UNIT I

Microcomputer: The term *microcomputer* is generally synonymous with personal computer, or a computer that depends on a microprocessor.

- Microcomputers are designed to be used by individuals, whether in the form of PCs, workstations or notebook computers.
- A microcomputer contains a CPU on a microchip (the microprocessor), a memory system (typically ROM and RAM), a bus system and I/O ports, typically housed in a motherboard.
- **Microprocessor**: A silicon chip that contains a CPU. In the world of personal computers, the terms *microprocessor* and CPU are used interchangeably.
- A **microprocessor** (sometimes abbreviated **µP**) is a digital electronic component with miniaturized transistors on a single semiconductor integrated circuit (IC).
- One or more microprocessors typically serve as a central processing unit (CPU) in a computer system or handheld device.
- Microprocessors made possible the advent of the microcomputer.
- At the heart of all personal computers and most working stations sits a microprocessor.
- Microprocessors also control the logic of almost all digital devices, from clock radios to fuel-injection systems for automobiles.
- Three basic characteristics differentiate microprocessors:
- **Instruction set**: The set of instructions that the microprocessor can execute.
- **Bandwidth**: The number of bits processed in a single instruction.
- **Clock speed**: Given in megahertz (MHz), the clock speed determines how many instructions per second the processor can execute.

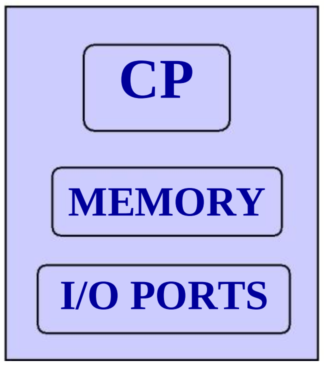
- In both cases, the higher the value, the more powerful the CPU. For example, a 32 bit microprocessor that runs at 50MHz is more powerful than a 16-bit microprocessor that runs at 25MHz.
- In addition to bandwidth and clock speed, microprocessors are classified as being either RISC (reduced instruction set computer) or CISC (complex instruction set computer).
- **Supercomputer**: A supercomputer is a computer that performs at or near the currently highest operational rate for computers.
- A supercomputer is typically used for scientific and engineering applications that must handle very large databases or do a great amount of computation (or both).
- At any given time, there are usually a few well-publicized supercomputers that operate at the very latest and always incredible speeds.
- The term is also sometimes applied to far slower (but still impressively fast) computers.
- Most supercomputers are really multiple computers that perform parallel processing.
- In general, there are two parallel processing approaches: symmetric multiprocessing (SMP) and massively parallel processing (MPP).
- **Microcontroller:** A highly integrated chip that contains all the components comprising a controller.
- Typically this includes a CPU, RAM, some form of ROM, I/O ports, and timers.
- Unlike a general-purpose computer, which also includes all of these components, a microcontroller is designed for a very specific task - to control a particular system.
- A microcontroller differs from a microprocessor, which is a general-purpose chip that is used to create a multi-function computer or device and requires multiple chips to handle various tasks.

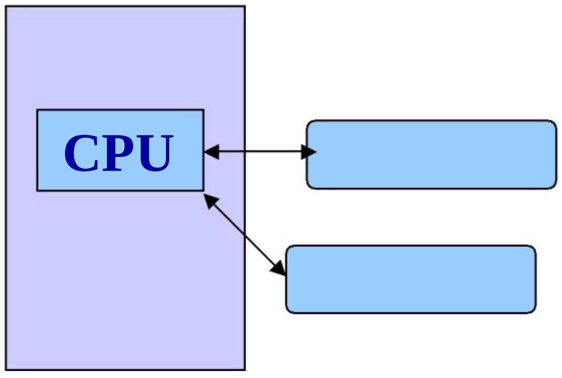
- A microcontroller is meant to be more self-contained and independent, and functions as a tiny, dedicated computer.
- The great advantage of microcontrollers, as opposed to using larger microprocessors, is that the parts-count and design costs of the item being controlled can be kept to a minimum.
- They are typically designed using CMOS (complementary metal oxide semiconductor) technology, an efficient fabrication technique that uses less power and is more immune to power spikes than other techniques.
- Microcontrollers are sometimes called *embedded microcontrollers*, which just means that they are part of an embedded system that is, one part of a larger device or system.
- **Controller:** A device that controls the transfer of data from a computer to a peripheral device and vice versa.
- For example, disk drives, display screens, keyboards and printers all require controllers.
- In personal computers, the controllers are often single chips.
- When you purchase a computer, it comes with all the necessary controllers for standard components, such as the display screen, keyboard, and disk drives.

- If you attach additional devices, however, you may need to insert new controllers that come on expansion boards.
- Controllers must be designed to communicate with the computer's expansion bus.
- There are three standard bus architectures for PCs - the AT bus, PCI (Peripheral Component Interconnect) and SCSI.
- When you purchase a controller, therefore, you must ensure that it conforms to the bus architecture that your computer uses.
- Short for ***P****eripheral* ***C****omponent* ***I****nterconnect,* a local bus standard developed by Intel Corporation.
- Most modern PCs include a PCI bus in addition to a more general IAS expansion bus.
- PCI is also used on newer versions of the Macintosh computer.
- PCI is a 64-bit bus, though it is usually implemented as a 32 bit bus. It can run at clock speeds of 33 or 66 MHz.
- At 32 bits and 33 MHz, it yields a throughput rate of 133 MBps.
- Short for ***s****mall* ***c****omputer* ***s****ystem* ***i****nterface*, a parallel interface standard used by Apple Macintosh computers, PCs, and many UNIX systems for attaching peripheral devices to computers.
- Nearly all Apple Macintosh computers, excluding only the earliest Macs and the recent iMac, come with a SCSI port for attaching devices such as disk drives and printers.
- SCSI interfaces provide for faster data transmission rates (up to 80 megabytes per second) than standard serial and parallel ports. In addition, you can attach many devices to a single SCSI port, so that SCSI is really an I/O bus rather than simply an interface
- Although SCSI is an ANSI standard, there are many variations of it, so two SCSI interfaces may be incompatible.
- For example, SCSI supports several types of connectors.
- While SCSI has been the standard interface for Macintoshes, the iMac comes with *IDE*, a less expensive interface, in which the controller is integrated into the disk or CD-ROM drive.
- The following varieties of SCSI are currently implemented:
- SCSI-1**:** Uses an 8-bit bus, and supports data rates of 4 MBps.

- SCSI-2**:** Same as SCSI-1, but uses a 50-pin connector instead of a 25-pin connector, and supports multiple devices. This is what most people mean when they refer to plain *SCSI*.
- Wide SCSI**:** Uses a wider cable (168 cable lines to 68 pins) to support 16-bit transfers.
- Fast SCSI**:** Uses an 8-bit bus, but doubles the clock rate to support data rates of 10 MBps.
- Fast Wide SCSI**:** Uses a 16-bit bus and supports data rates of 20 MBps.
- Ultra SCSI: Uses an 8-bit bus, and supports data rates of 20 MBps.
- Wide Ultra2 SCSI**:** Uses a 16-bit bus and supports data rates of 80 MBps.
- SCSI-3**:** Uses a 16-bit bus and supports data rates of 40 MBps. Also called *Ultra Wide SCSI*.
- Ultra2 SCSI**:** Uses an 8-bit bus and supports data rates of 40 MBps.
- **Embedded system**: A specialized computer system that is part of a larger system or machine.
- Typically, an embedded system is housed on a single microprocessor board with the programs stored in ROM.
- Virtually all appliances that have a digital Interface- watches, microwaves, VCRs, cars -utilize embedded systems.
- Some embedded systems include an operating system, but many are so specialized that the entire logic can be implemented as a single program.

MICRO CONTROLLER	**MICRO PROCESSER**
• It is a single chip	• It is a CPU
• Consists Memory, I/o ports	• Memory, I/O Ports to be connected externally





8085 Microprocessor
ContentsGeneral definitions

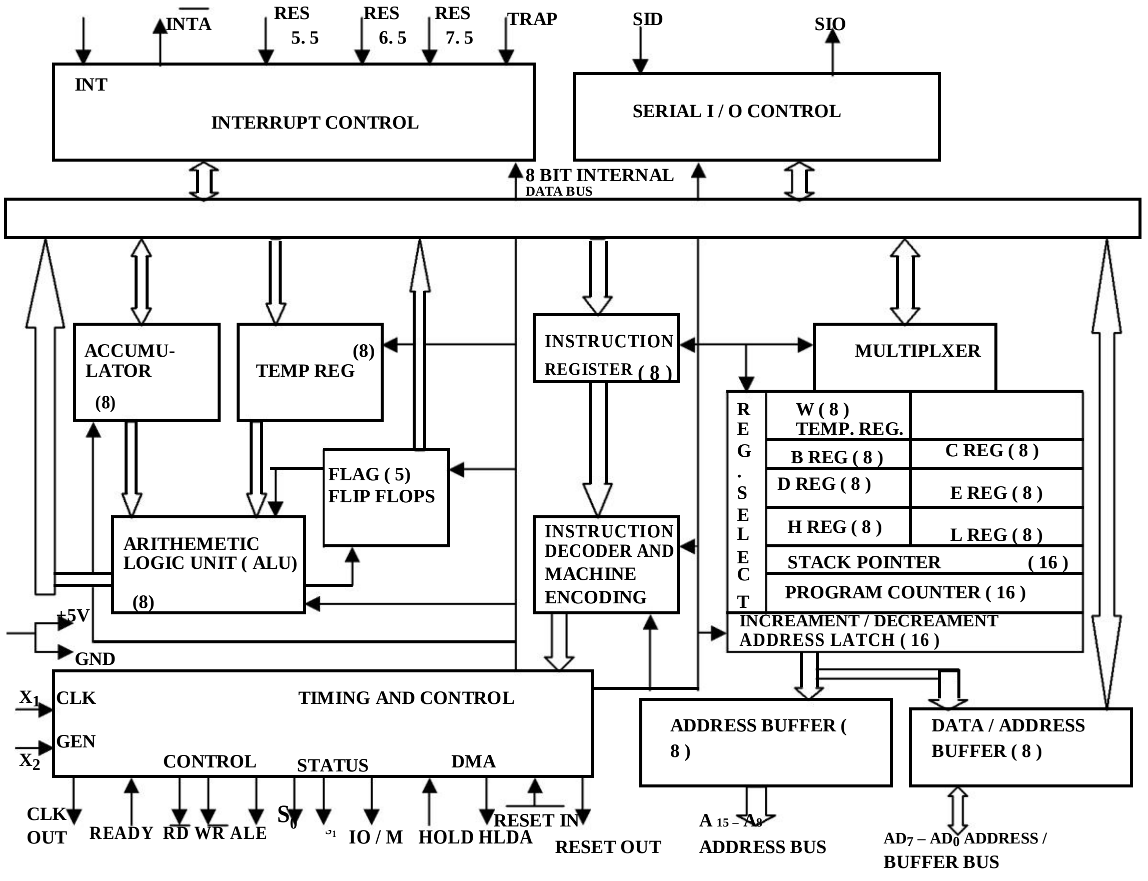
- Overview of 8085 microprocessor
- Overview of 8086 microprocessor

-
- Signals and pins of 8086 microprocessor

- The salient features of 8085 µp are:
- It is a 8 bit microprocessor.
- It is manufactured with N-MOS technology.
- It has 16-bit address bus and hence can address up to 216 = 65536 bytes (64KB) memory locations through A_0-A_{15}.
- The first 8 lines of address bus and 8 lines of data bus are multiplexed AD_0 – AD_7.
- Data bus is a group of 8 lines D_0 – D_7.
- It supports external interrupt request.
- A 16 bit program counter (PC)
- A 16 bit stack pointer (SP)
- Six 8-bit general purpose register arranged in pairs: BC, DE, HL.
- It requires a signal +5V power supply and operates at 3.2 MHZ single phase clock.
- It is enclosed with 40 pins DIP (Dual in line package).

Overview of 8085 microprocessor

- 8085 Architecture
- Pin Diagram
-



Block Diagram

Flag Registers

D7	D6	D5	D4	D3	D2	D1	D0
S	Z		AC		P		CY

General Purpose Registers

INDIVIDUAL	B, C, D, E, H, L

Memory

- Program, data and stack memories occupy the same memory space. The total addressable memory size is 64 KB.
- **Program memory** - program can be located anywhere in memory. Jump, branch and call instructions use 16-bit addresses, i.e. they can be used to jump/branch anywhere within 64 KB. All jump/branch instructions use absolute addressing.
- **Data memory** - the processor always uses 16-bit addresses so that data can be placed anywhere.
- **Stack memory** is limited only by the size of memory. Stack grows downward.
- First 64 bytes in a zero memory page should be reserved for vectors used by RST instructions.

Interrupts

- The processor has 5 interrupts. They are presented below in the order of their priority (from lowest to highest):
-
- **INTR** is maskable 8080A compatible interrupt. When the interrupt occurs the processor fetches from the bus one instruction, usually one of these instructions:
- One of the 8 RST instructions (RST_0 - RST_7). The processor saves current program counter into stack and branches to memory location N * 8 (where N is a 3-bit number from 0 to 7 supplied with the RST instruction).
- **CALL** instruction (3 byte instruction). The processor calls the subroutine, address of which is specified in the second and third bytes of the instruction.
- **RST5.5** is a maskable interrupt. When this interrupt is received the processor saves the contents of the PC register into stack and branches to 2CH (hexadecimal) address.
- **RST6.5** is a maskable interrupt. When this interrupt is received the processor saves the contents of the PC register into stack and branches to 34H (hexadecimal) address.
- **RST7.5** is a maskable interrupt. When this interrupt is received the processor saves the contents of the PC register into stack and branches to 3CH (hexadecimal) address.
- **TRAP** is a non-maskable interrupt. When this interrupt is received the processor saves the contents of the PC register into stack and branches to 24H (hexadecimal) address.
- All maskable interrupts can be enabled or disabled using EI and DI instructions. RST 5.5, RST6.5 and RST7.5 interrupts can be enabled or disabled individually using SIM instruction.

Reset Signals

- **RESET IN**: When this signal goes low, the program counter (PC) is set to Zero, µp is reset and resets the interrupt enable and HLDA flip-flops.
- The data and address buses and the control lines are 3-stated during RESET and because of asynchronous nature of RESET, the processor internal registers and flags may be altered by RESET with unpredictable results.
- RESET IN is a Schmitt-triggered input, allowing connection to an R-C network for power-on RESET delay.

- Upon power-up, RESET IN must remain low for at least 10 ms after minimum Vcc has been reached.
- For proper reset operation after the power – up duration, RESET IN should be kept low a minimum of three clock periods.
- The CPU is held in the reset condition as long as RESET IN is applied. Typical Power-on RESET RC values $R_1 = 75K\Omega$, $C_1 = 1\mu F$.
- **RESET OUT**: This signal indicates that µp is being reset. This signal can be used to reset other devices. The signal is synchronized to the processor clock and lasts an integral number of clock periods.

Serial communication Signal

- **SID - Serial Input Data Line**: The data on this line is loaded into accumulator bit 7 whenever a RIM instruction is executed.
- **SOD – Serial Output Data Line**: The SIM instruction loads the value of bit 7 of the accumulator into SOD latch if bit 6 (SOE) of the accumulator is 1.

DMA Signals

- **HOLD**: Indicates that another master is requesting the use of the address and data buses. The CPU, upon receiving the hold request, will relinquish the use of the bus as soon as the completion of the current bus transfer.
- Internal processing can continue. The processor can regain the bus only after the HOLD is removed.
- When the HOLD is acknowledged, the Address, Data RD, WR and IO/M lines are 3-stated.
- **HLDA: Hold Acknowledge**: Indicates that the CPU has received the HOLD request and that it will relinquish the bus in the next clock cycle.
- HLDA goes low after the Hold request is removed. The CPU takes the bus one half-clock cycle after HLDA goes low.
- **READY:** This signal Synchronizes the fast CPU and the slow memory, peripherals.
- If READY is high during a read or write cycle, it indicates that the memory or peripheral is ready to send or receive data.
- If READY is low, the CPU will wait an integral number of clock cycle for READY to go high before completing the read or write cycle.
- READY must conform to specified setup and hold times.

Registers

- **Accumulator** or A register is an 8-bit register used for arithmetic, logic, I/O and load/store operations.
- **Flag Register** has five 1-bit flags.
- **Sign** - set if the most significant bit of the result is set.
- **Zero** - set if the result is zero.
- **Auxiliary carry** - set if there was a carry out from bit 3 to bit 4 of the result.
- **Parity** - set if the parity (the number of set bits in the result) is even.
- **Carry** - set if there was a carry during addition, or borrow during subtraction/comparison/rotation.

General Registers

- 8-bit B and 8-bit C registers can be used as one 16-bit BC register pair. When used as a pair the C register contains low-order byte. Some instructions may use BC register as a data pointer.
- 8-bit D and 8-bit E registers can be used as one 16-bit DE register pair. When used as a pair the E register contains low-order byte. Some instructions may use DE register as a data pointer.
- 8-bit H and 8-bit L registers can be used as one 16-bit HL register pair. When used as a pair the L register contains low-order byte. HL register usually contains a data pointer used to reference memory addresses.
- **Stack pointer** is a 16 bit register. This register is always decremented/incremented by 2 during push and pop.
- **Program counter** is a 16-bit register.

Instruction Set

- 8085 instruction set consists of the following instructions:
- Data moving instructions.
- Arithmetic - add, subtract, increment and decrement.
- Logic - AND, OR, XOR and rotate.
- Control transfer - conditional, unconditional, call subroutine, return from subroutine and restarts.
- Input/Output instructions.
- Other - setting/clearing flag bits, enabling/disabling interrupts, stack operations, etc.

Addressing mode

- **Register** - references the data in a register or in a register pair.

  Register indirect - instruction specifies register pair containing address, where the data is located.

  Direct, Immediate - 8 or 16-bit data.

8086 Microprocessor

•It is a 16-bit µp.

•8086 has a 20 bit address bus can access up to 2^{20} memory locations (1 MB).

•It can support up to 64K I/O ports.

•It provides 14, 16 -bit registers.

•It has multiplexed address and data bus AD0- AD15 and A16 – A19.

•It requires single phase clock with 33% duty cycle to provide internal timing. •8086 is designed to operate in two modes, Minimum and Maximum.

•It can prefetches upto 6 instruction bytes from memory and queues them in order to speed up instruction execution.

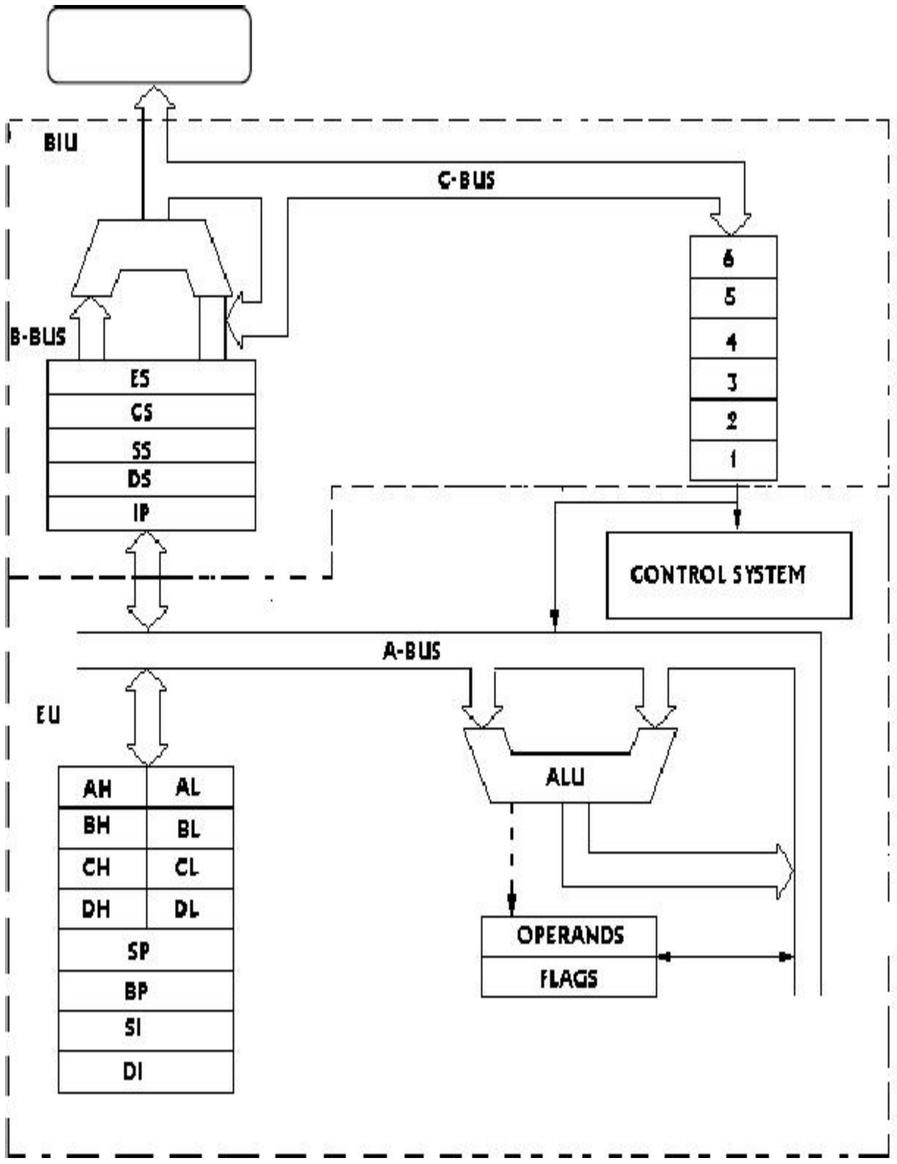
•It requires +5V power supply.

•A 40 pin dual in line package

Minimum and Maximum Modes:

•The minimum mode is selected by applying logic 1 to the MN / $\overline{MX}$ input pin. This is a single microprocessor configuration.

•The maximum mode is selected by applying logic 0 to the MN / MX input pin. This is a multi micro processors configuration.



Block Diagram of 8086

Internal Architecture of 8086

•8086 has two blocks BIU and EU.

•The BIU performs all bus operations such as instruction fetching, reading and writing operands for memory and calculating the addresses of the memory operands. The instruction bytes are transferred to the instruction queue.

•EU executes instructions from the instruction system byte queue.

•Both units operate asynchronously to give the 8086 an overlapping instruction fetch and execution mechanism which is called as Pipelining. This results in efficient use of the system bus and system performance.

•BIU contains Instruction queue, Segment registers, Instruction pointer, Address adder. •EU contains Control circuitry, Instruction decoder, ALU, Pointer and Index register, Flag register.

BUS INTERFACR UNIT:

•It provides a full 16 bit bidirectional data bus and 20 bit address bus.

•The bus interface unit is responsible for performing all external bus operations.

Specifically it has the following functions:

•Instruction fetch, Instruction queuing, Operand fetch and storage, Address relocation and Bus control.

•The BIU uses a mechanism known as an instruction stream queue to implement a ***pipeline architecture.***

•This queue permits prefetch of up to six bytes of instruction code. When ever the queue of the BIU is not full, it has room for at least two more bytes and at the same time the EU is not requesting it to read or write operands from memory, the BIU is free to look ahead in the program by prefetching the next sequential instruction.

•These prefetching instructions are held in its FIFO queue. With its 16 bit data bus, the BIU fetches two instruction bytes in a single memory cycle.

•After a byte is loaded at the input end of the queue, it automatically shifts up through the FIFO to the empty location nearest the output.

•The EU accesses the queue from the output end. It reads one instruction byte after the other from the output of the queue. If the queue is full and the EU is not requesting access to operand in memory.

•These intervals of no bus activity, which may occur between bus cycles are known as ***Idle state***.

•If the BIU is already in the process of fetching an instruction when the EU request it to read or write operands from memory or I/O, the BIU first completes the instruction fetch bus cycle before initiating the operand read / write cycle.

•The BIU also contains a dedicated adder which is used to generate the 20bit physical address that is output on the address bus. This address is formed by adding an appended 16 bit segment address and a 16 bit offset address.

•For example**:** The physical address of the next instruction to be fetched is formed by combining the current contents of the code segment CS register and the current contents of the instruction pointer IP register.

•The BIU is also responsible for generating bus control signals such as those for memory read or write and I/O read or write.

EXECUTION UNIT

The Execution unit is responsible for decoding and executing all instructions. •The EU extracts instructions from the top of the queue in the BIU, decodes them, generates operands if necessary, passes them to the BIU and requests it to perform the read or write bys cycles to memory or I/O and perform the operation specified by the instruction on the operands.

•During the execution of the instruction, the EU tests the status and control flags and updates them based on the results of executing the instruction.

•If the queue is empty, the EU waits for the next instruction byte to be fetched and shifted to top of the queue.

•When the EU executes a branch or jump instruction, it transfers control to a location corresponding to another set of sequential instructions.

•Whenever this happens, the BIU automatically resets the queue and then begins to fetch instructions from this new location to refill the queue.

Module 1 and learning unit 4:

Signal Description of 8086•The Microprocessor 8086 is a 16-bit CPU available in different clock rates and packaged in a 40 pin CERDIP or plastic package.

•The 8086 operates in single processor or multiprocessor configuration to achieve high performance. The pins serve a particular function in minimum mode (single processor

mode) and other function in maximum mode configuration (multiprocessor mode).
•The 8086 signals can be categorised in three groups. The first are the signal having common functions in minimum as well as maximum mode.
•The second are the signals which have special functions for minimum mode and third are the signals having special functions for maximum mode.
•**The following signal descriptions are common for both modes.**
•**AD15-AD0**: These are the time multiplexed memory I/O address and data lines.
• Address remains on the lines during T1 state, while the data is available on the data bus during T2, T3, Tw and T4.
•These lines are active high and float to a tristate during interrupt acknowledge and local bus hold acknowledge cycles.
•**A19/S6,A18/S5,A17/S4,A16/S3:** These are the time multiplexed address and status lines.
•During T1 these are the most significant address lines for memory operations.
•During I/O operations, these lines are low. During memory or I/O operations, status information is available on those lines for T2,T3,Tw and T4.
•The status of the interrupt enable flag bit is updated at the beginning of each clock cycle.
•The S4 and S3 combinedly indicate which segment register is presently being used for memory accesses as in below fig.
•These lines float to tri-state off during the local bus hold acknowledge. The status line S6 is always low.
•The address bit are separated from the status bit using latches controlled by the ALE signal.

• **$\overline{\text{BHE}}$ /S7:** The bus high enable is used to indicate the transfer of data over the higher order (D15-D8) data bus as shown in table. It goes low for the data transfer over D15-D8 and is used to derive chip selects of odd address memory bank or peripherals. BHE is low during T1 for read, write and interrupt acknowledge cycles, whenever a byte is to be transferred on higher byte of data bus. The status information is available during T2, T3 and T4. The signal is active low and tristated during hold. It is low during T1 for the first pulse of the interrupt acknowledges cycle.

• **$\overline{\text{RD}}$ Read:** This signal on low indicates the peripheral that the processor is performing s memory or I/O read operation. RD is active low and shows the state for T2, T3, Tw of any read cycle. The signal remains tristated during the hold acknowledge.

•**READY**: This is the acknowledgement from the slow device or memory that they have completed the data transfer. The signal made available by the devices is synchronized by the 8284A clock generator to provide ready input to the 8086. the signal is active high.
•**INTR-Interrupt Request**: This is a triggered input. This is sampled during the last clock cycles of each instruction to determine the availability of the request. If any interrupt request is pending, the processor enters the interrupt acknowledge cycle.
•This can be internally masked by resulting the interrupt enable flag. This signal is active high and internally synchronized.
• **$\overline{\text{TEST}}$** This input is examined by a 'WAIT' instruction. If the TEST pin goes low, execution will continue, else the processor remains in an idle state. The input is synchronized internally during each clock cycle on leading edge of clock.
•**CLK-** Clock Input: The clock input provides the basic timing for processor operation and bus control activity. Its an asymmetric square wave with 33% duty cycle.
•**MN/ $\overline{\text{MX}}$** : The logic level at this pin decides whether the processor is to operate in either minimum or maximum mode.
•**The following pin functions are for the minimum mode operation of 8086.**
•**M/ $\overline{\text{IO}}$ – Memory/IO**: This is a status line logically equivalent to S2 in maximum mode. When it is low, it indicates the CPU is having an I/O operation, and when it is high, it indicates that the CPU is having a memory operation. This line becomes active high in the previous T4 and remains active till final T4 of the current cycle. It is tristated during local bus "hold acknowledge ".
• **$\overline{\text{INTA}}$ Interrupt Acknowledge**: This signal is used as a read strobe for interrupt acknowledge cycles. i.e. when it goes low, the processor has accepted the interrupt.
•**ALE – Address Latch Enable**: This output signal indicates the availability of the valid address on the address/data lines, and is connected to latch enable input of latches. This signal is active high and is never tristated.
•**DT/ $\overline{\text{R}}$ – Data Transmit/Receive**: This output is used to decide the direction of data flow through the transreceivers (bidirectional buffers). When the processor sends out data, this signal is high and when the processor is receiving data, this signal is low.
•**DEN – Data Enable**: This signal indicates the availability of valid data over the address/data lines. It is used to enable the transreceivers (bidirectional buffers) to separate the data from the multiplexed address/data signal. It is active from the middle of T2 until the middle of T4. This is tristated during ' hold acknowledge' cycle.
•**HOLD, HLDA- Acknowledge**: When the HOLD line goes high, it indicates to the processor that another master is requesting the bus access.
•The processor, after receiving the HOLD request, issues the hold acknowledge signal on HLDA pin, in the middle of the next clock cycle after completing the current bus cycle.•At the same time, the processor floats the local bus and control lines. When the processor detects the HOLD line low, it lowers the HLDA signal. HOLD is an asynchronous input, and is should be externally synchronized.
•If the DMA request is made while the CPU is performing a memory or I/O cycle, it will release the local bus during T4 provided:
1.The request occurs on or before T2 state of the current cycle.
2.The current cycle is not operating over the lower byte of a word.
3.The current cycle is not the first acknowledge of an interrupt acknowledge sequence.

4. A Lock instruction is not being executed.

•***The following pin function are applicable for maximum mode operation of 8086***.

•**S2, S 1, S0 – Status Lines**: These are the status lines which reflect the type of operation, being carried out by the processor. These become activity during T4 of the previous cycle and active during T1 and T2 of the current bus cycles.

S_2	S_1	S_0	Indication
0	**0**	**0**	**Interrupt Acknowledge**
0	**0**	**1**	**Read I/O port**
0	**1**	**0**	**Write I/O port**
0	**1**	**1**	**Halt**
1	**0**	**0**	**Code Access**
1	**0**	**1**	**Read memory**
1	**1**	**0**	**Write memory**
1	**1**	**1**	**Passive**

• $\overline{\text{LOCK}}$ This output pin indicates that other system bus master will be prevented from gaining the system bus, while the LOCK signal is low.

•The LOCK signal is activated by the 'LOCK' prefix instruction and remains active until the completion of the next instruction. When the CPU is executing a critical instruction which requires the system bus, the LOCK prefix instruction ensures that other processors connected in the system will not gain the control of the bus.

•The 8086, while executing the prefixed instruction, asserts the bus lock signal output, which may be connected to an external bus controller.

•**QS_1, QS_0 – Queue Status:** These lines give information about the status of the code-prefetch queue. These are active during the CLK cycle after while the queue operation is performed.

•This modification in a simple fetch and execute architecture of a conventional microprocessor offers an added advantage of pipelined processing of the instructions.

•The 8086 architecture has 6-byte instruction prefetch queue. Thus even the largest (6 - bytes) instruction can be prefetched from the memory and stored in the prefetch. This results in a faster execution of the instructions.

•In 8085 an instruction is fetched, decoded and executed and only after the execution of this instruction, the next one is fetched.

•By prefetching the instruction, there is a considerable speeding up in instruction execution in 8086. This is known as ***instruction pipelining***.

•At the starting the CS:IP is loaded with the required address from which the execution is to be started. Initially, the queue will be empty an the microprocessor starts a fetch operation to bring one byte (the first byte) of instruction code, if the CS:IP address is odd or two bytes at a time, if the CS:IP address is even.

•The first byte is a complete opcode in case of some instruction (one byte opcode instruction) and is a part of opcode, in case of some instructions (two byte opcode instructions), the remaining part of code lie in second byte.

•The second byte is then decoded in continuation with the first byte to decide the instruction length and the number of subsequent bytes to be treated as instruction data.

•The queue is updated after every byte is read from the queue but the fetch cycle is initiated by BIU only if at least two bytes of the queue are empty and the EU may be concurrently executing the fetched instructions.
•The next byte after the instruction is completed is again the first opcode byte of the next instruction. A similar procedure is repeated till the complete execution of the program.•The fetch operation of the next instruction is overlapped with the execution of the current instruction. As in the architecture, there are two separate units, namely Execution unit and Bus interface unit.
•While the execution unit is busy in executing an instruction, after it is completely decoded, the bus interface unit may be fetching the bytes of the next instruction from memory, depending upon the queue status.

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

• $\overline{\mathbf{RQ}}$ **/** $\overline{\mathbf{GT_0}}$ **,**$\overline{\mathbf{RQ}}$ **/** $\overline{\mathbf{GT_1}}$ **– Request/Grant:** These pins are used by the other local bus master in maximum mode, to force the processor to release the local bus at the end of the processor current bus cycle.
•Each of the pin is bidirectional with RQ/GT0 having higher priority than RQ/GT1.
•RQ/GT pins have internal pull-up resistors and may be left unconnected.
•**Request/Grant sequence is as follows:**
1.A pulse of one clock wide from another bus master requests the bus access to 8086.
2.During T4(current) or T1(next) clock cycle, a pulse one clock wide from 8086 to the requesting master, indicates that the 8086 has allowed the local bus to float and that it will enter the 'hold acknowledge' state at next cycle. The CPU bus interface unit is likely to be disconnected from the local bus of the system.
3.A one clock wide pulse from the another master indicates to the 8086 that the hold request is about to end and the 8086 may regain control of the local bus at the next clock cycle. Thus each master to master exchange of the local bus is a sequence of 3 pulses. There must be at least one dead clock cycle after each bus exchange.
•The request and grant pulses are active low.
•For the bus request those are received while 8086 is performing memory or I/O cycle, the granting of the bus is governed by the rules as in case of HOLD and HLDA in minimum mode.
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 utilisation of processor pins and it facilitates the use of 40 pin standard DIP package.

•The bus can be demultiplexed using a few latches and transreceivers, when ever 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.

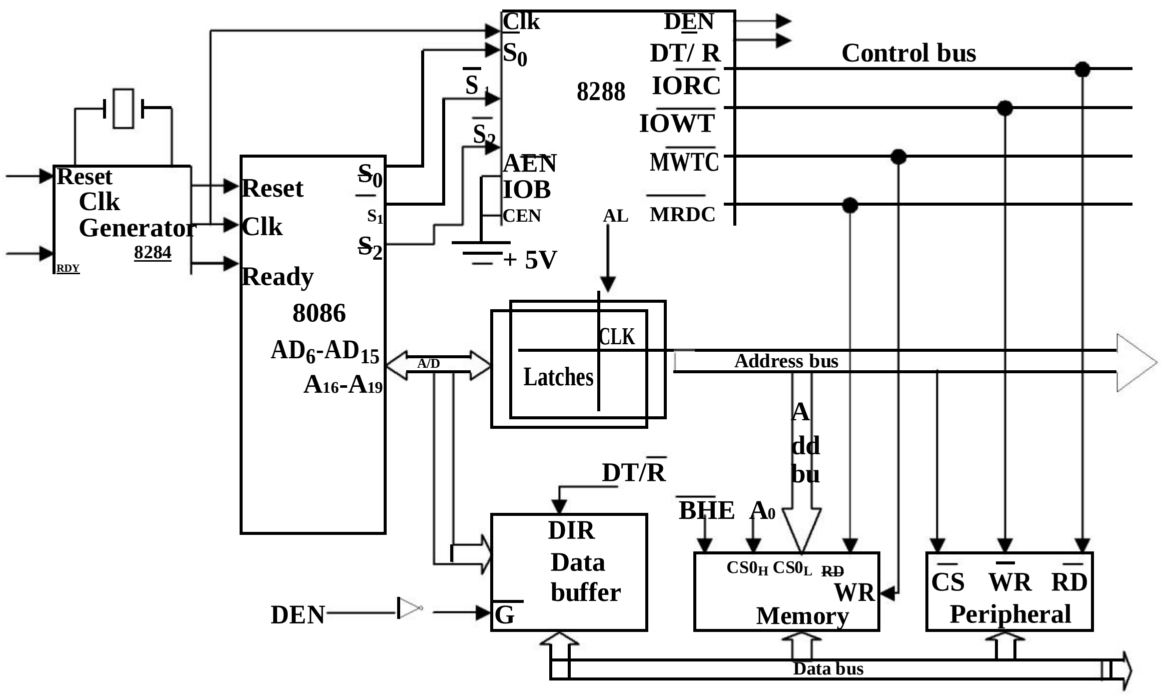
•The negative edge of this ALE pulse is used to separate the address and the data or status information. In maximum mode, the status lines S0, S1 and S2 are used to indicate the type of operation.
•Status bits S3 to S7 are multiplexed with higher order address bits and the BHE signal.
Address is valid during T1 while status bits S3 to S7 are valid during T2 through T4.

Minimum Mode 8086 System

•In a minimum mode 8086 system, the microprocessor 8086 is operated in minimum mode by strapping its MN/MX pin to logic 1.
•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, transreceivers, 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.
•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.

•Transreceivers are the bidirectional buffers and some times they are called as data amplifiers. They are required to separate the valid data from the time multiplexed address/data signals.
•They are controlled by two signals namely, DEN and DT/R.
•The DEN signal indicates the direction of data, i.e. from or to the processor. The system contains memory for the monitor and users program storage.
•Usually, EPROM are used for monitor storage, while RAM for users program storage. A system may contain I/O devices.
•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.
•The read cycle begins in T1 with the assertion of address latch enable (ALE) signal and also M / IO signal. During the negative going edge of this signal, the valid address is latched on the local bus.
•The BHE and A0 signals address low, high or both bytes. From T1 to T4 , the M/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 tristated. The read (RD) control signal is also activated in T2.
•The read (RD) signal causes the address device to enable its data bus drivers. After 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 tristate its bus drivers.
•A write cycle also begins with the assertion of ALE and the emission of the address. The M/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 WR becomes active at the beginning of T2 (unlike RD is somewhat delayed in T2 to provide time for floating).
•The BHE and A0 signals are used to select the proper byte or bytes of memory or I/O word to be read or write.
•The M/IO, RD and WR signals indicate the type of data transfer

Maximum Mode 8086 System •In the maximum mode, the 8086 is operated by strapping the MN/MX pin to ground.
•In this mode, the processor derives the status signal S2, S1, S0. 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.
•The components in the system are same as in the minimum mode system.
•The basic function of the bus controller chip IC8288, is to derive control signals like RD and WR (for memory and I/O devices), DEN, DT/R, ALE etc. using the information by the processor on the status lines.
•The bus controller chip has input lines S2, S1, S0 and CLK. These inputs to 8288 are driven by CPU.
•It derives the outputs ALE, DEN, DT/R, MRDC, MWTC, AMWC, IORC, IOWC and AIOWC. The AEN, IOB and CEN pins are specially useful for multiprocessor systems.
•AEN and IOB are generally grounded. CEN pin is usually tied to +5V. The significance of the MCE/PDEN output depends upon the status of the IOB pin.
•If IOB is grounded, it acts as master cascade enable to control cascade 8259A, else it acts as peripheral data enable used in the multiple bus configurations.
•INTA pin used to issue two interrupt acknowledge pulses to the interrupt controller or to an interrupting device.
•IORC, 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 address port.
•The MRDC, MWTC are memory read command and memory write command signals respectively and may be used as memory read or write signals.
•All these command signals instructs the memory to accept or send data from or to the bus.
•For both of these write command signals, the advanced signals namely AIOWC and AMWTC are available.
•Here the only difference between in timing diagram between minimum mode and maximum mode is the status signals used and the available control and advanced command signals.



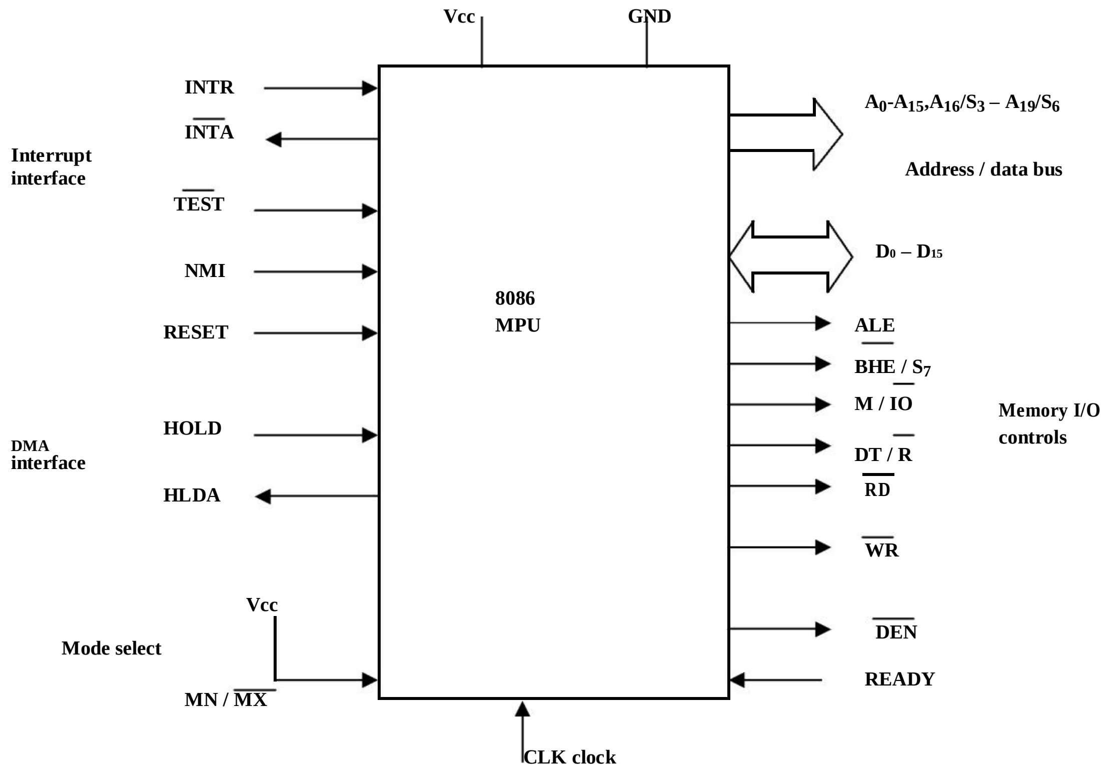
Maximum Mode 8086 System.

•R0, S1, S2 are set at the beginning of bus cycle.8288 bus controller will output a pulse as on the ALE and apply a required signal to its DT / R pin during T1.
•In T2, 8288 will set DEN=1 thus enabling transceivers, and for an input it will activate MRDC or IORC. These signals are activated until T4. For an output, the AMWC or AIOWC is activated from T2 to T4 and MWTC or IOWC is activated from T3 to T4. •The status bit S0 to S2 remains active until T3 and become passive during T3 and T 4. •If reader input is not activated before T3, wait state will be inserted between T3 and T4.

Minimum Mode Interface

•When the Minimum mode operation is selected, the 8086 provides all control signals needed to implement the memory and I/O interface.

•The minimum mode signal can be divided into the following basic groups: address/data bus, status, control, interrupt and DMA.
•**Address/Data Bus**: these lines serve two functions. As an address bus is 20 bits long and consists of signal lines A0 through A19. A19 represents the MSB and A0 LSB. A 20bit address gives the 8086 a 1Mbyte memory address space. More over it has an independent I/O address space which is 64K bytes in length.
•The 16 data bus lines D0 through D15 are actually multiplexed with address lines A0 through A15 respectively. By multiplexed we mean that the bus work as an address bus during first machine cycle and as a data bus during next machine cycles. D15 is the MSB and D0 LSB.
•When acting as a data bus, they carry read/write data for memory, input/output data for I/O devices, and interrupt type codes from an interrupt controller.



Block Diagram of the Minimum Mode 8086 MPU

•**Status signal:**

The four most significant address lines A19 through A16 are also multiplexed but in this case with status signals S6 through S3. These status bits are output on the bus at the same time that data are transferred over the other bus lines.

•Bit S4 and S3 together from a 2 bit binary code that identifies which of the 8086 internal segment registers are used to generate the physical address that was output on the address bus during the current bus cycle.

•Code S4S3 = 00 identifies a register known as ***extra segment register*** as the source of the segment address.

Status line S5 reflects the status of another internal characteristic of the 8086. It is the logic level of the internal enable flag. The last status bit S6 is always at the logic 0 level.

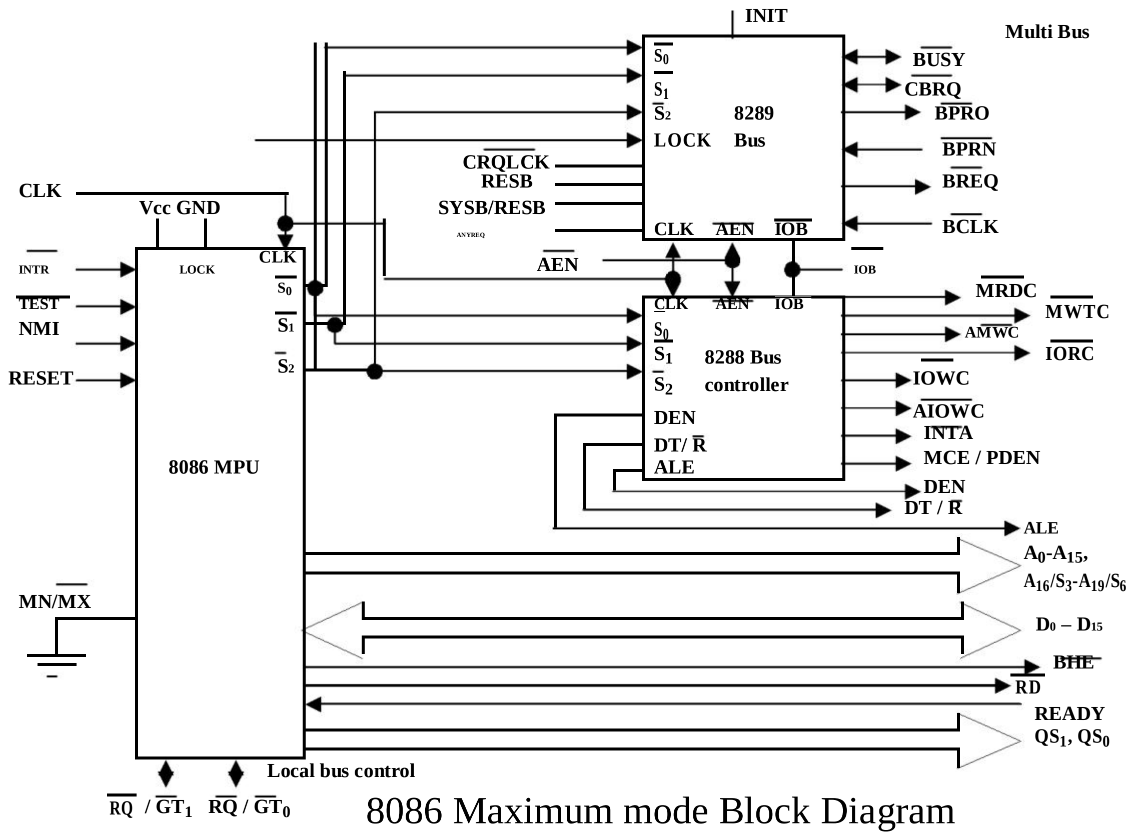
•**DMA Interface signals**:The direct memory access DMA interface of the 8086 minimum mode consist of the HOLD and HLDA signals.

•When an external device wants to take control of the system bus, it signals to the 8086 by switching HOLD to the logic 1 level. At the completion of the current bus cycle, the 8086 enters the hold state. In the hold state, signal lines AD0 through AD15, A16/S3 through A19/S6, BHE, M/IO, DT/R, RD, WR, DEN and INTR are all in the high Z state. The 8086 signals external device that it is in this state by switching its HLDA output to logic 1 level.

Maximum Mode Interface

•When the 8086 is set for the maximum-mode configuration, it provides signals for implementing a multiprocessor / coprocessor system environment.

•By multiprocessor environment we mean that one microprocessor exists in the system and that each processor is executing its own program.
•Usually in this type of system environment, there are some system resources that are common to all processors.
•They are called as ***global resources***. There are also other resources that are assigned to specific processors. These are known as ***local or private resources***.
•Coprocessor also means that there is a second processor in the system. In this two processor does not access the bus at the same time.
•One passes the control of the system bus to the other and then may suspend its operation.
•In the maximum-mode 8086 system, facilities are provided for implementing allocation of global resources and passing bus control to other microprocessor or coprocessor.



8086 Maximum mode Block Diagram

•**8288 Bus Controller – Bus Command and Control Signals**:
8086 does not directly provide all the signals that are required to control the memory, I/O and interrupt interfaces.
•Specially the WR, M/IO, DT/R, DEN, ALE and INTA, signals are no longer produced by the 8086. Instead it outputs three status signals S0, S1, S2 prior to the initiation of each bus cycle. This 3- bit bus status code identifies which type of bus cycle is to follow.
•S2S1S0 are input to the external bus controller device, the bus controller generates the appropriately timed command and control signals.

Queue status codes

•**Local Bus Control Signal – Request / Grant Signals**: In a maximum mode configuration, the minimum mode HOLD, HLDA interface is also changed. These two are replaced by request/grant lines RQ/ GT0 and RQ/ GT1, respectively. They provide a prioritized bus access mechanism for accessing the local bus.

Internal Registers of 8086

•The 8086 has four groups of the user accessible internal registers. They are the instruction pointer, four data registers, four pointer and index register, four segment registers.

•The 8086 has a total of fourteen 16-bit registers including a 16 bit register called the ***status register***, with 9 of bits implemented for status and control flags.

•Most of the registers contain data/instruction offsets within 64 KB memory segment. There are four different 64 KB segments for instructions, stack, data and extra data. To specify where in 1 MB of processor memory these 4 segments are located the processor uses 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.

•**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.

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.

•**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 and string manipulation.

•**Base** register consists of two 8-bit registers BL and BH, which can be combined together and used as a 16- bit register BX. BL in this case contains the low-order byte of the word, and BH contains the high-order byte. BX register usually contains a data pointer used for based, based indexed or register indirect addressing.

•**Count** register consists of two 8-bit registers CL and CH, which can be combined together and used as a 16-bit register CX. When combined, CL register contains the low - order byte of the word, and CH contains the high -order byte. Count register can be used in Loop, shift/rotate instructions and as a counter in string manipulation,.

•**Data** register consists of two 8-bit registers DL and DH, which can be combined together and used as a 16-bit register DX. When combined, DL register contains the low-order byte of the word, and DH contains the high -order byte. Data register can be used as a port number in I/O operations. In integer 32- bit multiply and divide instruction the DX register contains high-order word of the initial or resulting number.

•The following registers are both general and index registers:

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

•**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 address 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.

Other registers:

•**Instruction Pointer** (IP) is a 16-bit register.

•**Flags** is a 16-bit register containing 9 one bit flags.

•**Overflow Flag** (OF) - set if the result is too large positive number, or is too small negative number to fit into destination operand.

•**Direction Flag** (DF) - if set then string manipulation instructions will auto - decrement index registers. If cleared then the index registers will be auto-incremented.

•**Interrupt-enable Flag** (IF) - setting this bit enables maskable interrupts.

•**Single-step Flag** (TF) - if set then single-step interrupt will occur after the next instruction.

•**Sign Flag** (SF) - set if the most significant bit of the result is set.

•**Zero Flag** (ZF) - set if the result is zero.

•**Auxiliary carry Flag** (AF) - set if there was a carry from or borrow to bits 0-3 in the AL register.

•**Parity Flag** (PF) - set if parity (the number of "1" bits) in the low-order byte of the result is even.

•**Carry Flag** (CF) - set if there was a carry from or borrow to the most significant bit during last result calculation.

Addressing Modes

•**Implied** - the data value/data address is implicitly associated with the instruction.

•**Register** - references the data in a register or in a register pair.

•**Immediate** - the data is provided in the instruction.

•**Direct** - the instruction operand specifies the memory address where data is located.

•**Register indirect** - instruction specifies a register containing an address, where data is located. This addressing mode works with SI, DI, BX and BP registers.

•**Based** :- 8-bit or 16-bit instruction operand is added to the contents of a base register (BX or BP), the resulting value is a pointer to location where data resides.

•**Indexed**:- 8-bit or 16-bit instruction operand is added to the contents of an index register (SI or DI), the resulting value is a pointer to location where data resides

•**Based Indexed**: - the contents of a base register (BX or BP) is added to the contents of an index register (SI or DI), the resulting value is a pointer to location where data resides.

•**Based Indexed with displacement**:- 8-bit or 16-bit instruction operand is added to the contents of a base register (BX or BP) and index register (SI or DI), the resulting value is a pointer to location where data resides.

Memory •Program, data and stack memories occupy the same memory space. As the most of the processor instructions use 16-bit pointers the processor can effectively address only 64 KB of memory.

•To access memory outside of 64 KB the CPU uses special segment registers to specify where the code, stack and data 64 KB segments are positioned within 1 MB of memory (see the "Registers" section below).

•16-bit pointers and data are stored
as: address: low-order byte
address+1: high-order byte

•**Program memory** - program can be located anywhere in memory. Jump and call instructions can be used for short jumps within currently selected 64 KB code segment, as well as for far jumps anywhere within 1 MB of memory.

•All conditional jump instructions can be used to jump within approximately +127 to - 127 bytes from current instruction.

•**Data memory** - the processor can access data in any one out of 4 available segments, which limits the size of accessible memory to 256 KB (if all four segments point to different 64 KB blocks).
•Accessing data from the Data, Code, Stack or Extra segments can be usually done by prefixing instructions with the DS:, CS:, SS: or ES: (some registers and instructions by default may use the ES or SS segments instead of DS segment).
•Word data can be located at odd or even byte boundaries. The processor uses two memory accesses to read 16-bit word located at odd byte boundaries. Reading word data from even byte boundaries requires only one memory access.
•**Stack memory** can be placed anywhere in memory. The stack can be located at odd memory addresses, but it is not recommended for performance reasons (see "Data Memory" above).
Reserved locations:
•0000h - 03FFh are reserved for interrupt vectors. Each interrupt vector is a 32-bit pointer in format segment: offset.
•FFFF0h - FFFFFh - after RESET the processor always starts program execution at the FFFF0h address.
Interrupts
The processor has the following interrupts:

•**INTR** is a maskable hardware interrupt. The interrupt can be enabled/disabled using STI/CLI instructions or using more complicated method of updating the FLAGS register with the help of the POPF instruction.
•When an interrupt occurs, the processor stores FLAGS register into stack, disables further interrupts, fetches from the bus one byte representing interrupt type, and jumps to interrupt processing routine address of which is stored in location 4 * <interrupt type>. Interrupt processing routine should return with the IRET instruction.
•**NMI** is a non-maskable interrupt. Interrupt is processed in the same way as the INTR interrupt. Interrupt type of the NMI is 2, i.e. the address of the NMI processing routine is stored in location 0008h. This interrupt has higher priority then the maskable interrupt.
•**Software interrupts** can be caused by:
•INT instruction - breakpoint interrupt. This is a type 3 interrupt.
•INT <interrupt number> instruction - any one interrupt from available 256 interrupts.
•INTO instruction - interrupt on overflow
•Single-step interrupt - generated if the TF flag is set. This is a type 1 interrupt. When the CPU processes this interrupt it clears TF flag before calling the interrupt processing routine.
•**Processor exceptions**: Divide Error (Type 0), Unused Opcode (type 6) and Escape opcode (type 7).
•Software interrupt processing is the same as for the hardware interrupts.

UNIT II
Description of Instructions
Assembly directives
Algorithms with assembly software programs

Data Transfer Instructions GENERAL – PURPOSE BYTE OR WORD TRANSFER INSTRUCTIONS:
🕗MOV
🕗PUSH
🕗POP
🕗XCHG
🕗XLAT

SIMPLE INPUT AND OUTPUT PORT TRANSFER INSTRUCTIONS: 🕗IN
🕗OUT

SPECIAL ADDRESS TRANSFER INSTRUCTIONS 🕗LEA
🕗LDS
🕗LES

FLAG TRANSFER INSTRUCTIONS: 🕗LAHF
🕗SAHF
🕗PUSHF
🕗POPF

Arithmetic Instructions
ADITION INSTRUCTIONS:
🕗ADD
🕗ADC
🕗INC
🕗AAA
🕗DAA

SUBTRACTION INSTRUCTIONS:
🕗SUB
🕗SBB
🕗DEC
🕗NEG
🕗CMP
🕗AAS
🕗DAS

MULTIPLICATION INSTRUCTIONS: 🕗MUL
🕗IMUL
🕗AAM

DIVISION INSTRUCTIONS: 🕗DIV

🕘IDIV
🕘AAD
🕘CBW
🕘CWD

Bit Manipulation Instructions

LOGICAL INSTRUCTIONS:
🕘NOT
🕘AND
🕘OR
🕘XOR
🕘TEST

SHIFT INSTRUCTIONS:
🕘SHL / SAL
🕘SHR
🕘SAR

ROTATE
INSTRUCTIONS: 🕘ROL
🕘ROR
🕘RCL
🕘RCR

String Instructions

🕘REP
🕘REPE /
REPZ 🕘REPNE /
REPNZ
🕘MOVS / MOVSB /
MOVSW 🕘COMPS / COMPSB /
COMPSW 🕘SCAS / SCASB /
SCASW 🕘LODS / LODSB /
LODSW 🕘STOS / STOSB /
STOSW

Program Execution Transfer Instructions

UNCONDITIONAL TRANSFER
INSTRUCTIONS: 🕘CALL
🕘RET
🕘JMP

CONDITIONAL TRANSFER INSTRUCTIONS:
🕘JA / JNBE
🕘JAE / JNB
🕘JB / JNAE
🕘JBE /
JNA 🕘JC
🕘JE /
JZ 🕘JG /
JNLE 🕘J
GE /
JNL 🕘JL /
JNGE 🕘J
LE /
JNG 🕘JN
C 🕘JNE

🕗JNO 🕗J
NP /
JPO 🕗JN
S
🕗JO 🕗
JP /
JPE 🕗J
S
ITERATION CONTROL
INSTRUCTIONS: 🕗LOOP
🕗LOOPE /
LOOPZ 🕗LOOPNE
/ LOOPNZ 🕗JCXZ
INTERRUPT INSTRUCTIONS:
🕗INT
🕗INTO
🕗IRET

Process Control Instructions

FLAG SET / CLEAR INSTRUCTIONS:
🕗STC
🕗CLC
🕗CMC
🕗STD
🕗CLD
🕗STI
🕗CLI
EXTERNAL HARDWARE SYNCHRONIZATION
INSTRUCTIONS: 🕗HLT
🕗WAIT
🕗ESC
🕗LOCK
🕗NOP

Instruction Description

🕗**AAA** Instruction - ASCII Adjust after Addition
🕗**AAD** Instruction - ASCII adjust before Division
🕗**AAM** Instruction - ASCII adjust after Multiplication
🕗**AAS** Instruction - ASCII Adjust for Subtraction
🕗**ADC** Instruction - Add with carry.
🕗**ADD** Instruction - ADD destination, source
🕗**AND** Instruction - AND corresponding bits of two operands

Example

🕗**AAA** Instruction:

AAA converts the result of the addition of two valid unpacked BCD digits to a valid 2-digit BCD number and takes the AL register as its implicit operand.

Two operands of the addition must have its lower 4 bits contain a number in the range from 0-9.The AAA instruction then adjust AL so that it contains a correct BCD digit. If the addition produce carry (AF=1), the AH register is incremented and the carry CF and auxiliary carry AF flags are set to 1. If the addition did not produce a decimal carry, CF and AF are cleared to 0 and AH is not altered. In both cases the higher 4 bits of AL are cleared to 0.

AAA will adjust the result of the two ASCII characters that were in the range from 30h (“0”) to 39h(“9”).This is because the lower 4 bits of those character fall in the range of 0-9.The result of addition is not a ASCII character but it is a BCD digit.

🕗**Example:**

```
MOV  AH, 0  ; Clear AH for MSD
MOV  AL, 6  ; BCD 6 in AL
ADD  AL, 5  ; Add BCD 5 to digit in AL
AAA         ; AH=1, AL=1 representing BCD 11.
```

🕗**AAD Instruction:** ADD converts unpacked BCD digits in the AH and AL register into a single binary number in the AX register in preparation for a division operation.

Before executing AAD, place the Most significant BCD digit in the AH register and Last significant in the AL register. When AAD is executed, the two BCD digits are combined into a single binary number by setting AL=(AH*10)+AL and clearing AH to 0.

🕗**Example:**

```
MOV AX, 0205h    ; The unpacked BCD
AAD              number 25 ; After AAD,
                 AH=0 and
                 ; AL=19h (25)
```

After the division AL will then contain the unpacked BCD quotient and AH will contain the unpacked BCD remainder.

🕗**Example:**

```
                ; AX=0607 unpacked BCD for 67 decimal
                ; CH=09H
AAD             ; Adjust to binary before division
                ; AX=0043 = 43H =67 decimal
DIV CH          ; Divide AX by unpacked BCD in CH
                ; AL = quotient = 07 unpacked BCD
                ; AH = remainder = 04 unpacked BCD
```

🕗**AAM** Instruction - AAM converts the result of the multiplication of two valid unpacked BCD digits into a valid 2-digit unpacked BCD number and takes AX as an implicit operand.

To give a valid result the digits that have been multiplied must be in the range of 0 – 9 and the result should have been placed in the AX register. Because both operands of multiply are required to be 9 or less, the result must be less than 81 and thus is completely contained in AL.

AAM unpacks the result by dividing AX by 10, placing the quotient (MSD) in AH and the remainder (LSD) in AL.

🕗**Example:**

```
MOV     AL, 5
MOV     BL, 7
MUL     BL    ; Multiply AL by BL, result in AX
AAM           ; After AAM, AX =0305h (BCD 35)
```

🕗**AAS** Instruction: AAS converts the result of the subtraction of two valid unpacked BCD digits to a single valid BCD number and takes the AL register as an implicit operand.

The two operands of the subtraction must have its lower 4 bit contain number in the range from 0 to 9.The AAS instruction then adjust AL so that it contain a correct BCD digit.

```
        MOV  AX, 0901H      ; BCD 91
        SUB    AL, 9        ; Minus 9
        AAS                 ; Give AX =0802 h (BCD 82)


                            ( a )

                            ; AL =0011 1001 =ASCII 9
                            ; BL=0011 0101 =ASCII 5
        SUB    AL, BL       ; (9 - 5) Result:
                            ; AL = 00000100 = BCD 04, CF = 0
        AAS                 ; Result:
                            ; AL=00000100 =BCD 04
                            ; CF = 0 NO Borrow required


                       ( b )
                            ; AL = 0011 0101 =ASCII 5
                            ; BL = 0011 1001 = ASCII 9
SUB AL, BL                  ; ( 5 - 9 ) Result:
                            ; AL = 1111 1100 = - 4
                            ; in 2's complement CF = 1
AAS                         ; Results:
                            ; AL = 0000 0100 =BCD 04
                            ; CF = 1 borrow needed.
```

🕘**ADD** Instruction:

These instructions add a number from source to a number from some destination and put the result in the specified destination. The add with carry instruction ADC, also add the status of the carry flag into the result.

The source and destination must be of same type, means they must be a byte location or a word location. If you want to add a byte to a word, you must copy the byte to a word location and fill the upper byte of the word with zeroes before adding.

🕘**EXAMPLE:**

```
        ADD   AL, 74H       ; Add immediate number 74H to content of AL
        ADC   CL, BL                  ; Add contents of BL plus
                            ; carry status to contents of CL.
                            ; Results in CL
        ADD   DX, BX        ; Add contents of BX to contents
                            ; of DX
 ADD  DX, [SI]         ; Add word from memory at
                       ; offset [SI] in DS to contents of DX
                             ; Addition of Un Signed numbers
        ADD CL, BL ; CL = 01110011 =115 decimal
                            • + BL = 01001111 = 79 decimal
                            • Result in CL = 11000010 = 194 decimal
                            • Addition of Signed numbers
         ADD CL, BL ; CL = 01110011 = + 115 decimal
```

+ BL = 01001111 = +79 decimal
Result in CL = 11000010 = - 62 decimal

• **Incorrect because result is too large to fit in 7 bits.**

◷**AND** Instruction:

This Performs a bitwise Logical AND of two operands. The result of the operation is stored in the op1 and used to set the flags.

AND op1, op2

To perform a bitwise AND of the two operands, each bit of the result is set to 1 if and only if the corresponding bit in both of the operands is 1, otherwise the bit in the result I cleared to 0.

```
AND   BH, CL       ; AND byte in CL with byte in BH
                   ; result in BH
AND   BX, 00FFh    ; AND word in BX with immediate
                           ; 00FFH. Mask upper byte, leave
                           ; lower unchanged
AND   CX, [SI]     ; AND word at offset [SI] in data
                   ; segment with word in CX
                   ; register. Result in CX register.

                   ; BX = 10110011 01011110
AND   BX, 00FFh    ; Mask out upper 8 bits of BX
                   ; Result BX = 00000000 01011110
                   ; CF =0, OF = 0, PF = 0, SF = 0,
                   ; ZF = 0
```

◷**CALL** Instruction

•Direct within-segment (near or intrasegment)
•Indirect within-segment (near or intrasegment)
•Direct to another segment (far or intersegment)
•Indirect to another segment (far or intersegment)

◷**CBW** Instruction	-	Convert signed Byte to signed word
◷**CLC** Instruction	-	Clear the carry flag
◷**CLD** Instruction	-	Clear direction flag
◷**CLI** Instruction	-	Clear interrupt flag
◷**CMC** Instruction	-	Complement the carry flag
◷**CMP** Instruction	-	Compare byte or word - CMPdestination, source.
◷**CMPS/CMPSB/ CMPSW** Instruction	-	Compare string bytes or string words
◷**CWD** Instruction	-	Convert Signed Word to - Signed Double word

Example

◷**CALL** Instruction:

This Instruction is used to transfer execution to a subprogram or procedure. There are two basic types of CALL's: Near and Far.

A Near CALL is a call to a procedure which is in the same code segment as the CALL instruction.

When 8086 executes the near CALL instruction it decrements the stack pointer by two and copies the offset of the next instruction after the CALL on the stack. This offse saved on the stack is referred as the return address, because this is the address that execution will returns to after the procedure executes. A near CALL instruction will also load the instruction pointer with the offset of the first instruction in the procedure.

A RET instruction at the end of the procedure will return execution to the instruction after the CALL by coping the offset saved on the stack back to IP.

A Far CALL is a call to a procedure which is in a different from that which contains the CALL instruction. When 8086 executes the Far CALL instruction it decrements the stack pointer by two again and copies the content of CS register to the stack. It then decrements the stack pointer by two again and copies the offset contents offset of the instruction after the CALL to the stack.

Finally it loads CS with segment base of the segment which contains the procedure and IP with the offset of the first instruction of the procedure in segment. A RET instruction at end of procedure will return to the next instruction after the CALL by restoring the saved CS and IP from the stack.

; Direct within-segment (near or intrasegment)

CALL MULTO ; MULTO is the name of the procedure. The assembler determines displacement of MULTO from the instruction after the CALL and codes this displacement in as part of the instruction.

; Indirect within-segment (near or intrasegment)

CALL BX ; BX contains the offset of the first instruction of the procedure. Replaces contents of word of IP with contents o register BX.

CALL WORD PTR [BX] ; Offset of first instruction of procedure is in two memory addresses in DS. Replaces contents of IP with contents of word memory location in DS pointed to by BX.

; Direct to another segment- far or intersegment.

CALL SMART ; SMART is the name of the Procedure

SMART PROC FAR; Procedure must be declare as an far

🕗**CBW** Instruction - CBW converts the signed value in the AL register into an equivalent 16 bit signed value in the AX register by duplicating the sign bit to the left. This instruction copies the sign of a byte in AL to all the bits in AH. AH is then said to be the sign extension of AL.

Example:

; AX = 00000000 10011011 = - 155 decimal

CBW ; Convert signed byte in AL to signed word in AX.

Result in AX = 11111111 10011011

= - 155 decimal

🕗**CLC** Instruction:

CLC clear the carry flag (CF) to 0 This instruction has no affect on the processor, registers, or other flags. It is often used to clear the CF before returning from a procedure to indicate a successful termination. It is also use to clear the CF during rotate operation involving the CF such as ADC, RCL, RCR.

Example:

CLC ; Clear carry flag.

🕗**CLD** Instruction:

This instruction reset the designation flag to zero. This instruction has no effect on the registers or other flags. When the direction flag is cleared / reset SI and DI wil automatically be incremented when one of the string instruction such as MOVS, CMPS, SCAS, MOVSB and STOSB executes.

Example:

CLD ; Clear direction flag so that string pointers auto increment

🕘**CLI** Instruction:

This instruction resets the interrupt flag to zero. No other flags are affected. If the interrupt flag is reset, the 8086 will not respond to an interrupt signal on its INTR input. This CLI instruction has no effect on the nonmaskable interrupt input, NMI

🕘**CMC** Instruction:

If the carry flag CF is a zero before this instruction, it will be set to a one after the instruction. If the carry flag is one before this instruction, it will be reset to a zero after the instruction executes. CMC has no effect on other flags.

Example:

CMC; Invert the carry flag.

🕘**CWD** Instruction:

CWD converts the 16 bit signed value in the AX register into an equivalent 32 bit signed value in DX: AX register pair by duplicating the sign bit to the left.

The CWD instruction sets all the bits in the DX register to the same sign bit of the AX register. The effect is to create a 32- bit signed result that has same integer value as the original 16 bit operand.

Example:

Assume AX contains C435h. If the CWD instruction is executed, DX will contain FFFFh since bit 15 (MSB) of AX was 1. Both the original value of AX (C435h) and resulting value of DX: AX (FFFFC435h) represents the same signed number.

Example:

```
                 ; DX = 00000000 00000000
                 ; AX = 11110000 11000111 = - 3897 decimal
      CWD        ; Convert signed word in AX to signed double
                 ; word in DX:AX
                 ; Result DX = 11111111 11111111
                 ; AX = 11110000 11000111 = -3897 decimal.
```

🕘**DAA** Instruction - Decimal Adjust Accumulator

🕘**DAS** Instruction - Decimal Adjust after Subtraction

🕘**DEC** Instruction - Decrement destination register or memory DEC destination.

🕘**DIV** Instruction - Unsigned divide-Div source

🕘**ESC** Instruction

When a double word is divided by a word, the most significant word of the double word must be in DX and the least significant word of the double word must be in AX. After the division AX will contain the 16 –bit result (quotient) and DX will contain a 16 bit remainder. Again, if an attempt is made to divide by zero or quotient is too large to fit in AX (greater than FFFFH) the 8086 will do a type of 0 interrupt.

Example:

```
DIV   CX    ; (Quotient) AX= (DX: AX)/CX
            : (Reminder) DX= (DX: AX)%CX
```

For DIV the dividend must always be in AX or DX and AX, but the source of the divisor can be a register or a memory location specified by one of the 24 addressing modes.

If you want to divide a byte by a byte, you must first put the dividend byte in AL and fill AH with all 0's. The SUB AH, AH instruction is a quick way to do.

If you want to divide a word by a word, put the dividend word in AX and fill DX with all 0's. The SUB DX, DX instruction does this quickly.

🕘**Example:**

```
                        ; AX = 37D7H = 14, 295 decimal
                        ; BH = 97H = 151 decimal
            DIV BH      ; AX / BH
                        ; AX = Quotient = 5EH = 94 decimal
                        ; AH = Remainder = 65H = 101 decimal
```

🕘**ESC** Instruction - Escape instruction is used to pass instruction to a coprocessor such as the 8087 math coprocessor which shares the address and data bus with an 8086. Instruction for the coprocessor is represented by a 6 bit code embedded in the escape instruction. As the 8086 fetches instruction byte, the coprocessor also catches these bytes from data bus and puts them in its queue. The coprocessor treats all of the 8086 instruction as an NOP. When 8086 fetches an ESC instruction, the coprocessor decodes the instruction and carries out the action specified by the 6 bit code. In most of the case 8086 treats ESC instruction as an NOP.

🕘**HLT** Instruction - HALT processing

🕘**IDIV** Instruction - Divide by signed byte or word IDIV source

🕘**IMUL** Instruction - Multiply signed number-IMUL source

🕘**IN** Instruction - Copy data from a port
IN accumulator, port

🕘**INC** Instruction - Increment - INC destination

🕘**HALT** Instruction - The HLT instruction will cause the 8086 to stop fetching and executing instructions. The 8086 will enter a halt state. The only way to get the processor out of the halt state are with an interrupt signal on the INTR pin or an interrupt signal on NMI pin or a reset signal on the RESET input.

🕘**IDIV** Instruction - This instruction is used to divide a signed word by a signed byte or to divide a signed double word by a signed word.

🕘**Example:**

```
    IDIV   BL     ; Signed word in AX is divided by signed  byte in BL
```

🕘**IMUL** Instruction - This instruction performs a signed multiplication.

IMUL op ; In this form the accumulator is the multiplicand and op is the multiplier. op may be a register or a memory operand.

IMUL op1, op2 ; In this form op1 is always be a register operand and op2 may be a register or a memory operand.

🕘**Example:**

```
    IMUL BH     ; Signed byte in AL times multiplied by
                ; signed byte in BH and result in AX.
```

🕘**Example:**

```
                              ; 69 * 14
                              ; AL = 01000101 = 69 decimal
                              ; BL = 00001110 = 14 decimal



    IMUL BL                   ; AX = 03C6H        = + 966 decimal
```

```
                              ; MSB = 0 because positive result

                              ; - 28 * 59
                              ; AL = 11100100 = - 28 decimal
                              ; BL = 00001110 = 14 decimal
    IMUL BL                   ; AX = F98Ch = - 1652 decimal
                              ; MSB = 1 because negative result
```

🕘**IN** Instruction: This IN instruction will copy data from a port to the AL or AX register. For the Fixed port IN instruction type the 8 – bit port address of a port is specified
directly in the instruction.

🕘**Example:**

```
    IN    AL, 0C8H     ; Input a byte from port 0C8H to AL
    IN    AX, 34H      ; Input a word from port 34H to AX
    A_TO_D      EQU  4AH
    IN    AX, A_TO_D ; Input a word from port 4AH to AX
```

For a variable port IN instruction, the port address is loaded in DX register before IN instruction. DX is 16 bit. Port address range from 0000H – FFFFH.

🕘**Example:**

```
    MOV  DX, 0FF78H           ; Initialize DX point to port
    IN   AL, DX               ; Input a byte from a 8 bit port
                              ; 0FF78H to AL
    IN   AX, DX               ; Input a word from 16 bit port to
                              ; 0FF78H to AX.
```

🕘**INC** Instruction:

INC instruction adds one to the operand and sets the flag according to the result. INC instruction is treated as an unsigned binary number.

🕘**Example:**

```
                        ; AX = 7FFFh
        INC AX          ; After this instruction AX = 8000h
        INC    BL       ; Add 1 to the contents of BL register
        INC    CL       ; Add 1 to the contents of CX register.
```

🕘**INT** Instruction	-	Interrupt program
🕘**INTO** Instruction	-	Interrupt on overflow.
🕘**IRET** Instruction	-	Interrupt return
🕘**JA/JNBE** Instruction	-	Jump if above/Jump if not below nor equal.
🕘**JAE/JNB/JNC** Instructions	-	Jump if above or equal/ Jump if not below/ Jump if no carry.

🕘**JA / JNBE** - This instruction performs the Jump if above (or) Jump if not below or equal operations according to the condition, if CF and ZF = 0.

🕘**Example:**

(1)

```
                 CMP   AX, 4371H     ; Compare ( AX – 4371H)
                 JNBE  RUN_PRESS ; Jump to label RUN_PRESS if
                                     ; AX not below or equal to 4371H
```

⌚**JAE / JNB / JNC** - This instructions performs the Jump if above or equal, Jump if not below, Jump if no carry operations according to the condition, if CF = 0.

⌚**Examples**
:

```
1.   CMP   AX, 4371H     ; Compare ( AX – 4371H)
     JAE   RUN           ; Jump to the label RUN if AX is
                         ; above or equal to 4371H.
2.   CMP   AX, 4371H     ; Compare ( AX – 4371H)
     JNB   RUN_1         ; Jump to the label RUN_1 if AX
                         ; is not below than 4371H
3.   ADD   AL, BL        ; Add AL, BL. If result is with in JNC OK
                         ; acceptable range, continue
```

⌚**JB/JC/JNAE** Instruction - Jump if below/Jump if carry/ Jump if not above nor equal

⌚**JBE/JNA** Instructions- Jump if below or equal / Jump if not above

⌚**JCXZ** Instruction - Jump if the CX register is zero

⌚**JE/JZ** Instruction - Jump if equal/Jump if zero

⌚**JG/JNLE** Instruction- Jump if greater/Jump if not less than nor equal

⌚**JB/JC/JNAE** Instruction - This instruction performs the Jump if below (or) Jump if carry (or) Jump if not below/ equal operations according to the condition, if CF = 1

⌚**Example:**

```
1. CMP     AX, 4371H     ; Compare (AX – 4371H)
     JB    RUN_P         ; Jump to label RUN_P if AX is
                         ; below 4371H

2. ADD     BX, CX                ; Add two words and Jump to
     JC    ERROR         ; label ERROR if CF = 1
```

⌚**JBE/JNA** Instruction - This instruction performs the Jump if below or equal (or) Jump if not above operations according to the condition, if CF and ZF = 1

⌚**Example:**

```
     CMP   AX, 4371H     ; Compare (AX – 4371H )
     JBA   RUN           ; Jump to label RUN if AX is
                         ; below or equal to 4371H
     CMP   AX, 4371H     ; Compare ( AX – 4371H )
     JNA   RUN_R         ; Jump to label RUN_R if AX is
                         ; not above than 4371H
```

⌚**JCXZ** Instruction:

This instruction performs the Jump if CX register is zero. If CX does not contain all zeros, execution will simply proceed to the next instruction.

🕗**Example:**

```
    JCXZ SKIP_LOOP; If CX = 0, skip the process
    NXT: SUB [BX], 07H      ; Subtract 7 from data value
         INC BX             ; BX point to next value
    LOOP      NXT           ; Loop until CX = 0
         SKIP_LOOP          ; Next instruction
```

🕗**JE/JZ** Instruction:

This instruction performs the Jump if equal (or) Jump if zero operations according to the condition if ZF = 1

🕗**Example:**

```
   NXT: CMP       BX, DX ; Compare ( BX – DX )
         JE       DONE ; Jump to DONE if BX = DX,
         SUB      BX, AX ; Else subtract Ax
         INC      CX      ; Increment counter
         JUMP     NXT     ; Check again
        DONE: MOV AX, CX; Copy count to AX
```

🕗**Example:**

```
        IN    AL, 8FH      ; read data from port 8FH
        SUB   AL, 30H      ; Subtract minimum value
        JZ    STATR        ; Jump to label if result of
                           ; subtraction was 0
```

🕗**JG/JNLE** Instruction:

This instruction performs the Jump if greater (or) Jump if not less than or equal operations according to the condition if ZF =0 and SF = OF

🕗**Example:**

```
        CMP        BL, 39H      ; Compare by subtracting
                                ; 39H from BL
        JG         NEXT1        ; Jump to label if BL is
                                ; more positive than 39H
        CMP        BL, 39H      ; Compare by subtracting
                                          ; 39H from BL
        JNLE       NEXT2        ; Jump to label if BL is not
                                ; less than or equal 39H
```

🕗**JGE/JNL** Instruction - Jump if greater than or equal/ Jump if not less than

🕗**JL/JNGE** Instruction - Jump if less than/Jump if not greater than or equal

🕗**JLE/JNG** Instruction - Jump if less than or equal/ Jump if not greater

🕗**JMP** Instruction - Unconditional jump to - specified destination

🕗**JGE/JNL** Instruction - This instruction performs the Jump if greater than or equal / Jump if not less than operation according to the condition if SF = OF

🕗**Example:**

```
        CMP        BL, 39H      ; Compare by the
                                ; subtracting 39H from BL
        JGE        NEXT11       ; Jump to label if BL is
                                ; more positive than 39H
                                ; or equal to 39H
        CMP        BL, 39H      ; Compare by subtracting
```

```
                              ; 39H from BL
        JNL        NEXT22     ; Jump to label if BL is not
                              ; less than 39H
```

🕘**JL/JNGE** Instruction - This instruction performs the Jump if less than / Jump if not greater than or equal operation according to the condition, if SF ≠ OF

🕘**Example:**

```
        CMP   BL, 39H     ; Compare by subtracting 39H
                          ; from BL
        JL    AGAIN       ; Jump to the label if BL is more
                          ; negative than 39H
        CMP   BL, 39H     ; Compare by subtracting 39H
                          ; from BL
        JNGE AGAIN1       ; Jump to the label if BL is not
                          ; more positive than 39H or
                          ; not equal to 39H
```

🕘**JLE/JNG** Instruction - This instruction performs the Jump if less than or equal / Jump if not greater operation according to the condition, if ZF=1 and SF ≠ OF

🕘**Example:**

```
    CMP   BL, 39h      ; Compare by subtracting 39h
                       ; from BL
    JLE         NXT1 ; Jump to the label if BL is more
                ; negative than 39h or equal to 39h
    CMP   BL, 39h      ; Compare by subtracting 39h
                       ; from BL
    JNG   AGAIN2       ; Jump to the label if BL is not
                       ; more positive than 39h
```

🕘**JNA/JBE** Instruction - Jump if not above/Jump if below or equal

🕘**JNAE/JB** Instruction - Jump if not above or equal/ Jump if below

🕘**JNB/JNC/JAE** Instruction - Jump if not below/Jump if no carry/Jump if above or equal

🕘**JNE/JNZ** Instruction - Jump if not equal/Jump if not zero

🕘**JNE/JNZ** Instruction - This instruction performs the Jump if not equal / Jump if not zero operation according to the condition, if ZF=0

🕘**Example:**

```
 NXT: IN    AL, 0F8H       ; Read data value from port
            CMP  AL, 72    ; Compare ( AL – 72 )
```

```
                 JNE   NXT          ; Jump to NXT if AL ≠ 72
                 IN    AL, 0F9H     ; Read next port when AL = 72
                 MOV BX, 2734H      ; Load BX as counter
NXT_1:          ADD   AX, 0002H     ; Add count factor to AX
                DEC   BX            ; Decrement BX
        JNZ     NXT_1          ; Repeat until BX = 0
```

🕘**JNG/JLE** Instruction - Jump if not greater/ Jump if less than or equal

🕘**JNGE/JL** Instruction - Jump if not greater than nor equal/Jump if less than

🕘**JNL/JGE** Instruction - Jump if not less than/ Jump if greater than or equal

🕘**JNLE/JG** Instruction - Jump if not less than nor equal to /Jump if greater than

🕘**JNO** Instruction – Jump if no overflow

🕘**JNP/JPO** Instruction – Jump if no parity/ Jump if parity odd

🕘**JNS** Instruction - Jump if not signed (Jump if positive)

🕘**JNZ/JNE** Instruction - Jump if not zero / jump if not equal

🕘**JO** Instruction - Jump if overflow

🕘**JNO** Instruction – This instruction performs the Jump if no overflow operation according to the condition, if OF=0

🕘**Example:**

```
                ADD   AL, BL        ; Add signed bytes in AL and BL
                JNO   DONE          ; Process done if no overflow        -
                MOV  AL, 00H        ; Else load error code in AL
        DONE: OUT 24H, AL           ; Send result to display
```

🕘**JNP/JPO** Instruction – This instruction performs the Jump if not parity / Jump if parity odd operation according to the condition, if PF=0

🕘**Example:**

```
                IN    AL, 0F8H      ; Read ASCII char from UART
                OR    AL, AL        ; Set flags
                JPO   ERROR1        ; If even parity executed, if not
                                    ; send error message
```

🕘**JNS** Instruction - This instruction performs the Jump if not signed (Jump if positive) operation according to the condition, if SF=0

🕘**Example:**

```
        DEC   AL     ; Decrement counter
        JNS   REDO ; Jump to label REDO if counter has not
                   ; decremented to FFH
```

🕘**JO** Instruction - This instruction performs Jump if overflow operation according to the condition OF = 0

🕘**Example:**

```
        ADD   AL, BL         ; Add signed bits in AL and BL
        JO    ERROR          ; Jump to label if overflow occur
                             ; in addition
```

```
                              ; location named SUM
```
🕘**JPE/JP** Instruction - Jump if parity even/ Jump if parity

🕘**JPO/JNP** Instruction - Jump if parity odd/ Jump if no parity

🕘**JS** Instruction - Jump if signed (Jump if negative)

🕘**JZ/JE** Instruction - Jump if zero/Jump if equal

🕘**JPE/JP** Instruction - This instruction performs the Jump if parity even / Jump if parity operation according to the condition, if PF=1

Example:

```
          IN     AL, 0F8H     ; Read ASCII char from UART
          OR     AL, AL       ; Set flags
          JPE    ERROR2       ; odd parity is expected, if not
                              ; send error message
```

🕘**JS** Instruction - This instruction performs the Jump if sign operation according to the condition, if SF=1

🕘**Example:**

```
          ADD    BL, DH       ; Add signed bytes DH to BL
          JS     JJS_S1       ; Jump to label if result is
                              ; negative
```

🕘**LAHF** Instruction - Copy low byte of flag register to AH

🕘**LDS** Instruction - Load register and Ds with words from memory – LDS register, memory address of first word

🕘**LEA** Instruction - Load effective address-LEA register, source

🕘**LES** Instruction

Load register and ES with words from memory –LES register, memory address of first word.

🕘**LAHF** Instruction:

LAHF instruction copies the value of SF, ZF, AF, PF, CF, into bits of 7, 6, 4, 2, 0 respectively of AH register. This LAHF instruction was provided to make conversion of assembly language programs written for 8080 and 8085 to 8086 easier.

🕘**LDS** Instruction:

This instruction loads a far pointer from the memory address specified by op2 into the DS segment register and the op1 to the register.

LDS op1, op2

🕘**Example:**

LDS BX, [4326] ; copy the contents of the memory at displacement 4326H in DS to BL, contents of the 4327H to BH. Copy contents of 4328H and 4329H in DS to DS register.

🕘**LEA** Instruction - This instruction indicates the offset of the variable or memory location named as the source and put this offset in the indicated 16 – bit register.

🕘**Example:**

```
          LEA    BX, PRICE    ; Load BX with offset of PRICE
                              ; in DS
          LEA    BP, SS:STAK; Load BP with offset of STACK
                              ; in SS
```

```
LEA    CX, [BX][DI] ; Load CX with EA=BX + DI
```

◷**LOCK** Instruction - Assert bus lock signal

◷**LODS/LODSB/**
LODSW Instruction - Load string byte into AL or
Load string word into AX.

◷**LOOP** Instruction - Loop to specified
label until CX = 0

◷**LOOPE /**
LOOPZ Instruction - loop while CX ≠ 0 and
ZF = 1

◷**LODS/LODSB/LODSW** Instruction - This instruction copies a byte from a string location pointed to by SI to AL or a word from a string location pointed to by SI to AX. If DF is cleared to 0, SI will automatically incremented to point to the next element of string.

◷**Example:**

```
CLD   ; Clear direction flag so SI is auto incremented
MOV  SI, OFFSET SOURCE_STRING    ; point SI at  start of the string
LODS SOUCE_STRING      ; Copy byte or word from
                       ; string to AL or AX
```

◷**LOOP** Instruction - This instruction is used to repeat a series of instruction some number of times

◷**Example:**

```
         MOV  BX, OFFSET PRICE
                        ; Point BX at first element in array
         MOV  CX, 40          ; Load CX with number of
                              ; elements in array
NEXT: MOV AL, [BX]           ; Get elements from array
         ADD   AL, 07H        ; Ad correction factor
         DAA                  ; decimal adjust result
         MOV  [BX], AL        ; Put result back in array
         LOOP NEXT            ; Repeat until all elements
                              ; adjusted.
```

◷**LOOPE / LOOPZ** Instruction - This instruction is used to repeat a group of instruction some number of times until CX = 0 and ZF = 0

◷**Example:**

```
MOV  BX, OFFSET ARRAY
                      ; point BX at start of the array
DEC   BX
MOV  CX, 100          ; put number of array elements in
                      ; CX
NEXT:INC    BX        ; point to next element in array



CMP   [BX], 0FFH      ; Compare array elements FFH
LOOP NEXT
```

◷**LOOPNE/LOOPNZ** Instruction - This instruction is used to repeat a group of instruction some number of times until CX = 0 and ZF = 1

◷**Example:**

```
MOV  BX, OFFSET ARRAY1
                      ; point BX at start of the array
DEC   BX
```

```
                MOV  CX, 100        ; put number of array elements in
                                    ; CX
        NEXT:INC    BX              ; point to next elements in array
                CMP  [BX], 0FFH     ; Compare array elements 0DH
                LOOPNE NEXT
```

◴**MOV** Instruction - MOV destination, source

◴**MOVS/MOVSB/**
MOVSW Instruction - Move string byte or string word-MOVS destination, source

◴**MUL** Instruction - Multiply unsigned bytes or words-MUL source

◴**NEG** Instruction - From 2's complement – NEG destination

◴**NOP** Instruction - Performs no operation.

◴**MOV** Instruction - The MOV instruction copies a word or a byte of data from a specified source to a specified destination.

MOV op1, op2

◴**Example:**

```
        MOV  CX, 037AH    ; MOV 037AH into the CX.
        MOV  AX, BX       ; Copy the contents of register BX
                          ; to AX
        MOV  DL, [BX]     ; Copy byte from memory at BX
                          ; to DL, BX contains the offset of byte in DS.
```

◴**MUL** Instruction:

This instruction multiplies an unsigned multiplication of the accumulator by the operand specified by op. The size of op may be a register or memory operand.

```
                    MUL  op
        Example:          ; AL = 21h (33 decimal)
                          ; BL = A1h(161 decimal )
                MUL  BL   ; AX =14C1h (5313 decimal) since AH≠0,
                          ; CF and OF will set to 1.
                MUL  BH   ; AL times BH, result in AX
                MUL  CX   ; AX times CX, result high word in DX,
                          ; low word in AX.
```

◴**NEG** Instruction - NEG performs the two's complement subtraction of the operand from zero and sets the flags according to the result.

```
                    ; AX = 2CBh
                NEG  AX   ; after executing NEG result AX =FD35h.
```

◴**NOP** Instruction:

This instruction simply uses up the three clock cycles and increments the instruction pointer to point to the next instruction. NOP does not change the status of any flag. The NOP instruction is used to increase the delay of a delay loop.

◴**NOT** Instruction - Invert each bit of operand –NOT destination.

◴**OR** Instruction - Logically OR corresponding of two operands- OR destination, source.

◴**OUT** Instruction - Output a byte or word to a port – OUT port, accumulator AL or AX.

◴**POP** Instruction - POP destination

◴**NOT** Instruction - NOT perform the bitwise complement of op and stores the result back into op.

NOT op

Example:

NOT BX ; Complement contents of BX register.

; DX =F038h

NOT DX ; after the instruction DX = 0FC7h

🕗**OR** Instruction - OR instruction perform the bit wise logical OR of two operands.Each bit of the result is cleared to 0 if and only if both corresponding bits in each operand are 0, other wise the bit in the result is set to 1.

OR op1, op2

Examples:

```
OR      AH, CL       ; CL ORed with AH, result in AH.
                     ; CX = 00111110 10100101
OR      CX, FF00h    ; OR CX with immediate FF00h
                     ; result in CX = 11111111 10100101
                     ; Upper byte are all 1's lower bytes
                             ; are unchanged.
```

🕗**OUT** Instruction - The OUT instruction copies a byte from AL or a word from AX or a double from the accumulator to I/O port specified by op. Two forms of OUT instruction are available: **(1)** Port number is specified by an immediate byte constant, (0 - 255).It is also called as fixed port form. **(2)** Port number is provided in the DX register (0 – 65535)

🕗**Example**: **(1)**

```
OUT 3BH, AL        ; Copy the contents of the AL to port 3Bh
OUT 2CH, AX        ; Copy the contents of the AX to port 2Ch
                                  (2)
        MOV DX, 0FFF8H            ; Load desired port address in DX
        OUT DX, AL         ; Copy the contents of AL to
                           ; FFF8h

        OUT DX, AX         ; Copy content of AX to port
                           ; FFF8H
```

🕗**POP** Instruction:

POP instruction copies the word at the current top of the stack to the operand specified by op then increments the stack pointer to point to the next stack.

🕗**Example**:

```
        POP   DX     ; Copy a word from top of the stack to
                     ; DX and increments SP by 2.
        POP   DS     ; Copy a word from top of the stack to
                     ; DS and increments SP by 2.
        POP   TABLE [BX]
                     ; Copy a word from top of stack to memory in DS with
                     ; EA = TABLE + [BX].
```

🕗**POPF** Instruction - Pop word from top of stack to flag - register.

🕗**PUSH** Instruction - PUSH source

🕗**PUSHF** Instruction - Push flag register on the stack

🕗**RCL** Instruction - Rotate operand around to the left through CF – RCL destination, source.

🕗 **RCR** Instruction - Rotate operand around to the right through CF- RCR destination, count

🕗**POPF** Instruction - This instruction copies a word from the two memory location at the top of the stack to flag register and increments the stack pointer by 2.

🕗**PUSH** Instruction:

PUSH instruction decrements the stack pointer by 2 and copies a word from a specified source to the location in the stack segment where the stack pointer pointes.

🕘**Example:**

```
        PUSH BX ; Decrement SP by 2 and copy BX to stack
        PUSH DS ; Decrement SP by 2 and copy DS to stack
        PUSH TABLE[BX] ; Decrement SP by 2 and copy word
                                    from memory in DS at
                                    EA = TABLE + [BX] to stack.
```

🕘**PUSHF** Instruction:

This instruction decrements the SP by 2 and copies the word in flag register to the memory location pointed to by SP.

🕘**RCL** Instruction:

RCL instruction rotates the bits in the operand specified by op1 towards left by the count specified in op2.The operation is circular, the MSB of operand is rotated into a carry flag and the bit in the CF is rotated around into the LSB of operand.

RCR op1, op2

🕘**Example:**

```
        CLC            ; put 0 in CF
        RCL   AX, 1    ; save higher-order bit of AX in CF
        RCL   DX, 1    ; save higher-order bit of DX in CF
        ADC   AX, 0    ; set lower order bit if needed.
```

🕘**Example:**

```
        RCL   DX, 1          ; Word in DX of 1 bit is moved to left, and
                                     ; MSB of word is given to CF and
                             ; CF to LSB.
                             ; CF=0, BH = 10110011
        RCL   BH, 1          ; Result: BH =01100110
                             ; CF = 1, OF = 1 because MSB changed
                              ; CF =1, AX =00011111 10101001
        MOV  CL, 2           ; Load CL for rotating 2 bit position
        RCL   AX, CL         ; Result: CF =0, OF undefined
                              ; AX = 01111110 10100110
```

🕘**RCR** Instruction - RCR instruction rotates the bits in the operand specified by op1 towards right by the count specified in op2. **RCR op1, op2**

🕘**Example: (1)**

```
        RCR   BX, 1  ; Word in BX is rotated by 1 bit towards
                     ; right and CF will contain MSB bit and
                     ; LSB contain CF bit.
 ( 2)
                     ; CF = 1, BL = 00111000
        RCR   BL, 1 ; Result: BL = 10011100, CF =0
                     ; OF = 1 because MSB is changed to 1.
```

🕘**REP/REPE/REPZ/**
REPNE/REPNZ - (Prefix) Repeat String instruction until specified condition exist

🕘**RET** Instruction – Return execution from procedure to calling program.

🕘**ROL** Instruction - Rotate all bits of operand left, MSB to LSB

ROL destination, count.

🕘**ROL** Instruction - ROL instruction rotates the bits in the operand specified by op1 towards left by the count specified in op2. ROL moves each bit in the operand to next higher bit position. The higher order bit is moved to lower order position. Last bit rotated is copied into carry flag.

ROL op1, op2

🕘**Example: (1)**

```
ROL   AX, 1  ; Word in AX is moved to left by 1 bit
               and MSB bit is to LSB, and CF
               CF =0, BH =10101110
ROL BH, 1 ; Result: CF, Of =1, BH = 01011101
```

🕘**Example: (2)**

```
                        ; BX = 01011100 11010011
                        ; CL = 8 bits to rotate
ROL   BH, CL            ; Rotate BX 8 bits towards left
                        ; CF =0, BX =11010011 01011100
```

🕘**ROR** Instruction - Rotate all bits of operand right, LSB to MSB – ROR destination, count

🕘**SAHF** Instruction – Copy AH register to low byte of flag register

🕘**ROR** Instruction - ROR instruction rotates the bits in the operand op1 to wards right by count specified in op2. The last bit rotated is copied into CF.

ROR op1, op2

🕘**Example:**

```
              ( 1 )
ROR   BL, 1           ; Rotate all bits in BL towards right by 1
                      ; bit position, LSB bit is moved to MSB
                      ; and CF has last rotated bit.
              ( 2 )
                      ; CF =0, BX = 00111011 01110101
ROR   BX, 1           ; Rotate all bits of BX of 1 bit position
                      ; towards right and CF =1,
BX = 10011101 10111010
```

🕘**Example** **(3)**

```
                        ; CF = 0, AL = 10110011,
MOVE        CL, 04H     ; Load CL
ROR         AL, CL      ; Rotate all bits of AL towards right
                        ; by 4 bits, CF = 0, AL = 00111011
```

🕘**SAHF** Instruction:

SAHF copies the value of bits 7, 6, 4, 2, 0 of the AH register into the SF, ZF, AF, PF, and CF respectively. This instruction was provided to make easier conversion of assembly language program written for 8080 and 8085 to 8086.

🕘**SAL/SHL** Instruction - Shift operand bits left, put zero in LSB(s)
SAL/AHL destination, count

🕘**SAR** Instruction - Shift operand bits right, new MAB = old MSB
SAR destination, count.

🕘**SBB** Instruction - Subtract with borrow SBB destination, source

🕘**SAL / SHL** Instruction - SAL instruction shifts the bits in the operand specified by op1 to its left by the count specified in op2. As a bit is shifted out of LSB position a 0 is kept in LSB position. CF will contain MSB bit.

SAL op1, op2

🕘**Example:**

```
                    ; CF = 0, BX = 11100101 11010011
    SAL   BX, 1     ; Shift BX register contents by 1 bit
                    ; position towards left
                    ; CF = 1, BX = 11001011 1010011
```

🕘**SAR** Instruction - SAR instruction shifts the bits in the operand specified by op1 towards right by count specified in op2.As bit is shifted out a copy of old MSB is taken in MSB

MSB position and LSB is shifted to CF.

SAR op1, op2

🕘**Example:** **(1)**

```
                 ; AL = 00011101 = +29 decimal, CF = 0
    SAR   AL, 1  ; Shift signed byte in AL towards right
                 ; ( divide by 2 )

          ( 2 )
                     ; BH = 11110011 = - 13 decimal, CF = 1
    SAR   BH, 1  ; Shifted signed byte in BH to right
                     ; BH = 11111001 = - 7 decimal, CF = 1
```

🕘**SBB** Instruction - SUBB instruction subtracts op2 from op1, then subtracts 1 from op1 is CF flag is set and result is stored in op1 and it is used to set the flag.

🕘**Example:**

```
    SUB   CX, BX       ; CX – BX. Result in CX
    SUBB CH, AL        ; Subtract contents of AL and
                       ; contents CF from contents of CH.
                       ; Result in CH
    SUBB AX, 3427H ; Subtract immediate number
                 ; from AX
```

🕘**Example:**

•**Subtracting unsigned number**

```
                              ; CL = 10011100 = 156 decimal
                              ; BH = 00110111 = 55 decimal
    SUB   CL, BH              ; CL = 01100101 = 101 decimal
                              ; CF, AF, SF, ZF = 0, OF, PF = 1
```

•**Subtracting signed number**

```
                              ; CL = 00101110 = + 46 decimal
                              ; BH = 01001010= + 74 decimal
    SUB   CL, BH              ; CL = 11100100 = - 28 decimal
                              ; CF = 1, AF, ZF =0,
                              ; SF = 1 result negative
```

🕘**STD** Instruction	- Set the direction flag to 1
🕘**STI** Instruction	- Set interrupt flag (IF)
🕘**STOS/STOSB/ STOSW** Instruction	- Store byte or word in string.
🕘**SCAS/SCASB/ SCASW** Instruction	- Scan string byte or a string word.
🕘**SHR** Instruction	- Shift operand bits right, put

zero in MSB

🕗**STC** Instruction - Set the carry flag to 1

🕗**SHR** Instruction - SHR instruction shifts the bits in op1 to right by the number of times specified by op2.

🕗**Example:**

(1)

```
SHR    BP, 1  ; Shift word in BP by 1 bit position to right
              ; and 0 is kept to MSB
```

(2)

```
MOV  CL, 03H              ; Load desired number of shifts into CL
SHR    BYTE PYR[BX]       ; Shift bytes in DS at offset BX and
```

(3)

```
                          ; SI = 10010011 10101101, CF = 0
SHR    SI, 1              ; Result: SI = 01001001 11010110
                          ; CF = 1, OF = 1, SF = 0, ZF = 0
```

🕗**TEST** Instruction – AND operand to update flags

🕗**WAIT** Instruction - Wait for test signal or interrupt signal

🕗**XCHG** Instruction - Exchange XCHG destination, source

🕗**XLAT/**
XLATB Instruction - Translate a byte in AL

🕗**XOR** Instruction - Exclusive OR corresponding bits of two operands – XOR destination, source

🕗**TEST** Instruction - This instruction ANDs the contents of a source byte or word with the contents of specified destination word. Flags are updated but neither operand is changed. TEST instruction is often used to set flags before a condition jump instruction

🕗**Examples:**

```
TEST  AL, BH         ; AND BH with AL. no result is
                     ; stored. Update PF, SF, ZF
TEST  CX, 0001H      ; AND CX with immediate
                     ; number
                     ; no result is stored, Update PF,
                     ; SF
```

🕗**Example:**

```
                     ; AL = 01010001
TEST  Al, 80H        ; AND immediate 80H with AL to
                     ; test f MSB of AL is 1 or 0
                     ; ZF = 1 if MSB of AL = 0
                     ; AL = 01010001 (unchanged)
                     ; PF = 0, SF = 0
                     ; ZF = 1 because ANDing produced is 00
```

🕗**WAIT** Instruction - When this WAIT instruction executes, the 8086 enters an idle condition. This will stay in this state until a signal is asserted on TEST input pin or a valid interrupt signal is received on the INTR or NMI pin.

```
FSTSW        STATUS      ; copy 8087 status word to memory
FWAIT                    ; wait for 8087 to finish before-
                         ; doing next 8086 instruction
MOV  AX, STATUS          ; copy status word to AX to
                         ; check bits
```

🕗In this code we are adding up of FWAIT instruction so that it will stop the execution

of the command until the above instruction is finishes it's work.so that you are not loosing data and after that you will allow to continue the execution of instructions.
🕗**XCHG** Instruction - The Exchange instruction exchanges the contents of the register with the contents of another register (or) the contents of the register with the contents of the memory location. Direct memory to memory exchange are not supported.

The both operands must be the same size and one of the operand must always be a register.

Example:

```
XCHG    AX, DX           ; Exchange word in AX with word in DX
XCHG    BL, CH           ; Exchange byte in BL with byte in CH
XCHG    AL, Money [BX]   ; Exchange byte in AL with byte
                         ; in memory at EA.
```

🕗**XOR** Instruction - XOR performs a bit wise logical XOR of the operands specified by op1 and op2. The result of the operand is stored in op1 and is used to set the flag.

XOR op1, op2

Example: (Numerical)

```
                    ; BX = 00111101 01101001
                    ; CX = 00000000 11111111
XOR  BX, CX         ; Exclusive OR CX with BX
                    ; Result BX = 00111101 10010110
```

Assembler Directives🕗ASSUME

🕗**DB**	-	Defined Byte.
🕗**DD**	-	Defined Double Word
🕗**DQ**	-	Defined Quad Word
🕗**DT**	-	Define Ten Bytes
🕗**DW**	-	Define Word

🕗**ASSUME Directive**:

The ASSUME directive is used to tell the assembler that the name of the logical segment should be used for a specified segment. The 8086 works directly with only 4 physical segments: a Code segment, a data segment, a stack segment, and an extra segment.

🕗**Example:**

ASUME CS:CODE ; This tells the assembler that the logical segment named CODE contains the instruction statements for the program and should be treated as a code segment.

ASUME DS:DATA ; This tells the assembler that for any instruction which refers to a data in the data segment, data will found in the logical segment DATA.

🕗**DB:** DB directive is used to declare a byte-type variable or to store a byte in memory location.

🕗**Example:**

```
1.   PRICE    DB   49h, 98h, 29h ; Declare an array of 3 bytes,
                                 ; named as PRICE and initialize.
2.   NAME     DB   'ABCDEF'      ; Declare an array of 6 bytes and
                                 ; initialize with ASCII code for letters
3.   TEMP DB  100 DUP(?)         ; Set 100 bytes of storage in memory and
```

give it the name as TEMP, but leave the 100 bytes uninitialized. Program instructions will load values into these locations.

🕘**DW**: The DW directive is used to define a variable of type word or to reserve storage location of type word in memory.

🕘**Example:**

MULTIPLIER DW 437Ah ; this declares a variable of type word and named it as MULTIPLIER. This variable is initialized with the value 437Ah when it is loaded into memory to run.

EXP1 DW 1234h, 3456h, 5678h ; this declares an array of 3 words and initialized with specified values.

STOR1 DW 100 DUP(0); Reserve an array of 100 words of memory and initialize all words with 0000.Array is named as STOR1.

🕘**END**: END directive is placed after the last statement of a program to tell the assembler that this is the end of the program module. The assembler will ignore any statement after an END directive. Carriage return is required after the END directive.

🕘**ENDP**: ENDP directive is used along with the name of the procedure to indicate the end of a procedure to the assembler

🕘**Example:**

```
        SQUARE_NUM       PROCE ; It start the procedure
                                     ; Some steps to find the square root of a number

        SQUARE_NUMENDP ; Hear it is the End for the procedure
```

🕘**END**	-	End Program
🕘**ENDP**	-	End Procedure
🕘**ENDS**	-	End Segment
🕘**EQU**	-	Equate
🕘**EVEN**	-	Align on Even Memory Address
🕘**EXTRN**		

🕘**ENDS** - This ENDS directive is used with name of the segment to indicate the end of that logic segment.

🕘**Example:**

```
            CODE        SEGMENT   ; Hear it Start the logic
                                  ; segment containing code
                                  ; Some instructions statements to perform ;
                                  the logical operation

            CODE        ENDS      ; End of segment named as
                                          ; CODE
```

🕘**EQU**: This EQU directive is used to give a name to some value or to a symbol. Each time the assembler finds the name in the program, it will replace the name with the value or symbol you given to that name.

🕘**Example:**

FACTOR EQU 03H ; you has to write this statement at the starting of your program and later in the program you can use this as follows

ADD AL, FACTOR ; When it codes this instruction the assembler will code it as ADDAL, 03H

; The advantage of using EQU in this manner is, if FACTOR is used many no of times in a program and you want to change the value, all you had to do is change the EQU statement at beginning, it will changes the rest of all.

🕘**EVEN**: This **EVEN** directive instructs the assembler to increment the location of the counter to the next even address if it is not already in the even address. If the word is at even address 8086 can read a memory in 1 bus cycle.

If the word starts at an odd address, the 8086 will take 2 bus cycles to get the data. A series of words can be read much more quickly if they are at even address. When EVEN is used the location counter will simply incremented to next address and NOP instruction is inserted in that incremented location.

🕘**Example**:

DATA1 SEGMENT

Location counter will point to 0009 after assembler reads next statement

SALES DB 9 DUP(?) ; declare an array of 9 bytes

EVEN ; increment location counter to 000AH

RECORD DW 100 DUP(0) ; Array of 100 words will start from an even address for quicker read

DATA1 ENDS

🕘**GROUP** - Group Related Segments

🕘**LABLE**

🕘**NAME**

🕘**OFFSET**

🕘**ORG** - Originate

🕘**GROUP** - The **GROUP** directive is used to group the logical segments named after the directive into one logical group segment.

🕘**INCLUDE** - This **INCLUDE** directive is used to insert a block of source code from the named file into the current source module.

🕘**PROC** - Procedure

🕘**PTR** - Pointer

🕘**PUBLC**

🕘**SEGMENT**

🕘**SHORT**

🕘**TYPE**

🕘**PROC**: The **PROC** directive is used to identify the start of a procedure. The term near or far is used to specify the type of the procedure.

🕘**Example:**

SMART PROC FAR ; This identifies that the start of a procedure named as SMART and instructs the assembler that the procedure is far.

SMART ENDP

This PROC is used with ENDP to indicate the break of the procedure.

🕘**PTR**: This **PTR** operator is used to assign a specific type of a variable or to a label.

🕘**Example**:

INC [BX] ; this instruction will not know whether to increment the byte pointed to by BX or a word pointed to by BX.

This PTR operator can also be used to override the declared type of variable. If we want to access the a byte in an array **WORDS DW 437Ah, 0B97h,**

MOV AL, BYTE PTR WORDS

🕘**PUBLIC** - The **PUBLIC** directive is used to instruct the assembler that a specified name or label will be accessed from other modules.

🕘**Example:**

PUBLIC DIVISOR, DIVIDEND; these two variables are public so these are available to all modules.

If an instruction in a module refers to a variable in another assembly module, we can access that module by declaring as **EXTRN** directive. 🕘**TYPE** - **TYPE** operator instructs the assembler to determine the type of a variable and determines the number of bytes specified to that variable.

🕗**Example:**

Byte type variable – assembler will give a value 1
Word type variable – assembler will give a value 2
Double word type variable – assembler will give a value 4
ADD BX, TYPE WORD_ ARRAY; hear we want to increment BX to point to next word in an array of words.

DOS Function Calls🕗**AH 00H** : Terminate a Program

🕗**AH 01H**	: Read the Keyboard
🕗**AH 02H**	: Write to a Standard Output Device
🕗**AH 08H**	: Read a Standard Input without Echo
🕗**AH 09H**	: Display a Character String
🕗**AH 0AH**	: Buffered keyboard Input
🕗**INT 21H**	: Call DOS Function.

UNIT III

PIO 8255

- The parallel input-output port chip 8255 is also called as programmable ***peripheral input-output port.*** The Intel's 8255 is designed for use with Intel's 8-bit, 16-bit and higher capability microprocessors. It has 24 input/output lines which may be individually programmed in two groups of twelve lines each, or three groups of eight lines.
-
- The two groups of I/O pins are named as Group A and Group B. Each of these two groups contains a subgroup of eight I/O lines called as 8-bit port and another subgroup of four lines or a 4-bit port. Thus Group A contains an 8-bit port A along with a 4-bit port C upper.
-
- The port A lines are identified by symbols PA_0-PA_7 while the port C lines are identified as PC_4-PC_7. Similarly, Group B contains an 8-bit port B, containing lines PB0-PB7 and a 4-bit port C with lower bits PC0- PC3. The port C upper and port C lower can be used in combination as an 8-bit port C.
-
- Both the port C are assigned the same address. Thus one may have either three 8-bit I/O ports or two 8-bit and two 4-bit ports from 8255. All of these ports can function independently either as input or as output ports. This can be achieved by programming the bits of an internal register of 8255 called as control word register (CWR).

- The internal block diagram and the pin configuration of 8255 are shown in fig.
-
- The 8-bit data bus buffer is controlled by the read/write control logic. The read/write control logic manages all of the internal and external transfers of both data and control words.
-
- RD , WR , A_1, A_0 and RESET are the inputs provided by the microprocessor to the READ/ WRITE control logic of 8255. The 8-bit, 3-state bidirectional buffer is used to interface the 8255 internal data bus with the external system data bus.
-
- This buffer receives or transmits data upon the execution of input or output

instructions by the microprocessor. The control words or status information is also transferred through the buffer.

-
- The signal description of 8255 are briefly presented as follows :
-
- **PA_7-PA_0**: These are eight port A lines that acts as either latched output or buffered input lines depending upon the control word loaded into the control word register.
- **PC_7-PC_4** : Upper nibble of port C lines. They may act as either output latches or input buffers lines.
-
- This port also can be used for generation of handshake lines in mode 1 or mode 2.
-
- **PC_3-PC_0** : These are the lower port C lines, other details are the same as PC7-PC4 lines.
- **PB_0-PB_7** : These are the eight port B lines which are used as latched output lines or buffered input lines in the same way as port A.
-
- **RD** : This is the input line driven by the microprocessor and should be low to indicate read operation to 8255.
-
- **WR** : This is an input line driven by the microprocessor. A low on this line indicates write operation.
-
- **CS** : This is a chip select line. If this line goes low, it enables the 8255 to respond to RD and WR signals, otherwise RD and WR signal are neglected.
- **A_1-A_0** : These are the address input lines and are driven by the microprocessor.
- These lines A_1-A_0 with $\overline{RD}$, $\overline{WR}$ and $\overline{CS}$ from the following operations for 8255. These address lines are used for addressing any one of the four registers, i.e. three ports and a control word register as given in table below.
-
- In case of 8086 systems, if the 8255 is to be interfaced with lower order data bus, the A_0 and A_1 pins of 8255 are connected with A_1 and A_2 respectively.
-
- **D_0-D_7** : These are the data bus lines those carry data or control word to/from the microprocessor.
-
- **RESET** : A logic high on this line clears the control word register of 8255. All ports are set as input ports by default after reset

$\overline{RD}$	$\overline{WR}$	$\overline{CS}$	A_1	A_0	Input (Read) cycle
0	1	0	0	0	Port A to Data bus
0	1	0	0	1	Port B to Data bus
0	1	0	1	0	Port C to Data bus
0	1	0	1	1	CWR to Data bus

$\overline{RD}$	$\overline{WR}$	$\overline{CS}$	A_1	A_0	Output (Write) cycle
1	0	0	0	0	Data bus to Port A
1	0	0	0	1	Data bus to Port B
1	0	0	1	0	Data bus to Port C
1	0	0	1	1	Data bus to CWR

$\overline{RD}$	$\overline{WR}$	$\overline{CS}$	A_1	A_0	Function
X	X	1	X	X	Data bus tristated
1	1	0	X	X	Data bus tristated

Control Word Register

Block Diagram of 8255 (Architecture)

- It has a 40 pins of 4 groups. 1. Data bus buffer
2. Read Write control logic
3. Group A and Group B controls
4. Port A, B and C

- ***Data bus buffer***: This is a tristate bidirectional buffer used to interface the 8255 to system databus. Data is transmitted or received by the buffer on execution of input or output instruction by the CPU.
- Control word and status information are also transferred through this unit.
- ***Read/Write control logic***: This unit accepts control signals ($\overline{RD}$, $\overline{WR}$) and also inputs from address bus and issues commands to individual group of control blocks (Group A, Group B).
- It has the following pins.

a) CS – Chipselect : A low on this PIN enables the communication between CPU and 8255.

b) $\overline{RD}$ (Read) – A low on this pin enables the CPU to read the data in the ports or the status word through data bus buffer.

c) WR (Write) : A low on this pin, the CPU can write data on to the ports or on to the control register through the data bus buffer.

d) **RESET**: A high on this pin clears the control register and all ports are set to the input mode

e) **A_0** and **A_1** (Address pins): These pins in conjunction with $\overline{RD}$ and $\overline{WR}$ pins control the selection of one of the 3 ports.

- ***Group A and Group B controls*** : These block receive control from the CPU and issues commands to their respective ports.
- Group A - PA and PCU (PC_7 –PC_4)
- Group B - PCL (PC3 – PC0)
- Control word register can only be written into no read operation of the CW register is allowed.
- a) **Port A**: This has an 8 bit latched/buffered O/P and 8 bit input latch. It can be programmed in 3 modes – mode 0, mode 1, mode 2.

b) **Port B**: This has an 8 bit latched / buffered O/P and 8 bit input latch. It can be programmed in mode 0, mode1.

c) **Port C** : This has an 8 bit latched input buffer and 8 bit out put latched/buffer. This port can be divided into two 4 bit ports and can be used as control signals for port A and port B. it can be programmed in mode 0.

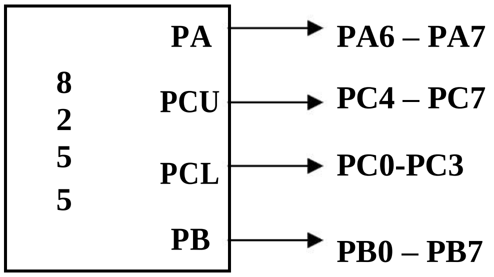
Modes of Operation of 8255

- These are two basic modes of operation of 8255. I/O mode and Bit Set-Reset mode (BSR).
- In I/O mode, the 8255 ports work as programmable I/O ports, while in BSR mode only port C (PC0-PC7) can be used to set or reset its individual port bits.
- Under the I/O mode of operation, further there are three modes of operation of 8255, so as to support different types of applications, mode 0, mode 1 and mode 2.
- ***BSR Mode***: In this mode any of the 8-bits of port C can be set or reset depending on D0 of the control word. The bit to be set or reset is selected by bit select flags D3, D2 and D1 of the CWR as given in table.
- **I/O Modes** :

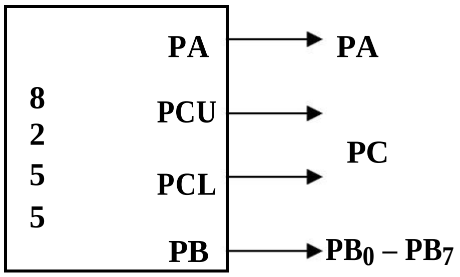
a) ***Mode 0 (Basic I/O mode):*** This mode is also called as basic input/output mode. This mode provides simple input and output capabilities using each of the three ports. Data can be simply read from and written to the input and output ports respectively, after appropriate initialisation.

D_3	D_2	D_1	Selected bits of port C
0	0	0	D_0
0	0	1	D_1
0	1	0	D_2
0	1	1	D_3
1	0	0	D_4
1	0	1	D_5
1	1	0	D_6
1	1	1	D_7

BSR Mode : CWR Format



All Output



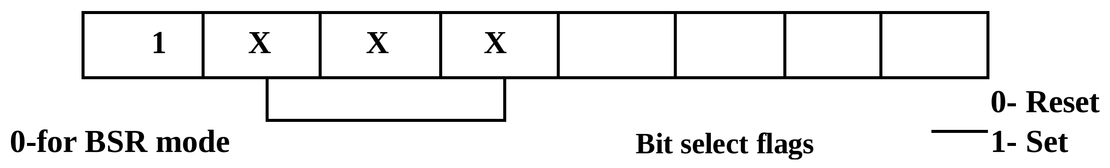
Port A and Port C acting as O/P. Port B acting as I/P

Mode 0

- The salient features of this mode are as listed below:

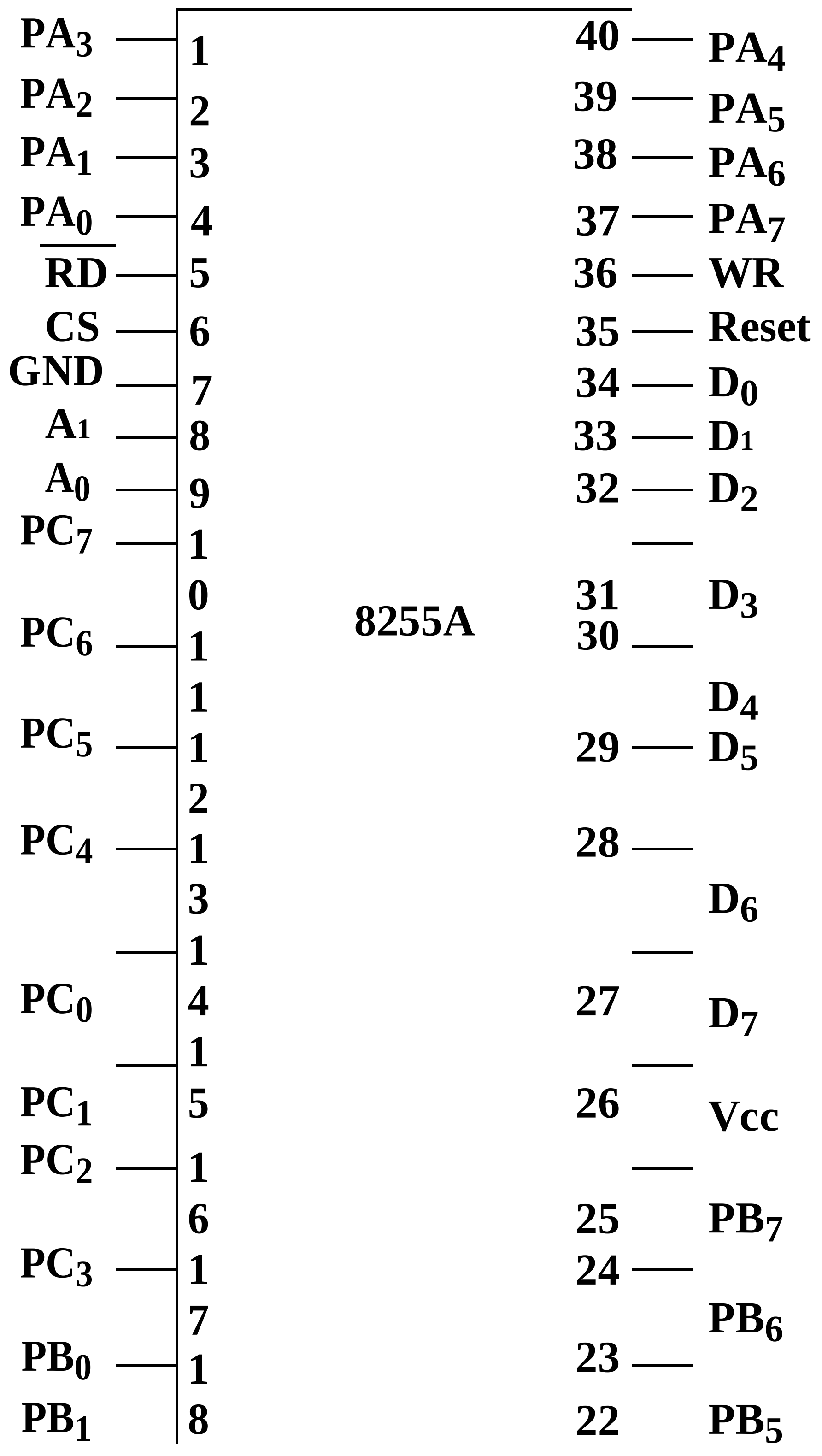
1. Two 8-bit ports (port A and port B)and two 4-bit ports (port C upper and lower) are available. The two 4-bit ports can be combinedly used as a third 8-bit port.
2. Any port can be used as an input or output port.
3. Output ports are latched. Input ports are not latched.
4. A maximum of four ports are available so that overall 16 I/O configuration are possible.

- All these modes can be selected by programming a register internal to 8255 known as CWR.
- The control word register has two formats. The first format is valid for I/O modes of operation, i.e. modes 0, mode 1 and mode 2 while the second format is valid for bit set/reset (BSR) mode of operation. These formats are shown in following fig.



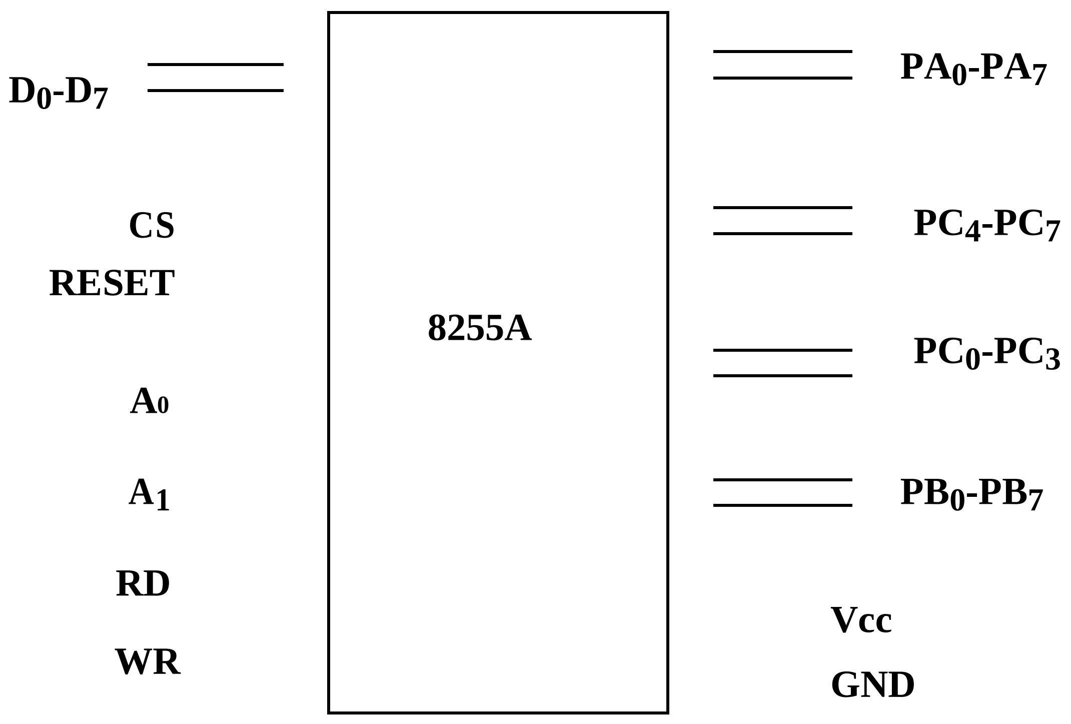
D_3, D_2, D_1 are from 000 to 111 for bits PC_0 TO PC_7

I/O Mode Control Word Register Format and
BSR Mode Control Word Register Format



1
9
2
0
PB$_2$
PB$_4$
21
PB$_3$

8255A Pin Configuration



Control Word Format of 8255

b) Mode 1: ***(Strobed input/output mode)*** In this mode the handshaking control the input and output action of the specified port. Port C lines PC0-PC2, provide strobe or handshake lines for port B. This group which includes port B and PC0-PC2 is called as group B for Strobed data input/output. Port C lines PC3-PC5 provide strobe lines for port A.

This group including port A and PC3-PC5 from group A. Thus port C is utilized for generating handshake signals. The salient features of mode 1 are listed as follows:

- Two groups – group A and group B are available for strobed data transfer.
- Each group contains one 8-bit data I/O port and one 4-bit control/data port.
- The 8-bit data port can be either used as input and output port. The inputs and outputs both are latched.
- Out of 8-bit port C, PC0-PC2 are used to generate control signals for port B and PC3-PC5 are used to generate control signals for port A. the lines PC6, PC7 may be used as independent data lines.
- **The control signals for both the groups in input and output modes are explained as follows**:

Input control signal definitions (mode 1):

- $\overline{STB}$ (Strobe input) – If this lines falls to logic low level, the data available at 8-bit input port is loaded into input latches.
- **IBF** (Input buffer full) – If this signal rises to logic 1, it indicates that data has been loaded into latches, i.e. it works as an acknowledgement. IBF is set by a low on $\overline{STB}$ and is reset by the rising edge of $\overline{RD}$ input.
- **INTR** (Interrupt request) – This active high output signal can be used to interrupt the CPU whenever an input device requests the service. INTR is set by a high

STB pin and a high at IBF pin. INTE is an internal flag that can be controlled by the bit set/reset mode of either PC4(INTEA) or PC2(INTEB) as shown in fig.

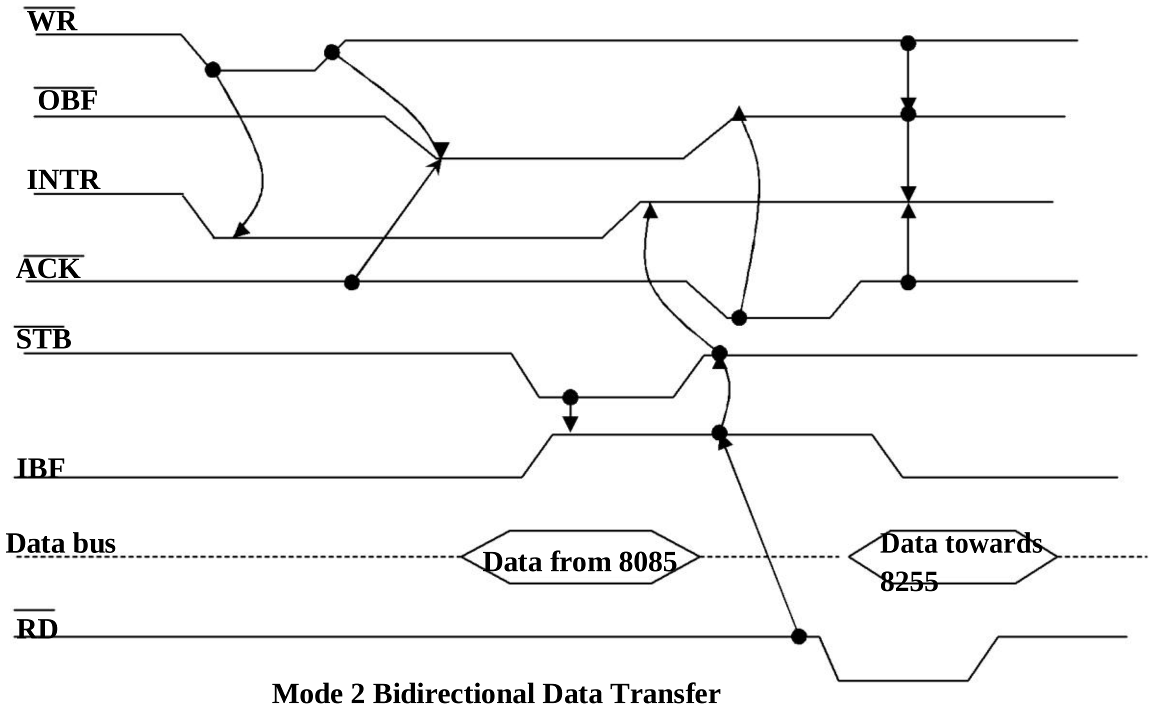
- INTR is reset by a falling edge of RD input. Thus an external input device can be request the service of the processor by putting the data on the bus and sending the strobe signal.

Output control signal definitions (mode 1) :

- **OBF** (Output buffer full) – This status signal, whenever falls to low, indicates that CPU has written data to the specified output port. The OBF flip-flop will be set by a rising edge of $\overline{WR}$ signal and reset by a low going edge at the $\overline{ACK}$ input.
- $\overline{ACK}$ (Acknowledge input) – $\overline{ACK}$ signal acts as an acknowledgement to be given by an output device. $\overline{ACK}$ signal, whenever low, informs the CPU that the data transferred by the CPU to the output device through the port is received by the output device.
- **INTR** (Interrupt request) – Thus an output signal that can be used to interrupt the CPU when an output device acknowledges the data received from the CPU. INTR is set when ACK, OBF and INTE are 1. It is reset by a falling edge on WR input. The INTEA and INTEB flags are controlled by the bit set-reset mode of PC6 and PC2 respectively.

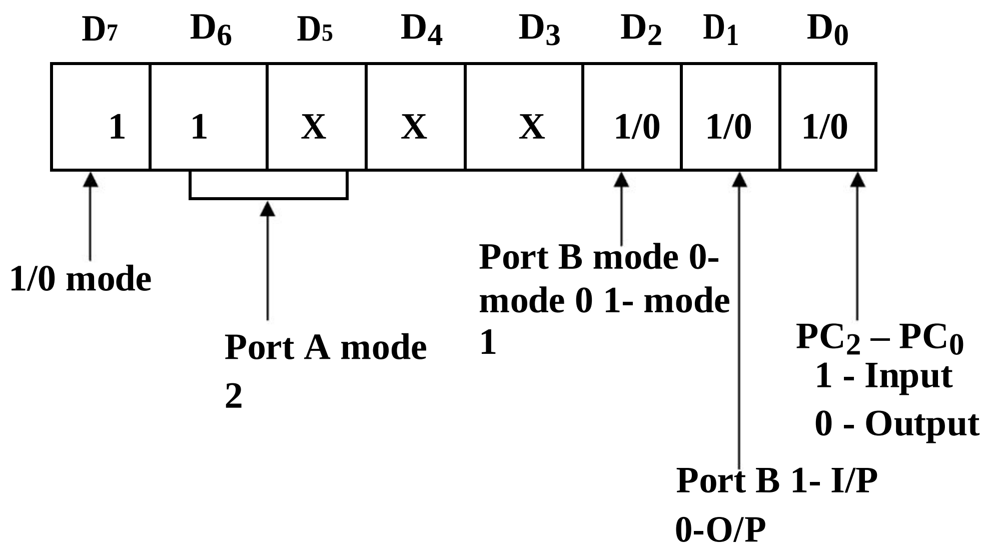
- between the data transmitter and receiver. The interrupt generation and other functions are similar to mode 1.
- In this mode, 8255 is a bidirectional 8-bit port with handshake signals. The Rd and WR signals decide whether the 8255 is going to operate as an input port or output port.
- The Salient features of Mode 2 of 8255 are listed as follows:
- The single 8-bit port in group A is available.
- The 8-bit port is bidirectional and additionally a 5-bit control port is available.
- Three I/O lines are available at port C.(PC2 – PC0)
- Inputs and outputs are both latched.
- The 5-bit control port C (PC3-PC7) is used for generating / accepting handshake signals for the 8-bit data transfer on port A.
- ***Control signal definitions in mode 2***:
- **INTR** – (Interrupt request) As in mode 1, this control signal is active high and is used to interrupt the microprocessor to ask for transfer of the next data byte to/from it. This signal is used for input (read) as well as output (write) operations.
- ***Control Signals for Output operations***:
- $\overline{OBF}$ (Output buffer full) – This signal, when falls to low level, indicates that the CPU has written data to port A.

- $\overline{ACK}$ (Acknowledge) This control input, when falls to logic low level, acknowledges that the previous data byte is received by the destination and next byte may be sent by the processor. This signal enables the internal tristate buffers to send the next data byte on port A.
- **INTE1** (A flag associated with OBF) This can be controlled by bit set/reset mode with PC6.
- ***Control signals for input operations :***
- $\overline{STB}$ (Strobe input) A low on this line is used to strobe in the data into the input latches of 8255.
- **IBF** (Input buffer full) When the data is loaded into input buffer, this signal rises to logic '1'. This can be used as an acknowledge that the data has been received by the receiver.
- The waveforms in fig show the operation in Mode 2 for output as well as input port.
- Note: $\overline{WR}$ must occur before $\overline{ACK}$ and $\overline{STB}$ must be activated before $\overline{RD}$.

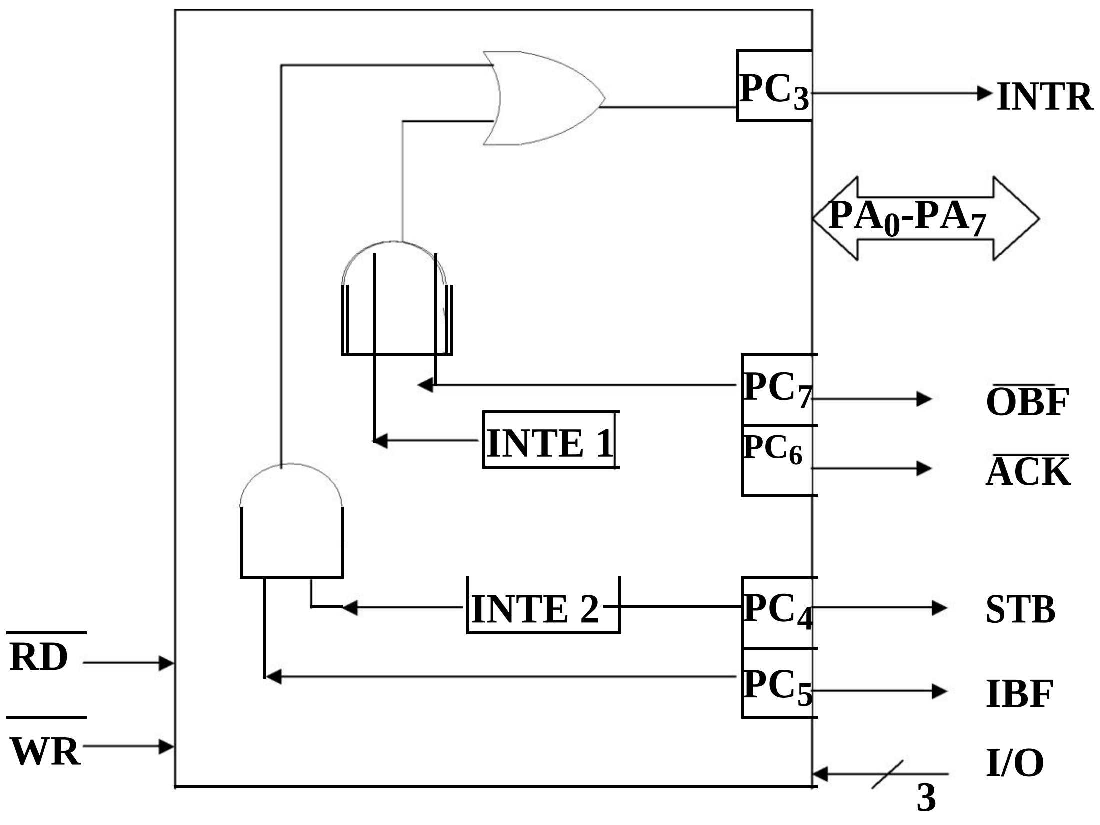


Mode 2 Bidirectional Data Transfer

0 The following fig shows a schematic diagram containing an 8-bit bidirectional port, 5-bit control port and the relation of INTR with the control pins. Port B can either be set to Mode 0 or 1 with port A(Group A) is in Mode 2.
1 Mode 2 is not available for port B. The following fig shows the control word.
2 The INTR goes high only if either IBF, INTE2, STB and RD go high or OBF, INTE1, ACK and WR go high. The port C can be read to know the status of the peripheral device, in terms of the control signals, using the normal I/O instructions.



Mode 2 control word



Mode 2 pins

8254

- Compatible with All Intel and Most other Microprocessors
- Handles Inputs from DC to 10 MHz
- 8 MHz 8254
- 10 MHz 8254-2
- Status Read-Back Command
- Six Programmable Counter Modes
- Three Independent 16-Bit Counters
- Binary or BCD Counting
- Single a 5V Supply
- Standard Temperature Range
- The Intel 8254 is a counter/timer device designed to solve the common timing control problems in microcomputer system design.
- It provides three independent 16-bit counters, each capable of handling clock inputs up to 10 MHz.
- All modes are software programmable. The 8254 is a superset of the 8253.The 8254 uses HMOS technology and comes in a 24-pin plastic or CERDIP package.

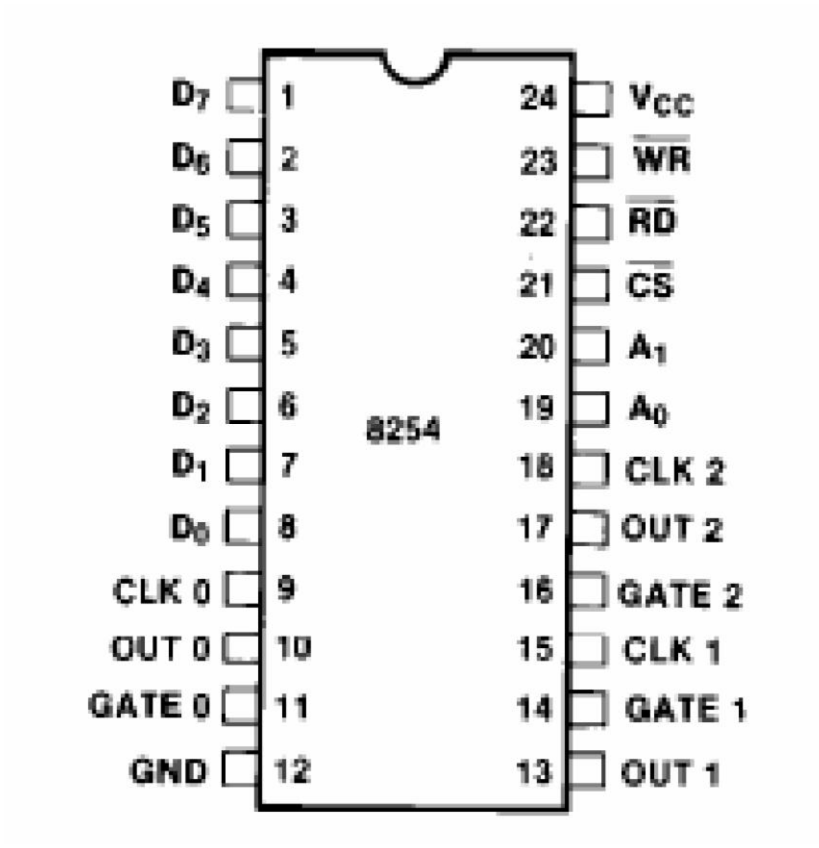


Figure 1. Pin Configuration

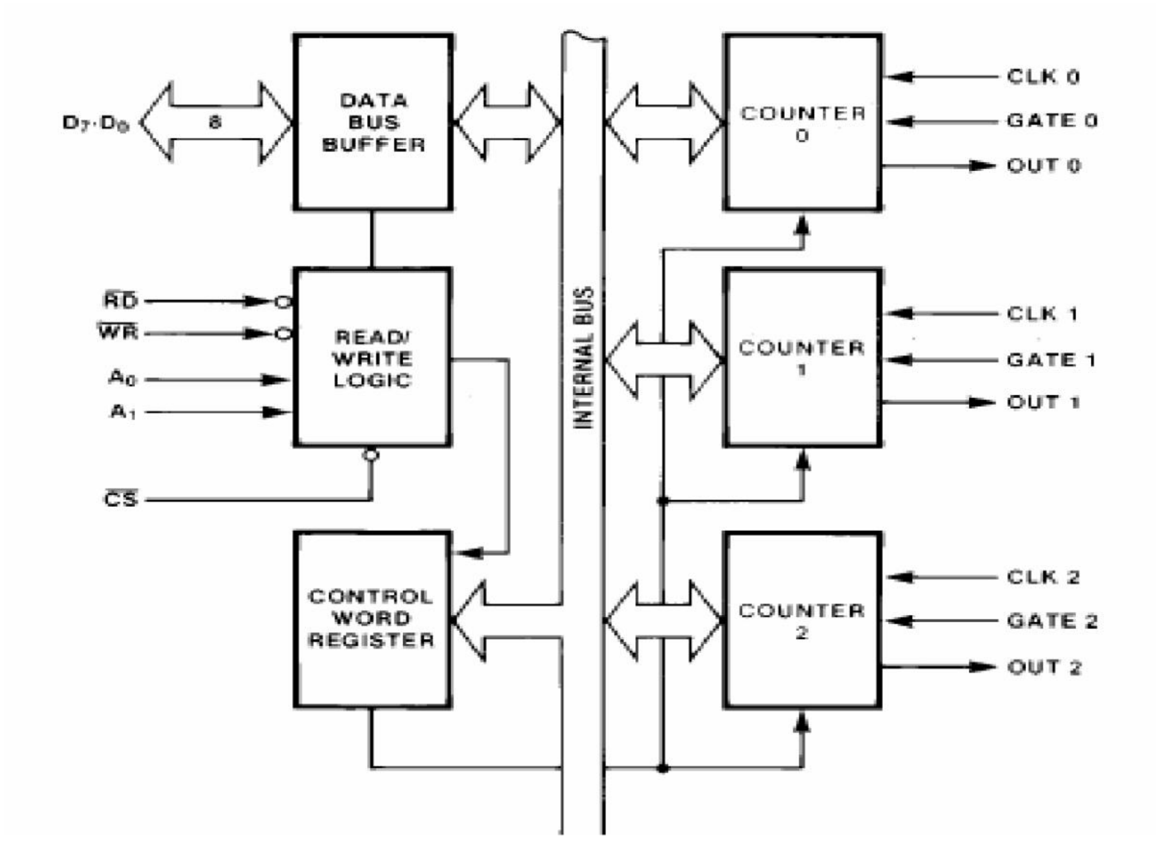


Figure 2. 8254 Block

Pin Description

Symbol	Pin No.	Type	Name and Function
D7-D0	**1 - 8**	**I/O**	**DATA: Bi-directional three state data bus lines, connected to system data bus.**
CLK 0	**9**	**I**	**CLOCK 0: Clock input of Counter 0.**
OUT 0	**10**	**O**	**OUTPUT 0: Output of Counter 0.**
GATE 0	**11**	**I**	**GATE 0: Gate input of Counter 0.**
GND	**12**		**GROUND: Power supply connection.**
VCC	**24**		**POWER: A 5V power supply connection.**
$\overline{\text{WR}}$	**23**	**I**	**WRITE CONTROL: This input is low during CPU write operations.**
RD	**22**	**I**	**READ CONTROL: This input is low during CPU read operations.**

CS	21	I	CHIP SELECT: A low on this input enables the 8254 to respond to RD and WR signals. RD and WR are ignored otherwise.
A1, A0	20 – 9	I	ADDRESS: Used to select one of the three Counters or the Control Word Register for read or write operations. Normally connected to the system address bus.

A1	A0	Selects
0	0	Counter 0
0	1	Counter 1
1	0	Counter 2
1	1	Control Word Register

CLK 2	18	I	CLOCK 2: Clock input of Counter 2.
OUT 2	17	O	OUT 2: Output of Counter 2.
GATE 2	16	I	GATE 2: Gate input of Counter 2.
CLK 1	15	I	CLOCK 1: Clock input of Counter 1.
GATE 1	14	I	GATE 1: Gate input of Counter 1.
OUT 1	OUT 1	O	OUT 1: Output of Counter 1.

Functional Description

- The 8254 is a programmable interval timer/counter designed for use with Intel microcomputer systems.
- It is a general purpose, multi-timing element that can be treated as an array of I/O ports in the system software.
- The 8254 solves one of the most common problems in any microcomputer system, the generation of accurate time delays under software control. Instead of setting up timing loops in software, the programmer configures the 8254 to match his requirements and programs one of the counters for the desired delay.
- After the desired delay, the 8254 will interrupt the CPU. Software overhead is minimal and variable length delays can easily be accommodated.
- Some of the other counter/timer functions common to microcomputers which can be implemented with the 8254 are:
- Real time clock
- Event-counter
- Digital one-shot
- Programmable rate generator
- Square wave generator
- Binary rate multiplier
- Complex waveform generator
- Complex motor controller

Block Diagram

- **DATA BUS BUFFER**: This 3-state, bi-directional, 8-bit buffer is used to interface the 8254 to the system bus, see the figure : Block Diagram Showing Data Bus Buffer and Read/Write Logic Functions.
- **READ/WRITE LOGIC** : The Read/Write Logic accepts inputs from the system bus and generates control signals for the other functional blocks of the 8254. A1 and A0 select one of the three counters or the Control Word Register to be read from/written into.
- A ``low" on the RD input tells the 8254 that the CPU is reading one of the counters.

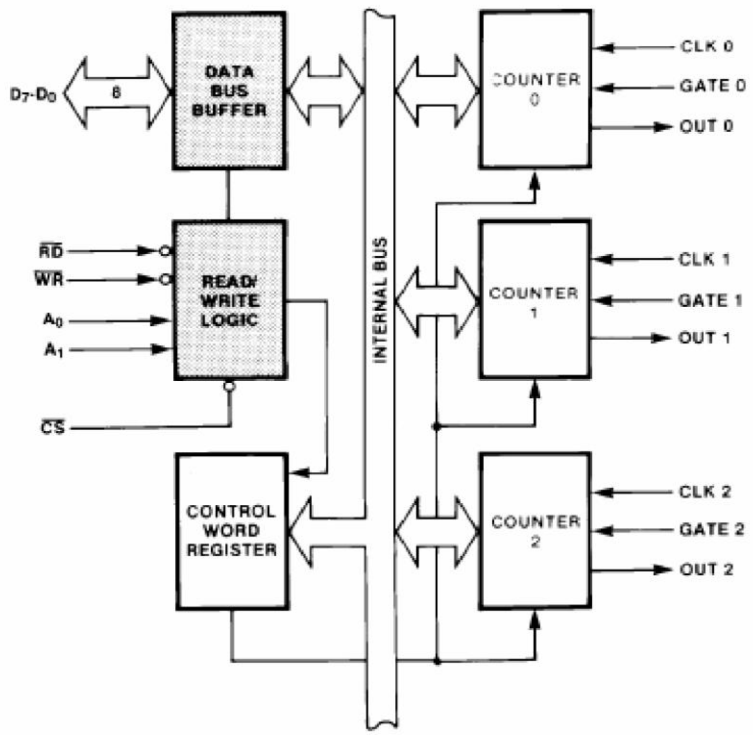


Figure 3. Block Diagram Showing Data Bus Buffer and Read/Write Logic Functions

- A ``low" on the WR input tells the 8254 that the CPU is writing either a Control Word or an initial count. Both RD and WR are qualified by CS; RD and WR are ignored unless the 8254 has been selected by holding CS low.
- **CONTROL WORD REGISTER :**The Control Word Register (see Figure 4) is selected by the Read/Write Logic when A_1,A_0 = 11. If the CPU then does a write operation to the 8254, the data is stored in the Control Word Register and is interpreted as a Control Word used to define the operation of the Counters.

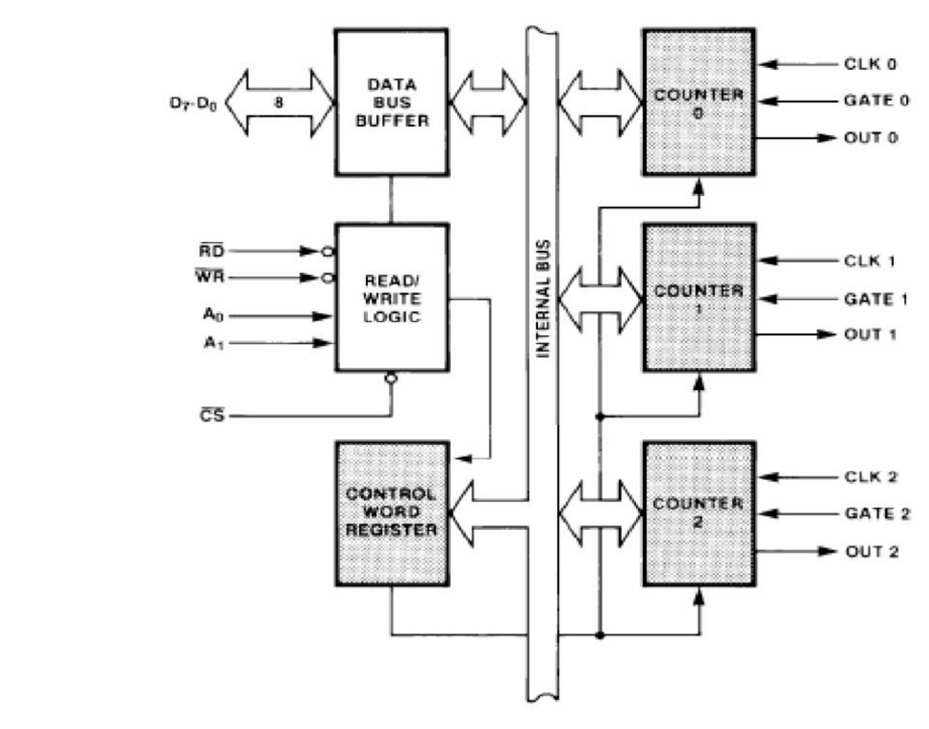


Figure 4. Block Diagram Showing Control Word Register and Counter Functions

- The Control Word Register can only be written to; status information is available with the Read-Back Command.
- **COUNTER 0, COUNTER 1, COUNTER 2** :These three functional blocks are identical in operation, so only a single Counter will be described. The internal block diagram of a single counter is shown in Figure 5.
- The Counters are fully independent. Each Counter may operate in a different Mode.
- The Control Word Register is shown in the figure, it is not part of the Counter itself, but its contents determine how the Counter operates.
- The status register, shown in Figure 5, when latched, contains the current contents of the Control Word Register and status of the output and null count flag. (See detailed explanation of the Read-Back command.)
- The actual counter is labelled CE (for ``Counting Element"). It is a 16-bit presettable synchronous down counter. OLM and OLL are two 8-bit latches. OL stands for ``Output Latch"; the subscripts M and L stand for ``Most significant byte" and ``Least significant byte’‘respectively.

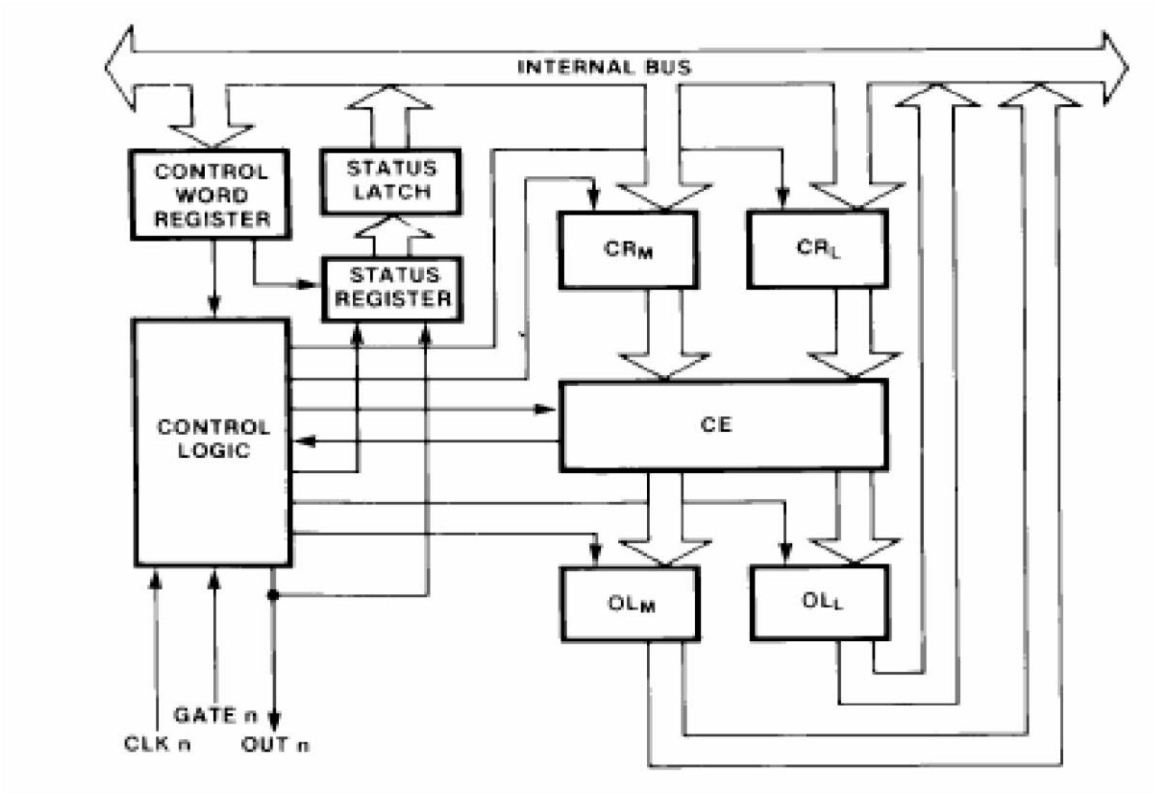


Figure 5. Internal Block Diagram of a Counter

- Both are normally referred to as one unit and called just OL. These latches normally ``follow'' the CE, but if a suitable Counter Latch Command is sent to the 8254, the latches ``latch" the present count until read by the CPU and then return to ``following" the CE.
- One latch at a time is enabled by the counter's Control Logic to drive the internal bus. This is how the 16-bit Counter communicates over the 8-bit internal bus. Note that the CE itself cannot be read; whenever you read the count, it is the OL that is being read.
- Similarly, there are two 8-bit registers called CRM and CRL (for ``Count Register"). Both are normally referred to as one unit and called just CR.
- When a new count is written to the Counter, the count is stored in the CR and later transferred to the CE. The Control Logic allows one register at a time to be loaded from the internal bus. Both bytes are transferred to the CE simultaneously.
- CRM and CRL are cleared when the Counter is programmed. In this way, if the Counter has been programmed for one byte counts (either most significant byte only or least significant byte only) the other byte will be zero.
- Note that the CE cannot be written into, whenever a count is written, it is written into the CR.
- The Control Logic is also shown in the diagram.
- CLK n, GATE n, and OUT n are all connected to the outside world through the Control Logic.

- **8254 SYSTEM INTERFACE: The** 8254 is a component of the Intel Microcomputer Systems and interfaces in the same manner as all other peripherals of the family.
- It is treated by the system's software as an array of peripheral I/O ports; three are counters and the fourth is a control register for MODE programming.
- Basically, the select inputs A0,A1 connect to the A0,A1 address bus signals of the CPU. The CS can be derived directly from the address bus using a linear select method. Or it can be connected to the output of a decoder, such as an Intel 8205 for larger systems.
- **Programming the 8254: Counters** are programmed by writing a Control Word and then an initial count.
- The Control Words are written into the Control Word Register, which is selected when A1,A0 = 11. The Control Word itself specifies which Counter is being programmed.

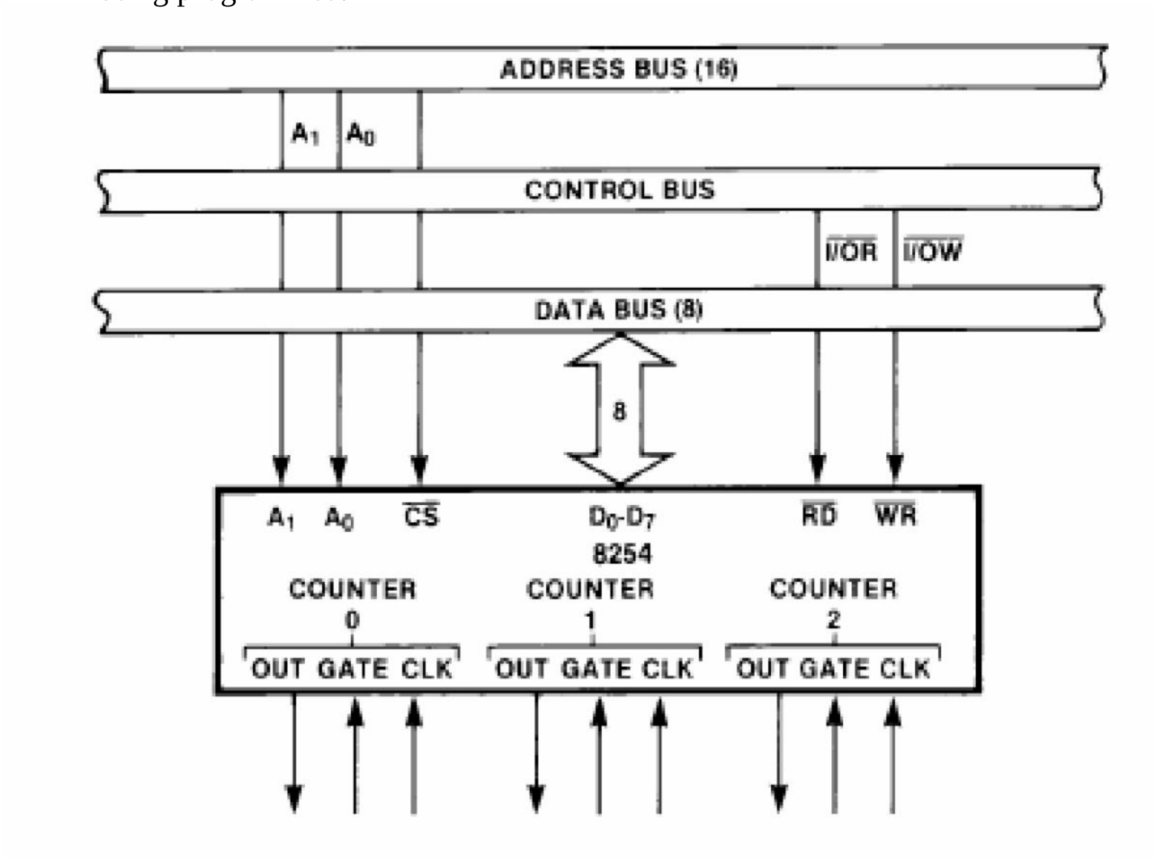


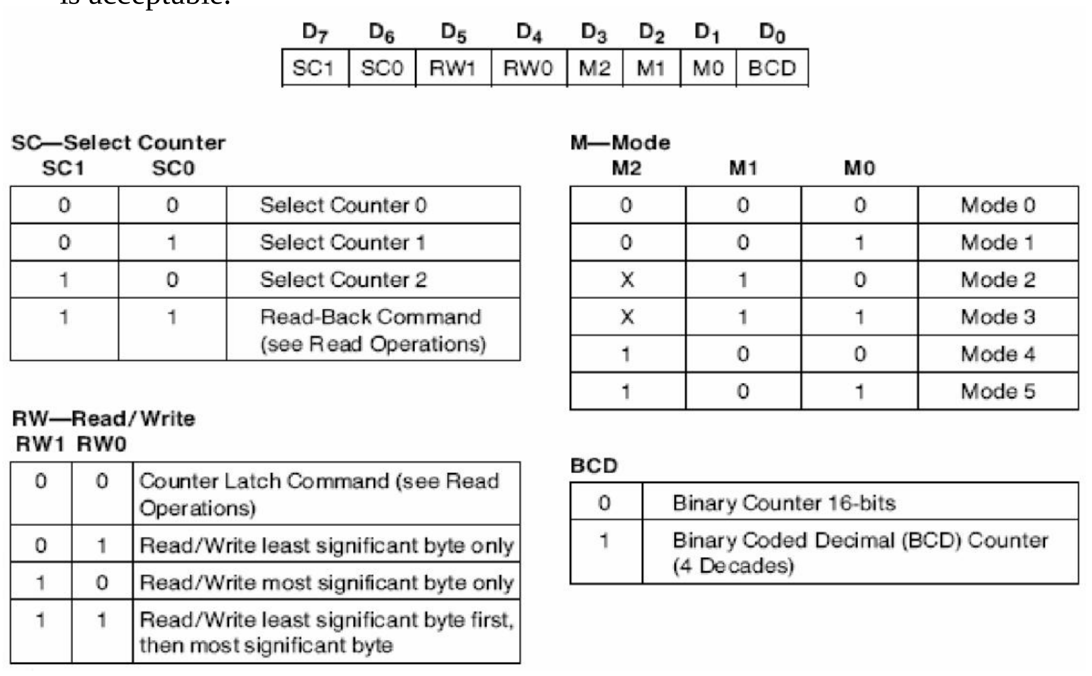
Figure 6. 8254 System Interface

- **Control Word Format:** A_1,A_0 = 11, CS = 0, RD = 1, WR = 0.
- By contrast, initial counts are written into the Counters, not the Control Word Register. The A_1,A_0 inputs are used to select the Counter to be written into. The format of the initial count is determined by the Control Word used.
- **Write Operations**: The programming procedure for the 8254 is very flexible. Only two conventions need to be remembered:

2) For each Counter, the Control Word must be written before the initial count is written.

2) The initial count must follow the count format specified in the Control Word (least significant byte only, most significant byte only, or least significant byte and then most significant byte).

- Since the Control Word Register and the three Counters have separate addresses (selected by the A_1,A_0 inputs), and each Control Word specifies the Counter it applies to (SC_0,SC_1 bits), no special instruction sequence is required.
- Any programming sequence that follows the conventions in Figure 7 is acceptable.



D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
SC1	SC0	RW1	RW0	M2	M1	M0	BCD

SC—Select Counter

SC1	SC0	
0	0	Select Counter 0
0	1	Select Counter 1
1	0	Select Counter 2
1	1	Read-Back Command (see Read Operations)

M—Mode

M2	M1	M0	
0	0	0	Mode 0
0	0	1	Mode 1
X	1	0	Mode 2
X	1	1	Mode 3
1	0	0	Mode 4
1	0	1	Mode 5

RW—Read/Write

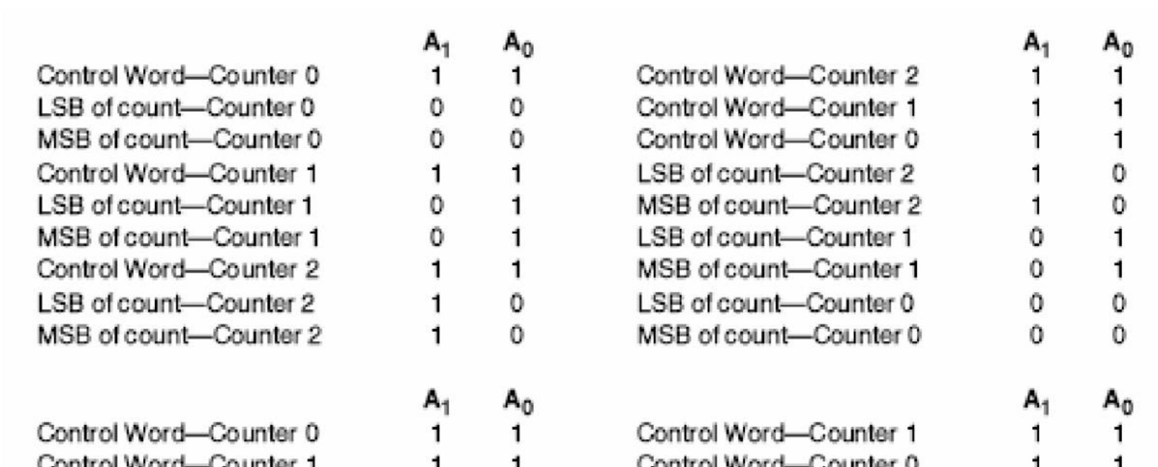
RW1	RW0	
0	0	Counter Latch Command (see Read Operations)
0	1	Read/Write least significant byte only
1	0	Read/Write most significant byte only
1	1	Read/Write least significant byte first, then most significant byte

BCD

0	Binary Counter 16-bits
1	Binary Coded Decimal (BCD) Counter (4 Decades)

NOTE: Don't care bits (X) should be 0 to insure compatibility with future Intel products. Figure 7. Control Word Format

- A new initial count may be written to a Counter at any time without affecting the Counter's programmed Mode in any way. Counting will be affected as described in the Mode definitions. The new count must follow the programmed count format.
- If a Counter is programmed to read/write two-byte counts, the following precaution applies: A program must not transfer control between writing the first and second byte to another routine which also writes into that same Counter. Otherwise, the Counter will be loaded with an incorrect count.

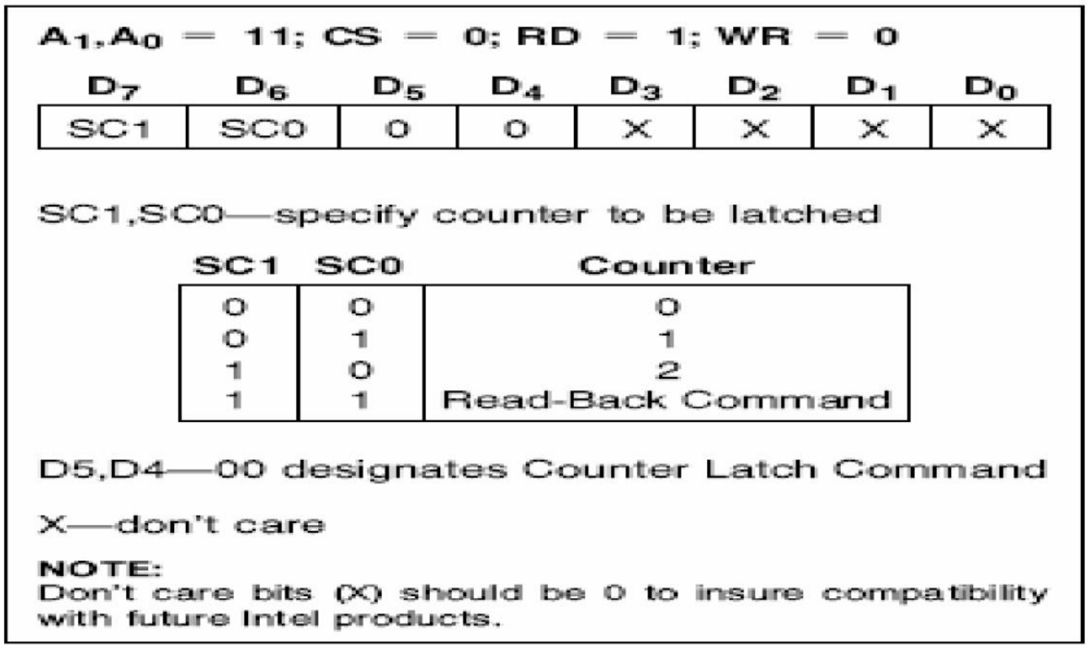


	A_1	A_0		A_1	A_0
Control Word—Counter 0	1	1	Control Word—Counter 2	1	1
LSB of count—Counter 0	0	0	Control Word—Counter 1	1	1
MSB of count—Counter 0	0	0	Control Word—Counter 0	1	1
Control Word—Counter 1	1	1	LSB of count—Counter 2	1	0
LSB of count—Counter 1	0	1	MSB of count—Counter 2	1	0
MSB of count—Counter 1	0	1	LSB of count—Counter 1	0	1
Control Word—Counter 2	1	1	MSB of count—Counter 1	0	1
LSB of count—Counter 2	1	0	LSB of count—Counter 0	0	0
MSB of count—Counter 2	1	0	MSB of count—Counter 0	0	0

	A_1	A_0		A_1	A_0
Control Word—Counter 0	1	1	Control Word—Counter 1	1	1
Control Word—Counter 1	1	1	Control Word—Counter 0	1	1

Figure 8. A Few Possible Programming Sequences

- **Read Operations**: It is often desirable to read the value of a Counter without disturbing the count in progress. This is easily done in the 8254.
- There are three possible methods for reading the counters: a simple read operation, the Counter Latch Command, and the Read-Back Command.
- Each is explained below. The first method is to perform a simple read operation. To read the Counter, which is selected with the A_1, A_0 inputs, the CLK input of the selected Counter must be inhibited by using either the GATE input or external logic.
- Otherwise, the count may be in the process of changing when it is read, giving an undefined result.
- **COUNTER LATCH COMMAND**: The second method uses the ``Counter Latch Command".
- Like a Control Word, this command is written to the Control Word Register, which is selected when A_1,A_0 = 11. Also like a Control Word, the SC_0, SC_1 bits select one of the three Counters, but two other bits, D_5 and D_4, distinguish this command from a Control Word.
- The selected Counter's output latch (OL) latches the count at the time the Counter Latch Command is received. This count is held in the latch until it is read by the CPU (or until the Counter is reprogrammed).



A_1,A_0 = 11; CS = 0; RD = 1; WR = 0

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
SC1	SC0	0	0	X	X	X	X

SC1,SC0—specify counter to be latched

SC1	SC0	Counter
0	0	0
0	1	1
1	0	2
1	1	Read-Back Command

D5,D4—00 designates Counter Latch Command

X—don't care

NOTE:
Don't care bits (X) should be 0 to insure compatibility with future Intel products.

Figure 9. Counter Latching Command Format

- The count is then unlatched automatically and the OL returns to ``following" the counting element (CE).
- This allows reading the contents of the Counters ``on the fly" without affecting counting in progress.
- Multiple Counter Latch Commands may be used to latch more than one Counter.

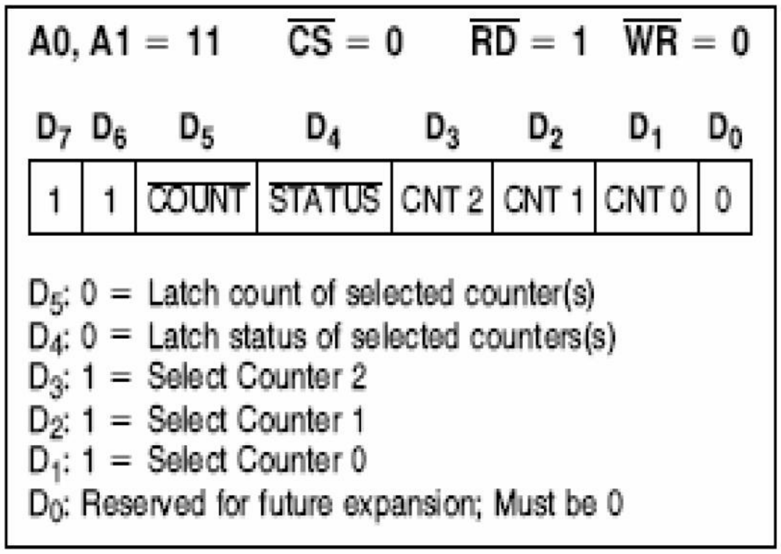
Each latched Counter's OL holds its count until it is read.

- Counter Latch Commands do not affect the programmed Mode of the Counter in any way.
- If a Counter is latched and then, some time later, latched again before the count is read, the second Counter Latch Command is ignored. The count read will be the count at the time the first Counter Latch Command was issued.
- With either method, the count must be read according to the programmed format; specifically, if the Counter is programmed for two byte counts, two bytes must be read. The two bytes do not have to be read one right after the other, read or write or programming operations of other Counters may be inserted between them.
- Another feature of the 8254 is that reads and writes of the same Counter may be interleaved.

- **Example**: If the Counter is programmed for two byte counts, the following sequence is valid.

1) Read least significant byte.
2) Write new least significant byte.
3) Read most significant byte.
4) Write new most significant byte.

- If a Counter is programmed to read/write two-byte counts, the following precaution applies: A program must not transfer control between reading the first and second byte to another routine which also reads from that same Counter. Otherwise, an incorrect count will be read.
- **READ-BACK COMMAND:** The third method uses the Read-Back Command. This command allows the user to check the count value, programmed Mode, and current states of the OUT pin and Null Count flag of the selected counter (s).
- The command is written into the Control Word Register and has the format shown in Figure 10. The command applies to the counters selected by setting their corresponding bits D_3, D_2, $D_1 = 1$.
- The read-back command may be used to latch multiple counter output latches (OL) by setting the COUNT bit $D_5 = 0$ and selecting the desired counter (s). This single command is functionally equivalent to several counter latch commands, one for each counter latched.



A0, A1 = 11 $\overline{CS} = 0$ $\overline{RD} = 1$ $\overline{WR} = 0$

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
1	1	$\overline{\text{COUNT}}$	$\overline{\text{STATUS}}$	CNT 2	CNT 1	CNT 0	0

D_5: 0 = Latch count of selected counter(s)
D_4: 0 = Latch status of selected counters(s)
D_3: 1 = Select Counter 2
D_2: 1 = Select Counter 1
D_1: 1 = Select Counter 0
D_0: Reserved for future expansion; Must be 0

- Each counter's latched count is held until it is read (or the counter is reprogrammed).
- The counter is automatically unlatched when read, but other counters remain

latched until they are read. If multiple count read-back commands are issued to the same counter without reading the count, all but the first are ignored; i.e., the count which will be read is the count at the time the first read-back command was issued.

- The read-back command may also be used to latch status information of selected counter (s) by setting STATUS bit $D_4 = 0$. Status must be latched to be read; status of a counter is accessed by a read from that counter.
- The counter status format is shown in Figure 11.

- Bits D_5 through D_0 contain the counter's programmed Mode exactly as written in the last Mode Control Word. OUTPUT bit D_7 contains the current state of the OUT pin.
- This allows the user to monitor the counter's output via software, possibly eliminating some hardware from a system. NULL COUNT bit D_6 indicates when the last count written to the counter register (CR) has been loaded into the counting element (CE).
- The exact time this happens depends on the Mode of the counter and is described in the Mode Definitions, but until the count is loaded into the counting element (CE), it can't be read from the counter.

8279

- While studying 8255, we have explained the use of 8255 in interfacing keyboards and displays with 8086. The disadvantages of this method of interfacing keyboard and display with 8086 is that the processor has to refresh the display and check the status of the keyboard periodically using polling technique. Thus a considerable amount of CPU time is wasted, reducing the system operating speed.
- Intel's 8279 is a general purpose keyboard display controller that simultaneously drives the display of a system and interfaces a keyboard with the CPU, leaving it free for its routine task.

Architecture and Signal Descriptions of 8279

- The keyboard display controller chip 8279 provides:

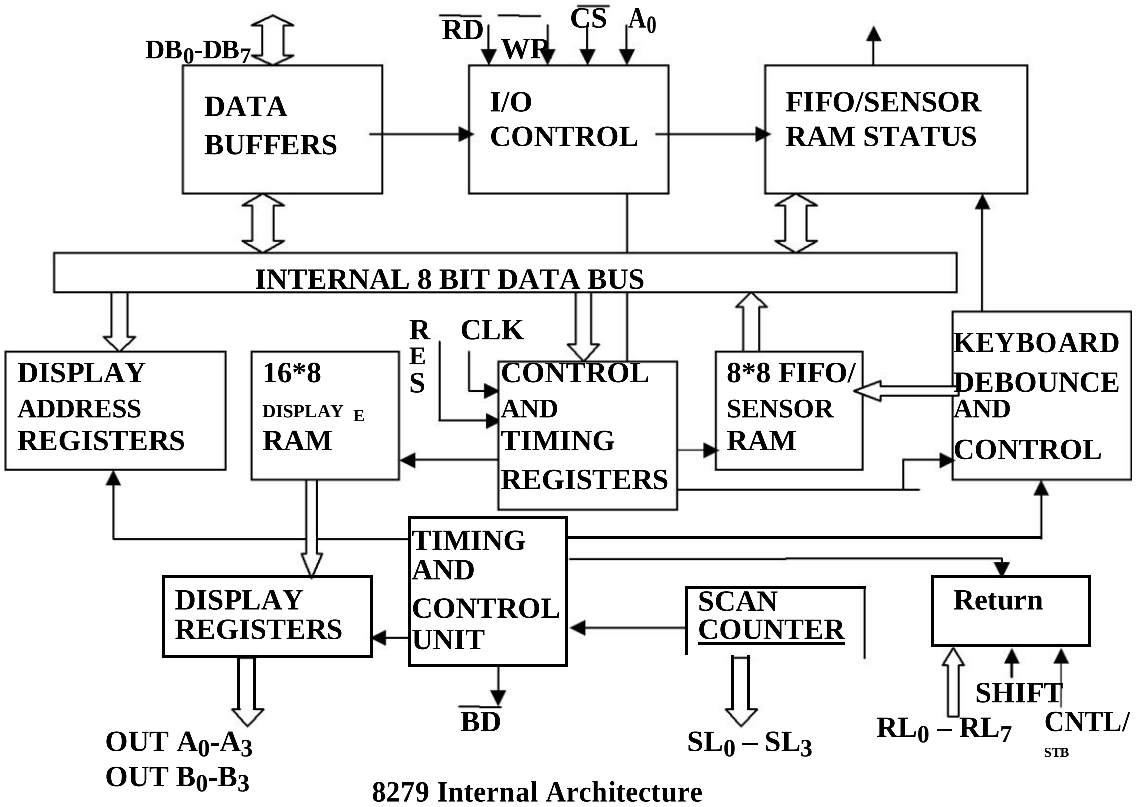
a) a set of four scan lines and eight return lines for interfacing keyboards

b) A set of eight output lines for interfacing display.

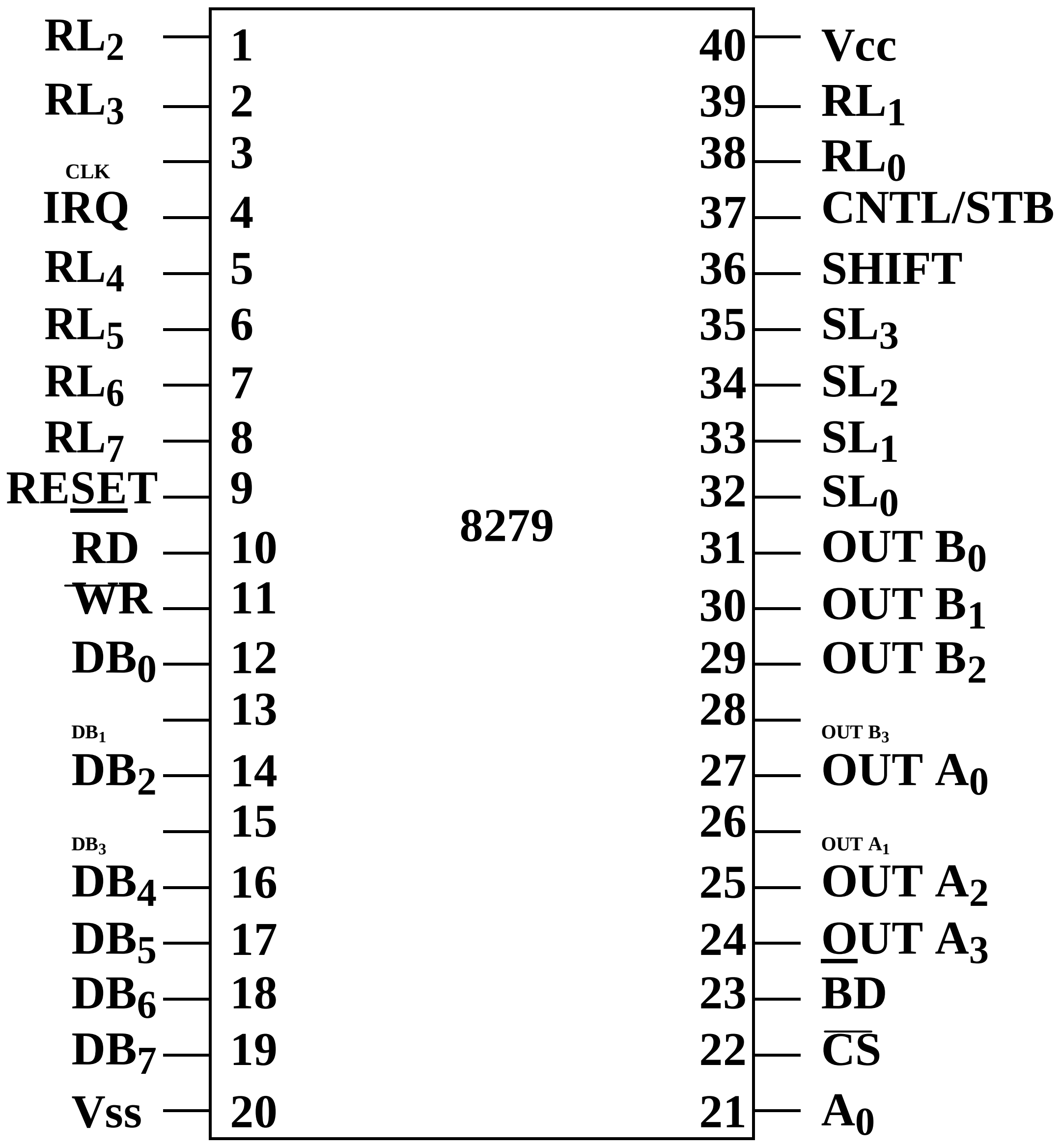
- Fig shows the functional block diagram of 8279 followed by its brief description.
- **I/O Control and Data Buffers** : The I/O control section controls the flow of data to/from the 8279. The data buffers interface the external bus of the system with internal bus of 8279.
- The I/O section is enabled only if CS is low. The pins A0, RD and WR select the command, status or data read/write operations carried out by the CPU with 8279.
- **Control and Timing Register and Timing Control** : These registers store the keyboard and display modes and other operating conditions programmed by CPU. The registers are written with A0=1 and WR=0. The Timing and control unit controls the basic timings for the operation of the circuit. Scan counter divide down the operating frequency of 8279 to derive scan keyboard and scan display frequencies.

- **Scan Counter** : The scan counter has two modes to scan the key matrix and refresh the display. In the encoded mode, the counter provides binary count that is to be externally decoded to provide the scan lines for keyboard and display (Four externally decoded scan lines may drive upto 16 displays). In the decode scan mode, the counter internally decodes the least significant 2 bits and provides a decoded 1 out of 4 scan on SL0 -SL3(Four internally decoded scan lines may drive upto 4 displays). The keyboard and display both are in the same mode at a time.

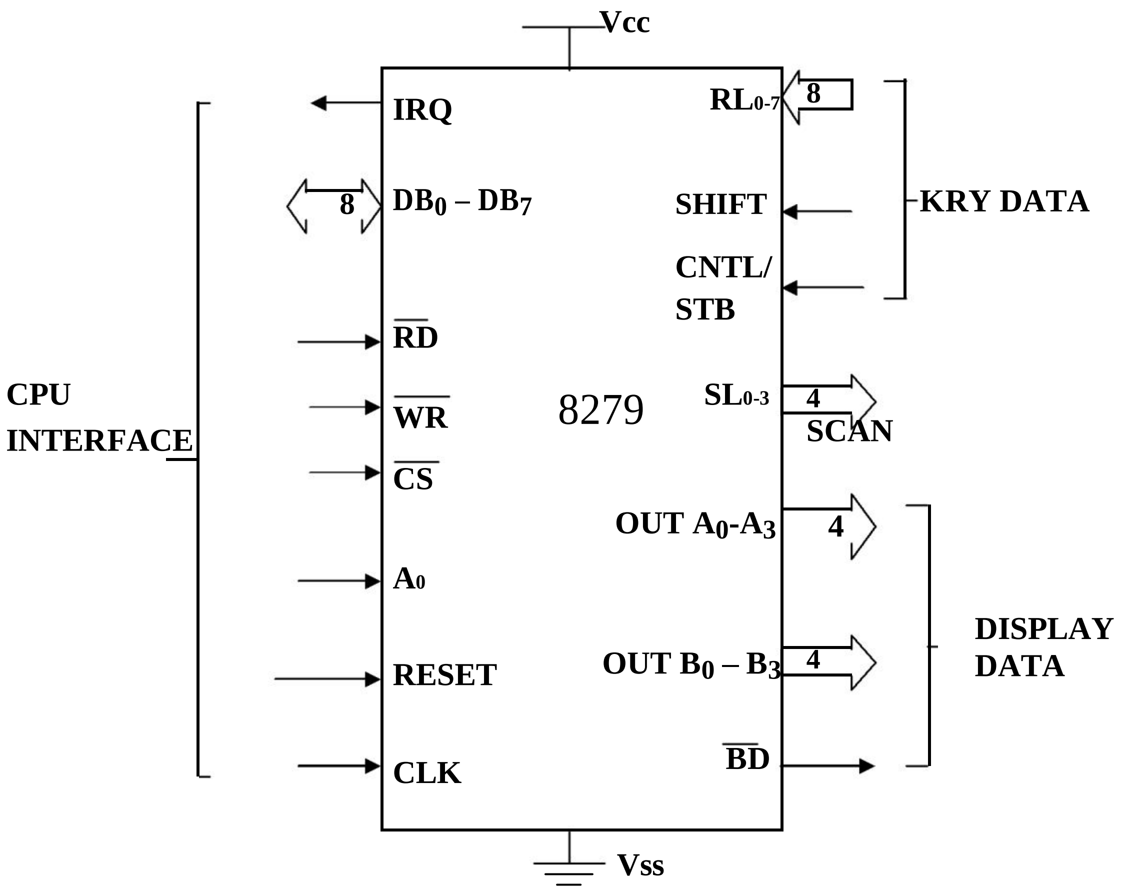
- **Return Buffers and Keyboard Debounce and Control**: This section for a key closure row wise. If a key closer is detected, the keyboard debounce unit debounces the key entry (i.e. wait for 10 ms). After the debounce period, if the key continues to be detected. The code of key is directly transferred to the sensor RAM along with SHIFT and CONTROL key status.
- **FIFO/Sensor RAM and Status Logic**: In keyboard or strobed input mode, this block acts as 8-byte first-in-first-out (FIFO) RAM. Each key code of the pressed key is entered in the order of the entry and in the mean time read by the CPU, till the RAM become empty.
- The status logic generates an interrupt after each FIFO read operation till the FIFO is empty. In scanned sensor matrix mode, this unit acts as sensor RAM. Each row of the sensor RAM is loaded with the status of the corresponding row of sensors in the matrix. If a sensor changes its state, the IRQ line goes high to interrupt the CPU.
- **Display Address Registers and Display RAM** : The display address register holds the address of the word currently being written or read by the CPU to or from the display RAM. The contents of the registers are automatically updated by 8279 to accept the next data entry by CPU.



8279 Internal Architecture



8279 Pin Configuration



- The signal discription of each of the pins of 8279 as follows :
- **DB0-DB7** : These are bidirectional data bus lines. The data and command words to and from the CPU are transferred on these lines.
- **CLK** : This is a clock input used to generate internal timing required by 8279.
- **RESET** : This pin is used to reset 8279. A high on this line reset 8279. After resetting 8279, its in sixteen 8-bit display, left entry encoded scan, 2-key lock out mode. The clock prescaler is set to 31.
- **CS** : Chip Select – A low on this line enables 8279 for normal read or write operations. Other wise, this pin should remain high.
- **A_0** : A high on this line indicates the transfer of a command or status information. A low on this line indicates the transfer of data. This is used to select one of the internal registers of 8279.
- **RD, WR (Input/Output) READ/WRITE** – These input pins enable the data buffers to receive or send data over the data bus.
- **IRQ** : This interrupt output lines goes high when there is a data in the FIFO sensor RAM. The interrupt lines goes low with each FIFO RAM read operation but if the FIFO RAM further contains any key-code entry to be read by the CPU, this pin again goes high to generate an interrupt to the CPU.
- **Vss, Vcc** : These are the ground and power supply lines for the circuit.
- **SL0-SL3-Scan Lines** : These lines are used to scan the key board matrix and display digits. These lines can be programmed as encoded or decoded, using the mode control register.

- **RL_0 - RL_7 - Return Lines** : These are the input lines which are connected to one terminal of keys, while the other terminal of the keys are connected to the decoded scan lines. These are normally high, but pulled low when a key is pressed.

- **SHIFT** : The status of the shift input lines is stored along with each key code in FIFO, in scanned keyboard mode. It is pulled up internally to keep it high, till it is pulled low with a key closure.
- **BD – Blank Display** : This output pin is used to blank the display during digit switching or by a blanking closure.
- **OUT A_0 – OUT A_3 and OUT B_0 – OUT B_3** – These are the output ports for two 16*4 or 16*8 internal display refresh registers. The data from these lines is synchronized with the scan lines to scan the display and keyboard. The two 4-bit ports may also as one 8-bit port.
- **CNTL/STB- CONTROL/STROBED I/P Mode** : In keyboard mode, this lines is used as a control input and stored in FIFO on a key closure. The line is a strobed lines that enters the data into FIFO RAM, in strobed input mode. It has an interrupt pull up. The lines is pulled down with a key closer.

Modes of Operation of 8279

- The modes of operation of 8279 are as follows

: 1. Input (Keyboard) modes.

2. Output (Display) modes.

- **Input (Keyboard) Modes :** 8279 provides three input modes. These modes are as follows:

1. **Scanned Keyboard Mode** : This mode allows a key matrix to be interfaced using either encoded or decoded scans. In encoded scan, an 8*8 keyboard or in decoded scan, a 4*8 keyboard can be interfaced. The code of key pressed with SHIFT and CONTROL status is stored into the FIFO RAM.
2. **Scanned Sensor Matrix** : In this mode, a sensor array can be interfaced with 8279 using either encoded or decoded scans. With encoded scan 8*8 sensor matrix or with decoded scan 4*8 sensor matrix can be interfaced. The sensor codes are stored in the CPU addressable sensor RAM.
3. **Strobed input**: In this mode, if the control lines goes low, the data on return lines, is stored in the FIFO byte by byte.

- **Output (Display) Modes** : 8279 provides two output modes for selecting the display options. These are discussed briefly.

1. **Display Scan** : In this mode 8279 provides 8 or 16 character multiplexed displays those can be organized as dual 4- bit or single 8-bit display units.
2. **Display Entry** : (right entry or left entry mode) 8279 allows options for data entry on the displays. The display data is entered for display either from the right side or from the left side.

.

Interfacing a Microprocessor To Keyboard

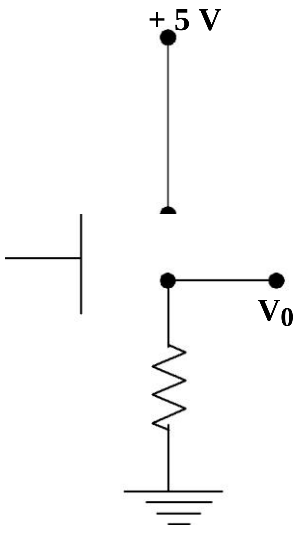
- When you press a key on your computer, you are activating a switch. There are many different ways of making these switches. An overview of the construction and operation of some of the most common types.

1. ***Mechanical key switches:*** In mechanical-switch keys, two pieces of metal are pushed together when you press the key. The actual switch elements are often made of a phosphor-bronze alloy with gold platting on the contact areas. The key switch usually contains a spring to return the key to the nonpressed position and perhaps a small piece of foam to help damp out bouncing.
2. Some mechanical key switches now consist of a molded silicon dome with a small piece of conductive rubber foam short two trace on the printed-circuit board to produce the key pressed signal.
3. Mechanical switches are relatively inexpensive but they have several disadvantages. First, they suffer from contact bounce. A pressed key may make and break contact several times before it makes solid contact.
4. Second, the contacts may become oxidized or dirty with age so they no longer make a dependable connection.

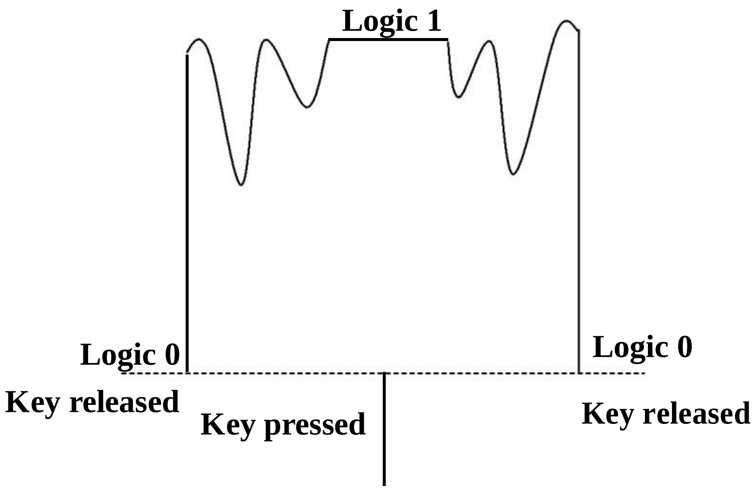
- Higher- quality mechanical switches typically have a rated life time of about 1 million keystrokes. The silicone dome type typically last 25 million keystrokes.

2. ***Membrane key switches:*** These switches are really a special type of mechanical switches. They consist of a three-layer plastic or rubber sandwich.

- The top layer has a conductive line of silver ink running under each key position. The bottom layer has a conductive line of silver ink running under each column of keys.



A Mechanical Key



Response

Interfacing To Alphanumeric Displays

- To give directions or data values to users, many microprocessor-controlled instruments and machines need to display letters of the alphabet and numbers. In systems where a large amount of data needs to be displayed a CRT is used to display the data. In system where only a small amount of data needs to be displayed, simple digit-type displays are often used.
- There are several technologies used to make these digit-oriented displays but we are discussing only the two major types.
- These are *light emitting diodes* (LED) and *liquid-crystal displays* (LCD).
- LCD displays use very low power, so they are often used in portable, battery-powered instruments. They do not emit their own light, they simply change the reflection of available light. Therefore, for an instrument that is to be used in low-light conditions, you have to include a light source for LCDs or use LEDs which emit their own light.
- Alphanumeric LED displays are available in three common formats. For displaying only number and hexadecimal letters, simple 7-segment displays such as that as shown in fig are used.
- To display numbers and the entire alphabet, 18 segment displays such as shown in fig or 5 by 7 dot-matrix displays such as that shown in fig can be used. The 7-segment type is the least expensive, most commonly used and easiest to interface with, so we will concentrate first on how to interface with this type.

1. ***Directly Driving LED Displays:*** Figure shows a circuit that you might connect to a parallel port on a microcomputer to drive a single 7-segment , common-anode display. For a common-anode display, a segment is tuned on by applying a logic low to it.

- The 7447 converts a BCD code applied to its inputs to the pattern of lows required to display the number represented by the BCD code. This circuit connection is referred to as a *static display* because current is being passed through the display at all times.

- Each segment requires a current of between 5 and 30mA to light. Let's assume you want a current of 20mA. The voltage drop across the LED when it is lit is about 1.5V.
- The output low voltage for the 7447 is a maximum of 0.4V at 40mA. So assume that it is about 0.2V at 20mA. Subtracting these two voltage drop from the supply voltage of 5V leaves 3.3V across the current limiting resistor. Dividing 3.3V by 20mA gives a value of 168Ω for the current-limiting resistor. The voltage drops across the LED and the output of 7447 are not exactly predictable and exact current through the LED is not critical as long as we don't exceed its maximum rating.

2. Software-Multiplexed LED Display:

- The circuit in fig works for driving just one or two LED digits with a parallel output port. However, this scheme has several problem if you want to drive, eight digits.
- The first problem is power consumption. For worst-case calculations, assume that all 8 digits are displaying the digit 8, so all 7 segments are all lit. Seven segment time 20mA per segment gives a current of 140mA per digit. Multiplying this by 8 digits gives a total current of 1120mA or 1.12A for 8 digits.
- A second problem of the static approach is that each display digit requires a separate 7447 decoder, each of which uses of another 13mA. The current required by the decoders and the LED displays might be several times the current required by the reset of the circuitry in the instrument.
- To solve the problem of the static display approach, we use a *multiplex method*, example for an explanation of the multiplexing.
- The fig shows a circuit you can add to a couple of microcomputer ports to drive some common anode LED displays in a multiplexed manner. The circuit has only one 7447 and that the segment outputs of the 7447 are bused in parallel to the segment inputs of all the digits.
- The question that may occur to you on first seeing this is: Aren't all the digits going to display the same number? The answer is that they would if all the digits were turned on at the same time. The tricky of multiplexing displays is that only one display digit is turned on at a time.
- The PNP transistor is series with the common anode of each digit acts as on/off switch for that digit. Here's how the multiplexing process works.
- The BCD code for digit 1 is first output from port B to the 7447. the 7447 outputs the corresponding 7-segment code on the segment bus lines. The transistor connected to digit 1 is then turned on by outputting a low to the appropriate bit of port A. All the rest of the bits of port A are made high to make sure no other digits are turned on. After 1 or 2 ms, digit 1 is turned off by outputting all highs to port A.
- The BCD code for digit 2 is then output to the 7447 on port B, and a word to turn on digit 2 is output on port A.
- After 1 or 2 ms, digit 2 is turned off and the process is repeated for digit 3. the process is continued until all the digits have had a turn. Then digit 1 and the following digits are lit again in turn.

- A procedure which is called on an interrupt basis every 2ms to keep these displays refreshed wit some values stored in a table. With 8 digits and 2ms per digit, you get back to digit 1 every 16ms or about 60 times a second.
- This refresh rate is fast enough so that the digits will each appear to be lit all time. Refresh rates of 40 to 200 times a second are acceptable.
- The immediately obvious advantages of multiplexing the displays are that only one 7447 is required, and only one digit is lit at a time. We usually increase the current per segment to between 40 and 60 mA for multiplexed displays so that they will appear as bright as they would if they were not multiplexed. Even with this increased segment current, multiplexing gives a large saving in power and parts.
- The software-multiplexed approach we have just described can also be used to drive 18-segment LED devices and dot-matrix LED device. For these devices, however you replace the 7447 in fig with ROM which generates the required segment codes when the ASCII code for a character is applied to the address inputs of the ROM

Interfacing Analog to Digital Data Converters

- In most of the cases, the PIO 8255 is used for interfacing the analog to digital converters with microprocessor.
- We have already studied 8255 interfacing with 8086 as an I/O port, in previous section. This section we will only emphasize the interfacing techniques of analog to digital converters with 8255.
- The analog to digital converters is treaded as an input device by the microprocessor, that sends an initialising signal to the ADC to start the analogy to digital data conversation process. The start of conversation signal is a pulse of a specific duration.
- The process of analog to digital conversion is a slow process, and the microprocessor has to wait for the digital data till the conversion is over. After the conversion is over, the ADC sends end of conversion EOC signal to inform the

  microprocessor that the conversion is over and the result is ready at the output buffer of the ADC. These tasks of issuing an SOC pulse to ADC, reading EOC signal from the ADC and reading the digital output of the ADC are carried out by the CPU using 8255 I/O ports.
- The time taken by the ADC from the active edge of SOC pulse till the active edge of EOC signal is called as the conversion delay of the ADC.
- It may range any where from a few microseconds in case of fast ADC to even a few hundred milliseconds in case of slow ADCs.
- The available ADC in the market use different conversion techniques for conversion of analog signal to digitals. Successive approximation techniques and dual slope integration techniques are the most popular techniques used in the integrated ADC chip.
- General algorithm for ADC interfacing contains the following steps:

1. Ensure the stability of analog input, applied to the ADC.
2. Issue start of conversion pulse to ADC
3. Read end of conversion signal to mark the end of conversion processes.
4. Read digital data output of the ADC as equivalent digital output.
5. Analog input voltage must be constant at the input of the ADC right from the start of conversion till the end of the conversion to get correct results. This may be ensured by a sample and hold circuit which samples the analog signal and holds it constant for a specific time duration. The microprocessor may issue a

hold signal to the sample and hold circuit.

6. If the applied input changes before the complete conversion process is over, the digital equivalent of the analog input calculated by the ADC may not be correct.

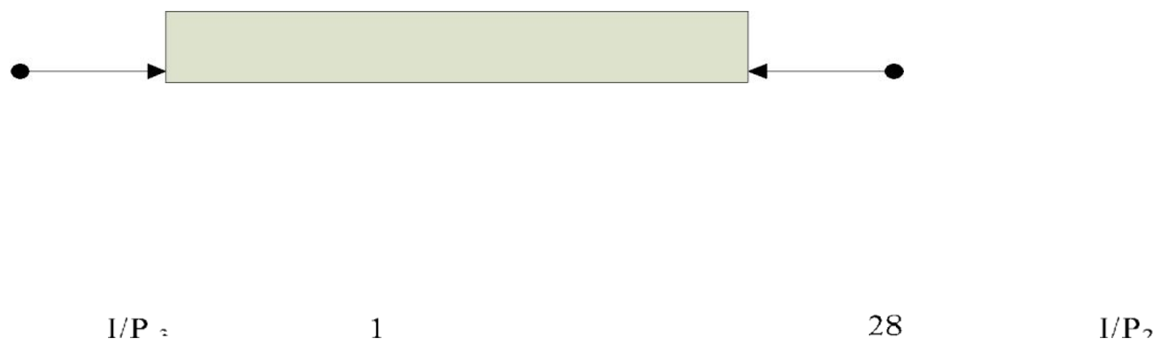
ADC 0808/0809 :

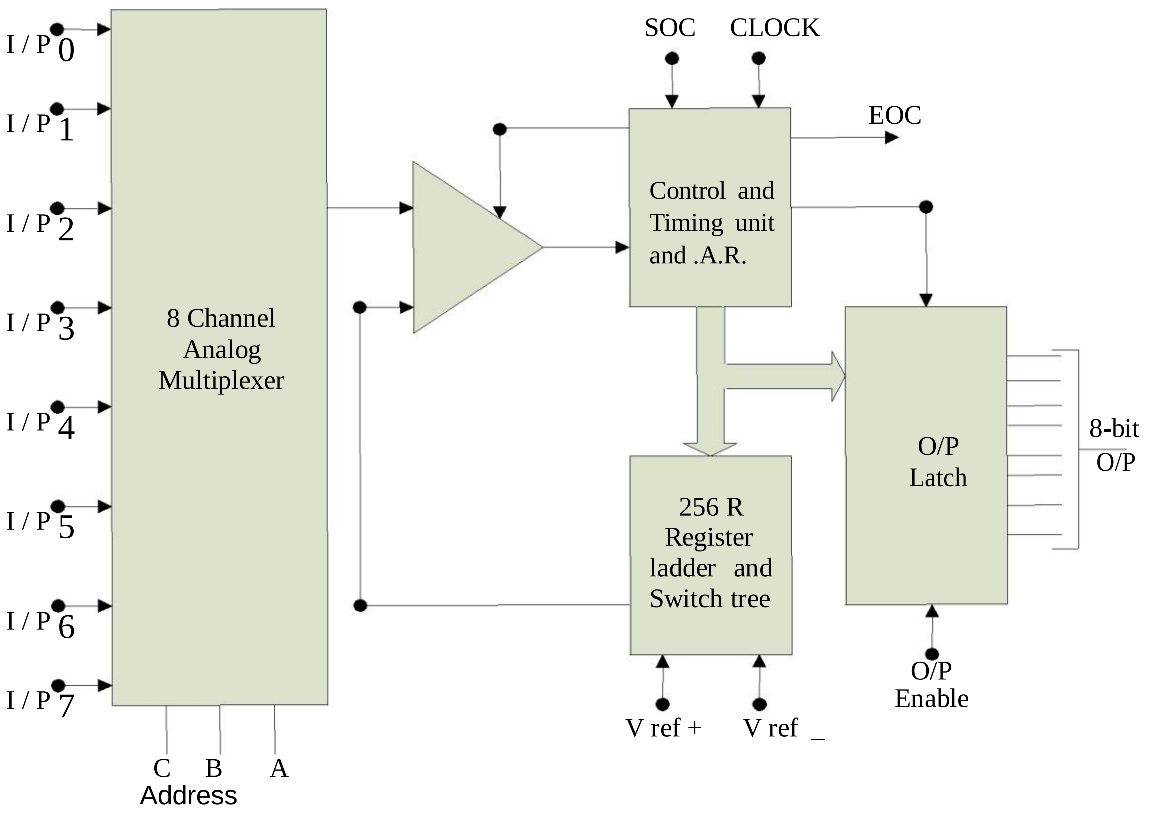
- The analog to digital converter chips 0808 and 0809 are 8-bit CMOS, successive approximation converters. This technique is one of the fast techniques for analog to digital conversion. The conversion delay is 100µs at a clock frequency of 640 KHz, which is quite low as compared to other converters. These converters do not need any external zero or full scale adjustments as they are already taken care of by internal circuits. These converters internally have a 3:8 analog multiplexer so that at a time eight different analog conversion by using address lines - ADD A, ADD B, ADD C. Using these address inputs, multichannel data acquisition system can be designed using a single ADC. The CPU may drive these lines using output port lines in case of multichannel applications. In case of single input applications, these may be hardwired to select the proper input.
- There are unipolar analog to digital converters, i.e. they are able to convert only positive analog input voltage to their digital equivalent. These chips do no contain any internal sample and hold circuit.

Analog /P selecte	Address lines		
	C	B	A
I / P_0	0	0	0
I / P_1	0	0	1
I / P_2	0	1	0
I / P_3	0	1	1
I / P_4	1	0	0
I / P_5	1	0	1
I / P_6	1	1	0
I / P_7	1	1	1

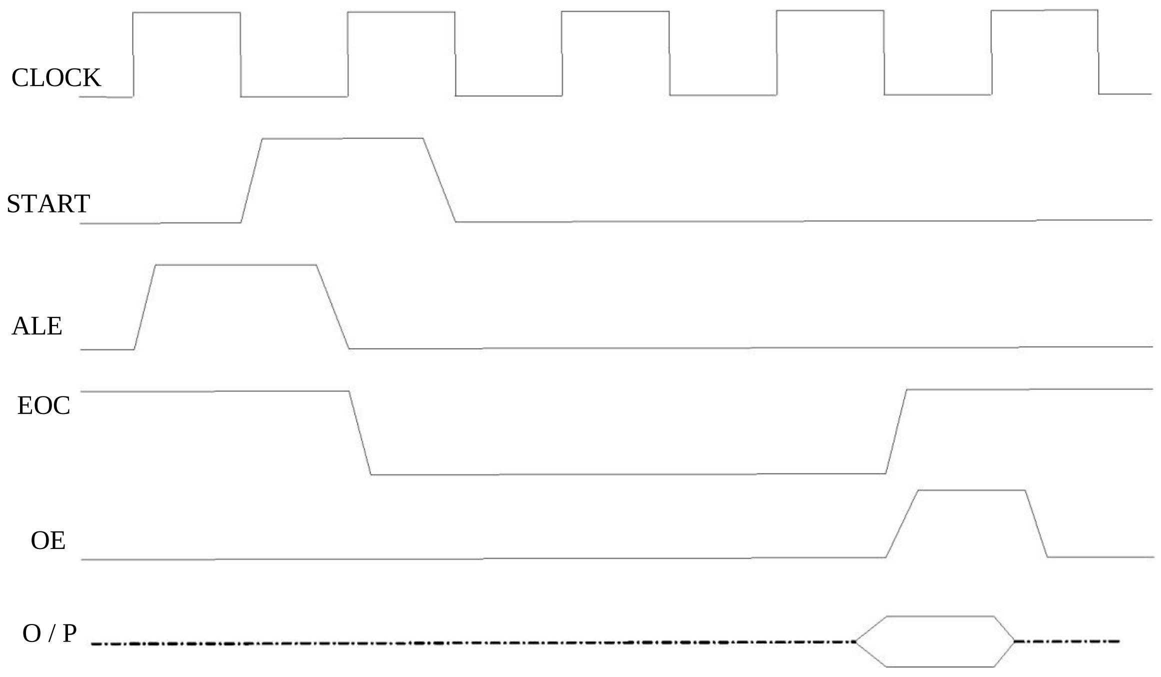
- If one needs a sample and hold circuit for the conversion of fast signal into equivalent digital quantities, it has to be externally connected at each of the analog inputs.

Vcc	Supply pins +5V
GND	GND
Vref +	**Reference voltage positive +5 Volts maximum.**
Vref _	**Reference voltage negative 0Volts minimum.**
I/P0 –I/P7	**Analog inputs**
ADD A,B,C	**Address lines for selecting analog inputs.**
O7 – O0	**Digital 8-bit output with O7 MSB and O0 LSB**
SOC	**Start of conversion signal pin**
EOC	**End of conversion signal pin**
OE	**Output latch enable pin, if high enables output**
CLK	**Clock input for ADC**





Block Diagram of ADC 0808 / 0809



Timing Diagram of ADC 0808

- ***Example:*** Interfacing ADC 0808 with 8086 using 8255 ports. Use port A of 8255 for transferring digital data output of ADC to the CPU and port C for control signals. Assume that an analog input is present at I/P2 of the ADC and a clock input of suitable frequency is available for ADC.

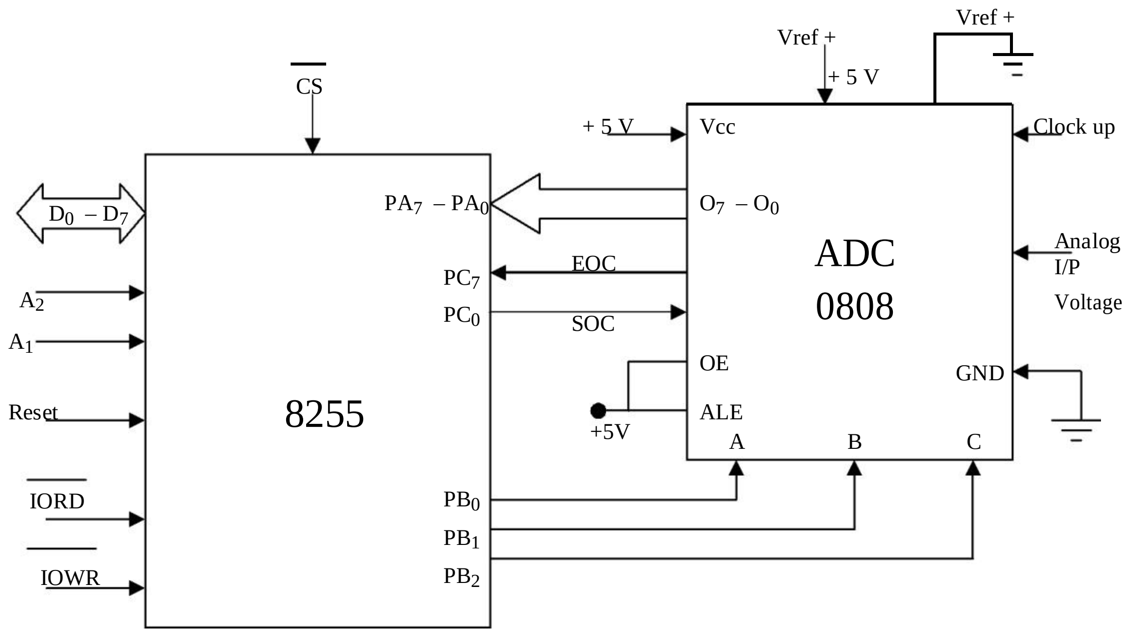
- **Solution**: The analog input I/P2 is used and therefore address pins A,B,C should be 0,1,0 respectively to select I/P2. The OE and ALE pins are already kept at +5V to select the ADC and enable the outputs. Port C upper acts as the input port to receive the EOC signal while port C lower acts as the output port to send SOC to the ADC.

- Port A acts as a 8-bit input data port to receive the digital data output from the ADC. The 8255 control word is written as follows:

  D7 D6 D5 D4 D3 D2 D1 D0
  1 0 0 1 1 0 0 0

- The required ALP is as follows:

```
        MOV   AL, 98h       ;initialise 8255 as
        OUT   CWR, AL       ;discussed above.
        MOV   AL, 02h       ;Select I/P2 as analog
        OUT   Port B, AL    ;input.
        MOV   AL, 00h       ;Give start of conversion
        OUT   Port C, AL    ; pulse to the ADC
        MOV   AL, 01h
        OUT   Port C, AL
        MOV   AL, 00h
        OUT   Port C, AL
WAIT:   IN    AL, Port C    ;Check for EOC by
        RCR                 ; reading port C upper and
        JNC   WAIT          ;rotating through carry.
        IN    AL, Port A    ;If EOC, read digital equivalent in AL
        HLT                 ;Stop.
```

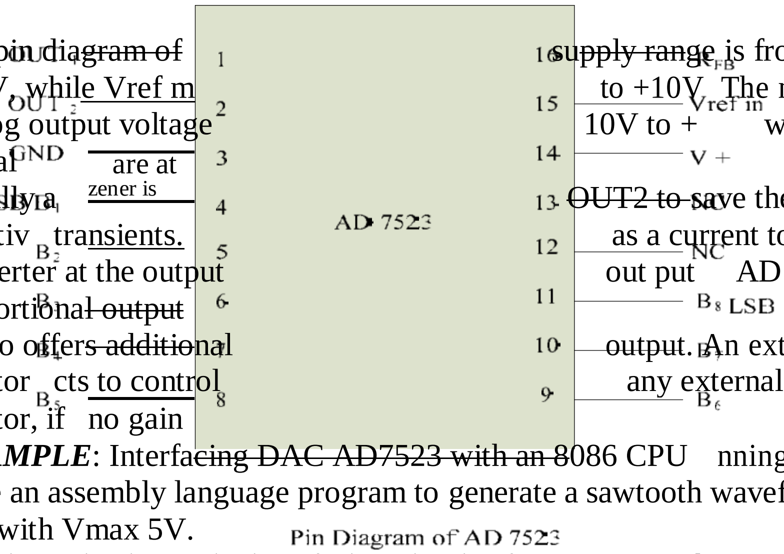


Interfacing 0808 with 8086

Interfacing Digital To Analog Converters

INTERFACING DIGITAL TO ANALOG CONVERTERS: The digital to analog converters convert binary number into their equivalent voltages. The DAC find applications in areas like digitally controlled gains, motors speed controls, programmable gain amplifiers etc.

AD 7523 8-bit Multiplying DAC : This is a 16 pin DIP, multiplying digital to analog converter, containing R- 2R ladder for D-A conversion along with single pole double thrown NMOS switches to connect the digital inputs to the ladder.



Pin Diagram of AD 7523

- The pin diagram of supply range is from +5V to +15V, while Vref m to +10V. The maximum analog output voltage 10V to + when all the digital are at
- Usually a zener is OUT2 to save the DAC from negativ transients. as a current to voltage converter at the output out put AD to a proportional output
- It also offers additional output. An external feedback resistor cts to control any external feedback resistor, if no gain
- ***EXAMPLE***: Interfacing DAC AD7523 with an 8086 CPU nning at 8MHZ and write an assembly language program to generate a sawtooth waveform of period 1ms with Vmax 5V.
- Solution: Fig shows the interfacing circuit of 74523 with 8086 using 8255. program gives an ALP to generate a sawtooth waveform using circuit.

```
ASSUME      CS:CODE
CODE SEGMENT
START       :MOV AL,80h         ;make all ports output
            OUT   CW, AL
AGAIN       :MOV  AL,00h        ;start voltage for ramp
BACK :      OUT   PA, AL
            INC   AL
            CMP   AL, 0FFh
            JB    BACK
            JMP   AGAIN
CODE ENDS
END         START
```

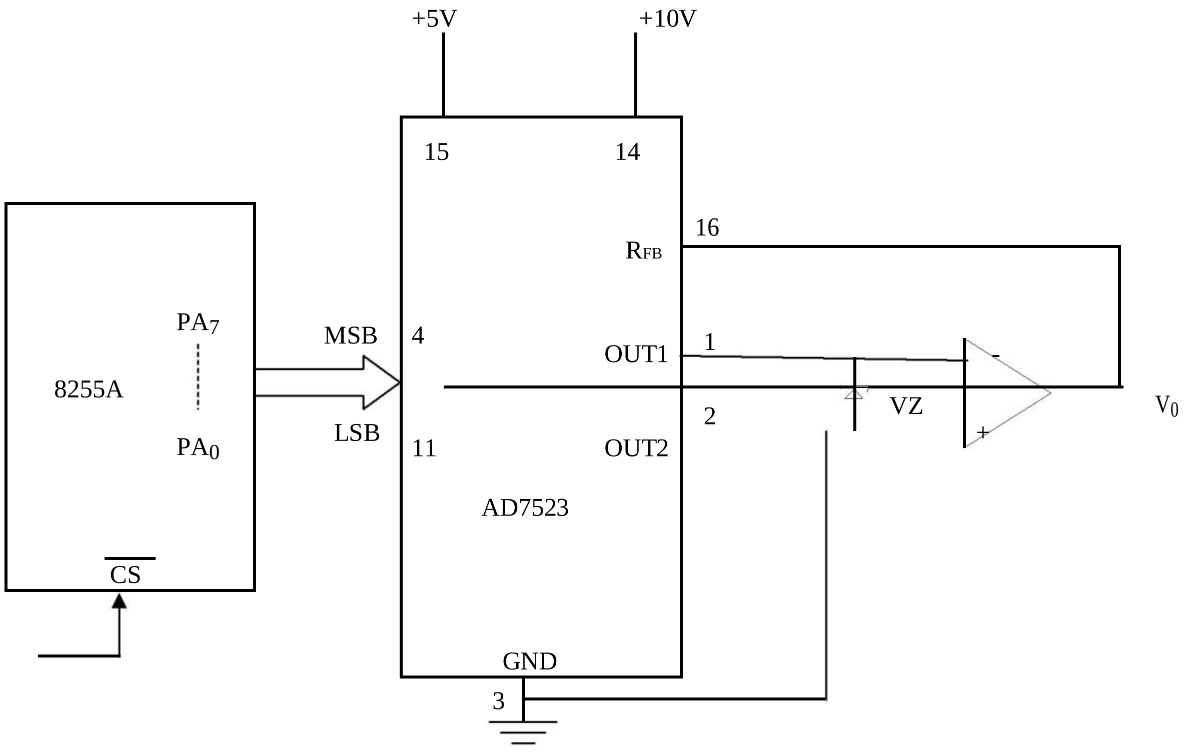


Fig: Interfacing of AD7523

- In the above program, port A is initialized as the output port for sending the digital data as input to DAC. The ramp starts from the 0V (analog), hence AL starts with 00H. To increment the ramp, the content of AL is increased during each execution of loop till it reaches F2H.
- After that the saw tooth wave again starts from 00H, i.e. 0V(analog) and the procedure is repeated. The ramp period given by this program is precisely 1.000625 ms. Here the count F2H has been calculated by dividing the required delay of 1ms by the time required for the execution of the loop once. The ramp slope can be controlled by calling a controllable delay after the OUT instruction.

UNIT IV

Microcontroller

Contents

•Introduction
•Inside 8051
•Instructions
•Interfacing

Introduction

- Definition of a Microcontroller
- Difference with a Microprocessor
- Microcontroller is used where ever

Definition

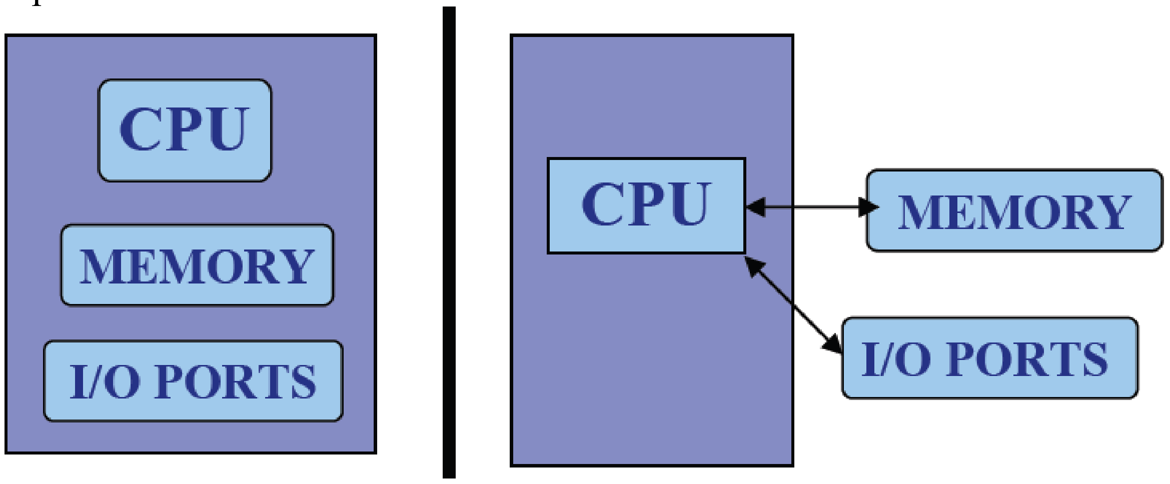
- It is a single chip
- Consists of Cpu, Memory
- I/O ports, timers and other peripherals

- It is a cpu
- Memory, I/O Ports to be connected externally.

MICRO PROCESSER

MICRO CONTROLLER

- It is a single chip
- Consists Memory,
- I/o ports



Where ever

- Small size
- Low cost
- Low power

Architecture

•Harvard university

The Architecture given by Harvard University has the following advantages:

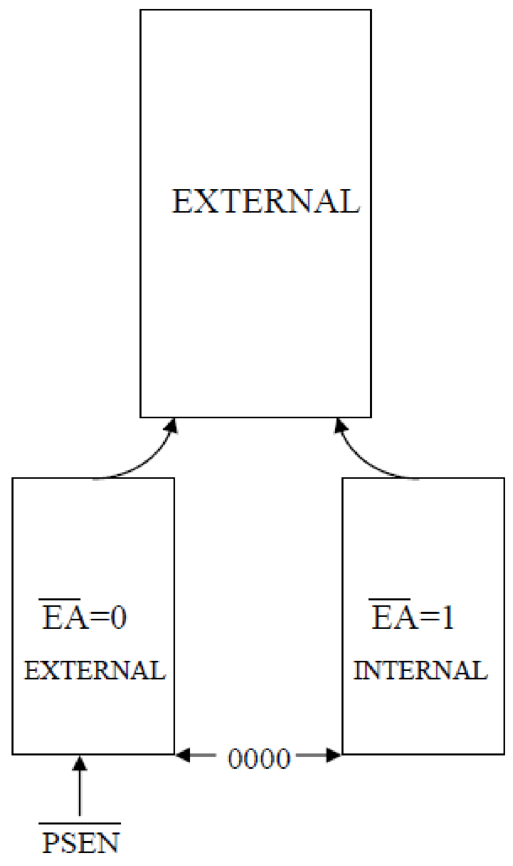
1: Data Space and Program Space are distinct

2: There is no Data corruption or loss of data

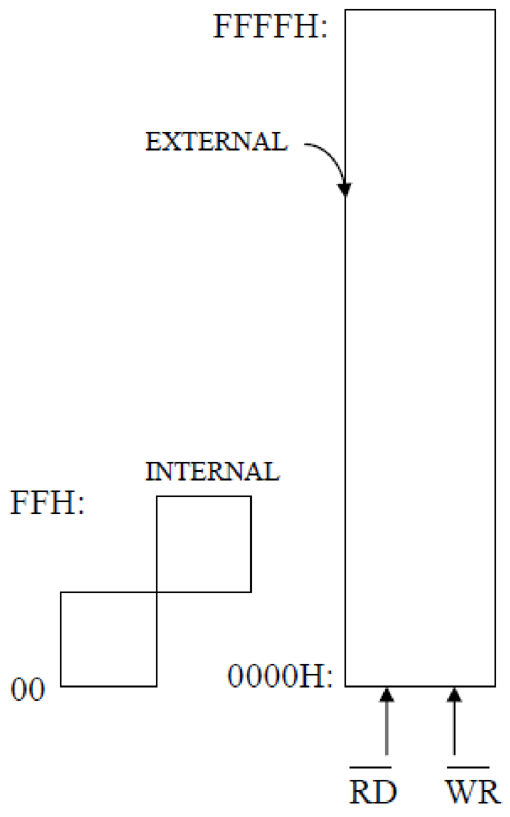
Disadvantage is:

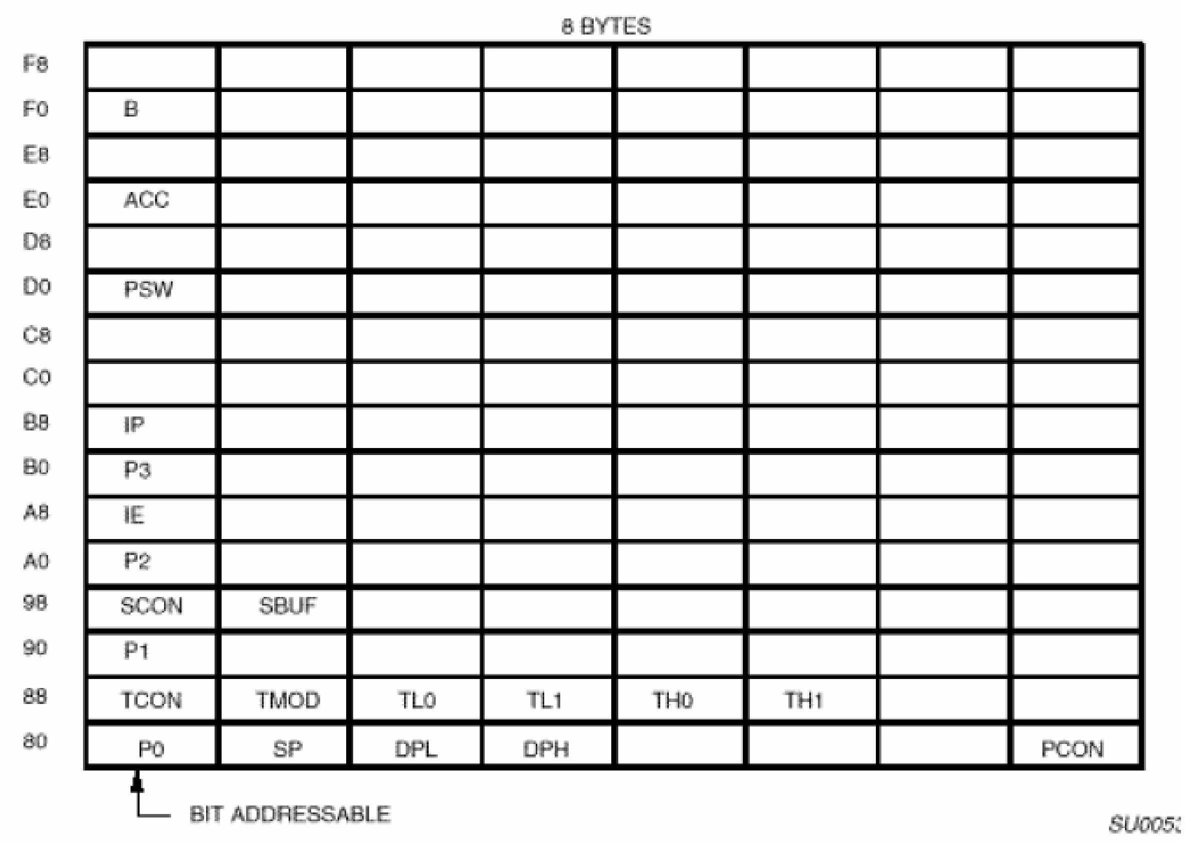
1: The circuitry is very complex.

Features


EXTERNAL
$\overline{EA}$=0
EXTERNAL
$\overline{EA}$=1
INTERNAL
0000
$\overline{PSEN}$

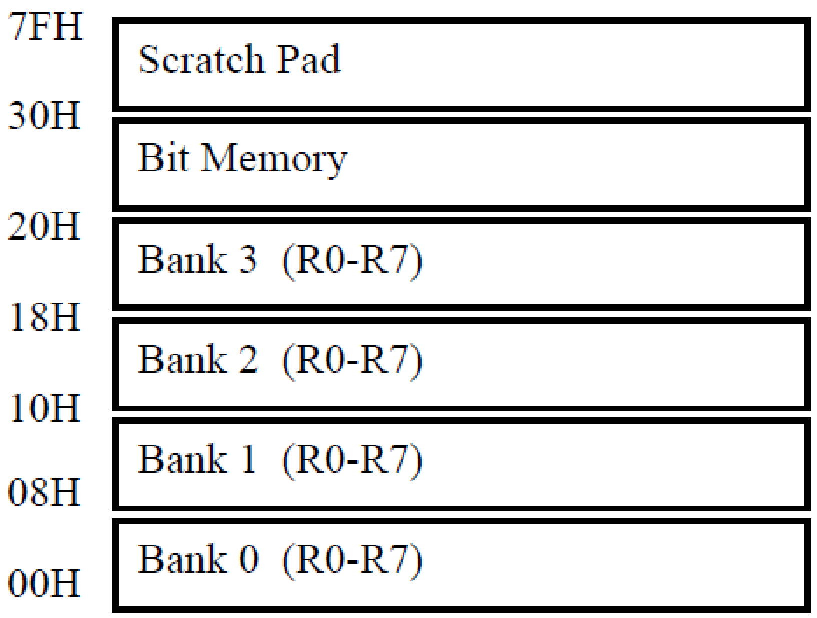
SFR Map


FFFFH:
EXTERNAL
INTERNAL
FFH:
00
0000H:
$\overline{RD}$
$\overline{WR}$


8 BYTES
BIT ADDRESSABLE
SU005

F8								
F0	B							
E8								
E0	ACC							
D8								
D0	PSW							
C8								
C0								
B8	IP							
B0	P3							
A8	IE							
A0	P2							
98	SCON	SBUF						
90	P1							
88	TCON	TMOD	TL0	TL1	TH0	TH1		
80	P0	SP	DPL	DPH				PCON

Internal Memory

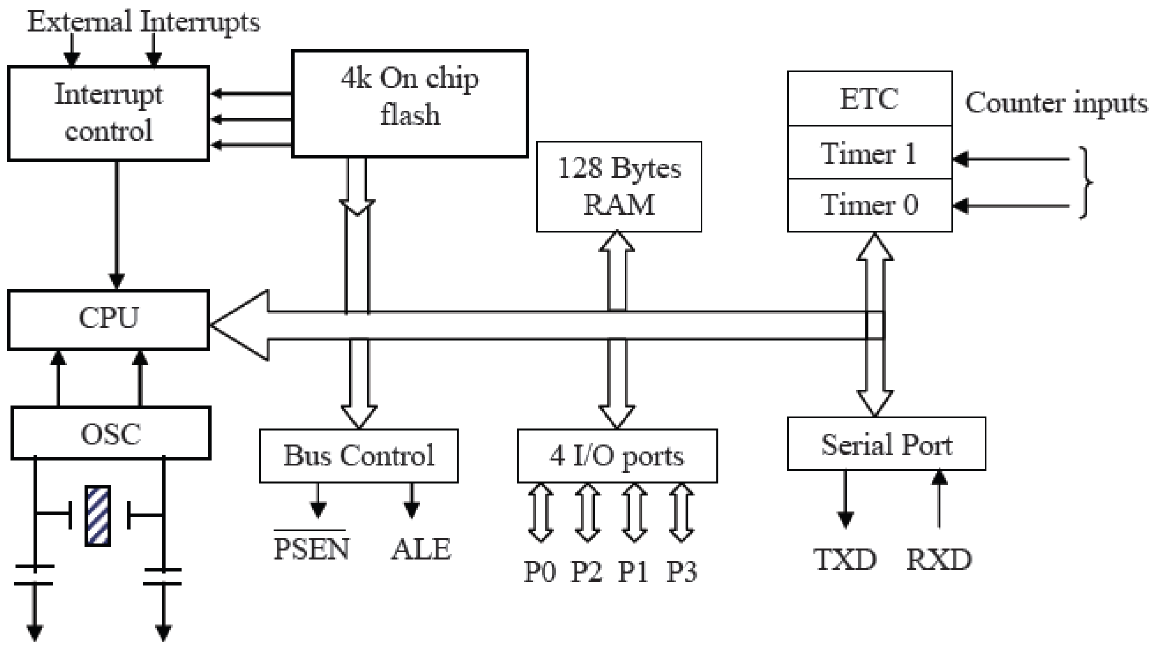


Pin connections

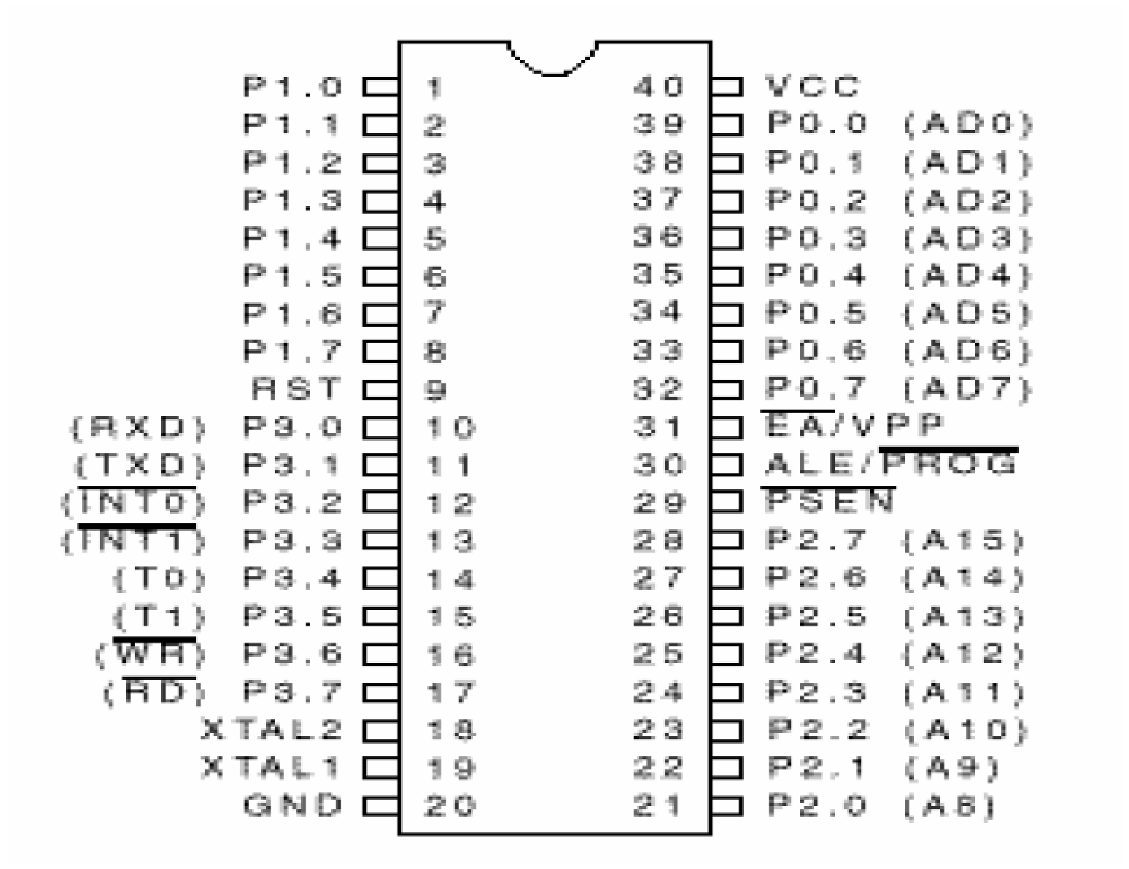
1

....8051 microcontroller

Block Diagram



Memory Architecture



8051: Types of Memory

The 8051 has three very general types of memory. To effectively program the 8051 it is necessary to have a basic understanding of these
memory types
.

On-Chip Memory refers to any memory(Code, RAM, or other) that physically exists on the microcontroller itself. On-chip memory can be of
several types, but we'll get into that shortly.

External Code Memory is code (or program) memory that resides off-chip. This is
often in the form of an external EPROM.

External RAM is RAM memory that resides off-chip. This is often in the form of
standard static RAM or flash RAM.

Code Memory

Code memory is the memory that holds the actual 8051 program that is to be run. This memory is limited to 64K and comes in many shapes and sizes: Code memory may be found
on-chip, either burned into the microcontroller as ROM or EPROM. Code may also be stored
completely *off-chip* in an external ROM or, more commonly, an external EPROM. Flash RAM is
also another popular method of storing a program. Various combinations of these memory types may
also be used--that is to say, it is possible to have4K of code memory *on-chip* and 64k of code
memory *off-chip* in an EPROM. When the program is stored on-chip the
64K maximum is often reduced to 4k, 8k, or 16k. This varies depending on the version of the chip
that is being used. Each version offers specific capabilities and one of the distinguishing factors

from chip to chip is how much ROM/EPROM space the chip has.
However, code memory is most commonly implemented as off-chip EPROM

Programming Tip: *Since code memory is restricted to 64K, 8051 programs are limited to 64K. Some assemblers and compilers offer ways to get around this limit when used with specially wired hardware. However, without such special compilers and hardware, programs are limited to 64K.*

External RAM
As an obvious opposite of *Internal RAM*, the 8051 also supports what is called *External RAM*. As the name suggests, External RAM is any

random access memory which is found *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 requires only 1 instruction and 1 instruction
cycle. To increment a 1-byte value stored in
External RAM requires 4 instructions and 7
instruction cycles. In this case, external memory is
7 times slower!
What External RAM loses in speed and flexibility it gains in quantity. While Internal RAM is limited to 128 bytes the 8051 supports External
RAM up to 64K.

Programming Tip: *The 8051 may only address 64k of RAM. To expand RAM beyond this limit requires programming and hardware tricks. You may have to do this "by hand" since many compilers and
assemblers, while providing support for programs in excess of 64k, do not support more than 64k of RAM.
This is rather strange since it has been my experiencethat programs can usually fit in 64k but often RAM is
what is lacking. Thus if you need more than 64k of RAM,
check to see if your compiler supports it-- but if it
doesn't, be prepared to do it by hand*
.

On-Chip Memory.
the 8051 includes a certain amount of onchipmemory. On-chip memory is really one of two (SFR) memory. the 8051 has bank of 128 bytes of *Internal RAM*. This Internal RAMis found *on-chip* on the 8051 so it is the fastest RAMavailable, and it is also the most flexible in terms ofreading, writing, and modifying it's contents. InternalRAM is volatile, so when the 8051 is reset this
memory is cleared.The 128 bytes of internal ram is subdivided
as shown on the memory map. The first 8 bytes (00h- 07h) are "register bank 0". By manipulating certainSFRs, a program may choose to use register banks1, 2, or 3. These alternative register banks arelocated in internal RAM in addresses 08h through1Fh.The 80 bytes remaining of Internal RAM, fromaddresses 30h through 7Fh, may be used by uservariables that need to be accessed frequently or athigh-speed. This area is also utilized by themicrocontroller as a storage area for the operating
stack. This fact severely limits the 8051's stack since,as illustrated in the memory map, the area reservedfor the stack is only 80 bytes--and usually it is lesssince this 80 bytes has to be shared between thestack and user variables.
Register Banks
The 8051 uses 8 "R" registers which areused in many of its instructions. These "R"registers are numbered from 0 through 7 (R0, R1,R2, R3, R4, R5, R6, and R7). These registers aregenerally used to assist in manipulating valuesand moving data from one memory location to
another. For example, to add the value of R4 tothe Accumulator, we would execute the following
instruction:ADD A,R4When the 8051 is first booted up, register bank 0(addresses 00h through 07h) is used by default.However, your program may instruct the 8051 to

use one of the alternate register banks; i.e. register banks 1, 2, or 3. In this case, R4 will no longer be the same as Internal RAM address 04h. For example, if your program instructs the 8051 to
use register bank 3, "R" register R4 will now be synonomous with Internal RAM address 1Ch.
The concept of register banks adds a great level of flexibility to the 8051, especially when dealing with interrupts However, always remember that
the register banks really reside in the first 32 bytes

Bit Memory

The 8051, being a communicationsorientedmicrocontroller, gives the user the ability to access a number of *bit variables*. Thesevariables may be either 1 or 0.
There are 128 bit variables available to theuser, numberd 00h through 7Fh. The user may make use of these variables with commands suchas SETB and CLR.
It is important to note that Bit Memory isreally a part of Internal RAM. In fact, the 128 bit variables occupy the 16 bytes of Internal RAMfrom 20h through 2Fh. Thus, if you write the value
FFh to Internal RAM address 20h you'veeffectively set bits 00h through 07h.But since the 8051 provides specialinstructions to access these 16 bytes of memoryon a bit by bit basis it is useful to think of it as aseparate type of memory. However, always keepin mind that it is just a subset of Internal RAM--anthat operations performed on Internal RAM canchange the values of the bit variables.
*use bit variables, you may use Internal RAM locations*Bit variables 00h through 7Fh are for userdefined
functions in their programs. However, bitvariables 80h and above are actually used to access certain SFRs on a bit-by-bit basis. Forexample, if output lines P0.0 through P0.7 are all
clear (0) and you want to turn on the P0.0 outputline you may either execute:
MOV P0,#01h || SETB 80hBoth these instructions accomplish thesame thing. However, using the SETB commandwill turn on the P0.0 line without effecting thestatus of any of the other P0 output lines. The
MOV command effectively turns off all the otheroutput lines which, in some cases, may not be
acceptable.

Special Function Register (SFR) Memory

Special Function Registers (SFRs) area of memory that control specific functionalityof the 8051 processor. For example, four SFRspermit access to the 8051's 32 input/output lines. Another SFR allows a program to read or write tothe 8051's serial port. Other SFRs allow the useto set the serial baud rate, control and accesstimers, and configure the 8051's interrupt system.
When programming, SFRs have theillusion of being Internal Memory. For example, ifyou want to write the value "1" to Internal RAMlocation 50 hex you would execute the instruction:
MOV 50h,#01Similarly, if you want to write the value "1"
to the 8051's serial port you would write this valueto the **SBUF** SFR, which has an SFR address of
99 Hex. Thus, to write the value "1" to the serialport you would execute the instruction:
MOV 99h,#01hAs you can see, it appears that the SFR ispart of Internal Memory. This is not the case.When using this method of memory access (it'scalled direct address), any instruction that has anaddress of 00h through 7Fh refers to an InternalRAM memory address; any instruction with an
address of 80h through FFh refers to an SFRcontrol register.

What Are SFRs?

The 8051 is a flexible microcontroller wita relatively large number of modes of operations. Your program may inspect and/or change theoperating mode of the 8051 by manipulating the
values of the 8051's Special Function Registers(SFRs).
SFRs are accessed as if they were normalInternal RAM. The only difference is that Internal RAM is from address 00h through 7Fh whereasSFR registers exist in the address range of 80h
through FFh.Each SFR has an address (80h throughFFh) and a name. The following chart provides a
graphical presentation of the 8051's SFRs, theirnames, and their address.

As you can see, although the addressrange of 80h through FFh offer 128 possible addresses, there are only 21 SFRs in a standard8051. All other addresses in the SFR range (80h
through FFh) are considered invalid. Writing to orreading from these registers may produce undefined values or behavior.

SFR Types
As mentioned in the chart itself, the SFRsthat have a blue background are SFRs related to the I/O ports. The 8051 has four I/O ports of 8 bits,for a total of 32 I/O lines. Whether a given I/O line
is high or low and the value read from the line arecontrolled by the SFRs in green.
The SFRs with yellow backgrouns areSFRs which in some way control the operation or the configuration of some aspect of the 8051. Forexample, **TCON** controls the timers, SCON
controls the serial portThe remaining SFRs, with greenbackgrounds, are "other SFRs." These SFRscanbe thought of as auxillary SFRs in the sense thatthey don't directly configure the 8051 but obviously
the 8051 cannot operate without them. Forexample, once the serial port has been configured
using **SCON**, the program may read or write to theserial port using the **SBUF** register.
SFR Descriptions
This section will endeavor to quicklyoverview each of the standard SFRs found in the above SFR chart map. It is not the intention of thissection to fully explain the functionality of each
SFR--this information will be covered in separatechapters of the tutorial. This section is to just give
you a general idea of what each SFR does.
P0 (Port 0, Address 80h, Bit-Addressable): This is input/output port 0. Each bit of this SFR
corresponds to one of the pins on the microcontroller. For example, bit 0 of port 0 is pin P0.0, bit 7 is pin
P0.7. Writing a value of 1 to a bit of this SFR will send a high level on the corresponding I/O pin whereas
a value of 0 will bring it to a low level.
Programming Tip: *While the 8051 has four I/O port (P0, P1, P2, and P3), if your hardware uses external*
RAM or external code memory (i.e., your program is stored in an external ROM or EPROM chip or if you are using
external RAM chips) you may not use P0 or P2. This is because the 8051 uses ports P0 and P2 to address the
external memory. Thus if you are using external RAM or code memory you may only use ports P1 and P3 for your
own use.
SP (Stack Pointer, Address 81h): This is the stack pointer of the microcontroller. This SFR indicates where the next value to be taken from the stack will be read from in Internal RAM. If you push a
value onto the stack, the value will be written to the address of SP + 1. That is to say, if SP holds the
value 07h, a PUSH instruction will push the value onto the stack at address 08h. This SFR is modified by
all instructions which modify the stack, such as PUSH, POP, LCALL, RET, RETI, and whenever interrupts
are provoked by the microcontroller.
Programming Tip: *The SP SFR, on startup, is initialized to 07h. This means the stack will start at 08h and*
start expanding upward in internal RAM. Since alternate register banks 1, 2, and 3 as well as the user bit variables
occupy internal RAM from addresses 08h through 2Fh, it is necessary to initialize SP in your program to some other
value if you will be using the alternate register banks and/or bit memory. It's not a bad idea to initialize SP to 2Fh as
the first instruction of every one of your programs unless you are 100% sure you will not be using the register banks
and bit variables.
DPL/DPH (Data Pointer Low/High, Addresses 82h/83h): The SFRs DPL and DPH work

together to represent a 16-bit value called the *Data Pointer*. The data pointer is used in operations
regarding external RAM and some instructions involving code memory. Since it is an unsigned two-byte
integer value, it can represent values from 0000h to FFFFh (0 through 65,535 decimal).
Programming Tip: *DPTR is really DPH and DPL taken together as a 16-bit value. In reality, you almost
always have to deal with DPTR one byte at a time. For example, to push DPTR onto the stack you must first push
DPL and then DPH. You can't simply plush DPTR onto the stack. Additionally, there is an instruction to "increment
DPTR." When you execute this instruction, the two bytes are operated upon as a 16-bit value. However, there is no
instruction that decrements DPTR. If you wish to decrement the value of DPTR, you must write your own code to do
so.*
PCON (Power Control, Addresses 87h): The Power Control SFR is used to control the 8051's
power control modes. Certain operation modes of the 8051 allow the 8051 to go into a type of "sleep"
mode which requires much less power. These modes of operation are controlled through PCON.
Additionally, one of the bits in PCON is used to double the effective baud rate of the 8051's serial port.
TCON (Timer Control, Addresses 88h, Bit-Addressable): The Timer Control SFR is used to
configure and modify the way in which the 8051's two timers operate. This SFR controls whether each of
the two timers is running or stopped and contains a flag to indicate that each timer has overflowed.
Additionally, some non-timer related bits are located in the TCON SFR. These bits are used to configure
the way in which the external interrupts are activated and also contain the external interrupt flags which
are set when an external interrupt has occured.
TMOD (Timer Mode, Addresses 89h): The Timer Mode SFR is used to configure the mode of
operation of each of the two timers. Using this SFR your program may configure each timer to be a 16-bit
timer, an 8-bit autoreload timer, a 13-bit timer, or two separate timers. Additionally, you may configure the
timers to only count when an external pin is activated or to count "events" that are indicated on an
external pin.
TL0/TH0 (Timer 0 Low/High, Addresses 8Ah/8Bh): These two SFRs, taken together, represent
timer 0. Their exact behavior depends on how the timer is configured in the TMOD SFR; however, these
timers always count up. What is configurable is how and when they increment in value.

TL1/TH1 (Timer 1 Low/High, Addresses 8Ch/8Dh): These two SFRs, taken together, represent
timer 1. Their exact behavior depends on how the timer is configured in the TMOD SFR; however, these
timers always count up. What is configurable is how and when they increment in value.
P1 (Port 1, Address 90h, Bit-Addressable): This is input/output port 1. Each bit of this SFR
corresponds to one of the pins on the microcontroller. For example, bit 0 of port 1 is pin P1.0, bit 7 is pin
P1.7. Writing a value of 1 to a bit of this SFR will send a high level on the corresponding I/O pin whereas
a value of 0 will bring it to a low level.
SCON (Serial Control, Addresses 98h, Bit-Addressable): The Serial Control SFR is used to

configure the behavior of the 8051's on-board serial port. This SFR controls the baud rate of the serial port, whether the serial port is activated to receive data, and also contains flags that are set when a byte is successfully sent or received.

Programming Tip: *To use the 8051's on-board serial port, it is generally necessary to initialize the following SFRs: SCON, TCON, and TMOD. This is because SCON controls the serial port. However, in most cases the program will wish to use one of the timers to establish the serial port's baud rate. In this case, it is necessary to configure timer 1 by initializing TCON and TMOD.*

SBUF (Serial Control, Addresses 99h): The Serial Buffer SFR is used to send and receive data via the on-board serial port. Any value written to SBUF will be sent out the serial port's TXD pin. Likewise, any value which the 8051 receives via the serial port's RXD pin will be delivered to the user program via SBUF. In other words, SBUF serves as the output port when written to and as an input port when read from.

P2 (Port 2, Address A0h, Bit-Addressable): This is input/output port 2. Each bit of this SFR corresponds to one of the pins on the microcontroller. For example, bit 0 of port 2 is pin P2.0, bit 7 is pin P2.7. Writing a value of 1 to a bit of this SFR will send a high level on the corresponding I/O pin whereas a value of 0 will bring it to a low level.

Programming Tip: *While the 8051 has four I/O port (P0, P1, P2, and P3), if your hardware uses external RAM or external code memory (i.e., your program is stored in an external ROM or EPROM chip or if you are using external RAM chips) you may not use P0 or P2. This is because the 8051 uses ports P0 and P2 to address the external memory. Thus if you are using external RAM or code memory you may only use ports P1 and P3 for your own use.*

IE (Interrupt Enable, Addresses A8h): The Interrupt Enable SFR is used to enable and disable specific interrupts. The low 7 bits of the SFR are used to enable/disable the specific interrupts, where as the highest bit is used to enable or disable ALL interrupts. Thus, if the high bit of IE is 0 all interrupts are disabled regardless of whether an individual interrupt is enabled by setting a lower bit.

P3 (Port 3, Address B0h, Bit-Addressable): This is input/output port 3. Each bit of this SFR corresponds to one of the pins on the microcontroller. For example, bit 0 of port 3 is pin P3.0, bit 7 is pin P3.7. Writing a value of 1 to a bit of this SFR will send a high level on the corresponding I/O pin whereas a value of 0 will bring it to a low level.

IP (Interrupt Priority, Addresses B8h, Bit-Addressable): The Interrupt Priority SFR is used to specify the relative priority of each interrupt. On the 8051, an interrupt may either be of low (0) priority or high (1) priority. An interrupt may only interrupt interrupts of lower priority. For example, if we configure the 8051 so that all interrupts are of low priority except the serial interrupt, the serial interrupt will always be able to interrupt the system, even if another interrupt is currently executing. However, if a serial interrupt is executing no other interrupt will be able to interrupt the serial interrupt routine since the serial interrupt routine has the highest priority.

PSW (Program Status Word, Addresses D0h, Bit-Addressable): The Program Status Word is
used to store a number of important bits that are set and cleared by 8051 instructions. The PSW SFR
contains the carry flag, the auxiliary carry flag, the overflow flag, and the parity flag. Additionally, the PSW
register contains the register bank select flags which are used to select which of the "R" register banks
are currently selected.
Programming Tip: *If you write an interrupt handler routine, it is a very good idea to always save the PSW*
SFR on the stack and restore it when your interrupt is complete. Many 8051 instructions modify the bits of PSW. If
your interrupt routine does not guarantee that PSW is the same upon exit as it was upon entry, your program is
bound to behave rather erradically and unpredictably--and it will be tricky to debug since the behavior will tend not to
make any sense.
8
ACC (Accumulator, Addresses E0h, Bit-Addressable): The Accumulator is one of the mostused
SFRs on the 8051 since it is involved in so many instructions. The Accumulator resides as an SFR
at E0h, which means the instruction **MOV A,#20h** is really the same as **MOV E0h,#20h**. However, it is a
good idea to use the first method since it only requires two bytes whereas the second option requires
three bytes.
B (B Register, Addresses F0h, Bit-Addressable): The "B" register is used in two instructions:
the multiply and divide operations. The B register is also commonly used by programmers as an auxiliary
register to temporarily store values.

8051 Basic Registers

The Accumulator

.
The Accumulator, as it's name suggests,is used as a general register to accumulate the results of a large number of instructions. It canhold an 8-bit (1-byte) value and is the mostversatile register the 8051 has due to the shearnumber of instructions that make use of theaccumulator. More than half of the 8051's 255instructions manipulate or use the accumulator insome way.
For example, if you want to add thenumber 10 and 20, the resulting 30 will be storein theAccumulator. Once you have a value in theAccumulator you may continue processing the
value or you may store it in another register or inmemory.
The "R" registers The "R" registers are a set of eightregisters that are named R0, R1, etc. up to and
including R7.These registers are used as auxillaryregisters in many operations. To continue with the
above example, perhaps you are adding 10 and20. The original number 10 may be stored in the
Accumulator whereas the value 20 may be storedin, say, register R4. To process the addition you
would execute the command:ADD A,R4After executing this instruction theAccumulator will contain the value 30.You may think of the "R" registers as veryimportant auxillary, or "helper", registers. The
Accumulator alone would not be very useful if itwere not for these "R" registers.
The "R" registers are also used totemporarily store values. For example, let's say
you want to add the values in R1 and R2 togetherand then subtract the values of R3 and R4. Oneway to do this would be:MOV A,R3 ;Move the value of R3 into the accumulator

```
ADD A,R4 ;Add the value of R4
MOV R5,A ;Store the resulting value temporarily in R5
MOV A,R1 ;Move the value of R1 into the accumulator
ADD A,R2 ;Add the value of R2
SUBB A,R5 ;Subtract the value of R5 (which now contains R3 + R4)
```

As you can see, we used R5 to
temporarily hold the sum of R3 and R4. Of course,
this isn't the most efficient way to calculate
(R1+R2) - (R3 +R4) but it does illustrate the use of
the "R" registers as a way to store values
temporarily.
The "B" Register The "B" register is very similar to theAccumulator in the sense that it may hold an 8-bit(1-byte) value.The "B" register is only used by two 8051instructions: MUL AB and DIV AB. Thus, if youwant to quickly and easily multiply or divide A byanother number, you may store the other number
in "B" and make use of these two instructionAside from the MUL and DIV instructions,the "B" register is often used as yet anothertemporary storage register much like a ninth "R"register.
The Data Pointer (DPTR)
The Data Pointer (DPTR) is the 8051'sonly user-accessable 16-bit (2-byte) register. TheAccumulator, "R" registers, and "B" register are all1-byte values.DPTR, as the name suggests, is used topoint to data. It is used by a number of commandswhich allow the 8051 to access external memory.
When the 8051 accesses external memory it willaccess external memory at the address indicatedby DPTR.While DPTR is most often used to point todata in external memory, many programmers often
take advantge of the fact that it's the only true 16-bit register available. It is often used to store 2-byte values which have nothing to do with memorylocations.

The Program Counter (PC)
The Program Counter (PC) is a 2-byteaddress which tells the 8051 where the nextinstruction to execute is found in memory. Whenthe 8051 is initialized PC always starts at 0000hand is incremented each time an instruction isexecuted. It is important to note that PC isn'talways incremented by one. Since someinstructions require 2 or 3 bytes the PC will beincremented by 2 or 3 in these cases.
The Program Counter is special in thatthere is no way to directly modify it's value. That isto say, you can't do something like PC=2430h. Onthe other hand, if you execute LJMP 2340h you've
effectively accomplished the same thing.It is also interesting to note that while you
may change the value of PC (by executing a jumpinstruction, etc.) there is no way to read the value
of PC. That is to say, there is no way to ask the8051 "What address are you about to execute?"As it turns out, this is not completely true: There isone trick that may be used to determine the
current value of PC
The Stack Pointer (SP)
The Stack Pointer, like all registers exceptDPTR and PC, may hold an 8-bit (1-byte) value. The Stack Pointer is used to indicate where thenext value to be removed from the stack should be
taken from.When you push a value onto the stack, the8051 first increments the value of SP and then
stores the value at the resulting memory location.
When you pop a value off the stack, the8051 returns the value from the memory locationindicated by SP, and then decrements the value ofSP.
This order of operation is important. Whenthe 8051 is initialized SP will be initialized to 07h. If
you immediately push a value onto the stack, thevalue will be stored in Internal RAM address 08h.
This makes sense taking into account what wasmentioned two paragraphs above: First the 8051
will increment the value of SP (from 07h to 08h)and then will store the pushed value at that memory address (08h).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 (more on interrupts later.

8051: Addressing Modes

An "addressing mode" refers to how youare addressing a given memory location. Insummary, the addressing modes are as follows,with an example of each:Immediate Addressing MOV A,#20h
Direct Addressing MOV A,30hIndirect Addressing MOV A,@R0External Direct MOVX A,@DPTR
Code Indirect MOVC A,@A+DPTREach of these addressing modes provides important flexibility.

Immediate Addressing

Immediate addressing is so-namedbecause the value to be stored in memory immediately follows the operation code in memory.For example, the instruction:
MOV A,#20hThis instruction uses ImmediateAddressing because the Accumulator will be loaded with the value that immediately follows; inthis case 20 (hexidecimal).
Immediate addressing is very fast sincethe value to be loaded is included in theinstruction. However, since the value to be loadedis fixed at compile-time it is not very flexible.

Direct Addressing

Direct addressing is so-named becausethe value to be stored in memory is obtained by directly retrieving it from another memory location.For example:
MOV A,30hThis instruction will read the data out ofInternal RAM address 30 (hexidecimal) and store it
in the Accumulator.Direct addressing is generally fast since,although the value to be loaded isn’t included inthe instruction, it is quickly accessable since it is
stored in the 8051’s Internal RAM. It is also muchmore flexible than Immediate Addressing since the
value to be loaded is whatever is found at thegiven address--which may be variable.FFh refers to SFRs, how can I access the upper128 bytes of Internal RAM that are available on the8052?" The answer is: You can’t access themusing direct addressing. As stated, if you directlyrefer to an address of 80h through FFh you will bereferring to an SFR. However, you may access the8052’s upper 128 bytes of RAM by using the nextaddressing mode, "indirect addressing."

Indirect Addressing

Indirect addressing is a very powerfuladdressing mode which in many cases provides an exceptional level of flexibility. Indirectaddressing is also the only way to access the extra 128 bytes of Internal RAM found on an 8052.Indirect addressing appears as follows:MOV A,@R0This instruction causes the 8051 toanalyze the value of the R0 register. The 8051 willthen load the accumulator with the value fromInternal RAM which is found at the addressindicated by R0.

External Direct

External Direct addressing mode:
MOVX A,@DPTR
MOVX @DPTR,A
As you can see, both commands utilizeDPTR. In these instructions, DPTR must first beloaded with the address of external memory thatyou wish to read or write. Once DPTR holds thecorrect external memory address, the firstcommand will move the contents of that externalmemory address into the Accumulator. Thesecond command will do the opposite: it will allowyou to write the value of the Accumulator to theexternal memory address pointed to by DPTR.

External Indirect

External memory can also be accessedusing a form of indirect addressing which I call External Indirect addressing. This form ofaddressing is usually only used in relatively smallprojects that have a very small amount of externalRAM. An example of this addressing mode is:
MOVX @R0,AOnce again, the value of R0 is first readand the value of the Accumulator is written to that
address in External RAM. Since the value of @R0can only be 00h through FFh the project would
effectively be limited to 256 bytes of ExternalRAM. There are relatively simplehardware/software tricks that can be implementedto access more than 256 bytes of memory using
External Indirect addressing; however, it is usuallyeasier to use External Direct addressing if yourproject has more than 256 bytes of External RAM.
"returns" from subroutines. Additionally, interrupts,when enabled, can cause the program flow todeviate from it’s otherwise sequential scheme.

Conditional Branching

The 8051 contains a suite of instructionswhich, as a group, are referred to as "conditionalbranching" instructions. These instructions causeprogram execution to follow a non-sequential pathif a certain condition is true.Take, for example, the JB instruction. Thisinstruction means "Jump if Bit Set." An example ofthe JB instruction might be:

```
JB 45h,HELLO
NOP
HELLO:
```

In this case, the 8051 will analyze thecontents of bit 45h. If the bit is set program execution will jump immediately to the labelHELLO, skipping the NOP instruction. If the bit is
not set the conditional branch fails and programexecution continues, as usual, with the NOP instruction which follows.Conditional branching is really thefundamental building block of program logic sinceall "decisions" are accomplished by using
conditional branching. Conditional branching canbe thought of as the "IF...THEN" structure in 8051
assembly language.
An important note worth mentioning aboutconditional branching is that the program may only
branch to instructions located withim 128 bytesprior to or 127 bytes following the address which
follows the conditional branch instruction. Thismeans that in the above example the label HELLO
must be within +/- 128 bytes of the memoryaddress which contains the conditional branching
instruction.

Direct Jumps

While conditional branching is extremelyimportant, it is often necessary to make a direct call to a given memory location without basing iton a given logical decision. This is equivalent tosaying "Goto" in BASIC. In this case you want theprogram flow to continue at a given memory
address without considering any conditions.This is accomplished in the 8051 using"Direct Jump and Call" instructions. As illustratedin the last paragraph, this suite of instructionscauses program flow to change unconditionally.Consider the example:

```
LJMP NEW_ADDRESS
...
NEW_ADDRESS:
```

The LJMP instruction in this examplemeans "Long Jump." When the 8051 executes thisinstruction the PC is loaded with the address ofNEW_ADDRESS and program executioncontinues sequentially from there.The obvious difference between the DirectJump and Call instructions and the conditionalbranching is that with Direct Jumps and Callsprogram flow always changes. With conditionalbranching program flow only changes if a certaincondition is true.
It is worth mentioning that, aside fromLJMP, there are two other instructions whichcause a direct jump to occur: the SJMP and AJMPcommands. Functionally, these two commandsperform the exact same function as the LJMPcommand--that is to say, they always causeprogram flow to continue at the address indicatedby the command. However, SJMP and AJMP differin the following ways:

- The SJMP command, like the conditionalbranching instructions, can only jump to anaddress within +/- 128 bytes of the SJMPcommand.
- The AJMP command can only jump to anaddress that is in the same 2k block ofmemory as the AJMP command. That is tosay, if the AJMP command is at code memory
location 650h, it can only do a jump toaddresses 0000h through 07FFh (0 through
2047, decimal).e: The LJMP command requiresthree bytes of code memory whereas both the
SJMP and AJMP commands require only two.

Direct Calls

.
When the 8051 executes an LCALLinstruction it immediately pushes the current Program Counter onto the stack and thencontinues executing code at the address indicatedby the LCALL instruction.

Returns from Routines

Another structure that can cause programflow to change is the "Return from Subroutine" instruction, known as RET in 8051 AssemblyLanguage.

The RET instruction, when executed,returns to the address following the instruction thatcalled the given subroutine. More accurately, itreturns to the address that is stored on the stack.

Interrupts

An interrupt is a special feature whichallows the 8051 to provide the illusion of "multitasking,"although in reality the 8051 is only doingone thing at a time. The word "interrupt" can oftenbe subsituted with the word "event."An interrupt is triggered whenever acorresponding event occurs. When the event
occurs, the 8051 temporarily puts "on hold" thenormal execution of the program and executes a
special section of code referred to as an interrupthandler. The interrupt handler performs whateverspecial functions are required to handle the eventand then returns control to the 8051 at which pointprogram execution continues as if it had neverbeen interrupted.

8051 Timers

The 8051 comes equipped with twotimers, both of which may be controlled, set, read,and configured individually. The 8051 timers havethree general functions:

1) Keeping time and/or calculating the amount oftime between events,

2) Counting the events themselves, or3) Generating baud rates for the serial port.

How does a timer count?

How does a timer count? The answer to
this question is very simple: A timer always countsup. It doesn't matter whether the timer is being
used as a timer, a counter, or a baud rategenerator: A timer is always incremented by themicrocontroller.

USING TIMERS TO MEASURE TIME

Obviously, one of the primary uses oftimers is to measure time.

How long does a timer take to count?First, it's worth mentioning that when a
timer is in interval timer mode (as opposed toevent counter mode) and correctly configured, itwill increment by 1 every machine cycle. As youwill recall from the previous chapter, a single
machine cycle consists of 12 crystal pulses. Thusa running timer will be incremented:11,059,000 / 12 = 921,583 times per second. Unlikeinstructions--some of which require 1 machinecycle, others 2, and others 4--the timers areconsistent: They will always be incremented once
per machine cycle. Thus if a timer has countedfrom 0 to 50,000 you may calculate: 50,000 / 921,583 = .0542.0542 seconds have passed. In plainEnglish, about half of a tenth of a second, or onetwentiethof a second.Obviously it's not very useful to know.0542 seconds have passed. If you want toexecute an event once per second you'd have towait for the timer to count from 0 to 50,000 18.45times.

Let's say we want to know how manytimes the timer will be incremented in .05 seconds.We can do simple multiplication: .05 * 921,583 =
46,079.15.This tells us that it will take .05 seconds(1/20th of a second) to count from 0 to 46,079.

Actually, it will take it .049999837 seconds--sowe're off by .000000163 seconds--however, that's
close enough for government work. Consider thatif you were building a watch based on the 8051and made the above assumption your watch wouldonly gain about one second every 2 months.Again, I think that's accurate enough for mostapplications--I wish my watch only gained onesecond every two months!Obviously, this is a little more useful. If youknow it takes 1/20th of a second to count from 0 to
46,079 and you want to execute some event everysecond you simply wait for the timer to count from0 to 46,079 twenty times; then you execute yourevent, reset the timers, and wait for the timer tocount up another 20 times. In this manner you willeffectively execute your event once per second,accurate to within thousandths of a second.Thus, we now have a system with which tomeasure time. All we need to review is how tocontrol the timers and initialize them to provide uswith the information we need.

The TMOD SFR

TMOD (Timer Mode). The TMOD SFR is used tocontrol the mode of operation of both timers. Each
bit of the SFR gives the microcontroller specificinformation concerning how to run a timer. Thehigh four bits (bits 4 through 7) relate to Timer 1whereas the low four bits (bits 0 through 3)perform the exact same functions, but for timer 0.The individual bits of TMOD have thefollowing functions:

The modes of operation are:

TxM1	TxM0	Timer Mode	Description of Mode
0	0	0	13-bit Timer.
0	1	1	16-bit Timer
1	0	2	8-bit auto-reload
1	1	3	Split timer mode

13-bit Time Mode (mode 0)

Timer mode "0" is a 13-bit timer. This is arelic that was kept around in the 8051 to maintain compatability with it's predecesor, the 8048.Generally the 13-bit timer mode is not used in new
development.When the timer is in 13-bit mode, TLx willcount from 0 to 31. When TLx is incremented from31, it will "reset" to 0 and increment THx. Thus,effectively, only 13 bits of the two timer bytes are
being used: bits 0-4 of TLx and bits 0-7 of THx.
This also means, in essence, the timer can onlycontain 8192 values. If you set a 13-bit timer to 0,
it will overflow back to zero 8192 machine cycleslater.

16-bit Time Mode mode 1)

Timer mode "1" is a 16-bit timer. This is avery commonly used mode. It functions just like 13-bit mode except that all 16 bits are used. TLxis incremented from 0 to 255. When TLx isincremented from 255, it resets to 0 and causesTHx to be incremented by 1. Since this is a full 16-bit timer, the timer may contain up to 65536distinct values. If you set a 16-bit timer to 0, it will
overflow back to 0 after 65,536 machine cycles.

8-bit Time Mode (mode 2)

Timer mode "2" is an 8-bit auto-reloadmode. What is that, you may ask? Simple. When atimer is in mode 2, THx holds the "reload value"and TLx is the timer itself. Thus, TLx startscounting up. When TLx reaches 255 and issubsequently incremented, instead of resetting to0 (as in the case of modes 0 and 1), it will be resetto the value stored in THx.For example, let's say TH0 holds the value
FDh and TL0 holds the value FEh. If we were towatch the values of TH0 and TL0 for a fewmachine cycles this is what we'd see:

Machine Cycle	TH0 Value	TL0 Value
1	FDh	FEh
2	FDh	FFh
3	FDh	FDh
4	FDh	FEh
5	FDh	FFh
6	FDh	FDh
7	FDh	FEh

As you can see, the value of TH0 neverchanged. In fact, when you use mode 2 you almost always set THx to a known value and TLxis the SFR that is constantly incremented.What's the benefit of auto-reload mode?Perhaps you want the timer to always have a
value from 200 to 255. If you use mode 0 or 1,you'd have to check in code to see if the timer had
overflowed and, if so, reset the timer to 200. Thistakes precious instructions of execution time to
check the value and/or to reload it. When you use
mode 2 the microcontroller takes care of this for
you. Once you've configured a timer in mode 2you don't have to worry about checking to see ifthe timer has overflowed nor do you have to worryabout resetting the value--the microcontroller
hardware will do it all for you.The auto-reload mode is very commonlyused for establishing a baud rate which we will talkmore about in the Serial Communications chapter.

Split Timer Mode (mode 3)

Timer mode "3" is a split-timer mode.When Timer 0 is placed in mode 3, it essentiallybecomes two separate 8-bit timers. That is to say,Timer 0 is TL0 and Timer 1 is TH0. Both timerscount from 0 to 255 and overflow back to 0. All thebits that are related to Timer 1 will now be tied to
TH0.While Timer 0 is in split mode, the realTimer 1 (i.e. TH1 and TL1) can be put into modes

0, 1 or 2 normally--however, you may not start orstop the real timer 1 since the bits that do that are
now linked to TH0. The real timer 1, in this case,will be incremented every machine cycle no matter
what.The only real use I can see of using splittimer mode is if you need to have two separatetimers and, additionally, a baud rate generator. Insuch case you can use the real Timer 1 as a baudrate generator and use TH0/TL0 as two separatetimers.

Reading the Timer

There are two common ways of readingthe value of a 16-bit timer; which you use dependson your specific application. You may either readthe actual value of the timer as a 16-bit number, oryou may simply detect when the timer haoverflowed.

Reading the value of a Timer

If your timer is in an 8-bit mode--that is,either 8-bit AutoReload mode or in split timer mode--then reading the value of the timer issimple. You simply read the 1-byte value of the timer and you’re done.However, if you’re dealing with a 13-bit or16-bit timer the chore is a little more complicated.Consider what would happen if you read the lowbyte of the timer as 255, then read the high byte ofthe timer as 15. In this case, what actuallyhappened was that the timer value was 14/255
(high byte 14, low byte 255) but you read 15/255.Why? Because you read the low byte as 255. Butwhen you executed the next instruction a smallamount of time passed--but enough for the timer to
increment again at which time the value rolled overfrom 14/255 to 15/0. But in the process you’ve
read the timer as being 15/255. Obviously there’sa problem there.
The solution? It’s not too tricky, really. Youread the high byte of the timer, then read the low byte, then read the high byte again. If the high byteread the second time is not the same as the high
byte read the first time you repeat the cycle. Incode, this would appear as:

```
REPEAT: MOV A,TH0
MOV R0,TL0
CJNE A,TH0,REPEAT
```

In this case, we load the accumulator withthe high byte of Timer 0. We then load R0 with thelow byte of Timer 0. Finally, we check to see if thehigh byte we read out of Timer 0--which is now
stored in the Accumulator--is the same as thecurrent Timer 0 high byte. If it isn’t it means we’vejust "rolled over" and must reread the timer’svalue--which we do by going back to REPEAT.When the loop exits we will have the low byte ofthe timer in R0 and the high byte in theAccumulator.
Another much simpler alternative is tosimply turn off the timer run bit (i.e. CLR TR0),read the timer value, and then turn on the timerrun bit (i.e. SETB TR0). In that case, the timer isn’trunning so no special tricks are necessary. Ofcourse, this implies that your timer will be stoppedfor a few machine cycles. Whether or not this istolerable depends on your specific application.

Detecting Timer Overflow

Often it is necessary to just know that thetimer has reset to 0. That is to say, you are notparticularly interest in the value of the timer butrather you are interested in knowing when thetimer has overflowed back to 0.Whenever a timer *overflows* from it’shighest value back to 0, the microcontrollerautomatically sets the TFx bit in the TCONregister. This is useful since rather than checkingthe exact value of the timer you can just check ifthe TFx bit is set. If TF0 is set it means that timer 0has overflowed; if TF1 is set it means that timer 1
has overflowed.We can use this approach to cause theprogram to execute a fixed delay. As you’ll recall,we calculated earlier that it takes the 8051 1/20thof a second to count from 0 to 46,079. However,the TFx flag is set when the timer overflows backto 0. Thus, if we want to use the TFx flag toindicate when 1/20th of a second has passed wemust set the timer initially to 65536 less 46079, or
19,457. If we set the timer to 19,457, 1/20th of asecond later the timer will overflow. Thus we come
up with the following code to execute a pause of
1/20th of a second:

```
MOV TH0,#76 ;High byte of 19,457 (76 * 256 = 19,456)
MOV TL0,#01 ;Low byte of 19,457 (19,456 + 1 = 19,457)
MOV TMOD,#01 ;Put Timer 0 in 16-bit mode
SETB TR0 ;Make Timer 0 start counting
JNB TF0,$ ;If TF0 is not set, jump back to this same instruction
```

In the above code the first two linesinitialize the Timer 0 starting value to 19,457. The

next two instructions configure timer 0 and turn it
on. Finally, the last instruction **JNB TF0,$**, reads
"Jump, if TF0 is not set, back to this sameinstruction." The "$" operand means, in most assemblers, the address of the current instruction.Thus as long as the timer has not overflowed andthe TF0 bit has not been set the program will keepexecuting this same instruction. After 1/20th of asecond timer 0 will overflow, set the TF0 bit, and
program execution will then break out of the loop.

8051 Serial Communication

One of the 8051's many powerful features
is it's integrated *UART*, otherwise known as aserial port. The fact that the 8051 has an integrated serial port means that you may veryeasily read and write values to the serial port. If itwere not for the integrated serial port, writing abyte to a serial line would be a rather tediousprocess requring turning on and off one of the I/Olines in rapid succession to properly "clock out"
each individual bit, including start bits, stop bits,and parity bits.

Setting the Serial Port Mode

The first thing we must do when using the8051's integrated serial port is, obviously, configure it. This lets us tell the 8051 how manydata bits we want, the baud rate we will be using,
and how the baud rate will be determined.First, let's present the "Serial Control"
(SCON) SFR and define what each bit of the SFRrepresents:

Bit	**Name**	**Bit Addres**	**Explanation of Function**
7	SM0	9Fh	Serial port mode bit 0
6	SM1	9Eh	Serial port mode bit 1.
5	SM2	9Dh	Mutliprocessor Communications Enable (explained later)
4	REN	9Ch	Receiver Enable. This bit must be set in order to receive characters.
3	TB8	9Bh	Transmit bit 8. The 9th bit to transmit in mode 2 and 3.
2	RB8	9Ah	Receive bit 8. The 9th bit received in mode 2 and 3.
1	TI	99h	Transmit Flag. Set when a byte has been completely transmitted.
0	RI	98h	Receive Flag. Set when a byte has been completely received.

Additionally, it is necessary to define the function of SM0 and SM1 by an additional table:

SM0	**SM1**	**Serial Mode**	**Explanation**	**Baud Rate**
0	0	0	8-bit Shift Register	Oscillator / 12
0	1	1	8-bit UART	Set by Timer 1 (*)
1	0	2	9-bit UART	Oscillator / 32 (*)
1	1	3	9-bit UART	Set by Timer 1 (*)

Note: *The baud rate indicated in this table is
doubled if PCON.7 (SMOD) is set.*

The SCON SFR allows us to configure the
Serial Port. Thus, we'll go through each bit and
review it's function. The first four bits (bits 4
through 7) are configuration bits.

Bits **SM0** and **SM1** let us set the *serialmode* to a value between 0 and 3, inclusive. The four modes are defined in the chart immediatelyabove. As you can see, selecting the Serial Mode
selects the mode of operation (8-bit/9-bit, UART orShift Register) and also determines how the baudrate will be calculated. In modes 0 and 2 the baudrate is fixed based on the oscillator's frequency. Inmodes 1 and 3 the baud rate is variable based on
how often Timer 1 overflows. We'll talk more aboutthe various Serial Modes in a moment.
The next bit, **SM2**, is a flag for"Multiprocessor communication." Generally,
whenever a byte has been received the 8051 willset the "RI" (Receive Interrupt) flag. This lets the
program know that a byte has been received andthat it needs to be processed. However, when
SM2 is set the "RI" flag will only be triggered if the
9th bit received was a "1". That is to say, if SM2 isset and a byte is received whose 9th bit is clear,
the RI flag will never be set. This can be useful incertain advanced serial applications. For now it is
safe to say that you will almost always want toclear this bit so that the flag is set upon reception

of *any* character.The next bit, **REN**, is "Receiver Enable."
This bit is very straightforward: If you want to
receive data via the serial port, set this bit. You willalmost always want to set this bit.
The last four bits (bits 0 through 3) areoperational bits. They are used when actually
sending and receiving data--they are not used toconfigure the serial port.
The **TB8** bit is used in modes 2 and 3. Inmodes 2 and 3, a total of nine data bits are
transmitted. The first 8 data bits are the 8 bits ofthe main value, and the ninth bit is taken
from
TB8. If TB8 is set and a value is written to thserial port, the data's bits will be written to the
serial line followed by a "set" ninth bit. If TB8 isclear the ninth bit will be "clear."
The **RB8** also operates in modes 2 and 3and functions essentially the same way as TB8,
but on the reception side. When a byte is receivedin modes 2 or 3, a total of nine bits are
received. In
this case, the first eight bits received are the dataof the serial byte received and the value of
the
ninth bit received will be placed in RB8.**TI** means "Transmit Interrupt." When a
program writes a value to the serial port, a certainamount of time will pass before the
individual bits
of the byte are "clocked out" the serial port. If theprogram were to write another byte to the
serialport before the first byte was completely output,the data being sent would be garbled.
Thus, the8051 lets the program know that it has "clockedout" the last byte by setting the TI
bit. When the TI
bit is set, the program may assume that the serial
port is "free" and ready to send the next byte.Finally, the **RI** bit means "Receive
Interrupt." It funcions similarly to the "TI" bit, but itindicates that a byte has been received.
That is to
say, whenever the 8051 has received a complete
byte it will trigger the RI bit to let the program knowthat it needs to read the value quickly,
before
another byte is read.

Setting the Serial Port Baud Rate

Once the Serial Port Mode has beenconfigured, as explained above, the program must
configure the serial port's baud rate. This onlyapplies to Serial Port modes 1 and 3. The
Baud
Rate is determined based on the oscillator'sfrequency when in mode 0 and 2. In mode 0,
the
baud rate is always the oscillator frequencydivided by 12. This means if you're crystal is
11.059Mhz, mode 0 baud rate will always be921,583 baud. In mode 2 the baud rate is
alwaysthe oscillator frequency divided by 64, so a
11.059Mhz crystal speed will yield a baud rate of172,797.
In modes 1 and 3, the baud rate idetermined by how frequently timer 1 overflows.
The more frequently timer 1 overflows, the higherthe baud rate. There are many ways one
can
cause timer 1 to overflow at a rate that determinesa baud rate, but the most common
method is to
put timer 1 in 8-bit auto-reload mode (timer mode2) and set a reload value (TH1) that
causes Timer
1 to overflow at a frequency appropriate togenerate a baud rate.
To determine the value that must be
placed in TH1 to generate a given baud rate, we
may use the following equation (assuming
PCON.7 is clear).

TH1 = 256 - ((Crystal / 384) / Baud)

If PCON.7 is set then the baud rate is
effectively doubled, thus the equation becomes:

TH1 = 256 - ((Crystal / 192) / Baud)

For example, if we have an 11.059Mhz
crystal and we want to configure the serial port to
19,200 baud we try plugging it in the first equation:

TH1 = 256 - ((Crystal / 384) / Baud)
TH1 = 256 - ((11059000 / 384) / 19200)
TH1 = 256 - ((28,799) / 19200)
TH1 = 256 - 1.5 = 254.5

As you can see, to obtain 19,200 baud on

a 11.059Mhz crystal we'd have to set TH1 to
254.5. If we set it to 254 we will have achieved
14,400 baud and if we set it to 255 we will have
achieved 28,800 baud. Thus we're stuck...
But not quite... to achieve 19,200 baud we
simply need to set PCON.7 (SMOD). When we do
this we double the baud rate and utilize the second
equation mentioned above. Thus we have:

```
TH1 = 256 - ((Crystal / 192) / Baud)
TH1 = 256 - ((11059000 / 192) / 19200)
TH1 = 256 - ((57699) / 19200)
TH1 = 256 - 3 = 253
```

Here we are able to calculate a nice, even
TH1 value. Therefore, to obtain 19,200 baud with
an 11.059MHz crystal we must:
1) Configure Serial Port mode 1 or 3.
2) Configure Timer 1 to timer mode 2 (8-bit autoreload).
3) Set TH1 to 253 to reflect the correct frequency
for 19,200 baud.
4) Set PCON.7 (SMOD) to double the baud rate.

Writing to the Serial Port

Once the Serial Port has been propertlyconfigured as explained above, the serial port is ready to be used to send data and receive data. Ifyou thought that configuring the serial port was
simple, using the serial port will be a breeze.To write a byte to the serial port one must simply write the value to the **SBUF** (99h) SFR. Forexample, if you wanted to send the letter "A" to the
serial port, it could be accomplished as easily as:

```
MOV SBUF,#'A'
```

Upon execution of the above instructionthe 8051 will begin transmitting the character via the serial port. Obviously transmission is notinstantaneous--it takes a measureable amount of
time to transmit. And since the 8051 does not havea serial output buffer we need to be sure that a
character is completely transmitted before we tryto transmit the next character.
The 8051 lets us know when it is donetransmitting a character by setting the **TI** bit inSCON. When this bit is set we know that the lastcharacter has been transmitted and that we maysend the next character, if any. Consider thefollowing code segment:

```
CLR TI ;Be sure the bit is initially clear
MOV SBUF,#'A' ;Send the letter 'A' to the serial
port
JNB TI,$ ;Pause until the RI bit is set.
```

The above three instructions willsuccessfully transmit a character and wait for theTI bit to be set before continuing. The lastinstruction says "Jump if the TI bit is not set to $"--
$, in most assemblers, means "the same addressof the current instruction." Thus the 8051 will
pause on the JNB instruction until the TI bit is setby the 8051 upon successful transmission of the
character.

Reading the Serial Port

Reading data received by the serial port isequally easy. To read a byte from the serial port one just needs to read the value stored in the**SBUF** (99h) SFR after the 8051 has automatically
set the **RI** flag in SCON.
For example, if your program wants to waitfor a character to be received and subsequentlyread it into the Accumulator, the following codesegment may be used:

```
JNB RI,$ ;Wait for the 8051 to set the RI flag
MOV A,SBUF ;Read the character from the serial port
```

The first line of the above code segmentwaits for the 8051 to set the RI flag; again, the8051 sets the RI flag automatically when itreceives a character via the serial port. So as longas the bit is not set the program repeats the "JNB"instruction continuously.Once the RI bit is set upon characterreception the above condition automatically failsand program flow falls through to the "MOV"instruction which reads the value.

8051 Tutorial: Interrupts

However, interrupts give us amechanism to "put on hold" the normal programflow, execute a subroutine, and then resumenormal program flow as if we had never left it. Thissubroutine, called an interrupt handler, is onlyexecuted when a certain event (interrupt) occurs.The event may be one of the timers "overflowing,"receiving a character via the serial port,transmitting a character via the serial port, or oneof two "external events." The 8051 may beconfigured so that when any of these events occurthe main program is temporarily suspended andcontrol passed to a special section of code whichpresumably would execute some function relatedto the event that occured. Once complete, controlwould be returned to the original program. Themain program never even knows it wasinterrupted.The ability to interrupt normal programexecution when certain events occur makes it
much easier and much more efficient to handlecertain conditions. If it were not for interrupts we
would have to manually check in our mainprogram whether the timers had overflown,whether we had received another character via theserial port, or if some external event had occured.

What Events Can Trigger Interrupts, and where do they go?

We can configure the 8051 so that any ofthe following events will cause an interrupt:

- Timer 0 Overflow.
- Timer 1 Overflow.
- Reception/Transmission of Serial
Character.
- External Event 0.
- External Event 1.

In other words, we can configure the 8051so that when Timer 0 Overflows or when a character is sent/received, the appropriateinterrupt handler routines are called. Obviously we need to be able todistinguish between various interrupts andexecuting different code depending on whatinterrupt was triggered. This is accomplished by jumping to a fixed address when a given interruptoccurs.

Interrupt	Flag	Interrupt Handler Address
External 0	IE0	0003h
Timer 0	TF0	000Bh
External 1	IE1	0013h
Timer 1	TF1	001Bh
Serial	RI/TI	0023h

By consulting the above chart we see thatwhenever Timer 0 overflows (i.e., the TF0 bit is set), the main program will be temporarilysuspended and control will jump to 000BH. It is assumed that we have code at address 0003Hthat handles the situation of Timer 0 overflowing.

Setting Up Interrupts

By default at powerup, all interrupts aredisabled. This means that even if, for example, the TF0 bit is set, the 8051 will not execute theinterrupt. Your program must specifically tell the 8051 that it wishes to enable interrupts andspecifically which interrupts it wishes to enable. Your program may enable and disableinterrupts by modifying the IE SFR (A8h):

Bit	Name	Bit Address	Explanation of Function
7	EA	AFh	Global Interrupt Enable/Disable
6	-	AEh	Undefined
5	-	ADh	Undefined
4	ES	ACh	Enable Serial Interrupt
3	ET1	ABh	Enable Timer 1 Interrupt
2	EX1	AAh	Enable External 1 Interrupt
1	ET0	A9h	Enable Timer 0 Interrupt
0	EX0	A8h	Enable External 0 Interrupt

As you can see, each of the 8051'sinterrupts has its own bit in the IE SFR. You enable a given interrupt by setting thecorresponding bit. For example, if you wish to enable Timer 1 Interrupt, you would executeeither:

```
MOV IE,#08h || SETB ET1
```

Both of the above instructions set bit 3 ofIE, thus enabling Timer 1 Interrupt. Once Timer 1Interrupt is enabled, whenever the TF1 bit is set,the 8051 will automatically put "on hold" the mainprogram and execute the Timer 1 InterruptHandler at address 001Bh.However, before Timer 1 Interrupt (or anyother interrupt) is truly enabled, you must also setbit 7 of IE. Bit 7, the Global Interupt
Enable/Disable, enables or disables all interruptssimultaneously. That is to say, if bit 7 is clearedthen no interrupts will occur, even if all the otherbits of IE are set. Setting bit 7 will

enable all theinterrupts that have been selected by setting otherbits in IE. This is useful in program execution if youhave time-critical code that needs to execute. Inthis case, you may need the code to execute fromstart to finish without any interrupt getting in theway. To accomplish this you can simply clear bit 7
of IE (CLR EA) and then set it after your timecriticialcode is done.
So, to sum up what has been stated in this
section, to enable the Timer 1 Interrupt the most
common approach is to execute the following two
instructions:

```
SETB ET1
SETB EA
```

Thereafter, the Timer 1 Interrupt Handlerat 01Bh will automatically be called whenever the TF1 bit is set (upon Timer 1 overflow).

Polling Sequence

The 8051 automatically evaluates whether
an interrupt should occur after every instruction.
When checking for interrupt conditions, it checks
them in the following order:

1) External 0 Interrupt
2) Timer 0 Interrupt
3) External 1 Interrupt
4) Timer 1 Interrupt
5) Serial Interrupt

Interrupt Priorities

The 8051 offers two levels of interruptpriority: high and low. By using interrupt prioritiesyou may assign higher priority to certain interrupt
conditions.For example, you may have enabledTimer 1 Interrupt which is automatically called
every time Timer 1 overflows. Additionally, youmay have enabled the Serial Interrupt which is
called every time a character is received via theserial port. However, you may consider thatreceiving a character is much more important thanthe timer interrupt. In this case, if Timer 1 Interruptis already executing you may wish that the serialinterrupt itself interrupts the Timer 1 Interrupt.When the serial interrupt is complete, controlpasses back to Timer 1 Interrupt and finally backto the main program. You may accomplish this byassigning a high priority to the Serial Interrupt anda low priority to the Timer 1 Interrupt.Interrupt priorities are controlled by the **IP**SFR (B8h). The IP SFR has the following format:

Bit	**Name**	**Bit Address**	**Explanation of Function**
7	-	-	Undefined
6	-	-	Undefined
5	-	-	Undefined
4	PS	BCh	Serial Interrupt Priority
3	PT1	BBh	Timer 1 Interrupt Priority
2	PX1	BAh	External 1 Interrupt Priority
1	PT0	B9h	Timer 0 Interrupt Priority
0	PX0	B8h	External 0 Interrupt Priority

When considering interrupt priorities, the
following rules apply:

- Nothing can interrupt a high-priority interrupt--not even another high priority interrupt.
- A high-priority interrupt may interrupt a lowpriorityinterrupt.
- A low-priority interrupt may only occur if noother interrupt is already executing.
- If two interrupts occur at the same time, theinterrupt with higher priority will execute first. Ifboth interrupts are of the same priority theinterrupt which is serviced first by pollingsequence will beexecuted first.

What Happens When an Interrupt Occurs?

When an interrupt is triggered, the
following actions are taken automatically by the
microcontroller:

- The current Program Counter is saved on thestack, low-byte first.
- Interrupts of the same and lower priority areblocked.
- In the case of Timer and External interrupts,the corresponding interrupt flag is set.
- Program execution transfers to the corresponding interrupt handler vector address.

 The Interrupt Handler Routine executes.Take special note of the third step: If theinterrupt being handled is a Timer or Externalinterrupt, the microcontroller automatically clearsthe interrupt flag before passing control to yourinterrupt handler routine.

What Happens When an Interrupt Ends?

An interrupt ends when your programexecutes the RETI instruction. When the RETIinstruction is executed the following actions aretaken by the microcontroller:

 Two bytes are popped off the stack into theProgram Counter to restore normal programexecution.

 Interrupt status is restored to its pre-interruptstatus.

Serial Interrupts

Serial Interrupts are slightly different thanthe rest of the interrupts. This is due to the factthat there are two interrupt flags: RI and TI. Ifeither flag is set, a serial interrupt is triggered. Asyou will recall from the section on the serial port,the RI bit is set when a byte is received by theserial port and the TI bit is set when a byte hasbeen sent.This means that when your serial interruptis executed, it may have been triggered becausethe RI flag was set or because the TI flag was set--or because both flags were set. Thus, your routinemust check the status of these flags todeterminewhatactionisappropriate.Also,since the 8051does not automatically clear the RI and TIflags
you must clear these bits in your interrupt handler.

```
INT_SERIAL: JNB RI,CHECK_TI ;If the RI flag is not set, we jump to check TI
MOV A,SBUF ;If we got to this line, it's because the RI bit *was* set
CLR RI ;Clear the RI bit after we've processed it
CHECK_TI: JNB TI,EXIT_INT ;If the TI flag is not set, we jump to the exit point
CLR TI ;Clear the TI bit before we send another character
MOV SBUF,#'A' ;Send another character to the serial port
EXIT_INT: RETI
```

As you can see, our code checks thestatus of both interrupts flags. If both flags wereset, both sections of code will be executed. Alsonote that each section of code clears itscorresponding interrupt flag. If you forget to clearthe interrupt bits, the serial interrupt will beexecuted over and over until you clear the bit.Thus it is very important that you always clear theinterrupt flags in a serial interrupt.

Important Interrupt Consideration: Register Protection

One very important rule applies to allinterrupt handlers: Interrupts must leave theprocessor in the same state as it was in when theinterrupt initiated.Remember, the idea behind interrupts isthat the main program isn't aware that they areexecuting in the "background." However, considerthe following code:

```
CLR C ;Clear carry
MOV A,#25h ;Load the accumulator with 25h
ADDC A,#10h ;Add 10h, with carry
```

After the above three instructions areexecuted, the accumulator will contain a value of35h. But what would happen if right after theMOV instruction an interrupt occured. During thisinterrupt, the carry bit was set and the value of theaccumulator was changed to 40h. When theinterrupt finishedand control was passed back tothe main program, the ADDC would add 10h to40h, and additionally add an additional 1h becausethe carry bit is set. In this case, the accumulatorwill contain the value 51h at the end of execution.In this case, the main program hasseemingly calculated the wrong answer. How can25h + 10h yield 51h as a result? It doesn't makesense. A programmer that was unfamiliar withinterrupts would be convinced that themicrocontroller was damaged in some way,
provoking problems with mathematicalcalculations.
What has happened, in reality, is theinterrupt did not protect the registers it used.
Restated: *An interrupt must leave the*
processor in the same state as it was in when theinterrupt initiated.
What does this mean? It means if yourinterrupt uses the accumulator, it must insure thatthe value of the accumulator is the same at theend of the interrupt as it was at the beginning. Thisis generally accomplished with a PUSH and POPsequence. For example:

```
PUSH ACC
PUSH PSW
MOV A,#0FFh
ADD A,#02h
POP PSW
POP ACC
```

The *guts* of the interrupt is the MOVinstruction and the ADD instruction. However,these two instructions modify the Accumulator (theMOV instruction) and also modify the value of thecarry bit (the ADD instruction will cause the carrybit to be set). Since an interrupt routine mustguarantee that the registers remain unchanged bythe routine, the routine pushes the

original valuesonto the stack using the PUSH instruction. It isthen free to use the registers it protected to its
heart’s content. Once the interrupt has finished itstask, it pops the original values back into theregisters. When the interrupt exits, the mainprogram will never know the difference because
the registers are exactly the same as they werebefore the interrupt executed.

In general, your interrupt routine must
protect the following registers:

- PSW
- DPTR (DPH/DPL)
- PSW
- ACC
- B
- Registers R0-R7

Remember that PSW consists of manyindividual bits that are set by various 8051instructions. Unless you are absolutely sure ofwhat you are doing and have a completeunderstanding of what instructions set what bits, itis generally a good idea to *always* protect PSW bypushing and popping it off the stack at thebeginning and end of your interrupts.Note also that most assemblers (in fact,ALL assemblers that I know of) will not allow youto execute the instruction:

```
PUSH R0
```

This is due to the fact that depending onwhich register bank is selected, R0 may refer toeither internal ram address 00h, 08h, 10h, or 18h.R0, in and of itself, is not a valid memory addressthat the PUSH and POP instructions can use.Thus, if you are using any "R" register inyour interrupt routine, you will have to push thatregister’s absolute address onto the stack instead
of just saying **PUSH R0**. For example, instead of
PUSH R0 you would execute:

```
PUSH 00h
```

Of course, this only works if you’ve
selected the default register set. If you are using
an alternate register set, you must PUSH the
address which corresponds to the register you are
using.

Common Problems with Interrupts

Interrupts are a very powerful toolavailable to the 8051 developer, but when usedincorrectly they can be a source of a huge numberof debugging hours. Errors in interrupt routines are often very difficult to diagnose and correct.If you are using interrupts and your
program is crashing or does not seem to beperforming as you would expect, always reviewthe following interrupt-related issues:

- **Register Protection**: Make sure you areprotecting all your registers, as explainedabove. If you forget to protect a register thatyour main program is using, very strangeresults may occur. In our example above wesaw how failure to protect registers caused themain program to apparently calculate that 25h+ 10h = 51h. If you witness problems withregisters changing values unexpectedly oroperations producing "incorrect" values, it isvery likely that you’ve forgotten to protectregisters. *ALWAYS PROTECT YOURREGISTERS.*
- **Forgetting to restore protected values**:
Another common error is to push registersonto the stack to protect them, and then forgetto pop them off the stack before exiting theinterrupt. For example, you may push ACC, B,and PSW onto the stack in order to protectthem and subsequently pop only ACC andPSW off the stack before exiting. In this case,since you forgot to restore the value of "B", anextra value remains on the stack. When youexecute the RETI instruction the 8051 will usethat value as the return address instead of thecorrect value. In this case, your program willalmost certainly crash. ALWAYS MAKE SUREYOU POP THE SAME NUMBER OF VALUES
OFF THE STACK AS YOU PUSHED ONTO
IT.

Using RET instead of RETI: Rememberthat interrupts are always terminated with the RETIinstruction. It is easy to inadvertantly use the RETinstruction instead. However, the RET instruction

will not end your interrupt. Usually, using a RETinstead of a RETI will cause the illusion of yourmain program running normally, but your interruptwill only be executed once. If it appears that yourinterrupt

Ports configuration

Port 0

Dual-purpose port- 1. general purpose I/O Port.
2. multiplexed address & data bus

Port 1

Open drain outputs
Dedicated I/O port – Used solely for interfacing to external devices
Internal pull-up

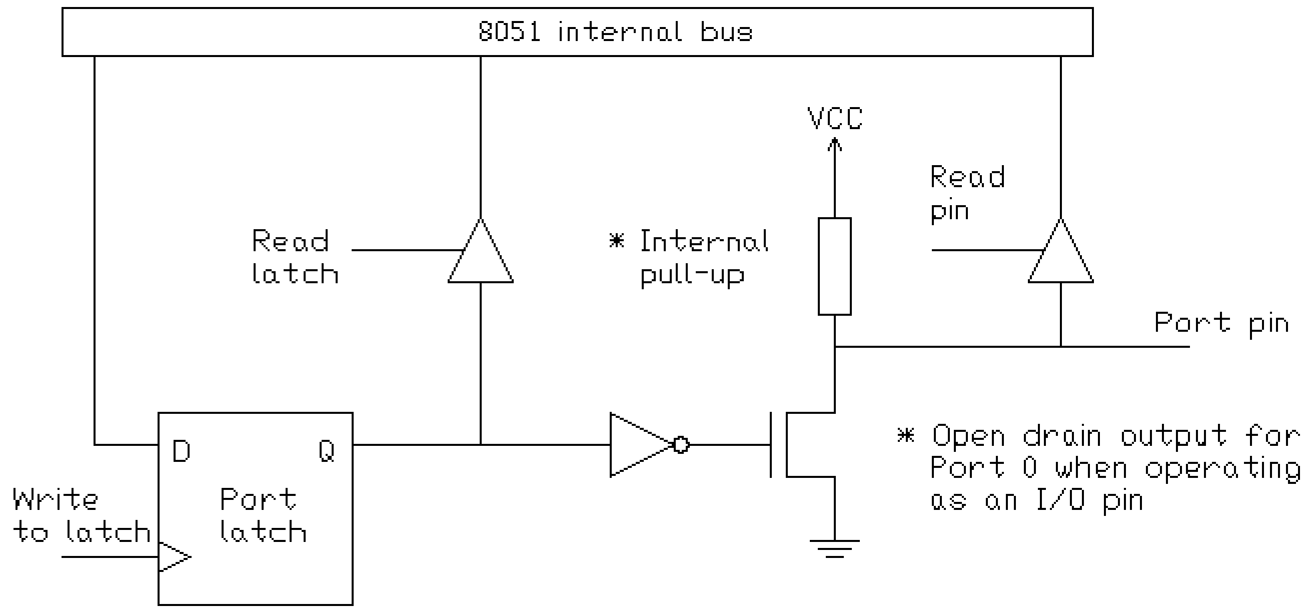
Port 2

Dual-purpose port- 1. general purpose I/O port.
2. a multiplexed address & data bus.
Internal pull-ups

Port 3

Dual-purpose port- 1. general purpose I/O port.
2. pins have alternate purpose related to special features of the 805
Internal pull-ups

The 8051 internal ports are partly bi-directional (Quasi-bi-directional). The following is the internal circuitry for the 8051 port pins:



1.Configuring for output

P0 is open drain.
– Has to be pulled high by external 10K resistors.
– Not needed if P0 is used for address lines

Writing to a port pin loads data into a port latch that drives a FET connected to the port pin.

P0: Note that the pull-up is absent on Port 0 except when functioning as the external address/data bus. When a "0" is written to a bit in port 0, the pin is pulled low. But when a "1" is written to it, it is in high impedance (disconnected) state. So when using port 0 for output, an external pull-up resistor is needed, depending on the input characteristics of the device driven by the port pin

P1, P2, P3 have internal pull-ups: When a "0" is written to a bit in these port , the pin is pulled low (FET-ON) ,also when 1 is written to a bit in these port pin becomes high (FET-OFF) thus using port P1,P2,P3 is simple.

2. Configuring for input

At power-on all are output ports by default
To configure any port for input, write all 1's (0xFF) to the port
Latch bit=1, FET=OFF, Read Pin asserted by read instruction

You can used a port for output any time. But for input, the FET must be off. Otherwise, you will be reading your own latch rather than the signal coming from the outside. Therefore, a "1" should be written to the pin if you want to use it as input, especially when you have used it for output before. If you don't do this input high voltage will get grounded through FET so you will read pin as low and not as high. An external device cannot easily drive it high

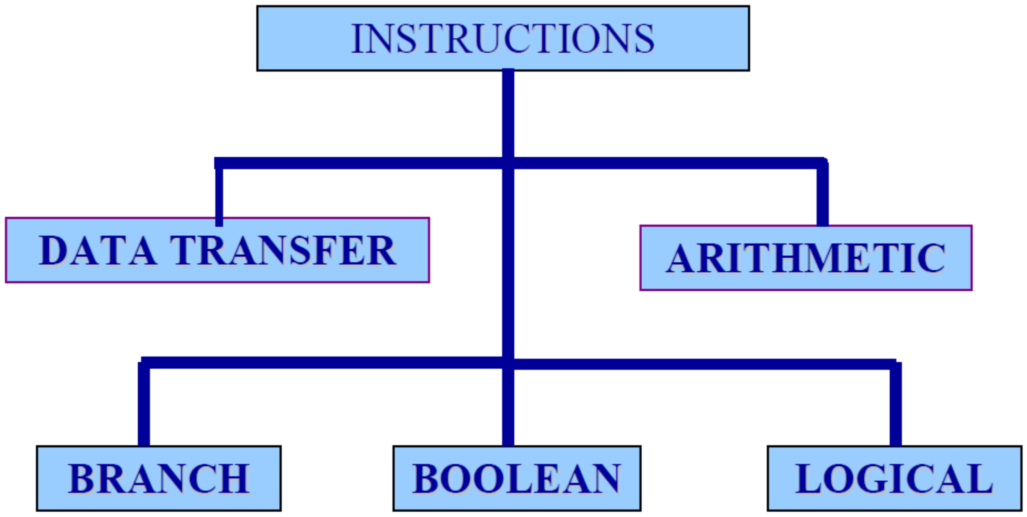
so, you should not tide a port high directly without any resistor. Otherwise, the FET would burn.

Some port pins serve multiple functions. Be careful writing to such ports. For example, P3.0 is the UART RXD (serial input), and P3.1 is the UART TXD (serial output). If you set P3.0 to a '0', an external buffer (such as an RS232 level translator) cannot drive it high. Therefore you have prevented receiving any serial input.

If an external interrupt such as EX1 on P3.3 is enabled, and set to be level sensitive, and you clear this pin's output latch to a zero, guess what? You've just caused a perpetual interrupt 1. The pin's input buffer will read the output of it's latch as always low. Your controller will spend all of its time in the interrupt handler code and will appear to have crashed, since it will have very little time for other tasks. In fact, it will get to execute a single instruction before re-entering the interrupt handler, so the rest of your program will execute very, very slowly.

mysteriously stops executing

Classification of Instructions



Data transfer Instructions

- Mov A, Rn
- Mov A, Direct
- Mov A, @Ri
- Mov A, #Data8
- Mov Dptr, #Data16
- Mov Rn, A
- Mov Rn, Direct
- Mov Rn, #Data**8**
- Mov Direct, A
- Mov Direct, Rn
- Mov Direct, #Data8
- Mov Direct, Direct
- Mov Direct, @Ri
- Mov Direct, # Data**8**
- Mov @Ri, A
- Mov @Ri, Direct
- Mov @Ri, #Data**8**
- Movx A, @Ri
- Movx A, @Dptr
- Movx @Ri, A
- Movx @dptr, A
- Movc A, @A+Dptr
- Movc A, @A+Pc
- Push Direct
- Pop Direct
- Xch A, Rn
- Xch A, Direct

- Xch A, @Ri
- Xchd A, @Ri

Boolean Instructions

- Clr C
- Clr Bit
- Setb C
- Setb Bit
- Cpl C
- Cpl Bit
- Anl C, Bit
- Anl C, /Bit
- Orl C, Bit
- Orl C, /Bit
- Mov C, Bit
- Mov Bit, C

Branch Instructions

- Jc Reladdr
- Jnc Reladdr
- Jb Bit, Reladdr
- Jnb Bit, Reladdr
- Jbc Bit, Reladdr

Arithmetic Instructions

- Add A, Rn
- Add A, Direct
- Add A, @Ri
- Add A, #Data**8**
- Addc A, Rn
- Addc A, Direct
- Addc A, @Ri
- Addc A, #Data**8**
- Subb A, Rn
- Subb A, Direct
- Subb A, @Ri
- Subb A, #Data**8**
- Inc A
- Inc Rn
- Inc Direct
- Inc @Ri
- Inc Dptr
- Dec A
- Dec Rn
- Dec Direct
- Dec @Ri
- Mul AB
- Div AB
- DA A

interrupt-related

Logical Instructions

- Anl A, Rn
- Anl A, Direct
- Anl A, @Ri
- Anl A, #Data**8**
- Anl Direct, A
- Anl Direct, #Data**8**
- Orl A, Rn
- Orl A, Direct
- Orl A, @Ri
- Orl A, #Data**8**
- Orl Direct, A
- Orl Direct, #Data**8**
- Xrl A, Rn
- Xrl A, Direct
- Xrl A, @Ri
- Xrl A, #Data**8**
- Xrl Direct, A
- Xrl Direct, #Data**8**
- Clr A
- Cpl A
- Rl A
- Rlc A
- Rr A
- Rrc A
- Swap A

Branch Instructions

- Acall Addr11
- Lcall Addr16
- Ret
- Reti
- Ajmp Addr11
- Ljmp Addr16
- Sjmp Reladdr
- Jmp @A+Dptr
- Jz Reladdr
- Jnz Reladdr
- Cjne Rn, #Data, Reladdr
- Cjne @Ri, #Data, Reladdr
- Cjne A, #Data, Reladdr
- Cjne A, Direct, Reladdr
- Djnz Rn, Reladdr
- Djnz Direct, Reladdr
- Nop

errors.

UNIT V

The purpose of an RTC or a real time clock is to provide precise time and date which can be used for various applications. RTC is an electronic device in the form of an Integrated Chip (IC) available in various packaging options. It is powered by an internal lithium battery. As a result of which even if the power of the system is turned off, the RTC clock keeps running. They play a very important role in the real time systems like digital clock, attendance system, digital camera etc.

The article presented here shows how RTC can be interfaced with the microcontroller AT89C51. It explores the basic operation of accessing the internal registers and extracting time from the RTC. The time is displayed on the hyper terminal using serial communication. The RTC used here is DS 12C887. 89C51 microcontroller is a very commonly used controller from the family of 8051 series of microcontroller.

. Port P2 is set as data port for LCD to send the data on the LCD while port P0 is set as data port for the RTC DS12C887. This is used to extract the time, date and other information from the RTC. The pins of port P1 and P3 of the controller AT89C51 are used to provide the input signals for setting the time, alarm etc. They also provide control signals for the RTC and the LCD.

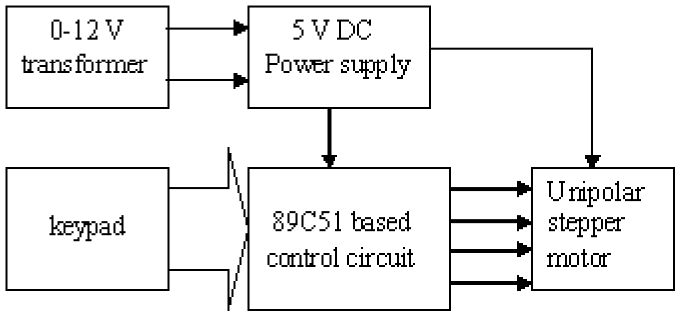
This project is an improved version of Digital clock using RTC DS12C887 and 8051 microcontroller (AT89C51) with time set with only difference that we can set alarm in it. It helps in understanding the concept of setting alarm in the RTC 12C887. The program uses the two external interrupt 0 and 1 for setting the time and alarm respectively. For setting the time the pin P3^2 of the microcontroller 8051 is made low, which is the external interrupt 0. For alarm setting the pin P3^3 of the microcontroller is made low which is the external interrupt 1.

When any of the interrupts occur, the microcontroller stops its current operation and gets ready to set the time or alarm depending on the interrupt activated. The alarm can be set by using the pins digalarm_hr1 and digalarm_min1. The time can be set by using the pins dig_hr1 and dig_min1. When the alarm time matches the clock time, 'Alarm' is displayed on the LCD and the alarmport pin is set high for 10 sec.

This pin can further be connected to a speaker or buzzer to sound the alarm.

General description and system block diagram:-

The complete board consists of transformer, control circuit, keypad and stepper motor as shown in snap. The given figure shows the block diagram of project.



The circuit has inbuilt 5 V power supply so when it is connected with transformer it will give the supply to circuit and motor both. The 8 Key keypad is connected with circuit through which user can give the command to control stepper motor. The control circuit includes micro controller 89C51, indicating LEDs, and current driver chip ULN2003A. One can program the controller to control the operation of stepper motor. He can give different commands through keypad like, run clockwise, run anticlockwise, increase/decrease RPM, increase/decrease revolutions, stop motor, change the mode, etc. before we start with project it is must that we first understood the operation of unipolar stepper motor.

Unipolar stepper motor:-

In the construction of unipolar stepper motor there are four coils. One end of each coil is tide together and it gives common terminal which is always connected with positive terminal of supply. The other ends of each coil are given for interface. Specific color code may also be given. Like in my motor orange is first coil (L1), brown is second (L2), yellow is third (L3), black is fourth (L4) and red for common terminal.

By means of controlling a stepper motor operation we can

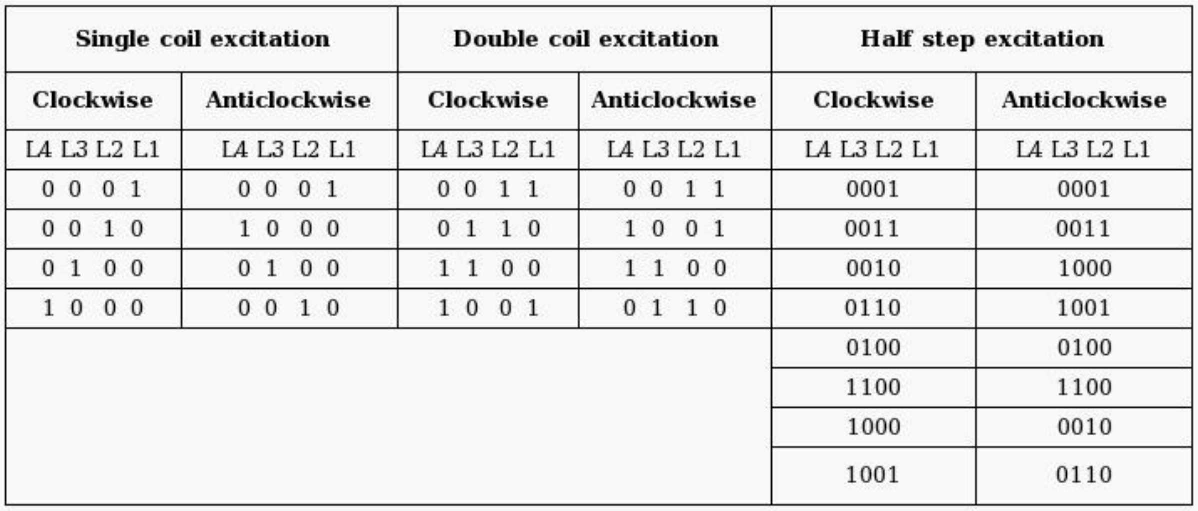
1. Increase or decrease the RPM (speed) of it
2. Increase or decrease number of revolutions of it
3. Change its direction means rotate it clockwise or anticlockwise

To vary the RPM of motor we have to vary the PRF (Pulse Repetition Frequency). Number of applied pulses will vary number of rotations and last to change direction we have to change pulse sequence.

So all these three things just depends on applied pulses. Now there are three different modes to rotate this motor

1. Single coil excitation
2. Double coil excitation
3. Half step excitation

The table given below will give you the complete idea that how to give pulses in each mode



Single coil excitation		Double coil excitation		Half step excitation	
Clockwise	Anticlockwise	Clockwise	Anticlockwise	Clockwise	Anticlockwise
L4 L3 L2 L1	L4 L3 L2 L1	L4 L3 L2 L1	L4 L3 L2 L1	L4 L3 L2 L1	L4 L3 L2 L1
0 0 0 1	0 0 0 1	0 0 1 1	0 0 1 1	0001	0001
0 0 1 0	1 0 0 0	0 1 1 0	1 0 0 1	0011	0011
0 1 0 0	0 1 0 0	1 1 0 0	1 1 0 0	0010	1000
1 0 0 0	0 0 1 0	1 0 0 1	0 1 1 0	0110	1001
				0100	0100
				1100	1100
				1000	0010
				1001	0110

Pulses for stepper motor module

Note:- In half step excitation mode motor will rotate at half the specified given step resolution. Means if step resolution is 1.8 degree then in this mode it will be 0.9 degree. Step resolution means on receiving on 1 pulse motor will rotate that much degree. If step resolution is 1.8 degree then it will take 200 pulses for motor to compete 1 revolution (360 degree).

Now let me give you the specification of the stepper motor that I have used.

Max rated voltage: - 5 V
Max rated current per coil: – 0.5 Amp
Step resolution: - 1.8 degree / pulse
Max RPM: - 20 in single/double coil excitation mode and 60 in half step mode
Torque: – 1.5 Kg/cm2

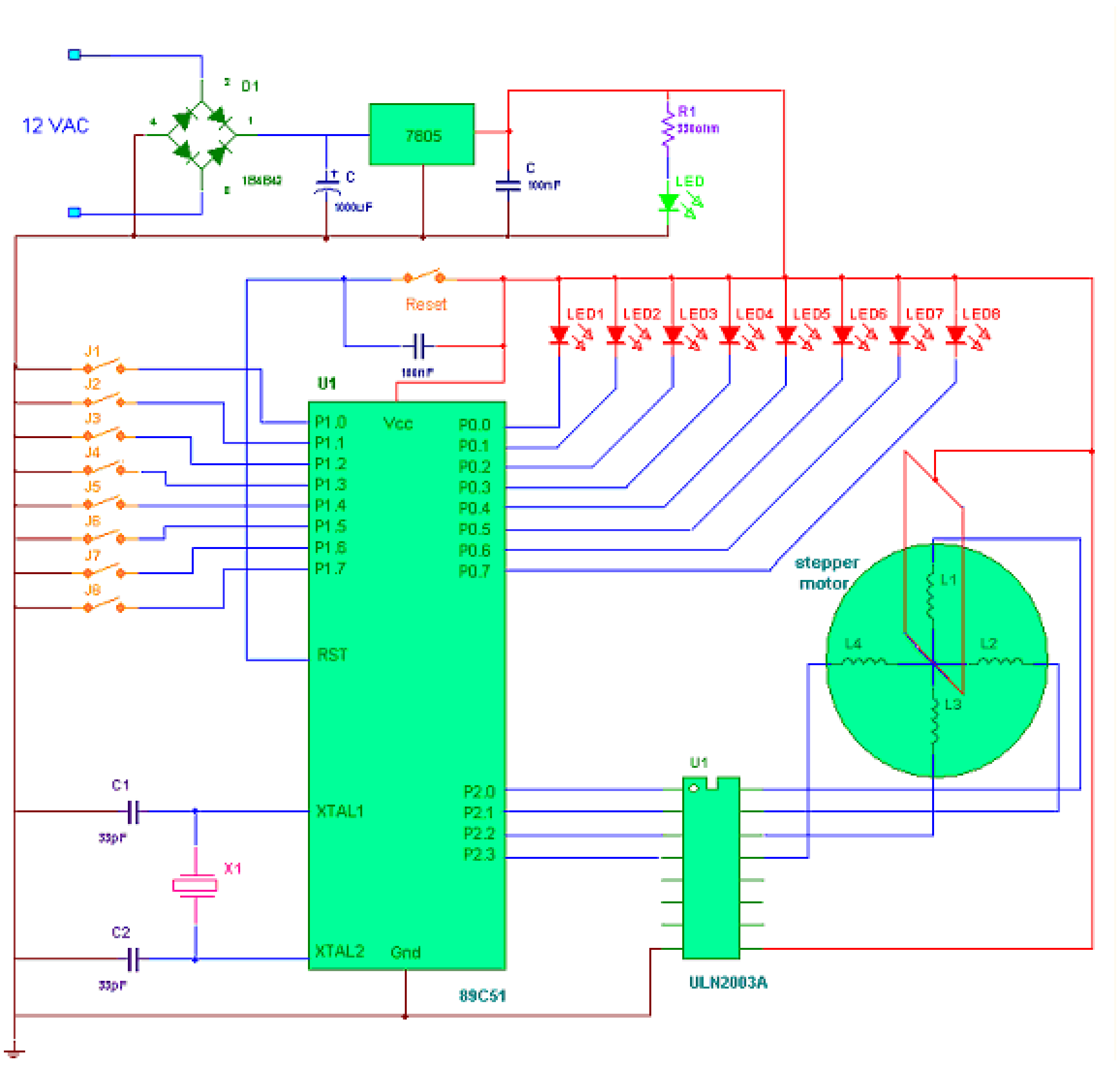
RPM calculation:-

One can calculate the exact RPM at which motor will run. We know that motor needs 200 pulses to complete 1 revolution. Means if 200 pulses applied in 1 second motor will complete 1 revolution in 1 second. Now 1 rev. in 1 sec means 60 rev. in 1 minute. That will give us 60 RPM. Now 200 pulses in 1 sec means the PRF is 200 Hz. And delay will be 5 millisecond (ms). Now lets see it reverse.

* If delay is 10 ms then PRF will be 100 Hz.
* So 100 pulses will be given in 1 sec
* Motor will complete 1 revolution in 2 second
* So the RPM will be 30.

In same manner as you change delay the PRF will be changed and it will change RPM

Stepper motor control board circuit:-



Stepper motor control board circuit

The circuit consists of very few components. The major components are 7805, 89C51 and ULN2003A.

Connections:-

1. The transformer terminals are given to bridge rectifier to generate rectified DC.
2. It is filtered and given to regulator IC 7805 to generate 5 V pure DC. LED indicates supply is ON.
3. All the push button micro switches J1 to J8 are connected with port P1 as shown to form serial keyboard.
4. 12 MHz crystal is connected to oscillator terminals of 89C51 with two biasing capacitors.
5. All the LEDs are connected to port P0 as shown
6. Port P2 drives stepper motor through current driver chip ULN2003A.
7. The common terminal of motor is connected to Vcc and rest all four terminals are connected to port P2 pins in sequence through ULN chip.

Now by downloading different programs in to 89C51 we can control the operation of stepper motor. Let us see all different kind of program.

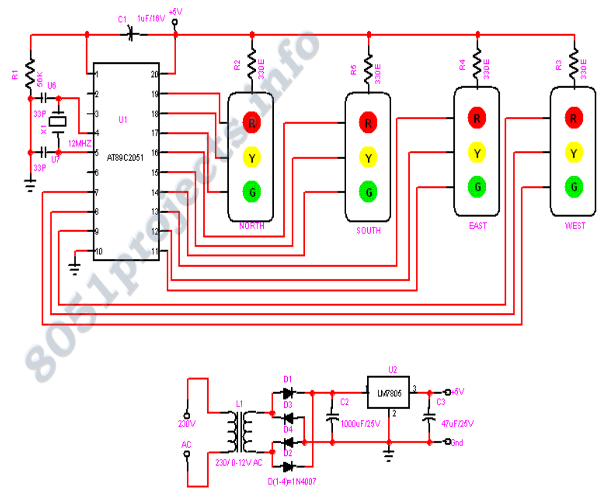
DC Motor Speed and Direction Control

Let us see the application of 8254 timer as a DC motor speed controller
As shown in the driver circuitry consists of IC 8254, a flip-flop, buffers and transistors.
When flip-flop outputs are : Q = 0 and Q = 1, inverters 1 and 3 give output logic 0 and inverters 2 and 4 give output logic 1. As a result, transistors Q_1 and Q_4 turn ON and transistors Q_2 and Q_3 turn OFF. Due to this V_{CC}, supply is applied to the positive lead of the DC motor and ground is applied to the negative lead of the DC motor. This connection causes DC motor to rotate in the forward direction.
When flip-flop outputs are : Q = 1 and Q = 0, the conditions of transistors are reversed and DC motor rotate in the reverse direction. Thus the direction of motor can be controlled by changing state of the flip-flop.
If the duty cycle of the flip-flop output is kept 50%, then motor current is positive and negative for equal amount of time. If flip-flop output is varied with enough frequency then due to inertia of motor, motor will not rotate at all. But it will exhibit some holding torque because of the current flowing through it.The variable duty cycle is achieved by programming two counters, counter 0 and counter 1 of 8254. The counter 0 and counter 1 are both programmed to divide the input clock by 25,600. The duty cycle now can be changed by changing the point at which counter 0 is started in relationship to counter 1. The clock input is divided by each counter by 25,600. This produces the basic operating frequency for the motor and as 25,600 is divisible by 256 a short program given below allows 256 different speeds.
.
TRAFFIC LIGHT CONTROLLER USING 8086
Traffic light controller interface module is designed to simulate the function of four way traffic light controller. Combinations of red, amber and green LED's are provided to indicate Halt, Wait and Go signals for vehicles. Combination of red and green LED's are provided for pedestrian crossing. 36 LED's are arranged in the form of an intersection. A typical junction is represented on the PCB with comprehensive legend printing.
At the left corner of each road, a group of five LED's (red, amber and 3 green) are arranged in the form of a T-section to control the traffic of that road. Each road is named North (N), South(S), East (E) and West (W). LED's L1, L10, L19 & L28 (Red) are for the stop signal for the vehicles on the road N, S, W, & E respectively. L2, L11, L20 & L29 (Amber) indicates wait state for vehicles on the road N, S, W, & E respectively. L3, L4 & L5 (Green) are for left, strait and right turn for the vehicles on road S. similarly L12-L13-L14, L23-L22-L21 & L32-L31-L30 simulates same function for the roads E, N, W respectively. A total of 16 LED's (2 Red & 2 Green at each road) are provided for pedestrian crossing. L7-L9.L16-L18, L25-L27 & L34-L36 (Green) when on allows pedestrians to cross and L6-L8, L15-L17, L24-L26 & L33-L35 (Red) when on alarms the pedestrians to wait.
To minimize the hardware pedestrian's indicator LED's (both red and green are connected to same port lines (PC4 to PC7) with red inverted. Red LED's L10 & L28 are connected to port lines PC2 & PC3 while L1 & L19 are connected to lines PC0 & PC1 after inversion. All other LED's (amber and green) are connected to port A & B.

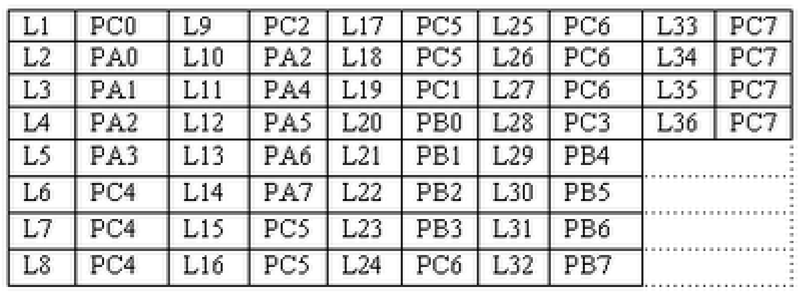
WORKING:-
8255 is interfaced with 8086 in I/O mapped I/O and all ports are output ports. The basic operation of the interface is explained with the help of the enclosed program. The enclosed program assumes no entry of vehicles from

North to West, from road East to South. At the beginning of the program all red LED’s are switch ON, and all other LED‘s are switched OFF. Amber LED is switched ON before switching over to proceed state from Halt state.
The sequence of traffic followed in the program is given below.
a) From road north to East
From road east to north
From road south to west
From road west to south
From road west to north
b) From road north to East
From road south to west
From road south to north
From road south to east
c) From road north to south
From road south to north
Pedestrian crossing at roads west & east
d) From road east to west
From road west to east
Pedestrian crossing at roads north & south

TRAFFIC LIGHT CONTROLLER USING 8051



Circuit Diagram



L1	PC0	L9	PC2	L17	PC5	L25	PC6	L33	PC7
L2	PA0	L10	PA2	L18	PC5	L26	PC6	L34	PC7
L3	PA1	L11	PA4	L19	PC1	L27	PC6	L35	PC7
L4	PA2	L12	PA5	L20	PB0	L28	PC3	L36	PC7
L5	PA3	L13	PA6	L21	PB1	L29	PB4		
L6	PC4	L14	PA7	L22	PB2	L30	PB5		
L7	PC4	L15	PC5	L23	PB3	L31	PB6		
L8	PC4	L16	PC5	L24	PC6	L32	PB7		