MICROPROCESSORS
&
MICROCOMPUTER

Prepared By:- Dr. Tapas Ranjan Baitharu

EINSTEIN ACADEMY OF TECHNOLOGY & MANAGEMENT
(Managed by Udayanath Educational & Charitable Trust, Bhubaneswar)

NAAC AICTE BPUT SCTE & VT www.eatm.in

Department of Computer Science and Engineering
Einstein Academy of Technology & Management
Baniatangi, Khordha, Odisha-752060

INTRODUCTION TO MICROPROCESSOR ARCHITECTURES

A Microprocessor is a multipurpose programmable logic device which reads the binary instructions from a storage device called 'Memory' accepts binary data as input and process data according to the instructions and gives the results as output. So, you can understand the Microprocessor as a programmable digital device, which can be used for both data processing and control applications. In view of a computer student, it is the CPU of a Computer or heart of the computer. A computer which is built around a microprocessor is called a microcomputer. A microcomputer system consists of a CPU (microprocessor), memories (primary and secondary) and I/O devices as shown in the block diagram in Fig 1. The memory and I/O devices are linked by data and address (control) buses. The CPU communicates with only one peripheral at a time by enabling the peripheral by the control signal. For example, to send data to the output device, the CPU places the device address on the address bus, data on the data bus and enables the output device. The other peripherals that are not enabled remain in high impedance state called tri-state.

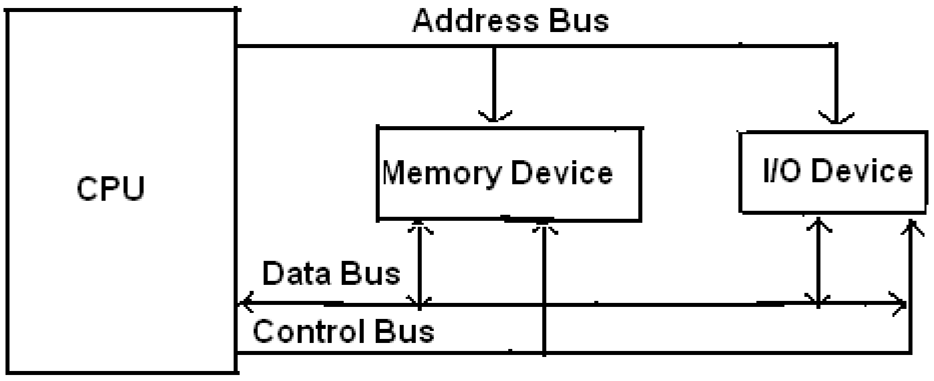


Fig.1 Block diagram of a Microcomputer

Evolution of Microprocessors

The first Microprocessor (4004) was designed by Intel Corporation which was founded by Moore and Noyce in 1968.

In the early years, Intel focused on developing semiconductor memories (DRAMs and EPROMs) for digital computers.

In 1969, a Japanese Calculator manufacturer, Busicom approached Intel with a design for a small calculator which need 12 custom chips. Ted Hoff, an Intel Engineer thought that a general purpose logic device could replace the multiple components.

This idea led to the development of the first so called microprocessor. So, Microprocessors started with a modest beginning of drivers for calculators.

With developments in integration technology Intel was able to integrate the additional chips like 8224 clock generator and the 8228 system controller along with 8080 microprocessors with in a single chip and released the 8 bit microprocessor 8085 in the year 1976. The 8085 microprocessors consisted of 6500 MOS transistors and could work at clock frequencies of 3-5 MHz It works on a single +5 volts supply. The other improved 8 bit microprocessors include Motorola MC 6809, Zilog Z-80 and RCA COSMAC.

In 1978, Intel introduced the 16 bit microprocessor 8086 and 8088 in 1979. IBM selected the Intel 8088 for their personal computer (IBM-PC).8086 microprocessor made up of 29,000 MOS transistors and could work at a clock speed of 5-10 MHz. It has a 16-bit ALU with 16-bit data bus and 20-bit address bus. It can address up to 1MB of address space. The pipelining concept was used for the first time to improve the speed of the processor. It had a pre-fetch queue of 6 instructions where in the instructions to be executed were fetched during the execution of an instruction. It means 8086 architecture supports parallel processing. The 8088 microprocessor is similar to 8086 processor in architecture ,but the basic difference is it has only 8-bit data bus even though the ALU is of 16-bit.It has a pre-fetch queue of 4-instructions only.

In 1982 Intel released another 16-bit processor called 80186 designed by a team under the leadership of Dave Stamm. This is having higher reliability and faster operational speed but at a lower cost. It had a pre-fetch queue of 6-instructions and it is suitable for high volume applications such as computer workstations, word-processor and personal computers. It is made up of 134,000 MOS transistors and could work at clock rates of 4 and 6 MHz. This also comes under first generation of Microprocessors.

Intel released another 16 bit microprocessor 80286 having 1, 34,000 transistors in 1981. It was used as CPU in PC-ATs in 1982. It is the second generation microprocessor, more advanced to 80186 processor. It could run at clock speeds of 6 to 12.5 MHz .It has a 16-bit data bus and 24- bit address bus, so that it can address up to 16MB of address space and 1GB of virtual memory. It had a pre-fetch queue of 6 instructions .Intel introduced the concept of protected mode and virtual mode to ensure proper operation. It also had on-chip memory management unit (MMU) .This was popularly called as Intel 286 in those days.

In 1985, Intel released the first 32 bit processor 80386, with 275,000 transistors. It has 32-bit data bus and 32-bit address bus so that it can address up to a total of 4GB memory also a virtual memory space of 64TB.It could process five million instructions per second and could work with all popular operating systems including Windows. It has a pre-fetch queue of length 16-bytes with extensive memory management capabilities. It is incorporated with a concept called paging in addition to segmentation technique. It uses a math co-processor called 80387.

Intel introduced 80486 microprocessors with a built-in math's co-processor and with 1.2 million transistors. It could run at the clock speed of 50 MHz This is also a 32 bit processor but it is twice as fast as 80386.The additional features in 486 processor are the built-in Cache and built-in math co-processors. The address bus here is bidirectional because of presence of cache memory.

On 19th October, 1992, Intel released the Pentium-I Processor with 3.1 million transistors. So, the Pentium began as fifth generation of the Intel x86 architecture. This Pentium was a backward compatible while offering new features. The revolutionary technology followed is that the CPU is able to execute two instructions at the same time. This is known as super scalar technology. The Pentium uses a 32-bit expansion bus; however, the data bus is 64 bits.

The 7.5 million transistors based chip; Intel Pentium II processor was released in 1997. It works at a clock speed of 300M.Hz. Pentium II uses the Dynamic Execution Technology which consists of three different facilities namely, Multiple branch prediction, Data flow analysis, and Speculative execution unit. Another important feature is a thermal sensor located on the mother board can monitor the die temperature of the processor. For thermal management applications.

Intel Celeron Processors were introduced in the year 1999. Pentium-III processor with 9.5 million transistors was introduced in 1999. It also uses dynamic execution micro-architecture, a unique combination of multiple branch prediction, dataflow analysis and speculative execution. The Pentium III has improved MMX and processor serial number feature. The improved MMX enables advanced imaging, 3D streaming audio and video, and speech recognition for enhanced Internet facility.

Pentium-IV with 42 million transistors and 1.5 GHz clock speed was released by Intel in November 2000. The Pentium 4 processor has a system bus with 3.2 G-bytes per second of bandwidth. This high bandwidth is a key reason for applications that stream data from memory. This bandwidth is achieved with 64 –bit wide bus capable of transferring data at a rate of 400 MHz. The Pentium 4 processor enables real-time MPEG2 video encoding and near real-time MPEG4 encoding, allowing efficient video editing and video conferencing.

Intel with partner Hewlett-Packard developed the next generation 64-bit processor architecture called IA-64 .This first implementation was named Itanium. Itanium processor which is the first in a family of 64 bit products was introduced in the year 2001.The Itanium processor was specially designed to provide a very high level of parallel processing ,to enable high performance without requiring very high clock frequencies .Key strengths of the Itanium architecture include ,up to 6 instructions/cycle. The Itanium processor can handle up to 6 simultaneous 64 –bit instructions per clock cycle.

The Itanium II is an IA-64 microprocessor developed jointly by Hewlett-Packard (HP) and Intel and released on July 8,2002...It is theoretically capable of performing nearly 8 times more work per clock cycle than other CISC and RISC architectures due to its parallel computing micro-architecture. The recent Itanium processor features a split L2 cache, adding a dedicated 1MB L2 cache for instructions and thereby effectively growing the original 256KBL2 cache, which becomes a dedicated data cache. The first Itanium 2 processor (code named McKinley) was more powerful than the original Itanium processor, with approximately two times performance.

Pentium 4EE was released by Intel in the year 2003 and Pentium 4E was released in the year 2004.

The Pentium Dual-Core brand was used for mainstream X86-architecture microprocessors from Intel from 2006 to 2009 The 64 bit Intel Core2 was released on July 27,2006. In terms of features, price and performance at a given clock frequency, Pentium Dual- Core processors were positioned above Celeron but below Core and Core 2 microprocessors in Intel's product range. The Pentium Dual-Core was also a very popular choice for over clocking, as it can deliver optimal performance (when over clocked) at a low price.

The Pentium Dual Core, which consists of 167 million transistors was released on January 21, 2007. Intel Core Duo consists of two cores on one die, a 2 MB L2 cache shared by both cores, and an arbiter bus that controls both L2 cache and FSB access.

Core 2 Quad processors are multi-chip modules consisting of two dies similar to those used in Core 2 Duo, forming a quad-core processor. While this allows twice the performance to a dual- core processors at the same clock frequency in ideal conditions, this is highly workload specific and requires applications to take advantage of the extra cores.

In September.2009, new Core i7 models based on the Lynnfield desktop quad-core processor and the Clarksfield quad-core mobile were added, and models based on the Arrandale dual- core mobile processor have been announced. The first six-core processor in the Core lineup is the Gulftown, which was launched on March 16, 2010. Both the regular Core i7 and the Extreme Edition are advertised as five stars in the Intel Processor Rating.

ASSEMBLY LANGUAGE PROGRAMMING EXAMPLES:

Addition Programs

Example 1: Addition of two 8-bit numbers whose sum is 8-bits.

Explanation: This assembly language program adds two 8-bit numbers stored in two memory locations .The sum of the two numbers is 8-bits only.The necessary algorithm and flow charts are given below.

ALGORITHM:

Step1. : Initialize H-L pair with memory address

XX00 (say: 9000). Step2. : Clear accumulator.

Step3. : Add contents of memory location M to

accumulator. Step4. : Increment memory pointer

(i.e. XX01).

Step5. : Add the contents of memory indicated by memory pointer to

accumulator. Step6. : Store the contents of accumulator in 9002.

Step7. : Halt

PROGRAM:

8000	**21,00,90**		**LXI**	**H, 9000**	**Initialise memory pointer to point the first data location 9000.**
8003	**3E**		**MVI**	**A, 00**	**Clear accumulator**
8004	**00**				
8005	**86**		**ADD**	**A, M**	**The first number is added to accumulator [A] ← [A] + M**
8006	**23**		**INX**	**H**	**Increment the memory pointer to next location of the Data.**
8007	**86**		**ADD**	**A, M**	**The 2nd number is added to contents of accumulator**
8008	**32**		**STA**	**9002**	**The contents of accumulator are stored in memory location 9002.**
8009	**02**				
800A	**90**				
800B	**76**		**HLT**		**Stop the execution**
8000	**21,00,90**		**LXI**	**H, 9000**	**Initialise memory pointer to point the first data location 9000.**
8003	**3E**		**MVI**	**A, 00**	**Clear accumulator**
8004	**00**				
8005	**86**		**ADD**	**A, M**	**The first number is added to accumulator [A] ← [A] + M**
8006	**23**		**INX**	**H**	**Increment the memory pointer to next location of the Data.**
8007	**86**		**ADD**	**A, M**	**The 2nd number is added to contents of accumulator**
8008	**32**		**STA**	**9002**	**The contents of accumulator are stored in memory location 9002.**
8009	**02**				
800A	**90**				
800B	**76**		**HLT**		**Stop the execution**

Ex: Input: Ex: (i) 9000 – 29 H **Ex :(ii)**
9000 –49 H

9001 – 16 H **9001 –32 H**

Result: Ex: (i) 9002 – 3F H   Ex :(ii) 9002 – 7B

Example 2: Addition of two 8-bit numbers whose sum is 16 bits.

Explanation: The first 8-bit number is stored in one memory location (say 8500) and the second 8-bit number is stored in the next location (8501).Add these two numbers and check for carry. Store the LSB of the sum in one memory location (8502) and the MSB (carry) in the other location(8503).

ALGORITHM:

Step1. : Initialize H-L pair with memory address X

(say: 8500). Step2. : Clear accumulator.

Step3. : Add contents of memory location M to

accumulator. Step4. : Increment memory pointer (i.e.

8501).

Step5. : Add the contents of memory indicated by memory pointer to accumulator.

Step6. : Check for Carry

Step 7 : Store the sum in 8502.

Step8 : Store the Carry in 8503

location Step 9 : Halt

PROGRAM:

Address of the memory location	Hex code	Label	Mnemonics		Comments
			Op-code	Operand	
8000	21,00,85		LXI	H, 8500 H	Initialise memory pointer to point the first data location 9000.
8003	3E		MVI	A,00	Clear accumulator
8004	00				
8005	86		ADD	A, M	The first number is added to accumulator [A]← [A]+M

8006	0E		MVI	C,00	Initial value of Carry is 0
8007	00				
8008	23		INX	H	Increment the memory pointer to next location of the Data.
8009	86		ADD	A, M	The 2nd number is added to contents of accumulator
800A	32		JNC	FWD	Is Carry exists ? No,go to the label FWD
800B	0E				
800C	80				
800D	0C		INR	C	Make carry =1
800E	32	FWD	STA	8502 H	The sum is stored in memory location 8502.
800F	02				
8010	85				
8011	79		MOV	A,C	
8012	32		STA	8503 H	Store the carry at 8503 location
8013	03				
8014	85				
8015	76		HLT		Stop the execution

Ex: Input: Ex : 8500 – 97 H RESULT: 8502 – 32 H

8501 – 98H 8503 -- 01 H

Example 3: Decimal addition of two 8-bit numbers whose sum is 16 bits.

Explanation: Decimal addition of two 8-bit numbers is same as that of two 8-bit numbers program. Except that the use of DAA instruction. The first 8-bit number is stored in one memory location (say 8500) and the second 8-bit number is stored in the next location(8501).Add these two numbers and use the DAA instruction to get the result in decimal. Also check for carry. Store the LSB of the sum in one memory location(8502) and the MSB (carry) in the other location(8503).

ALGORITHM:

Step1. : Initialize H-L pair with memory address XXXX (say: 8500).

Step2. : Clear Carry register C.

Step3. : Move contents of memory location M to accumulator.

Step4. : Increment memory pointer (i.e. 8501).

Step5. : Add the contents of memory indicated by memory pointer to accumulator.

Step6. : Apply the instruction DAA(Decimal adjust after addition)

Step7: Check for Carry

Step8: Store the sum in XX02.

Step9: Store the Carry in XX03 location

Step10: Halt

PROGRAM

Address of the memory location	Hex code	Label	Mnemonics		Comments
			Op-code	Operand	
8000	21, 00,85		LXI	H, 8500 H	Initialise memory pointer to point the first data location 9000.

8003	0E		MVI	C, 00	Clear accumulator
8004	00				
8005	7E		MOV	A, M	The first number is added to accumulator [A]🡐 [A]+M
8006	23		INX	H	Increment the memory pointer to next location of the Data.
8007	86		ADD	A, M	The 2nd number is added to contents of accumulator
8008	27		DAA		
8009	D2		JNC	FWD	Is Carry exists? No, go to the label FWD
	0D				
	80				
800C	0C		INR	C	Make carry =1
800D	32	FWD	STA	8502 H	The contents of accumulator are

800E	02				stored in memory location 8502.
800F	85				
8010	79		MOV	A, C	Carry is moved to accumulator
8011	32		STA	8503 H	A Carry is stored in the location 8503
8012	03				
8013	85				
8014	76		HLT		Stop the execution

Ex: Input: Ex : 8500 – 67 D RESULT: 8502 – 52 D

8501 – 85 D 8503 – 01 (Carry)

Example 4: Addition of two 16-bit numbers whose sum is 16 bits or more

Explanation: First 16-bit number is stored in two consecutive locations (Ex 8500 &8501) because in each location we can store only one 8-bit number. Store the second 16-bit number in the next two consecutive locations (For Ex: 8502 &8503).Add the LSB of the first number to the LSB of the second number and the MSB of the first number to the MSB of the second number using the DAD instruction. Store the sum in the next two locations and the carry (if any) in the third location

ALGORITHM:

Step1: First 16 bit number is in locations 8500 & 8501 respectively

Step2: Second 16-bit number is in locations 8502 & 8503

Step3: Add the two 16-bit numbers using DAD Instruction.

Step4: Sum is stored in locations 8504 & 8505.

Step5: Carry (if any) is stored in the location 8506.

Step6: Halt

PROGRAM:

ADDRESS	HEX – CODE	LABEL	MNEMONIC		COMMENTS
			OPCODE	OPERAND	
8000	2A,00,85		LHLD	8500 H	First 16-bit number in H-L pair
8001	00				
8002	85				
8003	EB		XCHG		Exchange first number to D-E Pair
8004	2A		LHLD	8502 H	
8005	02				
8006	85				
8007	0E		MVI	00	MSB of the sum is initially 00
8008	00				
8009	19		DAD	D	Add two 16 –bit numbers
800A	D2		JNC	FWD	Is Carry? If yes go to the next line .Else go to the 800E LOCATION
800B	0E				
800C	80				
800D	OC		INR	C	Increment carry
800E	22	FWD	SHLD	8504 H	Store the LSB of the Sum in 8504 & MSB in 8505 locations
800F	04				
8010	85				
8011	79		MOV	A,C	MSBs of the sum is in

					Accumulator
8012	32		STA	8506 H	Store the MSB (Carry) of the result in 8506 location
8013	06				
8014	85				
8015	76		HLT		Stop execution

Ex: INPUT:	8500- 12 H LSB of the Ist Number	RESULT : 8504 - 25H LSB of the Sum
	8501- 13 H MSB of the Ist Number	8505 – 25H MSB of the Sum
	8502 -13 H LSB of the IInd Number	8506 -- 00 Carry .
	8503 -12H MSB of the IInd number	

Subtraction Programs:

Example 5: Subtraction of two 8-bit numbers without borrows.

Explanation: It's a simple program similar to addition of two 8- bit numbers, except that we use the instruction SUB instead of ADD. The first 8-bit number is stored in XX00 memory location and the second 8-bit number is stored in the XX01 location .Use the SUB instruction and store the result in the XX02 location.

ALGORITHM:

Step1. : Initialise H-L pair with the address of minuend.

Step2. : Move the minuend into accumulator

Step3. : Increment H-L pair

Step4. : Subtract the subtrahend in memory location M from the minuend.

Step5. : Store the result in XX02.

Step6. : Stop the execution

ADDRESS	HEX CODE	LABEL	MNEMONIC		COMMENTS
			OPCODE	OPERAND	

8000	21		LXI	H, 8500	Initialise H-L pair and get the First number in to 8500 location
8001	00				
8002	85				
8003	7E		MOV	A,M	[A] ← [M]
8004	23		INX	H	[M+1] ← [M]
8005	96		SUB	M	A ← [A] – [M]
8006	23		INX	H	Next memory location
8007	77		MOV	M,A	Store the result in the location 8502
8008	76		HLT		Stop the execution

PROGRAM:

INPUT: Ex : 8500- 59H   Result: 8502 – 29H

8501- 30H

Example 6: Subtraction of two 8-bit Decimal numbers.

Explanation: In this program we can't use the DAA instruction after SUB or SBB instruction because it is decimal adjust after addition only. So, for decimal subtraction the number which is to be subtracted is converted to 10's complement and then DAA is applied.

ALGORITHM:

Step1. : Initialise H-L pair with the address of second number (XX01).

Step2. : Find its ten's complement

Step3. : Decrement the H-L pair for the first number (XX00)

Step4. : Add the first number to the 10's complement of second number.

Step5. : Store the result in XX02.

Step6. : Stop the execution

PROGRAM:

ADDRESS	HEX CODE	LABEL	MNEMONIC		COMMENTS
			OPCODE	OPERAND	
8000	21		LXI	H,8500	Initialise H-L pair and get theSecond number in to 8501 location
8001	00				
8002	85				
8003	3E		MVI	A,99	[A] ← 99
8004	99				
8005	96		SUB	M	9's complement of second number
8006	3C		INR	A	10's complement of second number
8007	2B		DCX	H	Address of the first number

8008	86		ADD	M	Add first number to 10’s complement of second number
8009	27		DAA		
800A	32		STA	8502	Store the result in the location 8502
800B	02				
800C	85				
800D	76		HLT		Stop the execution

Ex: Input: 8500 -76 D **Result: 8502 - 41 D**

8501- 35 D

Example 6: Subtraction of two 16 –bit numbers.

Explanation: It is very similar to the addition of two 16-bit numers.Here we use SUB &SBB instructions to get the result .The first 16-bit number is stored in two consecutive locations and the second 16-bit number is stored in the next two consecutive locations.The lsbs are subtracted using SUB instruction and the MSBs aare subtracted using SBB instruction.The result is stored in different locations.

ALGORITHM:

Step1. : Store the first number in the locations 8500 & 8501.

Step2. : Store the second number in the locations 8502 &8503.

Step4. : Subtract the second number from the first number with borrow.

Step5. : Store the result in locations 8504 & 8505.

Step6. : Store the borrow in location 8506

Step 7: Stop the execution

PROGRAM:

Ex: INPUT : 8500- FF H LSB of the I^st Number RESULT: 8504 - 11H LSB

ADDRESS	HEX CODE	LABEL	MNEMONIC		COMMENTS
			OPCODE	OPERAND	
8000	2A, 00,85		LHLD	8500 H	First 16-bit number in H-L pair
8003	EB		XCHG		Exchange first number to D-E Pair
8004	2A		LHLD	8502 H	Get the second 16-bit number in H-L pair
8005	02				
8006	85				
8007	7B		MOV	A, E	Get the lower byte of the First number in to Accumulator
8008	95		SUB	L	Subtract the lower byte of the second number
8009	6F		MOV	L, A	Store the result in L- register
800A			MOV	A, D	Get higher byte of the first number
800A	9C		SBB	H	Subtract higher byte of second number with borrow
800B	67		MOV	H, A	
800C	22		SHLD	8504	Store the result in memory locations with LSB in 8504 & MSB in 8505
800D	04				
80OE	85				
80OF	76		HLT		Stop execution

8501 - FF H MSB of the I^st Number 8505 – 11 H MSB

8502 -EE H LSB of the II^nd Number

8503 –EE H MSB of the II^nd number

Multiplication Programs

Example 7: Multiplication of two 8-bit numbers. Product is 16-bits.

Explanation: The multiplication of two binary numbers is done by successive addition. When multiplicand is multiplied by 1 the product is equal to the multiplicand, but when it is multiplied by zero, the product is zero. So, each bit of the multiplier is taken one by one and checked whether it is 1 or 0 .If the bit of the multiplier is 1 the multiplicand is added to the product and the product is shifted to left by one bit. If the bit of the multiplier is 0 , the product is simply shifted left by one bit. This process is done for all the 8-bits of the multiplier.

ALGORITHM:

Step 1 : Initialise H-L pair with the address of multiplicand.(say 8500)

Step 2 : Exchange the H-L pair by D-E pair. so that multiplicand is in D-E pair.

Step 3 : Load the multiplier in Accumulator.

Step 4 : Shift the multiplier left by one bit.

Step 5 : If there is carry add multiplicand to product.

Step 6 : Decrement the count.

Step 7 : If count ≠ 0; Go to step 4

Step 8 : Store the product i.e. result in memory location.

Step 9 : Stop the execution

PROGRAM:

ADDRESS	HEX-CODE	LABEL	MNEMONIC		COMMENTS
			OPCODE	OPERAND	
8000	2A, 00,85		LHLD	H, 8500	Load the multiplicand in to H-L pair
8003	EB		XCHG		Exchange the multiplicand in to D-E pair
8004	3A		LDA	8502	Multiplier in Accumulator
8005	02				
8006	85				
8007	21		LXI	H.0000	Initial value in H-L pair is 00
8008	00				
8009	00				
800A	0E		MVI	C,08	Count =08
800B	08				
800C	29	LOOP	DAD	H	Shift the partial product left by one bit.

800D	17		RAL		Rotate multiplier left by one bit
800E	D2		JNC	FWD	Is Multiplier bit =1? No go to label FWD
800F	12				
8010	80				
8011	19		DAD	D	Product =Product +Multiplicand
8012	0D	FWD	DCR	C	COUNT=COUNT-1
8013	C2		JNZ	LOOP	
8014	0C				
8015	80				
8016	22		SHLD	8503	Store the result in the locations 8503 & 8504
8017	03				
8018	85				
8019	76		HLT		Stop the execution

INPUT :

Address	Data
8500	8AH – LSB of Multiplicand
8501	00 H – MSB of Multiplicand
8502	52 H - Multiplier

Result:

8503	34 H – LSB of Product
8504	2C H – MSB of Product

Division Programs

Example 7: Division of a 16- bit number by a 8-bit number.

Explanation: The division of a 16/8-bit number by a 8-bit number follows the successive subtraction method. The divisor is subtracted from the MSBs of the dividend .If a borrow occurs, the bit of the quotient is set to 1 else 0.For correct subtraction process the dividend is shifted left by one bit before each subtraction. The dividend and quotient are in a pair of register H-L.The vacancy arised due to shifting is occupied by the quotient .In the present example the dividend is a 16-bit number and the divisor is a 8-bit number. The dividend is in locations 8500 &8501.Similarly the divisor is in the location 8502.The quotient is stored at 8503 and the remainder is stored at 8504 locations.

ALGORTHM:

STEP1. : Initialise H-L pair with address of dividend.

STEP2. : Get the divisor from 8502 to register A & then to Reg.B

STEP3. : Make count C=08

STEP4. : Shift dividend and divisor left by one bit

STEP 5: Subtract divisor from dividend.

STEP6. : If carry = 1 : goto step 8 else step7.

STEP7. : Increment quotient register.

STEP8. : Decrement count in C

STEP9. : If count not equal to zero go to step 4

STEP10: Store the quotient in 8503

STEP11: Store the remainder in 8504

STEP12: Stop execution.

PROGRAM:

ADDR-ESS	HEX – CODE	LABEL	MNEMONIC		COMMENTS
			OPCODE	OPERAND	
8000	21		LHLD	H, 8500	Initialize the H-L pair for dividend
8001	00				
8002	85				
8003	3A		LDA	8502 H	Load the divisor from location 8502 to accumulator
8004	02				
8005	85				
8006	47		MOV	B,A	Move Divisor to Reg.B from A
8007	0E		MVI	C,08	Count =08
8008	08				
8009	29	BACK	DAD	H	Shift dividend and quotient left by one bit

800A	7C		MOV	A,H	MSB of dividend in to accumulator
800B	90		SUB	B	Subtract divisor from MSB bits of divisor
800C	DA		JC	FWD	Is MSB part of dividend > divisor ? No,goto label FWD
800D	11				
800E	80				
800F	67		MOV	H,A	MSB of the dividend in Reg.H
8010	2C		INR	L	Increment quotient
8011	0D	FWD	DCR	C	Decrement count
8012	C2		JNZ	BACK	If count is not zero jump to8009 location
8013	09				
8014	80				
8015	22		SHLD	8503H	Store quotient in 8503 and remainder in 8504 locations
8016	03				
8017	85				
8018	76		HLT		Stop execution

Ex: Input & Result

Address	Data
8500	64 → LSB of Dividend
8501	00 → MSB of Dividend
8502	07 → Divisor
8503	0E → Quotient
8504	02 → Remainder

Largest & Smallest numbers in an Array

Example 8: To find the largest number in a data array

Explanation: To find the largest number in a data array of N numbers (say)first the count is placed in memory location (8500H) and the data are stored in consecutive locations.(8501….onwards).The first number is copied to Accumulator and it is compared with the second number in the memory location. The larger of the two is stored in Accumulator. Now the third number in the memory location is again compared with the accumulator. And the largest number is kept in the accumulator. Using the count, this process is completed , until all the numbers are compared .Finally the accumulator stores the smallest number and this number is stored in the memory location.85XX.

ALGORTHM:

Step1: Store the count in the Memory location pointed by H-L register.

Step2: Move the I st number of the data array in to accumulator

Step3: Compare this with the second number in Memory location.

Step4: The larger in the two is placed in Accumulator

Step5: The number in Accumulator is compared with the next number in memory .

Step 6: The larger number is stored in Accumulator.

Step 7; The process is repeated until the count is zero.

Step 8: Final result is stored in memory location.

Step 9: Stop the execution

PROGRAM

ADDR-ESS	HEX – CODE	LABEL	MNEMONIC		COMMENTS
			OPCODE	OPERAND	
8000	21,00,85		LXI	H, 8500	INITIALISE H-L PAIR
8003	7E		MOV	C,M	Count in the C register
8004	23		INX	H	First number in H-L pair
8005	4E		MOV	A,M	Move first number in to Accumulator
8006	0D		DCR	C	Decrement the count
8007	91	LOOP1	INX	H	Get the next number
8008	BE		CMP	M	Compare the next number with previous number

8009	D2		JNC	LOOP2	Is next number >previous maximum?No,go to the loop2
800A	0D				
800B	80				
800C	7E		MOV	A,M	If,yes move the large number in to Accumulator
800D	0D	LOOP2	DCR	C	Decrement the count
800E	C2		JNZ	LOOP1	If count not equal to zero,repeat
800F	07				
8011	80				
8012	78				
8013	32		STA	85XX	Store the largest number in the location 85XX
8014	XX				
8015	85				
8016	76		HLT		Stop the execution

Ex : Input : 8500- N(Say N=7) **Result : 8508 - 7F**

8501-05

8502-0A

8503-08

8504-14

8505 -7F

8506-25

8507-2D

Example 9 : To find the smallest number in a data array.

Explanation: To find the smallest number in a data array of N numbers (say)first the count is placed in memory location (8500H) and the data are stored in consecutive locations.(8501….onwards).The first number is copied to Accumulator and it is compared with the second number in the memory location.The smaller of the two is stored in Accumulator.Now the third number in the memory location is again compared with the accumulator.and the smallest number is kept in the accumulator.Using the count,this process is completed until all the numbers are compared .Finally the accumulator stores the smallest number and this number is stored in the memory location.85XX.

ALGORTHM :

Step1: Store the count in the Memory location pointed by H-L register.

Step2: Move the I st number of the data array in to accumulator

Step3: Compare this with the second number in Memory location.

Step4: The smaller in the two is placed in Accumulator

Step5: The number in Accumulator is compared with the next number in memory .

Step 6: The smaller number is stored in Accumulator.

Step 7; The process is repeated until the count is zero.

Step 8: Final result is stored in memory location.

Step 9: Stop the execution

PROGRAM

ADDR-ESS	HEX – CODE	LABEL	MNEMONIC		COMMENTS
			OPCODE	OPERAND	
8000	21		LXI	H, 8500	Initialise the H-L pair.
8001	00				
8002	85				
8003	7E		MOV	C,M	Count in the C register
8004	23		INX	H	First number in H-L pair
8005	4E		MOV	A,M	Move first number in to Accumulator
8006	0D		DCR	C	Decrement the count
8007	91	LOOP1	INX	H	Get the next number
8008	BE		CMP	M	Compare the next number with previous number
8009	D2		JC	LOOP2	Is next number <previous smallest ?If yes go to the loop2

800A	0D				
800B	80				
800C	7E		MOV	A,M	No,move the smaller number in to Accumulator
800D	0D	LOOP2	DCR	C	Decrement the count
800E	C2		JNZ	LOOP1	If count not equal to zero,repeat
800F	07				
8011	80				
8012	78				
8013	32		STA	85XX	Store the smallest number in the location 85XX
8014	XX				
8015	85				
8016	76		HLT		Stop the execution

Ex: Input : 8500 - N((Say N=7) **Result : 8508 – 04**

8501-09

8502-0A

8503-08

8504-14

8505 -7F

8506-04

8507-2D

Stack and Subroutines

Stack is a set of memory locations in the Read/Write memory which is used for temporary storage of binary information during the execution of a program. It is implemented in the Last- in-first-out (LIFO) manner. i.e., the data written first can be accessed last, One can put the data on the top of the stack by a special operation known as PUSH. Data can be read or taken out from the top of the stack by another special instruction known as POP.

Stack is implemented in two ways. In the first case, a set of registers is arranged in a shift register organization. One can PUSH or POP data from the top register. The whole block of data moves up or down as a result of push and pop operations respectively. In the second case, a block of RAM area is allocated to the stack. A special purpose register known as stack pointer (SP) points to the top of the stack. Whenever the stack is empty, it points to the bottom address. If a PUSH operation is performed, the data are stored at the location pointed to by SP and it is

decremented by one. Similarly if the POP operation is performed, the data are taken out of the location pointed at by SP and SP is incremented by one. In this case the data do not move but SP is incremented or decremented as a result of push or pop operations respectively.

Application of Stack: Stack provides a powerful data structure which has applications in many situations. The main advantage of the stack is that,

We can store data (PUSH) in it with out destroying previously stored data. This is not true in the case of other registers and memory locations.

stack operations are also very fast

The stack may also be used for storing local variables of subroutine and for the transfer of parameter addresses to a subroutine. This facilitates the implementation of re-entrant subroutines which is a very important software property.

The disadvantage is, as the stack has no fixed address, it is difficult to debug and document a program that uses stack.

Stack operation: Operations on stack are performed using the two instructions namely PUSH and POP. The contents of the stack are moved to certain memory locations after PUSH instruction. Similarly, the contents of the memory are transferred back to registers by POP instruction.

For example let us consider a Stack whose stack top is 4506 H. This is stored in the 16-bit Stack pointer register as shown in Fig.29

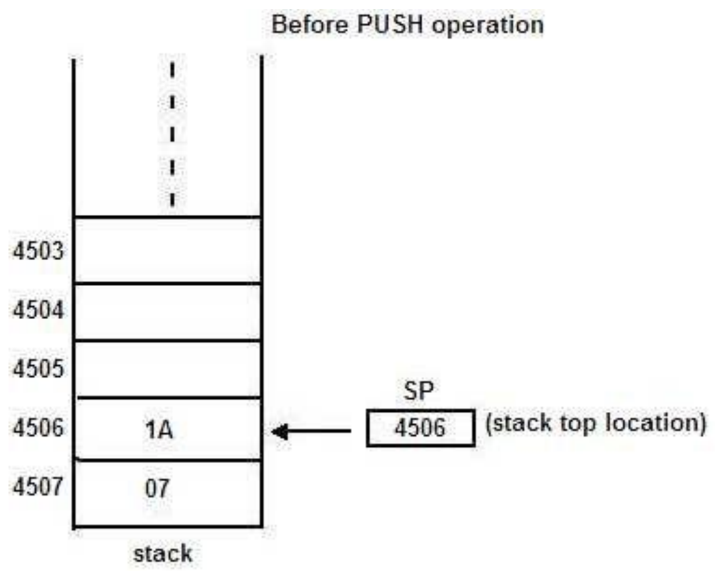


Figure.29 The PUSH operation of the Stack

Let us consider two registers (register pair) B & C whose contents are 25 & 62.

Reg. B	Reg. C
25	62

After PUSH operation the status of the Stack is as shown in Fig 3.30

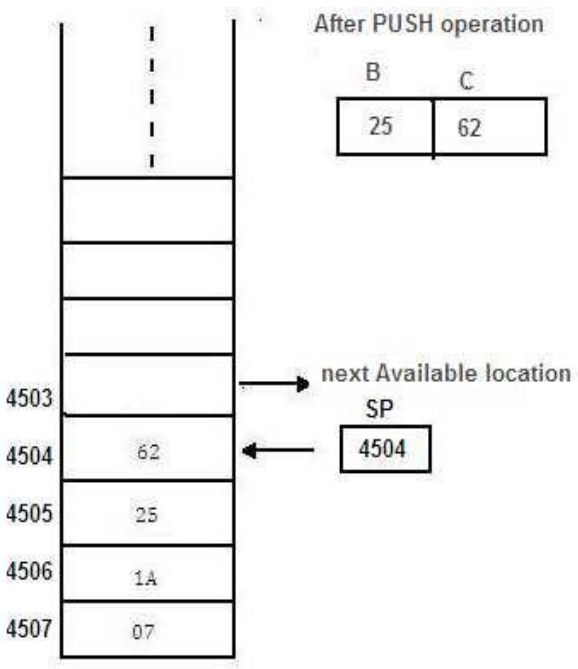


Figure .30 After PUSH operation the status of the stack

Let us now consider POP operation: The Figs 31 & 32 explains before and after the POP operation in detail

.

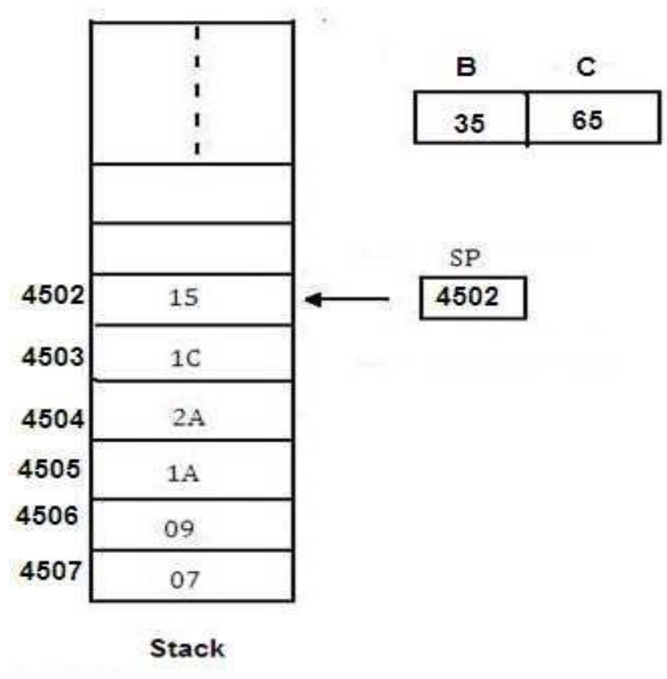


Figure 31 The POP operation of the Stack

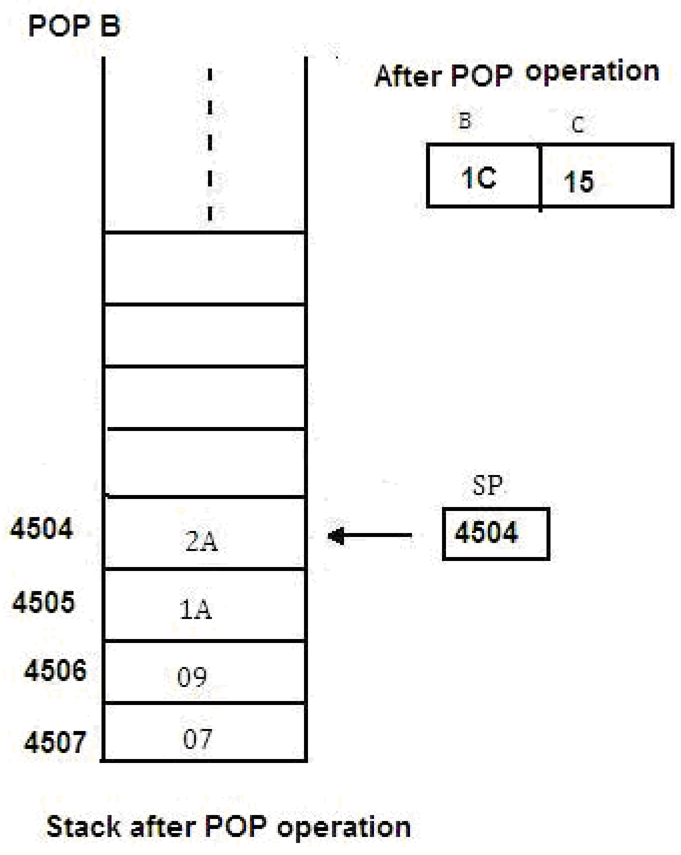


Figure 32 After POP operation the status of the stack

Before the operation the data 15 and 1C are in the locations 4502 & 4503 and after the pop operation the data is copied to B-C pair and now the SP register points to 4504 location. This is shown in Fig.3.32

Programming Example FOR PUSH & POP

Write a program to initialize the stack pointer (SP) and store the contents of the register pair H-L on stack by using PUSH instruction. Use the contents of the register pair for delay counter and at the end of the delay retrieve the contents of H-L using POP.

Memory Location	Label	Mnemonics	Operand	Comments
8000		LXI	SP, 4506 H	Initialize Stack pointer
8003		LXI	H,2565 H	
8006		PUSH	H	
8007				
.		DELAY	CALL	Push the

. . . 8.00A		. . . POP	. . . H	contents.

Subroutine: It is a set of instructions written separately from the main program to execute a function that occurs repeatedly in the main program.

For example, let us assume that a delay is needed three times in a program. Writing delay programs for three times in a main program is nothing but repetition. So, we can write a subroutine program called 'delay' and can be called any number of times we need

Similarly, in 8085 microprocessor we do not find the instructions for multiplication and division. For this purpose we write separate programs. So, in any main program if these operations are needed more than once, the entire program will become lengthy and complex. So, we write subroutine programs MUL & DIV separately from main program and use the instruction CALL MUL (or) CALL DIV in the main program. This can be done any number of times. At the end of every subroutine program there must be an instruction called 'RET'. This will take the control back to main program.

The 8085 microprocessor has two instructions to implement the subroutines. They are CALL and RET. The CALL instruction is used in the main program to call a subroutine and RET instruction is used at the end of the subroutine to return to the main program. When a subroutine is called, the contents of the program counter, which is the address of the instruction following the CALL instruction is stored on the stack and the program execution is transferred to the subroutine address. When the RET instruction is executed at the end of the subroutine, the memory address stored on the stack is retrieved and the sequence of execution is resumed in the main program.

Diagrammatic representation

Let us assume that the execution of the main program started at 8000 H. It continues until a CALL subroutine instruction at 8020 H is encountered. Then the program execution transfers to 8070 H. At the end of the subroutine 807B H. The RET instruction is present. After executing this RET, it comes back to main program at 8021 H as shown in the following Fig. 33

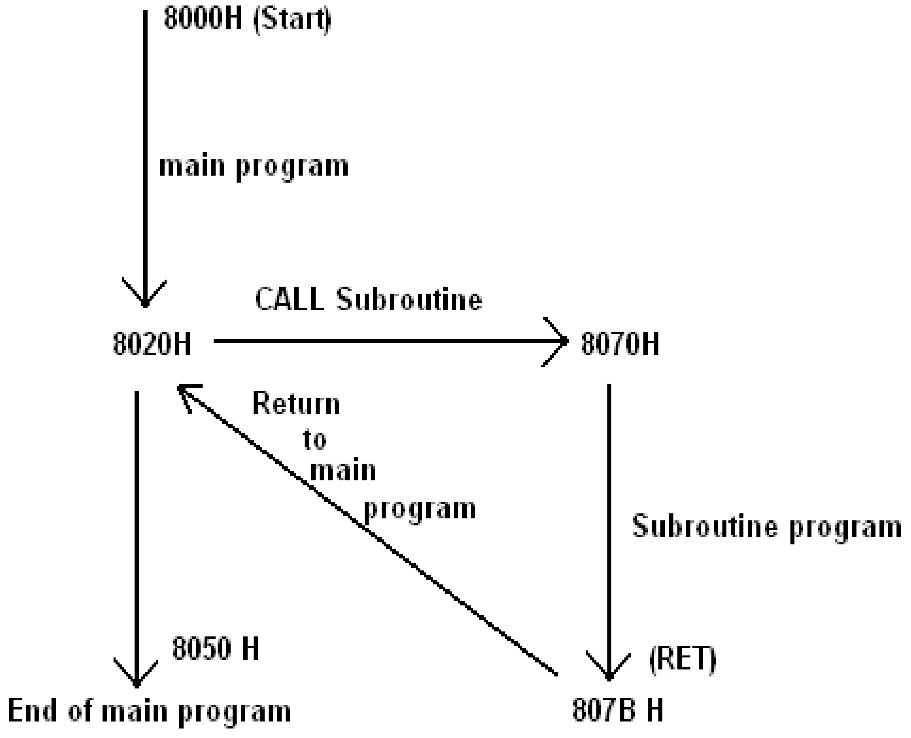


Fig.33 Diagrammatic representation of subroutine program execution

The same is explained using the assembly language program example.

Program Example:

Memory Address	**Mnemonics**	**Operand**	**Comments**
8000 **|** **|** **|**	**LXI**	**SP, 8400 H**	**Initialize the Stack pointer at 8400 H**
8020 **8021** **8022**	**CALL**	**8070 H**	**Call a subroutine program stored at the location 8070 H. (It is a three by Instruction)**
8023 **|** **|** **|** **802F**	**Next instruction** **|** **|** **|** **HLT**		**The address of the next instruction following CALL instruction.** **End of the main program** **.**

Memory Address Mnemonics Operand Comments

8070 | | | | 807B	Instructions RET		Beginning of the Subroutine. End of the program	Subroutine Program:
807C	Next Subroutine		Instructions of next subroutine if any	Delay programs:
807F	RET		End of the subroutine.	

In many situations it may be desired to provide some delay between the execution of two instructions by a microprocessor. The delay can be produced by either hardware chip like 8253 or by writing a software program using registers of the processor. Here we will discuss the software delay program. This delay program is not a part of the main program. Hence it is called delay sub-routine program. For small delays we can use only one register. But for longer delays one has to use two or three registers. The technique involved here is, a register is loaded with a number and then decremented by using the instruction DCR until it becomes zero. The time of execution of the microprocessor is equal to the delay time produced.

For example, we have constructed a display system where the LEDs receive the input from a microprocessor. Since the microprocessor is a very fast device it sends the signal at very high speeds there by our eye cannot recognize the display pattern. So, if you provide some delay between two input signals, the display can be visualized clearly. Similarly to observe the rotations of a stepper motor, a delay is needed between every two excitation signals applied to the motor.

Delay Subroutine with one register:

Program

Address	Label	Machine code	Mnemonics	Operand	Comments
9000			MVI	A, FF	Get FF in register A

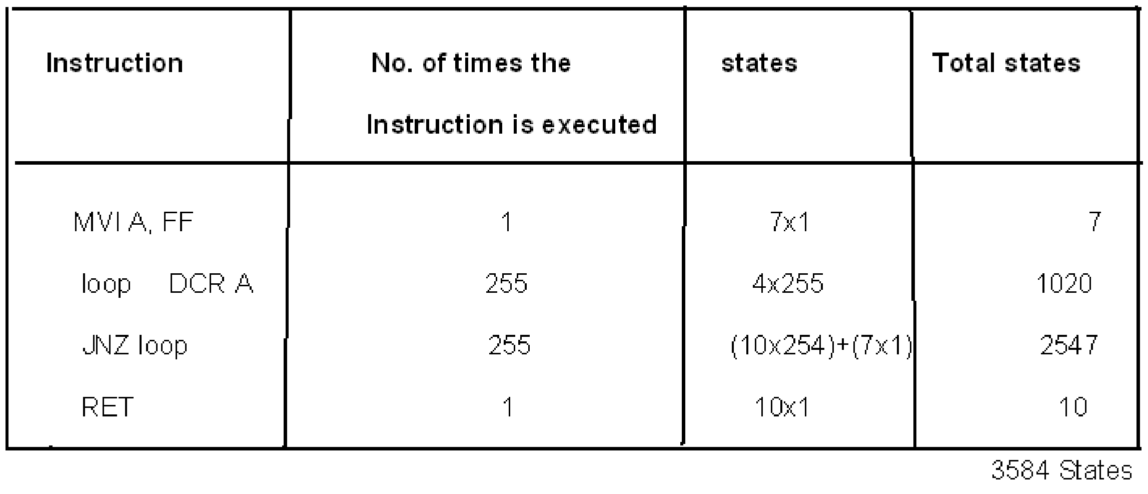
9002	**LOOP**		**DCR**	**A**	**Decrement register A.**
9003			**JNZ**	**LOOP**	**Has the content of register B becomes zero? No, jump to LOOP. Yes, proceed ahead.**
9006			**RET**		**Return to main program**

Calculation of Delay time for the above program:

In the above program register A is loaded by FFH B(255 decimal) and it is decremented in a loop until it becomes zero. The delay produced by this program is as follows

We should know the number of times each instruction of the above program is being executed. The number of states required for the execution of each instruction is as follows:

Instructions	**States**
MVI A, FFH	**7**
(loop) DCR A	**4**
JNZ loop	**7/10**
RET	**10**



Instruction	No. of times the Instruction is executed	states	Total states
MVI A, FF	1	7x1	7
loop DCR A	255	4x255	1020
JNZ loop	255	(10x254)+(7x1)	2547
RET	1	10x1	10

3584 States

Total T States=3584

The time required for one T-state in INTEL 8085 microprocessor is nearly 330n.sec

Delay time is= 3584 x 333n.sec

= 3.584 x 0.333 x 10^{-3} seconds

= 1.18272 x 10^{-3} seconds

= 1. 193472 milliseconds

Delay Subroutine with two registers

Program:

Address	Label	Machine Code	Mnemonic	Operand	Comments
8400			MVI	B, 10H	Get desired number in register B
8402	LOOP1		MVI	C, 56H	Get desired number in register
8404	LOOP2		DCR	C	Decrement C.
8405			JNZ	LOOP2	Is [C] zero? No, go to LOOP2. Yes, proceed further
8408			DCR	B	Decrement register B
8409			JNZ	LOOP1	Is [B] zero? No, go to LOOP1. Yes, proceed further
840C			RET		Return to main program.

Delay Subroutine using register pair

Program:

Address	Label	Machine Code	Mnemonic	Operand	Comments
8000			LXI	D, FFFF	Get FFFF in register pair D-E
	LOOP		DCX	D	Decrement count

			MOV	A, D	Move the contents of register D to accumulator
			ORA	E	Check if D and E are zero.
			JNZ	LOOP	If D-E is not zero, jump to LOOP
			RET		Return to main program

Delay Subroutine using three registers

Program:

Address	Label	Machine Code	Mnemonic	Operand	Comments
8400			MVI	A, 98H	Get control word
8402			OUT	03	Initialize port foe LED Display
8404			MVI	B, 50H	
8406			MVI	C, FFH	
8408			MVI	D, FFH	
840A			DCR	D	Delay Subroutine with three registers
840B			JNZ	LOOP3	
840E			DCR	C	
840F			JNZ	LOOP2	
8412			DCR	B	
8413			JNZ	LOOP1	
8416			MVI	A, 01	
8418			OUT	01	Output for LED
8419			HLT		Stop.

From the above discussion it is clear that with increase of T-states required for a delay subroutine ,the delay time also increases.

8086 Microprocessor

- It is a 16-bit µp.
- 8086 has a 20 bit address bus can access up to 220 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 up to 6 instruction bytes from memory and put them in instr queue in order to speed up instruction execution.
- It requires +5V power supply.
- A 40 pin dual in line package

Architectural Diagram of 8086:

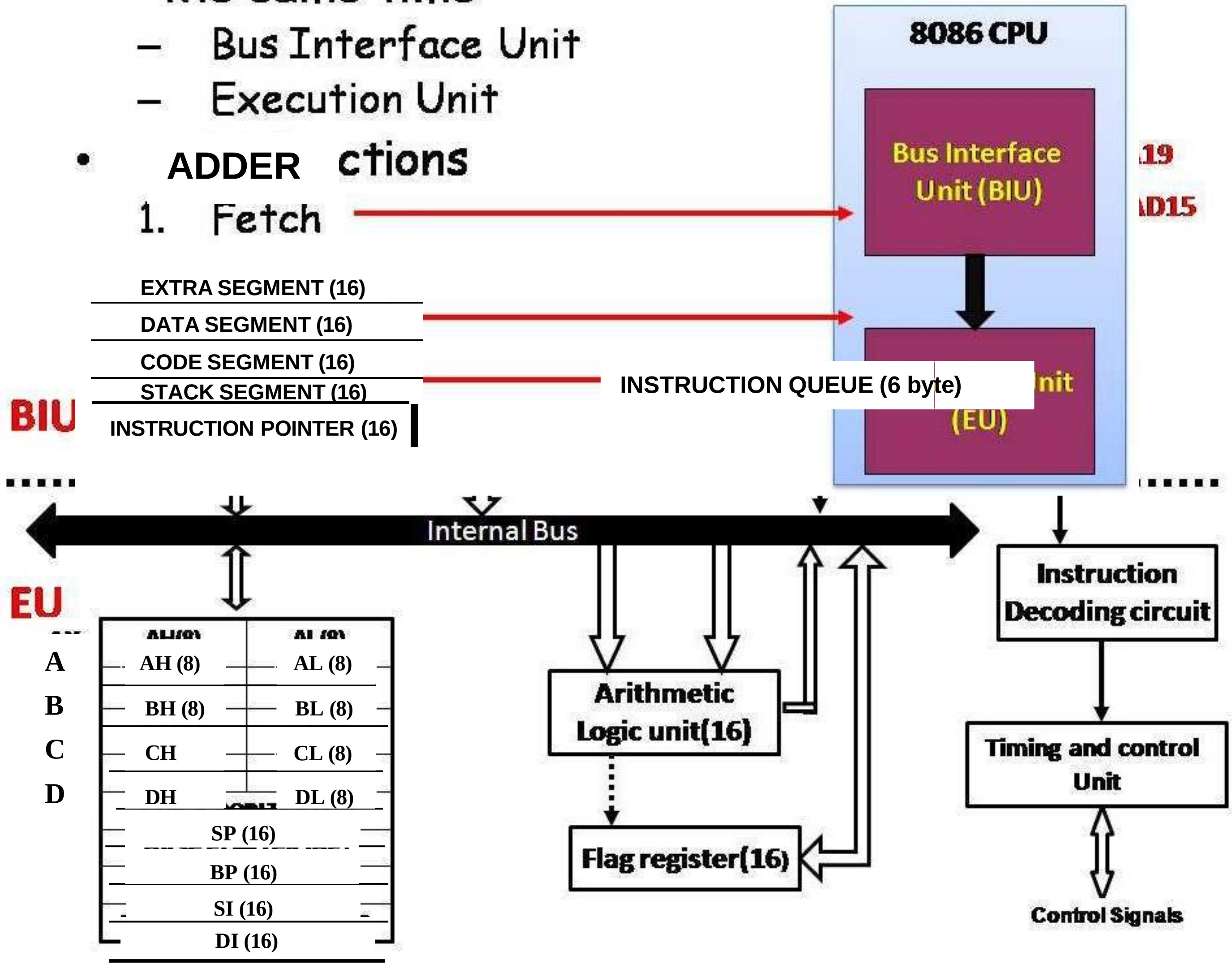
The 8086 has two parts, the Bus Interface Unit (BIU) and the Execution Unit (EU).

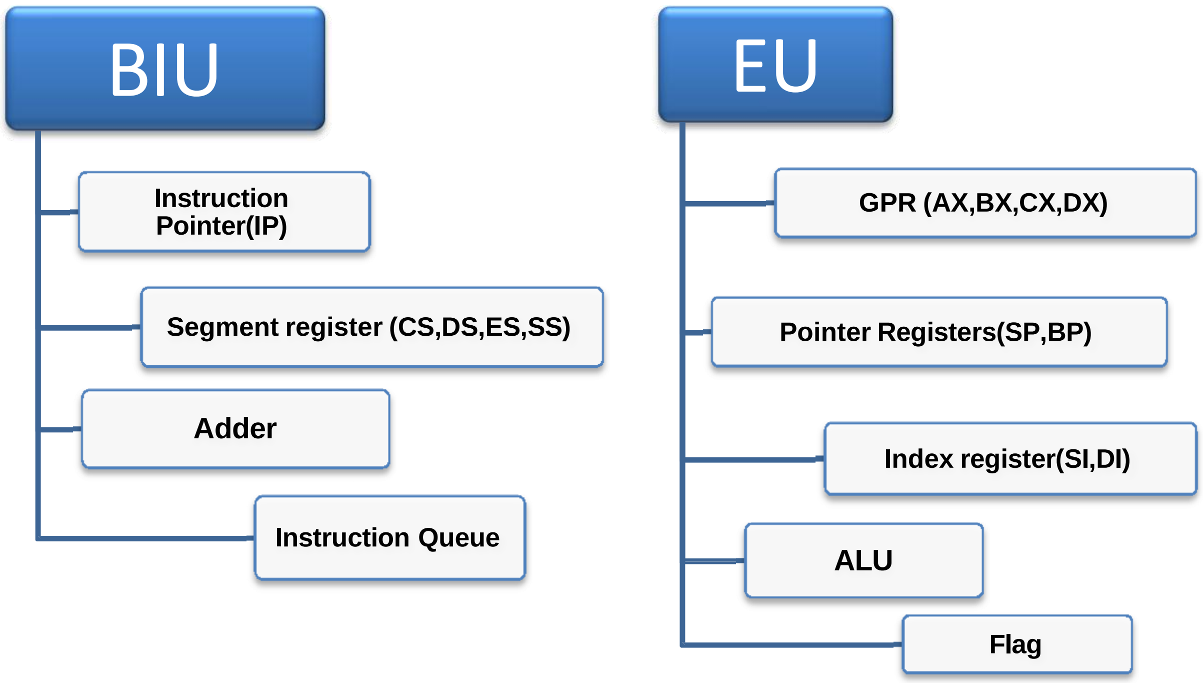
- The BIU fetches instructions, reads and writes data, and computes the 20-bit address.
- The EU decodes and executes the instructions using the 16-bit ALU.
- The two units functions independently.

Minimum and Maximum Modes:

- The minimum mode is selected by applying logic 1 to the MN / $\overline{\overline{M}}\overline{\overline{X}}$ input pin. This is a single microprocessor configuration.
- The maximum mode is selected by applying logic 0 to the MN / $\overline{\overline{M}}\overline{\overline{X}}$ input pin. This is a multi micro processors configuration.

- 8086 employs parallel processing
- 8086 CPU has two parts which operate at the same time
  - Bus Interface Unit
  - Execution Unit





Bus Interface Unit (BIU):

- 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.
- 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 calculation relocation and Bus control.
- The BIU uses a mechanism known as an instruction queue to implement a ***pipeline architecture.***
- This queue permits prefetch of up to six bytes of instruction code. Whenever the queue of the BIU is not full and it has room for at least two more bytes and at the same time 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.

EXECUTION UNIT (EU)

- 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

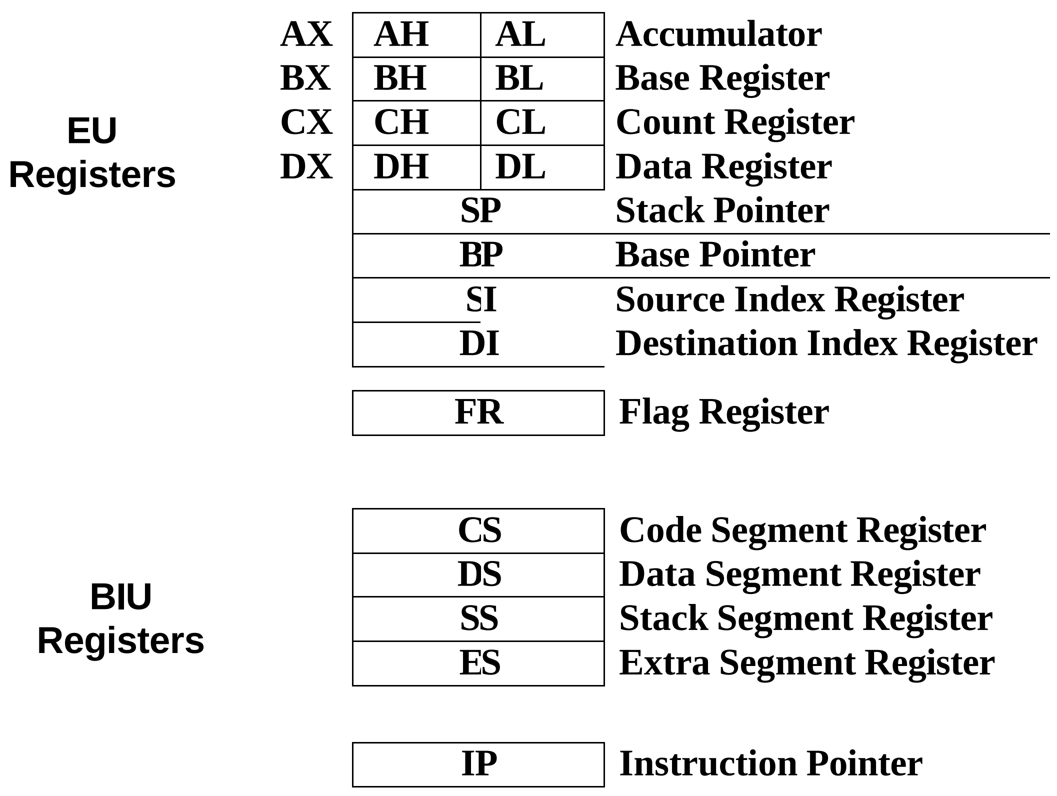
  perform the read or write bys cycles to memory or I/O and perform the operation specified by the instruction on the operands.
- During the execution of the instruction, the EU tests the status and control flags and updates them based on the results of executing the instruction.
- If the queue is empty, 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

The BIU contains the following registers:

IP - the Instruction Pointer
CS - the Code Segment Register
DS - the Data Segment Register
SS - the Stack Segment Register
ES - the Extra Segment Register

The BIU fetches instructions using the CS and IP, written CS:IP, to contract the 20-bit address. Data is fetched using a segment register (usually the DS) and an effective address (EA) computed by the EU depending on the addressing mode.

Internal Registers of 8086



- The 8086 has four groups of the user accessible internal registers.
- These are
  - Instruction pointer(IP)
  - Four General purpose registers(AX,BX,CX,DX)
  - Four pointer (SP,BP,SI,DI)
  - Four segment registers (CS,DS,SS,ES)
  - Flag Register(FR)

- The 8086 has a total of fourteen 16-bit registers including a 16 bit register called the ***status register (flag 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 addressable memory these 4 segments are located the processor uses four segment registers:

Segment Registers

1) **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.

2) **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.

3) **Data and Extra segment** (DS and ES) 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, and DX) and index register (SI, DI) is located in the data and Extra segment.

Data Registers

1) **AX (Accumulator)**
   - It is 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.
2) **BX (Base register)**

- It is 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 offset for data segment.

3) **CX (Count register)**
   - It is 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.
   - 8086 has the LOOP instruction which is used for conuter purpose when it is executed CX/CL is automatically decremented by 1.

     EX

```
        MOV CL, 05H
START   NOP
        LOOP START    (here CL is automatically decremented by 1without
                      DCR instruction.
```

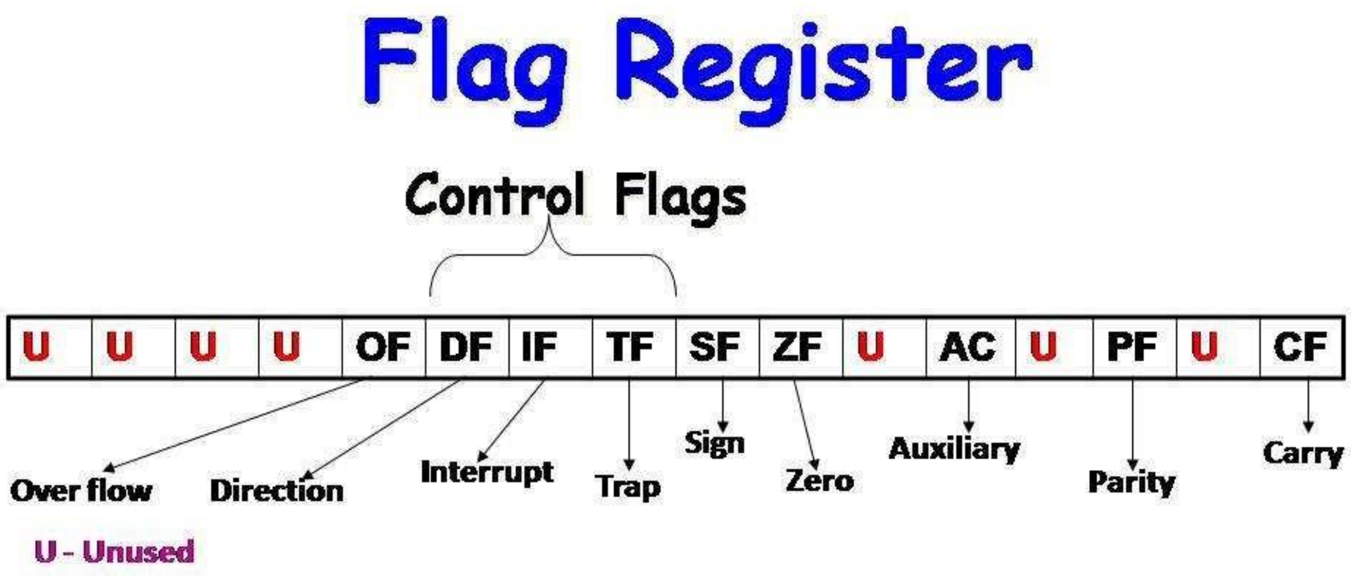
4) **DX (Data register)**
   - It is consists of two 8-bit registers DL and DH, which can be combined together and used as a 16-bit register DX. When combined, DL register contains the low-order byte of the word, and DH contains the high-order byte.
   - DX can be used as a port number in I/O operations.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

Pointer register

1. **Stack Pointer** (SP) is a 16-bit register is used to hold the offset address for stack segment.
2. **Base Pointer** (BP) is a 16-bit register is used to hold the offset address for stack segment.
   i. BP register is usually used for based, based indexed or register indirect addressing.
   ii. The difference between SP and BP is that the SP is used internally to store the address in case of interrupt and the CALL instrn.
3. **Source Index** (SI) and **Destination Index** (DI)
   These two 16-bit register is used to hold the offset address for DS and ES in case of string manipulation instrn.
   i. SI is used for indexed, based indexed and register indirect addressing, as well as a source data addresses in string manipulation instructions.
   ii. DI is used for indexed, based indexed and register indirect addressing, as well as a destination data addresses in string manipulation instructions.

Instruction Pointer (IP)

It is a 16-bit register. It acts as a program counter and is used to hold the offset address for CS.



A flag is a 16-bit register containing 9 one bit flags.

i. **Overflow Flag** (OF)
   - ➢ This flag is set if an overflow occurs. i.e. if the result of a signed operation is large enough to be accommodated in a destination register.

ii. **Direction Flag** (DF) –
   - ➢ This is used by string manipulation instructions. If this flag bit is '0', the string is processed beginning from the lowest address to the highest address. i.e. auto-incrementing mode.
   - ➢ Otherwise, the string is processed from the highest address towards the lowest address, i.e. auto-decrementing mode**.**

iii. **Interrupt-enable Flag** (IF) –
   - ➢ If this flag is set, the maskable interrupts are recognized by the CPU. Otherwise they are ignored. Setting this bit enables maskable interrupts.

iv. **Single-step Flag** (TF) –
   - ➢ If this flag is set, the processor enters the single step execution mode. In other words, a trap interrupt is generated after execution of each instruction. The processor executes the current instruction and the control is transferred to the Trap interrupt service routine.

v. **Sign Flag** (SF) –
   - ➢ This flag is set when the result of any computation is negative. For signed computations, the sign flag equals the MSB of the result.

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

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

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

ix. **Carry Flag** (CF) –
   - ➢ This flag is set when there is a carry out of MSB in case of addition or a borrow in case of subtraction. For example. When two numbers are added, a carry may

be generated out of the most significant bit position. The carry flag, in this case, will be set to 1'. In case, no carry is generated, it will be '0.

Segmented Memory

Reason for Segmented Memory:

- ✓ 8086 has a 20-bit address bus. So it can address a maximum of 1MB of memory and each memory location is addressed by a 20 bit address.
- ✓ To hold a 20-bit address there must be a 20-bit address register available within processor but 8086 only has 16-bit registers. So 20-bit address can't be stored inside the 16-bit register. To avoid this problem segmented memory is used in 8086.

Total 1MB memory can be divided into some equal size segments each of having capacity 64 KB.
So max no of segments is 16. (1mb/64 kb=16)
8086 can work with only four 64KB segments at a time within this 1MB range.
Each location in a particular segment can be expressed by two addresses.

i) ***Segment Address (16 bit):*** It refers the starting address of a segment and it is fixed for whole of the segment.

ii) ***Offset or Displacement Address (16 bit):*** It refers the individual location in that segment and it is varied location wise.

By using these two addresses the 20 bit physical address can be calculated as below:

Physical address (20 bit) = [Segment Address (16 bit) * 10]$_H$ + Offset Address(16 bit)

According to this formula segment address is multiplied by 10 and is added to offset. This is equivalent to shifting of segment register content towards left 4 times so that four zero are added to right side (MSB) of the segment address and added with the offset address to get the physical address which is 20 bit.

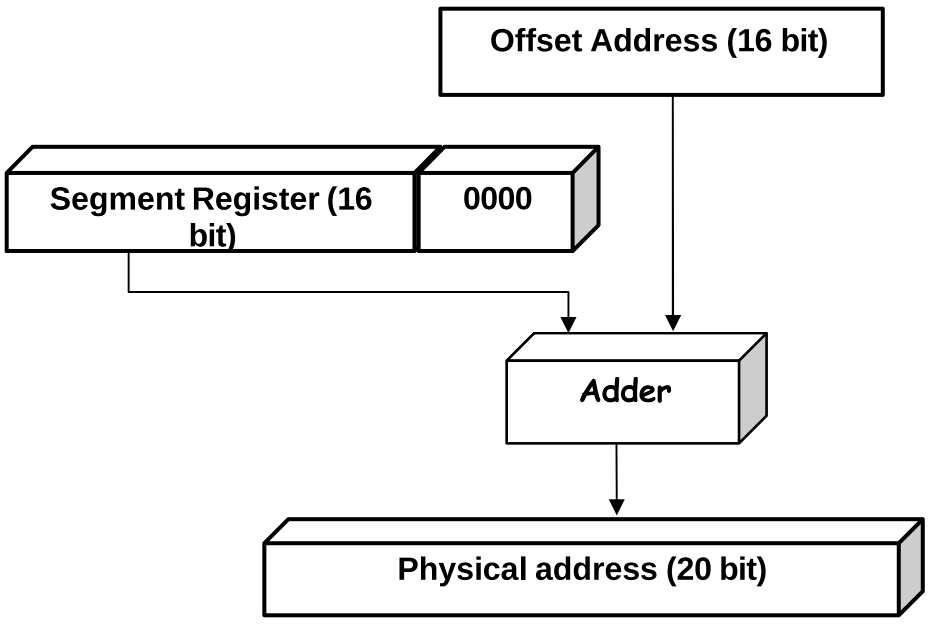


Figure 1 Fig: Physical address calculation

EX:-
Given Segment Address=3578H, Offset Address =6676H
So Physical address = [Segment Address * 10]$_H$ + Offset Address
= [3578 * 10]$_H$ + 6676H
= 35780+6676
= 3BDF6H

Types of Segments

There are four types of memory segments defined in 8086:

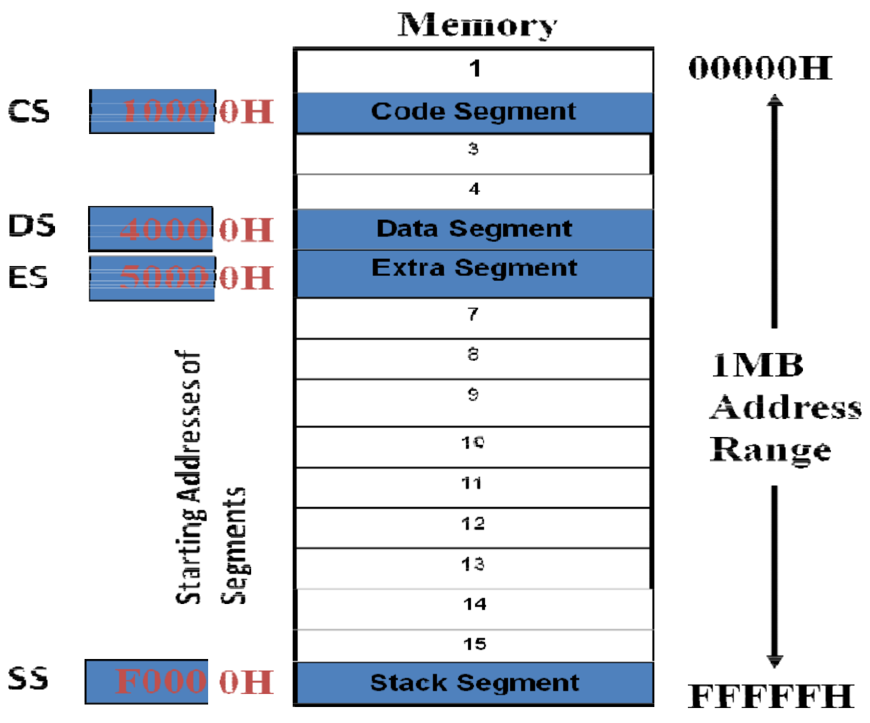
- Code segment(CS)
- Data segment (DS)
- Stack segment(SS)
- Extra segment(ES)

Code segment (CS): This segment is used to store code/program instructions.
Data and Extra segment (DS&ES): This segment is used to store data used in the program.
Stack segment (SS): This segment is used to store the stack contents.

Types of Segments Registers:

To hold the upper 16-bits of the starting address for each of the segments, there are four segment registers:

- **CS (Code Segment register)**
- **DS (Data Segment register)**
- **SS (Stack Segment register)**
- **ES (Extra Segment register)**



Advantage of memory Segmentation:

- Allows the memory capacity to be 1 Mbytes although the actual addresses to be handled are of 16-bit size.
- Allows the placing of code, data and stack portions of the same program in different parts (segments) of memory, for data and code protection.
- Permits a program and/or its data to be put into different areas of memory each time the program is executed. i.e., provision for relocation is done.

PIN Diagram

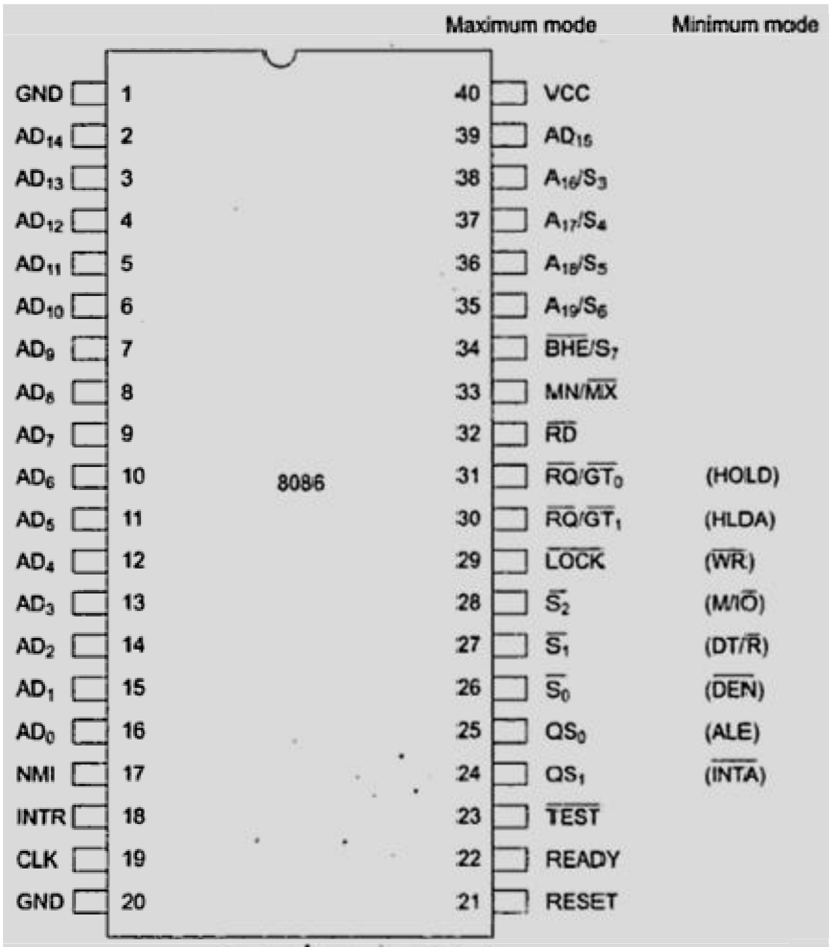


Fig. 1.5 *Pin Configuration of 8086*

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.

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.

- **A16/S3,A17/S4-**

  A16,A17 are multiplexed with segment identifier signals S3 and S4 which combinedly indicate which segment register is presently being used for memory accesses as in below fig..

S4	S3	Indication
0	0	Extra segment(ES)
0	1	Stack segment(SS)
1	0	Code segment(CS)
1	1	Data segment (DS)

- **A18/s5:** A18 is multiplexed with status S5 of the interrupt enable flag bit which is updated at the beginning of each clock cycle.
- **A19/s6:** A18 is multiplexed with status S6.

B̄H̄Ē / S7: (Bus High enable)

- 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.

B̄H̄Ē	**A0**	**Indication**
0	0	Whole Word
0	1	Upper byte from or to odd address
1	0	Lower byte from or to even address
1	1	None

$\overline{RD}$ **(Read):**

- This signal on low indicates the peripheral that the processor is performing s memory or I/O read operation. The signal is active low.

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. This signal is active high.
- enter into wait states and remain idle : READY = 0
- no effect on the operation of µ : READY = 1

INTR (Interrupt Request):

- This is a level triggered input and hardware interrupt pin.
- If any interrupt request is pending, the processor enters the interrupt acknowledge cycle. This can be internally masked by resulting the interrupt enable flag.

NMI : non-maskable interrupt

- This is a edge triggered input and hardware interrupt pin which causes Type 2 interrupt.

$\overline{TEST}$**:**

- 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.

CLK (Clock Input):

- The clock input provides the basic timing for processor operation and bus control activity.

V_{CC} (power supply) : +5.0V, ±10%

RESET:

- µ : reset if RESET is high

GND(Ground) : Two pins labeled GND(0 voltage

MN/$\overline{MX}$:

- The logic level at this pin decides whether the processor is to operate in either minimum or maximum mode.
- if **MN/$\overline{MX}$**= 1; Minimum Mode else Maximum Mode

Pin functions for the minimum mode operation of 8086

1. $M/\overline{IO}$
   - This is a status line logically equivalent to $\overline{S2}$ 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.

2. $\overline{INTA}$ **(Interrupt Acknowledge):**
   - when this signal it goes low, the processor has accepted the interrupt.
3. **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.
4. $DT/\overline{R}$ **(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.
5. $\overline{DEN}$ **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.
6. **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.
7. $\overline{WR}$ **(Write):**
   - When it is low the processor perform memory or Io write .

Pin functions for the maximum mode operation of 8086

1. $\overline{S2}$, $\overline{S1}$, $\overline{S0}$ **– Status Lines:**

- These signals are connected to 8288.These are the status lines which reflect the type of operation according to the below table, being carried out by the processor.

$\overline{S2}$	$\overline{S1}$	$\overline{S0}$	**Indication**
0	0	0	Interrupt acknowledge
0	0	1	Read I/O port
0	1	0	Write I/O port
0	1	1	Halt
0	0	0	Code access
0	0	1	Read memory
0	1	0	Write memory
0	1	1	Passive State

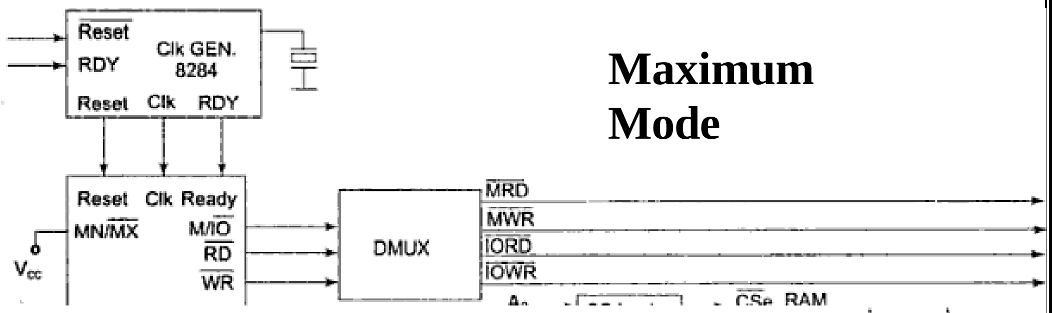
2. $\overline{LOCK}$:
    - This output pin indicates that other system bus master will be prevented from gaining the system bus, while the LOCK signal is low.
    - The LOCK signal is activated by the 'LOCK' prefix instruction and remains active until the completion of the next instruction. When the 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.
3. **QS1, QS0 (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.

QS1	**QS0**	**Indication**
0	0	No operation
0	1	First byte of opcode from the queue
1	0	Empty Queue
1	1	Subsequent byte from queue

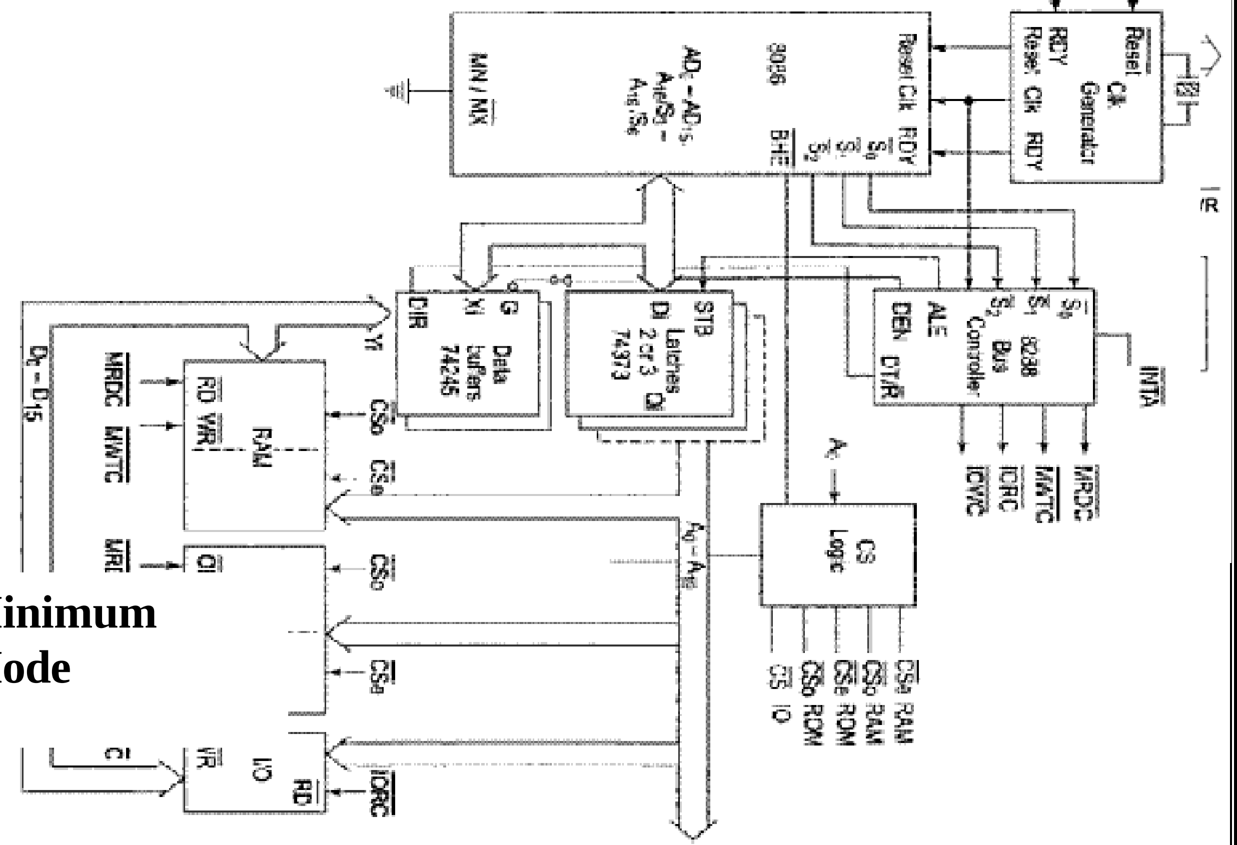
4. $\overline{RQ}/\overline{GT1}$,$\overline{RQ}/\overline{GT0}$ **(Request/Grant)**

- 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.

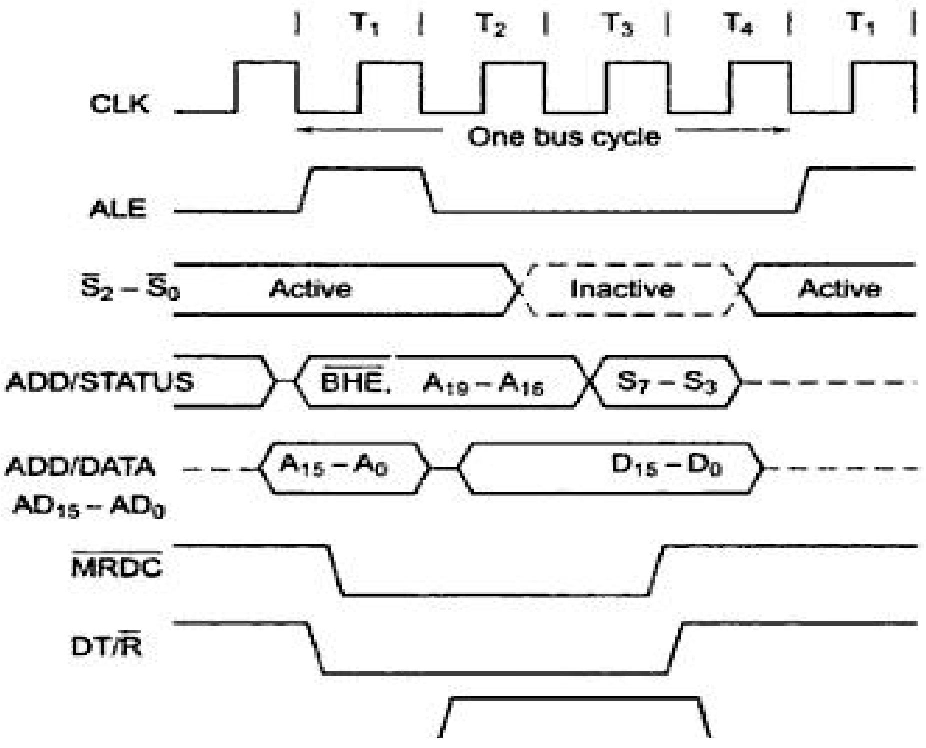
In maximum mode of operation signals like $\overline{WR}$, **ALE,** $\overline{DEN}$, **DT/**$\overline{R}$ etc are not available directly from the processor.
These signals are available from the controller 8288.



Maximum Mode



Minimum Mode



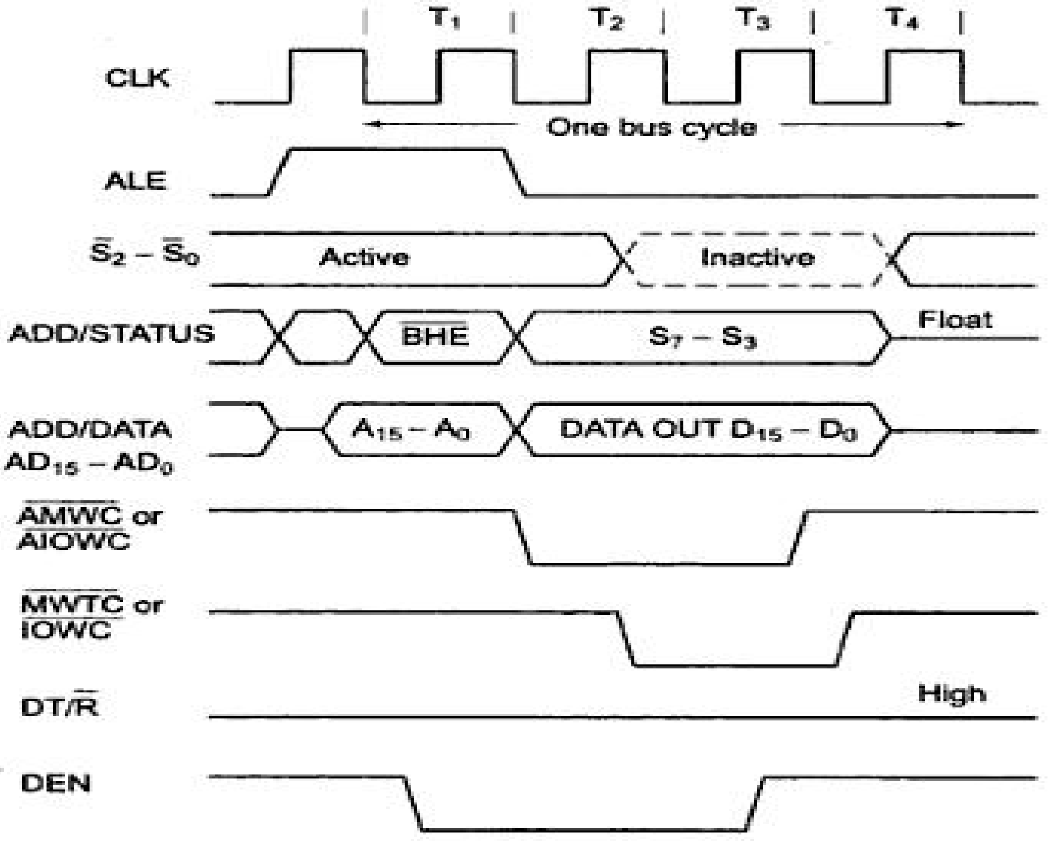


Fig. 1.16(b) *Memory Write Timing in Maximum Mode*

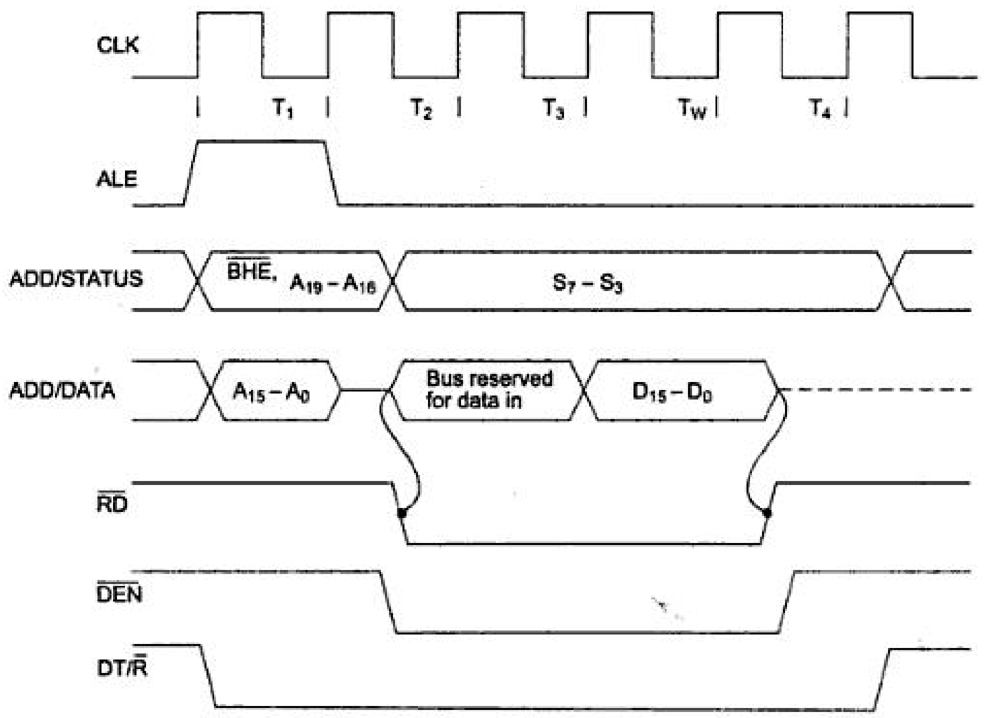


Fig.1.14(a) *Read Cycle Timing Diagram for Minimum Mode*

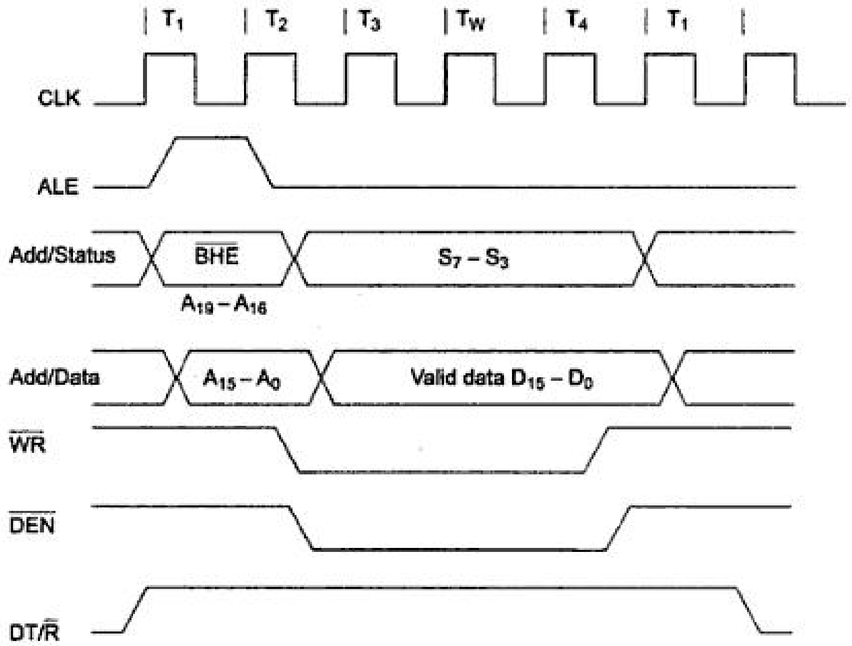


Fig. 1.14(b) *Write Cycle Timing Diagram for Minimum Mode Operation*

ADDRESSING MODES OF 8086

Addressing mode indicates a way of locating data or operands. Depending upon the data types used in the instruction and the memory addressing modes, any instruction may belong to one or more addressing modes, or some instruction may not belong to any of the addressing modes. Thus the addressing modes describe the types of operands and the way they are accessed for executing an instruction. Here, we will present the addressing modes of the instructions depending upon their types. According to the flow of instruction execution, the instructions may be categorized as

(i) Sequential control flow instructions and

(ii) Control transfer instructions.

Sequential control flow instructions are the instructions, which after execution, transfer control to the next instruction appearing immediately after it (in the sequence) in the program. For example, the arithmetic, logical, data transfer and processor control instructions are sequential control flow instructions. The control transfer instructions, on the other hand, transfer control to some predefined address somehow specified in the instruction after their execution. For example, INT, CALL, RET and JUMP instructions fall under this category.

The addressing modes for sequential control transfer instructions are explained as follows:

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

Example: MOV AX, 0005H

In the above example, 0005H is the immediate data. The immediate data may be 8-bit or 16-bit in size.

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

Example: MOV AX, [5000H]

Here, data resides in a memory location in the data segment, whose effective address may be computed using 5000H as the offset address and content of DS as segment address. The effective address, here, is 10H*DS+5000H.

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

Example: MOV BX, AX.

4. Register Indirect: Sometimes, the address of the memory location, which contains data or operand, is determined in an indirect way, using the offset registers. This mode of addressing is known as register indirect mode. In this addressing mode, the offset address of data is in either BX or SI or DI registers. The default segment is either DS or ES. The data is supposed to be available at the address pointed to by the content of any of the above registers in the default data segment.

Example: MOV AX, [BX]

Here, data is present in a memory location in DS whose offset address is in BX. The effective address of the data is given as 10H*DS+ [BX].

5. Indexed: In this addressing mode, offset of the operand is stored in one of the index registers. DS and ES are the default segments for index registers SI and DI respectively. This mode is a special case of the above discussed register indirect addressing mode.

Example: MOV AX, [SI]

Here, data is available at an offset address stored in SI in DS. The effective address, in this case, is computed as 10H*DS+ [SI].

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

Example: MOV Ax, 50H [BX]

Here, effective address is given as 10H*DS+50H+ [BX].

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

Example: MOV AX, [BX] [SI]

Here, BX is the base register and SI is the index register. The effective address is computed as 10H*DS+ [BX] + [SI].

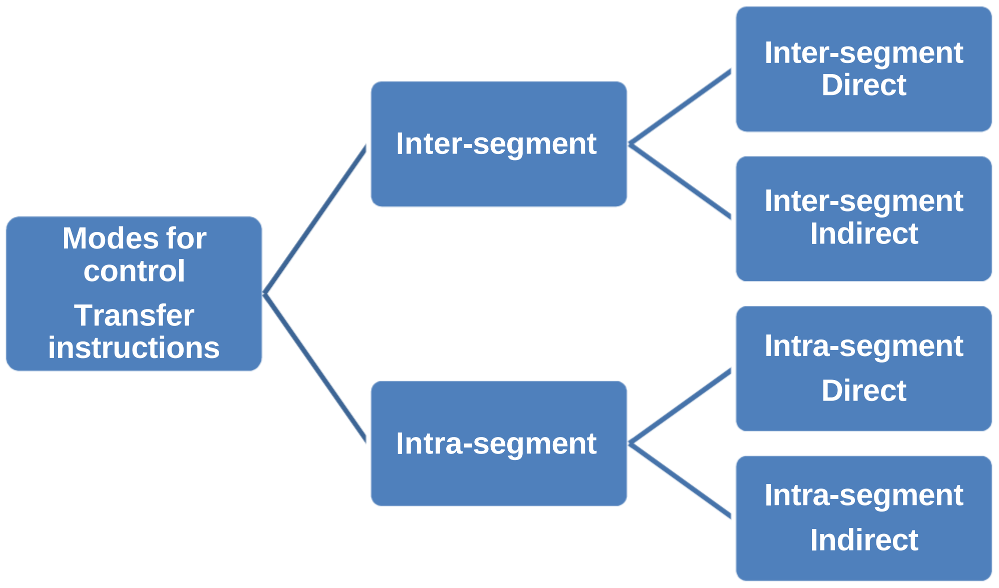
8. **Relative Based Indexed:** The effective address is formed by adding an 8-bit or 16-bit displacement with the sum of contents of any one of the bases registers (BX or BP) and any one of the index registers, in a default segment.

Example: MOV AX, 50H [BX] [SI]

Here, 50H is an immediate displacement, BX is a base register and SI is an index register. The effective address of data is computed as 160H*DS+ [BX] + [SI] + 50H.

For the control transfer instructions, the addressing modes depend upon whether the destination location is within the same segment or a different one. It also depends upon the method of passing the destination address to the processor. Basically, there are two addressing modes for the control transfer instructions, viz. inter-segment and intra-segment addressing modes.

If the location to which the control is to be transferred lies in a different segment other than the current one, the mode is called inter-segment mode. If the destination location lies in the same segment, the mode is called intra-segment.



ADDRESSING MODES FOR CONTROL TRANSFER INSTRUCTION

1. **Intra-segment direct mode:** In this mode, the address to which the control is to be transferred lies in the same segment in which the control transfer instruction lies and appears directly in the instruction as an immediate displacement value. In this addressing mode, the displacement is computed relative to the content of the instruction pointer IP.

The effective address to which the control will be transferred is given by the sum of 8 or 16 bit displacement and current content of IP. In case of jump instruction, if the signed displacement (d) is of 8 bits (i.e. $–128<d<+128$), we term it as short jump and if it is of

16 bits (i.e. $–32768<+32768$), it is termed as long jump.

2. **Intra-segment Indirect Mode:** In this mode, the displacement to which the control is to be transferred, is in the same segment in which the control transfer instruction lies, but it is passed to the instruction indirectly. Here, the branch address is found as the content of a register or a memory location. This addressing mode may be used in unconditional branch instructions.

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

4. **Inter-segment Indirect Mode:** In this mode, the address to which the control is to be transferred lies in a different segment and it is passed to the instruction indirectly, i.e. contents of a memory block containing four bytes, i.e. IP (LSB), IP (MSB), CS (LSB) and CS (MSB) sequentially. The starting address of the memory block may be referred using any of the addressing modes, except immediate mode.

8086 Instruction Set and Assembler Directives

The 8086 microprocessor supports 6 types of Instructions. They are

1. Data transfer instructions

2. Arithmetic instructions

3. Bit manipulation instructions

4. String instructions

5. Program Execution Transfer instructions (Branch & loop Instructions)

6. Processor control instructions

1. Data Transfer instructions: These instructions are used to transfer the data from source operand to destination operand. All the store, move, load, exchange, input and output instructions belong to this group.

General purpose byte or word transfer instructions:

MOV **:** Copy byte or word from specified source to specified destination

PUSH **:** Push the specified word to top of the stack

POP **:** Pop the word from top of the stack to the specified location

PUSHA **:** Push all registers to the stack

POPA **:** Pop the words from stack to all registers

XCHG **:** Exchange the contents of the specified source and destination operands one of which may be a register or memory location.

XLAT : Translate a byte in AL using a table in memory

Simple input and output port transfer instructions

1. IN : Reads a byte or word from specified port to the accumulator
2. OUT : Sends out a byte or word from accumulator to a specified port

Special address transfer instructions

1. LEA : Load effective address of operand into specified register
2. LDS : Load DS register and other specified register from memory
3. LES : Load ES register and other specified register from memory.

Flag transfer registers

1. LAHF : Load AH with the low byte of the flag register
2. SAHF : Store AH register to low byte of flag register
3. PUSHF : Copy flag register to top of the stack
4. POPF : Copy word at top of the stack to flag register

2. Arithmetic instructions : These instructions are used to perform various mathematical operations like addition, subtraction, multiplication and division etc….

Addition instructions

1. ADD **:** Add specified byte to byte or word to word
2. ADC **:** Add with carry
3. INC **:** Increment specified byte or specified word by 1
4. AAA **:** ASCII adjust after addition
5. DAA **:** Decimal (BCD) adjust after addition

Subtraction instructions

1. SUB : Subtract byte from byte or word from word
2. SBB : Subtract with borrow
3. DEC : Decrement specified byte or word by 1
4. NEG : Negate or invert each bit of a specified byte or word and add 1(2's complement)
5. CMP : Compare two specified byte or two specified words
6. AAS : ASCII adjust after subtraction
7. DAS : Decimal adjust after subtraction

Multiplication instructions

1. MUL **:** Multiply unsigned byte by byte or unsigned word or word.
2. IMUL **:** Multiply signed bye by byte or signed word by word
3. AAM **:** ASCII adjust after multiplication

Division instructions

1. DIV : Divide unsigned word by byte or unsigned double word by word
2. IDIV : Divide signed word by byte or signed double word by word
3. AAD : ASCII adjust after division

4. CBW : Fill upper byte of word with copies of sign bit of lower byte
5. CWD : Fill upper word of double word with sign bit of lower word.

3. Bit Manipulation instructions : These instructions include logical , shift and rotate instructions in which a bit of the data is involved.

Logical instructions

1. NOT :Invert each bit of a byte or word.
2. AND : ANDing each bit in a byte or word with the corresponding bit in another byte or word.
3. OR : ORing each bit in a byte or word with the corresponding bit in another byte or word.
4. XOR : Exclusive OR each bit in a byte or word with the corresponding bit in another byte or word.
5. TEST :AND operands to update flags, but don't change operands.

Shift instructions

1. SHL/SAL : Shift bits of a word or byte left, put zero(S) in LSBs.
2. SHR : Shift bits of a word or byte right, put zero(S) in MSBs.
3. SAR : Shift bits of a word or byte right, copy old MSB into new MSB.

Rotate instructions

1. ROL : Rotate bits of byte or word left, MSB to LSB and to Carry Flag [CF]
2. ROR : Rotate bits of byte or word right, LSB to MSB and to Carry Flag [CF]
3. RCR :Rotate bits of byte or word right, LSB TO CF and CF to MSB
4. RCL :Rotate bits of byte or word left, MSB TO CF and CF to LSB

4. String instructions

A string is a series of bytes or a series of words in sequential memory locations. A string often consists of ASCII character codes.

1. REP : An instruction prefix. Repeat following instruction until CX=0
2. REPE/REPZ : Repeat following instruction until CX=0 or zero flag ZF=1
3. REPNE/REPNZ : Repeat following instruction until CX=0 or zero flag ZF=1
4. MOVS/MOVSB/MOVSW: Move byte or word from one string to another
5. COMS/COMPSB/COMPSW: Compare two string bytes or two string words
6. INS/INSB/INSW: Input string byte or word from port
7. OUTS/OUTSB/OUTSW : Output string byte or word to port
8. SCAS/SCASB/SCASW: Scan a string. Compare a string byte with a byte in AL or a string word with a word in AX
9. LODS/LODSB/LODSW: Load string byte in to AL or string word into AX

5. Program Execution Transfer instructions

These instructions are similar to branching or looping instructions. These instructions include conditional & unconditional jump or loop instructions.

Unconditional transfer instructions

1. CALL : Call a procedure, save return address on stack
2. RET : Return from procedure to the main program.
3. JMP : Goto specified address to get next instruction

Conditional transfer instructions

1. JA/JNBE : Jump if above / jump if not below or equal
2. JAE/JNB : Jump if above /jump if not below
3. JBE/JNA : Jump if below or equal/ Jump if not above
4. JC : jump if carry flag CF=1
5. JE/JZ : jump if equal/jump if zero flag ZF=1
6. JG/JNLE : Jump if greater/ jump if not less than or equal
7. JGE/JNL : jump if greater than or equal/ jump if not less than
8. JL/JNGE : jump if less than/ jump if not greater than or equal
9. JLE/JNG : jump if less than or equal/ jump if not greater than
10. JNC : jump if no carry (CF=0)

11. JNE/JNZ : jump if not equal/ jump if not zero(ZF=0)
12. JNO : jump if no overflow(OF=0)
13. JNP/JPO : jump if not parity/ jump if parity odd(PF=0)
14. JNS : jump if not sign(SF=0)
15. JO : jump if overflow flag(OF=1)
16. JP/JPE : jump if parity/jump if parity even(PF=1)
17. JS : jump if sign(SF=1)

6. Iteration control instructions

These instructions are used to execute a series of instructions for certain number of times.

1. LOOP :Loop through a sequence of instructions until CX=0
2. LOOPE/LOOPZ : Loop through a sequence of instructions while ZF=1 and CX = 0
3. LOOPNE/LOOPNZ : Loop through a sequence of instructions while ZF=0 and CX =0
4. JCXZ : jump to specified address if CX=0

7. Interrupt instructions

1. INT : Interrupt program execution, call service procedure
2. INTO : Interrupt program execution if OF=1
3. IRET : Return from interrupt service procedure to main program

8.High level language interface instructions

1. ENTER : enter procedure
2. LEAVE :Leave procedure
3. BOUND : Check if effective address within specified array bounds

9.Processor control instructions

Flag set/clear instructions

1. STC : Set carry flag CF to 1
2. CLC : Clear carry flag CF to 0
3. CMC : Complement the state of the carry flag CF
4. STD : Set direction flag DF to 1 (decrement string pointers)
5. CLD : Clear direction flag DF to 0
6. STI : Set interrupt enable flag to 1(enable INTR input)
7. CLI : Clear interrupt enable Flag to 0 (disable INTR input)

10. External Hardware synchronization instructions

1. HLT : Halt (do nothing) until interrupt or reset
2. WAIT : Wait (Do nothing) until signal on the test pin is low
3. ESC : Escape to external coprocessor such as 8087 or 8089
4. LOCK : An instruction prefix. Prevents another processor from taking the bus while the adjacent instruction executes.

11. No operation instruction

1. NOP : No action except fetch and decode

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 number 25

AAD ; 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 - CMP destination, 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 offset 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 will

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 by subtracting 4371H ; from AX

JA RUN_PRESS ; Jump to label RUN_PRESS if ; AX above 4371H

(2) 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

MOV SUM, AL ; else put the result in memory ; 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.

Example:

NEG AL ; Replace number in AL with its 2's complement

NEG BX ; Replace word in BX with its 2's complement

NEG BYTE PTR[BX]; Replace byte at offset BX in

; DS with its 2's complement

- **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)

; AL = 00001110 = + 14 decimal, CF = 1

(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/** - Scan string byte or a
- **SCASW** Instruction 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

; rotate 3 bits to right and keep 3 0's in MSB

(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.

XCHG op1, op2

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 :

Assembler directives are the directions to the assembler which indicate how an operand or section of the program is to be processed. These are also called pseudo operations which are not executable by the microprocessor. The various directives are explained below.

1. ASSUME : The ASSUME directive is used to inform the assembler the name of the logical segment it should use for a specified segment.

Ex: ASSUME DS: DATA tells the assembler that for any program instruction which refers to the data segment ,it should use the logical segment called DATA.

2.DB -Define byte. It is used to declare a byte variable or set aside one or more storage locations of type byte in memory.

For example, CURRENT_VALUE DB 36H tells the assembler to reserve 1 byte of memory for a variable named CURRENT_ VALUE and to put the value 36 H in that memory location when the program is loaded into RAM .

3. DW -Define word. It tells the assembler to define a variable of type word or to reserve storage locations of type word in memory.

4. DD(define double word) :This directive is used to declare a variable of type double word or restore memory locations which can be accessed as type double word.

5.DQ (define quadword) : This directive is used to tell the assembler to declare a variable 4 words in length or to reserve 4 words of storage in memory .

6.DT (define ten bytes): It is used to inform the assembler to define a variable which is **10** bytes in length or to reserve 10 bytes of storage in memory.

7. EQU –Equate It is used to give a name to some value or symbol**.** Every time the assembler finds the given name in the program, it will replace the name with the value or symbol we have equated with that name

8.ORG -Originate : The ORG statement changes the starting offset address of the data.

It allows to set the location counter to a desired value at any point in the program.For example the statement ORG 3000H tells the assembler to set the location counter to 3000H.

9 .PROC- Procedure: It is used to identify the start of a procedure or subroutine.

10. END- End program .This directive indicates the assembler that this is the end of the program module.The assembler ignores any statements after an END directive.

11. ENDP- End procedure: It indicates the end of the procedure (subroutine) to the assembler.

12.ENDS-End Segment: This directive is used with the name of the segment to indicate the end of that logical segment.

Ex: CODE SEGMENT : Start of logical segment containing code

CODE ENDS : End of the segment named CODE.

Basic Peripherals and Their Interfacing with 8086

Interfacing with RAM And ROM

The **figure 2.1** shows a general block diagram of an 8086 memory array. In this, the 16-bit word memory is partitioned into high and low 8-bit banks on the upper halves of the data bus selected by BHE, and AO.

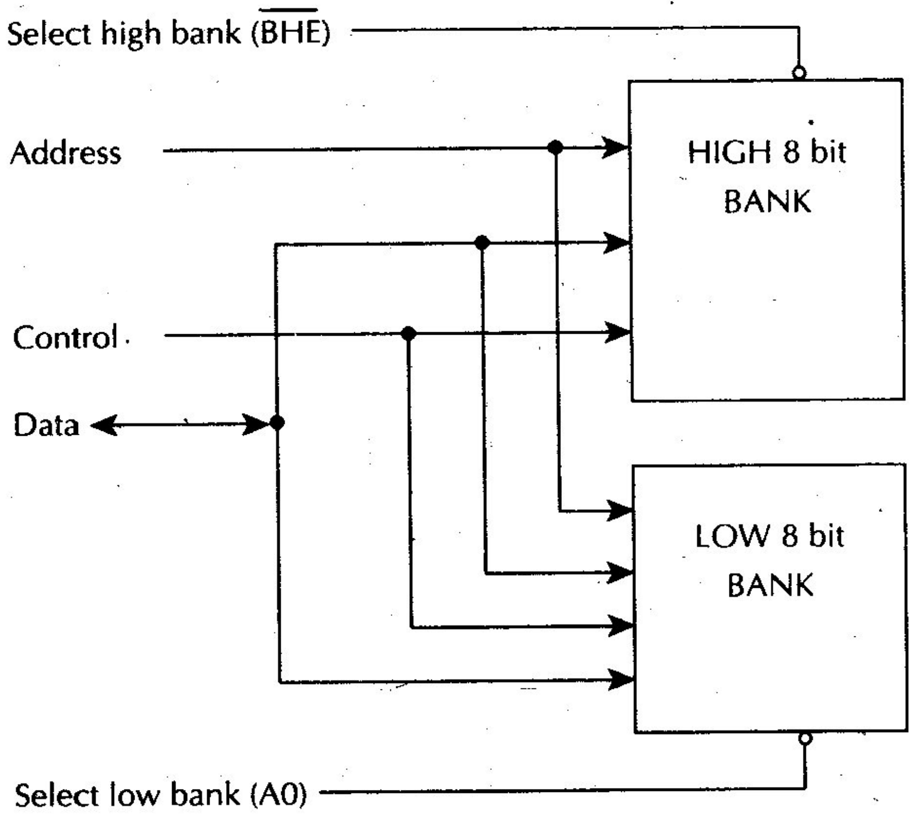


FIGURE 2.1– 8086 MEMORY ARRAY

a) ROM and EPROM

ROMS and EPROMs are the simplest memory chips to interface to the 8086. Since ROMs and EPROMs are read-only devices, A0 and BHE are not required to be part of the chip enable/select decoding. The 8086 address lines must be connected to the ROM/EPROM chip chips starting with A1 and higher to all the address lines of the ROM/EPROM chips. The 8086 unused address lines can be used as chip enable/select decoding. To interface the ROMs/RAMs directly to the 8086-multiplexed bus, they must have output enable signals. The **figure 3.5.2** shows the 8086 interfaced to two 2716s.

Byte accesses are obtained by reading the full 16-bit word onto the bus with the 8086 discarding the unwanted byte and accepting the desired byte.

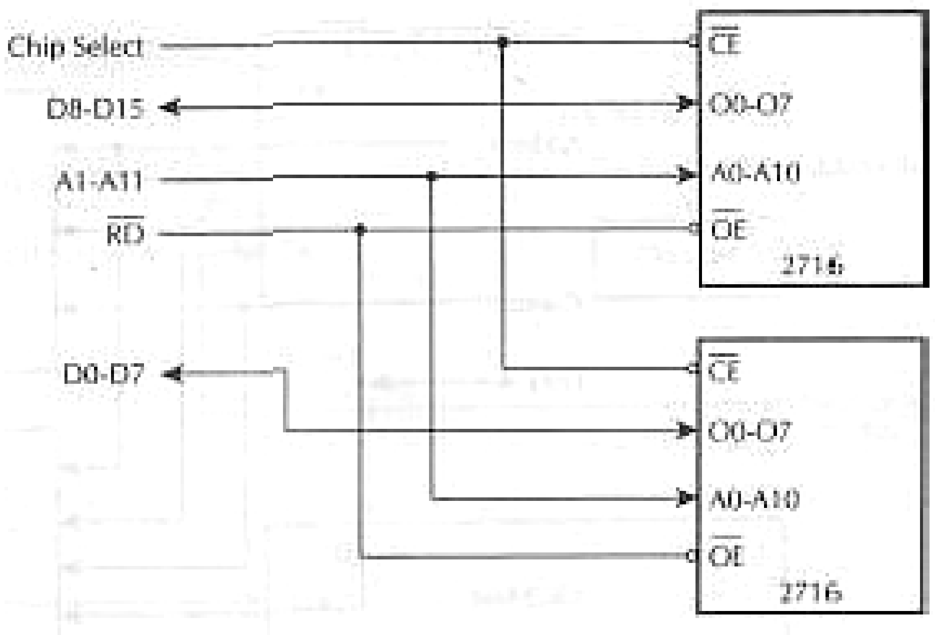


FIGURE 3.5.2

b) Static RAMS

Since static RAMs are read/write memories, both A0 and BHE must be included in the chip select/enable decoding of the devices and write timing must be considered in the compatibility analysis.

For each static RAM, the memory data lines must be connected to either the upper half AD15-AD0 or lower half AD7-AD0 of the 8086 data lines.

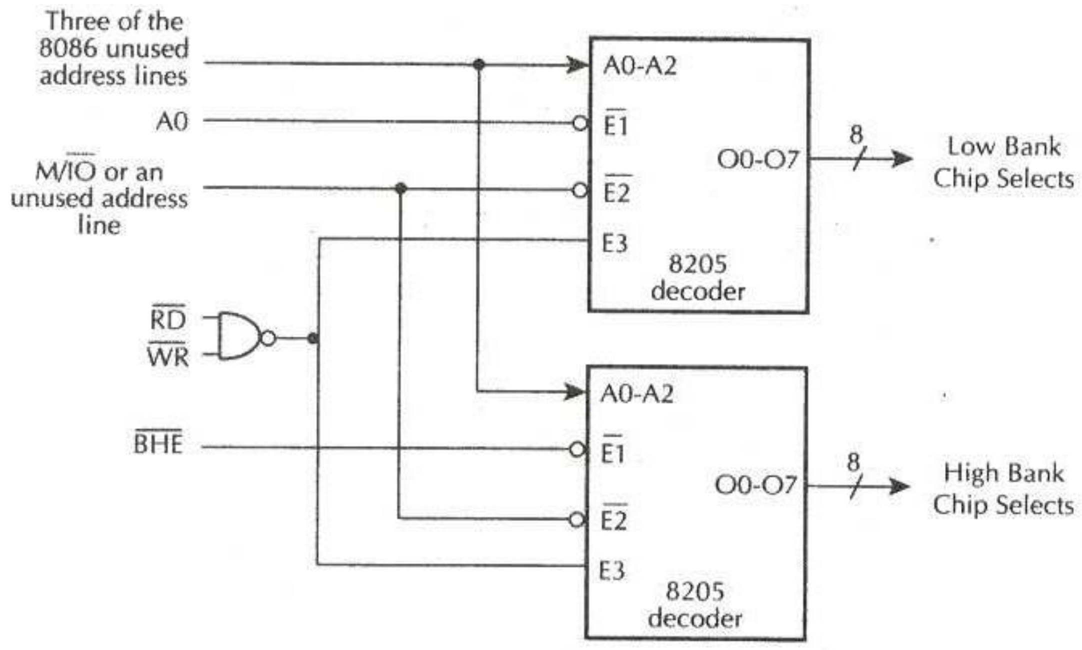
For static RAMs without output enable pins, read and write lines must be used as enables for chip select generation to avoid bus contention. If read and write lines are not used to activate the chip selects, static RAMs with common input/output data pins such as 2114 will face extreme bus contentions between chip selects and write active. The 8086 A0 and BHE pins must be used to enable the chip the chip selects. Note that Intel 8205 has three enables E1, E2, and E3, three inputs A0 and A2, and eight outputs O0- O7.

For devices with output enables such as 2142, one way to generate chip selects for the static RAMs is by gating the 8086 WR signal with BHE and A0 to provide upper and lower bank write strobes. A possible configuration is shown in the **figure 3.5.4**. Since the Intel 2142 is a 1024 * 4 bit static RAM, two chips for each bank with a total of 4 chips for 2K * 8 static RAM is required. Note that DATA is read from the 2142 when the output disable OD is low, WE is HIGH, and DATA is written into 2142. If multiple chip selects are available with the static RAM, BHE and A0 may be used directly as the chip selects. A possible configuration for 2K * 8 array is shown in the **figure 3.5.5**.

c) Dynamic RAM

Dynamic RAM store information as charges in capacitors. Since

capacitors can hold charges for a few milliseconds, refresh circuitry is necessary in dynamic RAMs for retaining these charges. Therefore, dynamic RAMs are complex devices to design a system. To relieve the designer of most of these complicated interfacing tasks, Intel provides the 8202 dynamic RAM controller as part of the 8086 families of peripheral devices. The 8202 can be interfaced with the 8086 to build a dynamic memory system.



put enables.

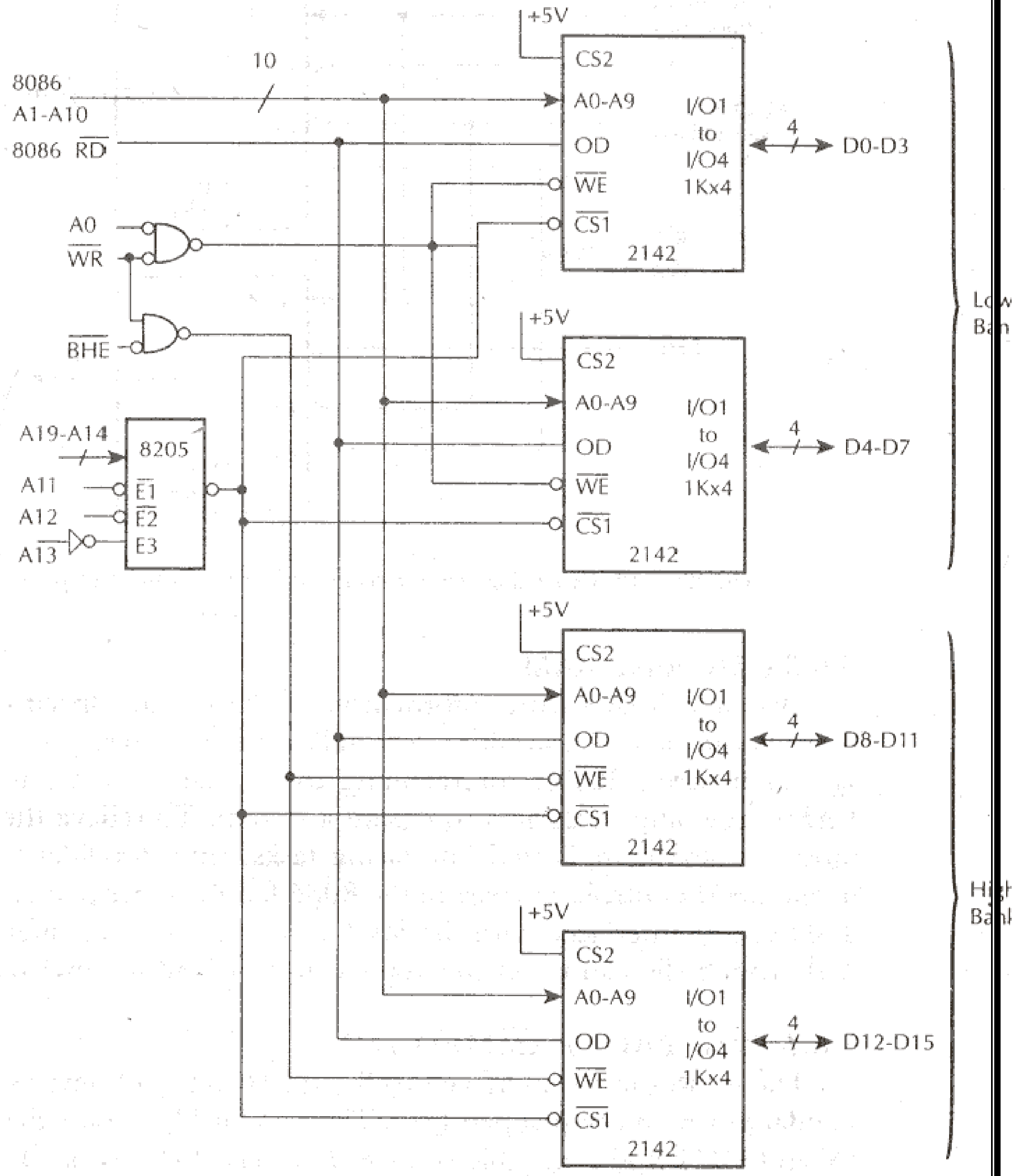
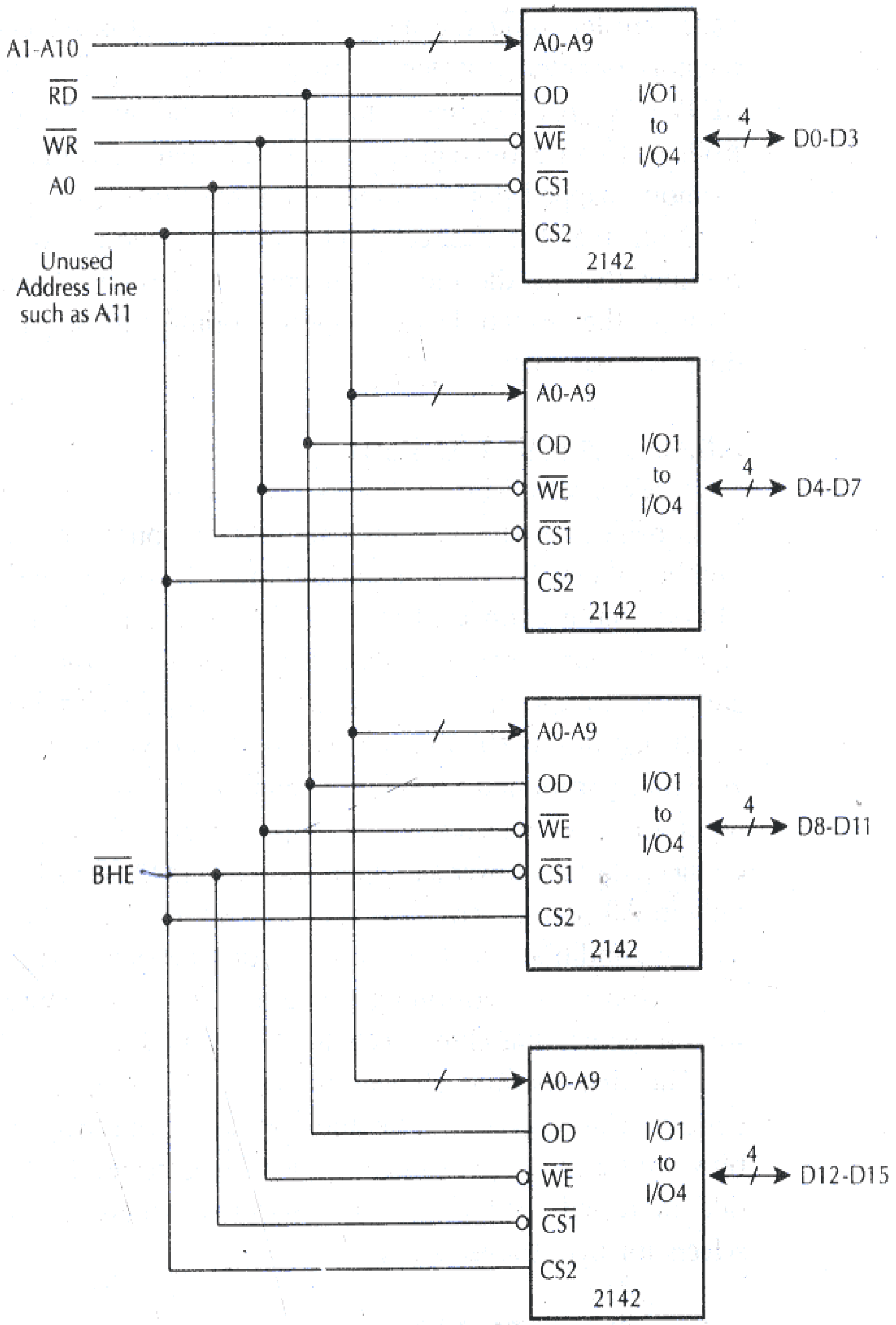


FIGURE 3.5.5 – 2K * 8 STATIC ARRAY WITH A0 and BHE AS DIRECT CHIP SELECT INPUTS



A1-A10
RD
WR
A0
Unused Address Line such as A11
BHE
A0-A9
OD
WE
CS1
CS2
2142
I/O1 to I/O4
4
D0-D3
D4-D7
D8-D11
D12-D15

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 PA0-PA7 while the port C lines are identified as PC4-PC7. 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.
- $\overline{RD}$, $\overline{WR}$, A1, A0 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 :
- **PA7-PA0**: 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.
- **PC7-PC4** : 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.
- **PC3-PC0** : These are the lower port C lines, other details are the same as PC7-PC4 lines.

- **PB0-PB7** : 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.
- **$\overline{RD}$** : This is the input line driven by the microprocessor and should be low to indicate read operation to 8255.
- **$\overline{WR}$** : This is an input line driven by the microprocessor. A low on this line indicates write operation.
- **$\overline{CS}$** : This is a chip select line. If this line goes low, it enables the 8255 to respond to $\overline{RD}$ and $\overline{WR}$ signals, otherwise $\overline{RD}$ and $\overline{WR}$ signal are neglected.
- **A1-A0** : These are the address input lines and are driven by the microprocessor.

  These lines A1-A0 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 A0 and A1 pins of 8255 are connected with A1 and A2 respectively.
- **D0-D7** : 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) $\overline{CS}$ – 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) $\overline{WR}$ (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) **A0** and **A1** (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 (PC7 –PC4)
- 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 output 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.

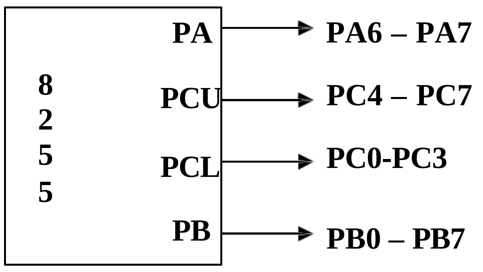
Modes of Operation of 8255

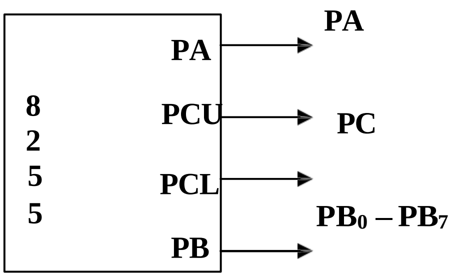
- These are two basic modes of operation of 8255. I/O mode and Bit Set-Reset mode (BSR).
- In I/O mode, the 8255 ports work as programmable I/O ports, while in BSR mode only port C (PC0-PC7) can be used to set or reset its individual port bits.
- Under the I/O mode of operation, further there are three 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 initialization.

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

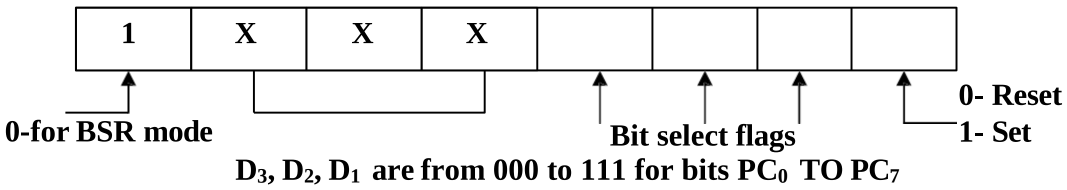
Port A and Port C acting as O/P. Port B acting as I/P

Mode 0

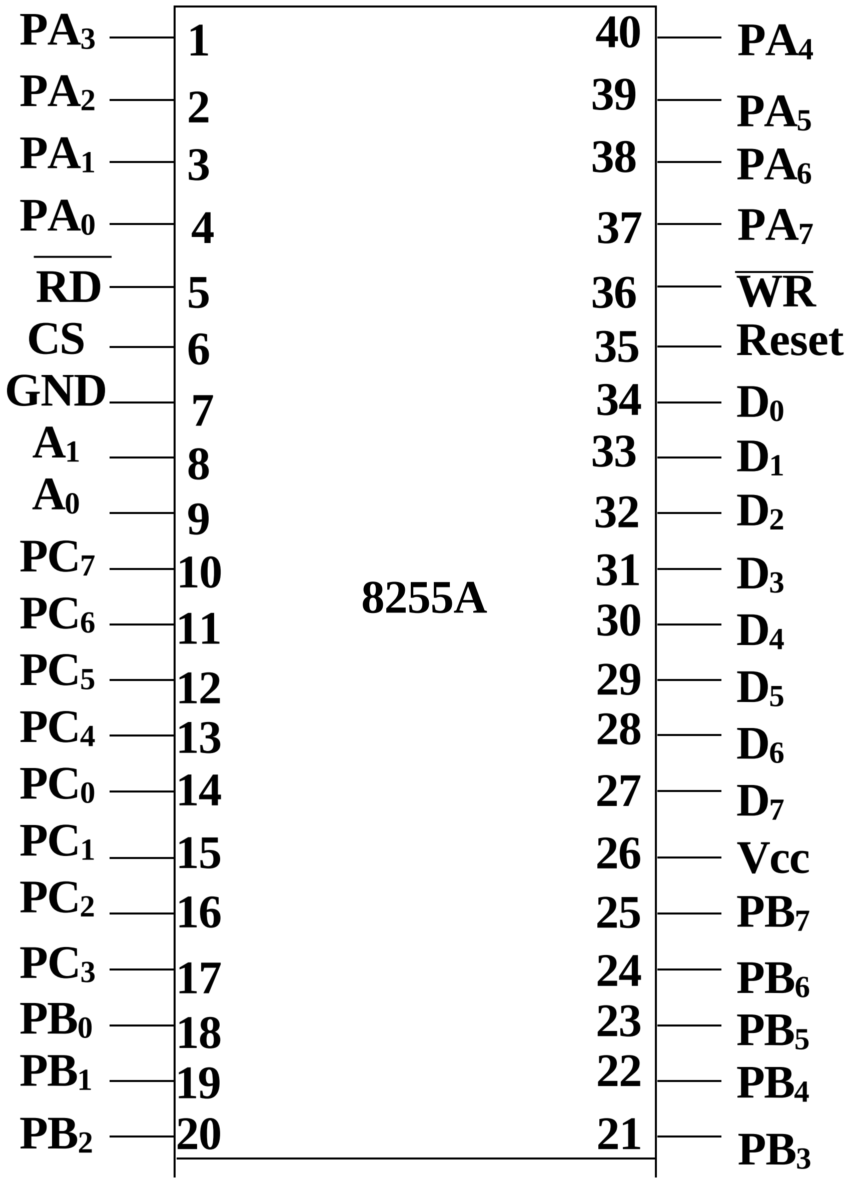
- The salient features of this mode are as listed below:

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.

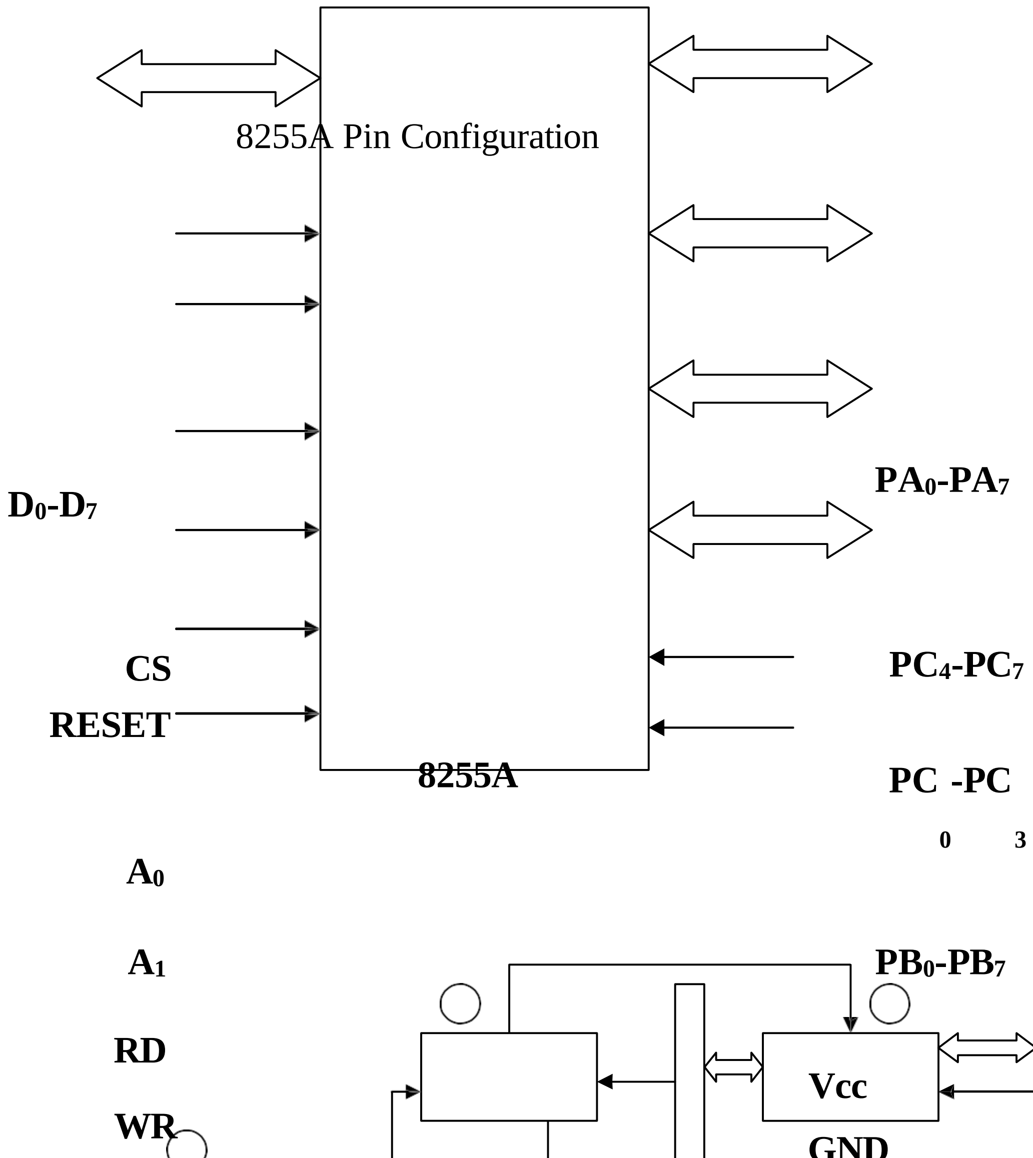
- All these modes can be selected by programming a register internal to 8255 known as CWR.
- The control word register has two formats. The first format is valid for I/O modes of operation, i.e. modes 0, mode 1 and mode 2 while the second format is valid for bit set/reset (BSR) mode of operation. These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



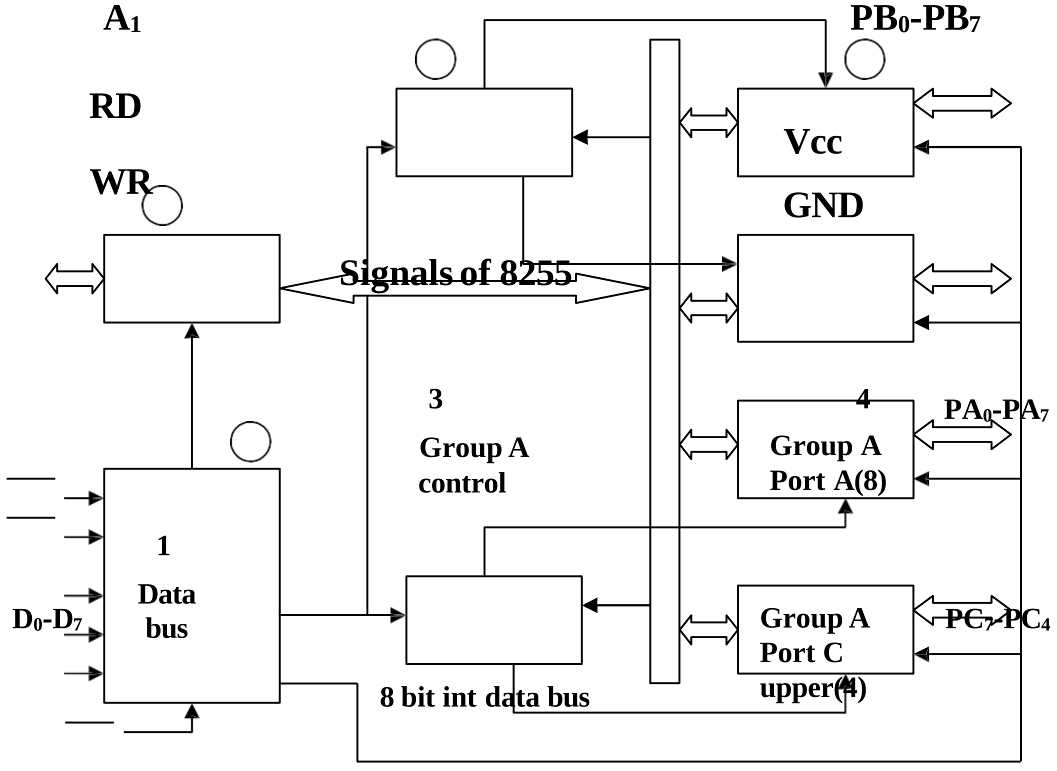
I/O Mode Control Word Register Format and
BSR Mode Control Word Register Format



8255A Pin Configuration



Signals of 8255



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:

1. Two groups – group A and group B are available for strobed data transfer.
2. Each group contains one 8-bit data I/O port and one 4-bit control/data port.
3. The 8-bit data port can be either used as input and output port. The inputs and outputs both are latched.
4. 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):

- $\overline{\text{STB}}$ (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 STB and is reset by the rising edge of 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 WR signal and reset by a low going edge at the ACK input.
- $\overline{\text{ACK}}$ (Acknowledge input) – $\overline{\text{ACK}}$ signal acts as an

acknowledgement to be given by an output device. 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.

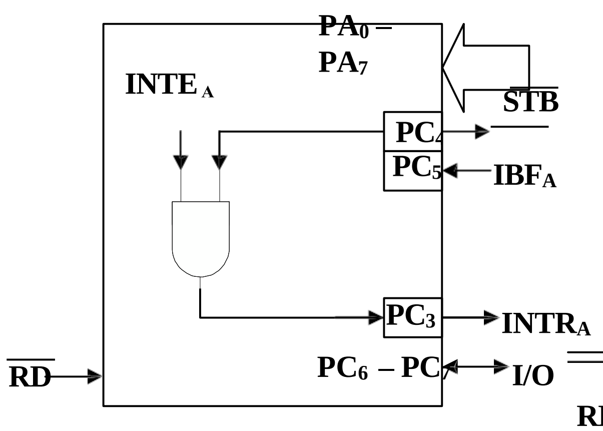
- **INTR** (Interrupt request) – Thus an output signal that can be used to interrupt the CPU when an output device acknowledges the data received from the CPU. INTR is set when ACK, OBF and INTE are 1. It is reset by a falling edge on WR input. The INTEA and INTEB flags are controlled by the bit set-reset mode of PC6 and PC2 respectively.

Input control signal definitions in Mode 1

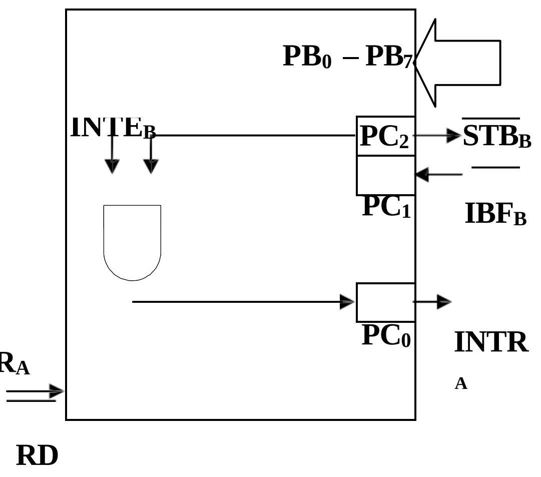
1	**0**	**1**	**0**	**1/0**	**X**	**X**	**X**
D_7	**D_6**	**D_5**	**D_4**	**D_3**	**D_2**	**D_1**	**D_0**

1 - Input
0 - Output
For PC_6 – PC_7

1	**X**	**X**	**X**	**X**	**1**	**1**	**X**
D_7	**D_6**	**D_5**	**D_4**	**D_3**	**D_2**	**D_1**	**D_0**



Mode 1 Control Word Group A I/P



Mode 1 Control Word Group B I/P

Programmable Interval Timer 8253

- The Intel 8253 is a programmable counter / timer chip designed for use as an Intel microcomputer peripheral. It uses nMOS technology with a single +5V supply and is packaged in a 24-pin plastic DIP.
- It is organized as 3 independent 16-bit counters, each with a counter rate up to 2 MHz. All modes of operation are software programmable.
- The 82C54 is pin compatible with the HMOS 8254, and is a superset of the 8253.
- Six programmable timer modes allow the 82C54 / 8253 to be used as an event counter, elapsed time indicator, programmable one-shot, and in many other applications.

Block diagram

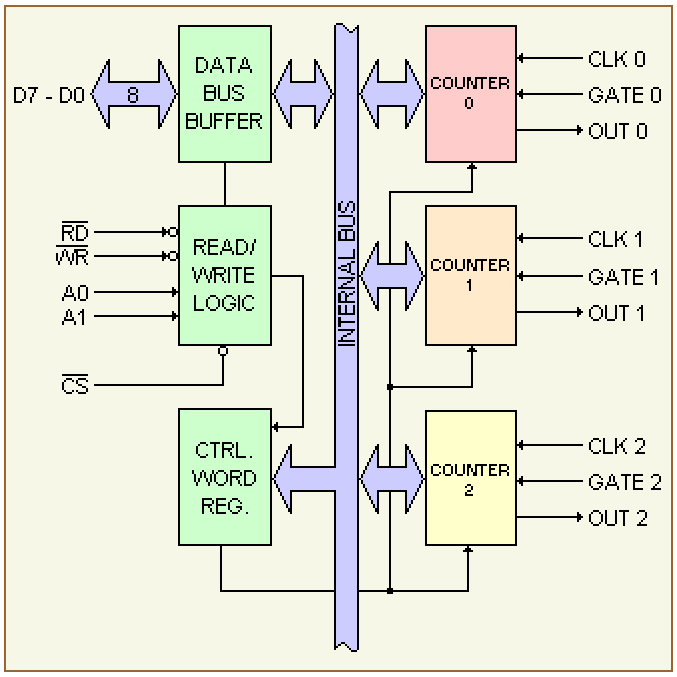


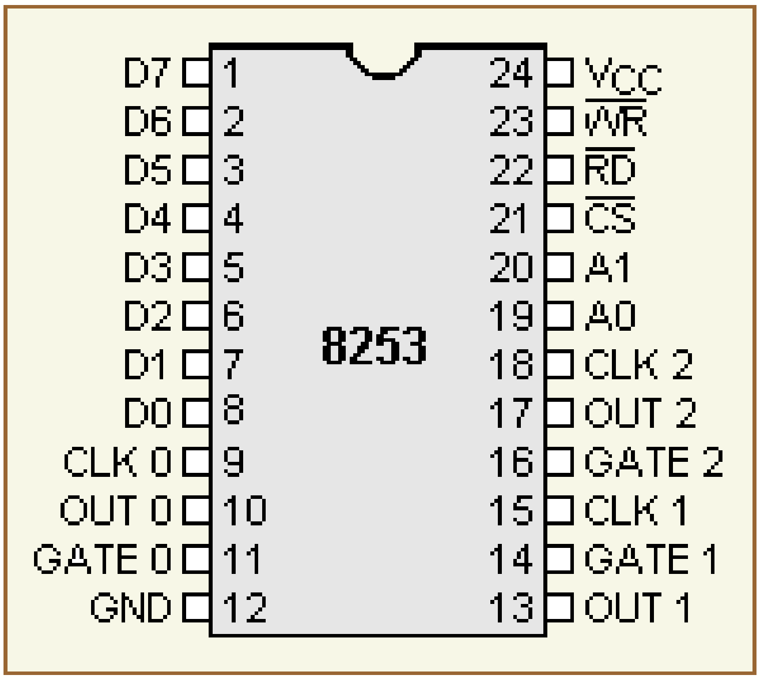
Fig: Block diagram of an 8253 programmable interval timer

The block labeled *data bus buffer* contains the logic to buffer the data bus to / from the microprocessor, and to the internal registers. The block labeled *read / write logic* controls the reading and the writing of the counter registers. The final block, the *control word register*, contains the programmed information that is sent to the device from the microprocessor. In effect this register defines how the 8253 logically works. The timer has three independent, programmable counters and they are all identical

Each counter in the block diagram has 3 logical lines connected to it. Two of these lines, clock and gate, are inputs. The third, labeled OUT is an output. The function of these lines changes and depends on how the device is initialized or programmed.

PIN configuration

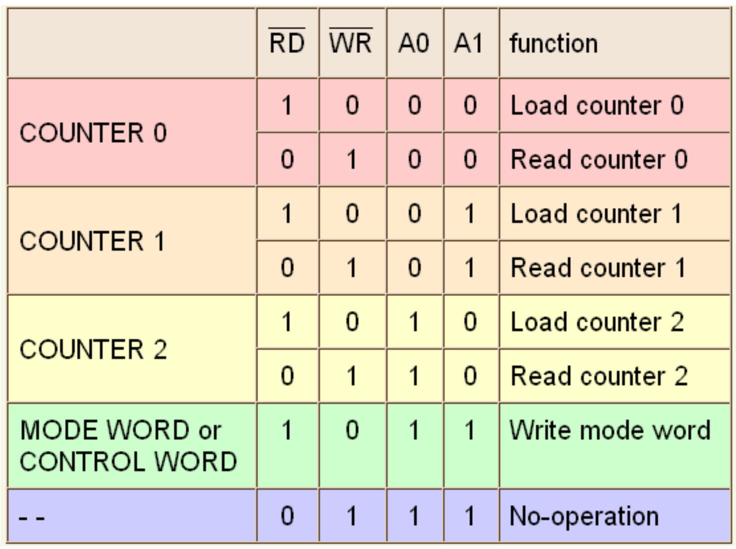
The following picture shows the pin configuration of the 8253 and a general definition of the lines follows:



- **Clock** This is the clock input for the counter. The counter is 16 bits. The maximum clock frequency is 1 / 380 nanoseconds or 2.6 megahertz. The minimum clock frequency is DC or static operation.
- **Out** This single output line is the signal that is the final programmed output of the device. Actual operation of the outline depends on how the device has been programmed.
- **Gate** This input can act as a gate for the clock input line, or it can act as a start pulse, depending on the programmed mode of the counter.

Internal 8253 register

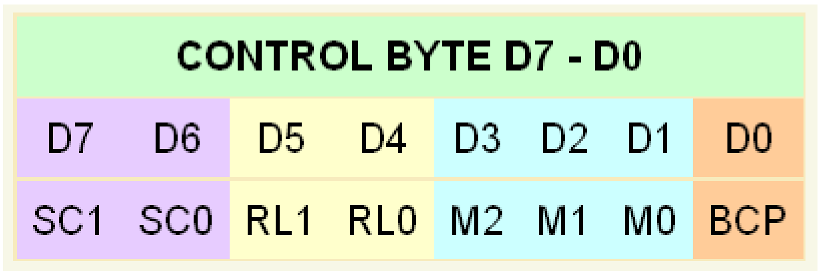
Here is a list of the internal 8253 registers that will program the internal counters of the 8253:



	$\overline{RD}$	$\overline{WR}$	A0	A1	function
COUNTER 0	1	0	0	0	Load counter 0
	0	1	0	0	Read counter 0
COUNTER 1	1	0	0	1	Load counter 1
	0	1	0	1	Read counter 1
COUNTER 2	1	0	1	0	Load counter 2
	0	1	1	0	Read counter 2
MODE WORD or CONTROL WORD	1	0	1	1	Write mode word
- -	0	1	1	1	No-operation

Counter #0, #1, #2 :Each counter is identical, and each consists of a 16-bit, pre-settable, down counter. Each is fully independent and can be easily read by the CPU. When the counter is read, the data within the counter will not be disturbed. This allows the system or your own program to monitor the counter's value at any time, without disrupting the overall function of the 8253.

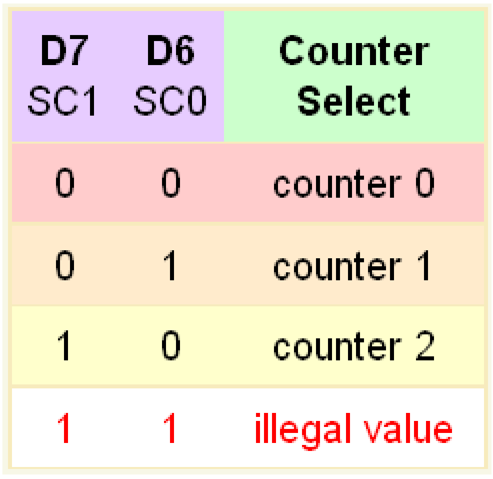
Control Word Register This internal register is used to write information to, prior to using the device. This register is addressed when A0 and A1 inputs are logical 1's. The data in the register controls the operation mode and the selection of either binary or BCD (binary coded decimal) counting format. The register can only be written to. You can't read information from the register.



CONTROL BYTE D7 - D0							
D7	D6	D5	D4	D3	D2	D1	D0
SC1	SC0	RL1	RL0	M2	M1	M0	BCP

Control Word Register

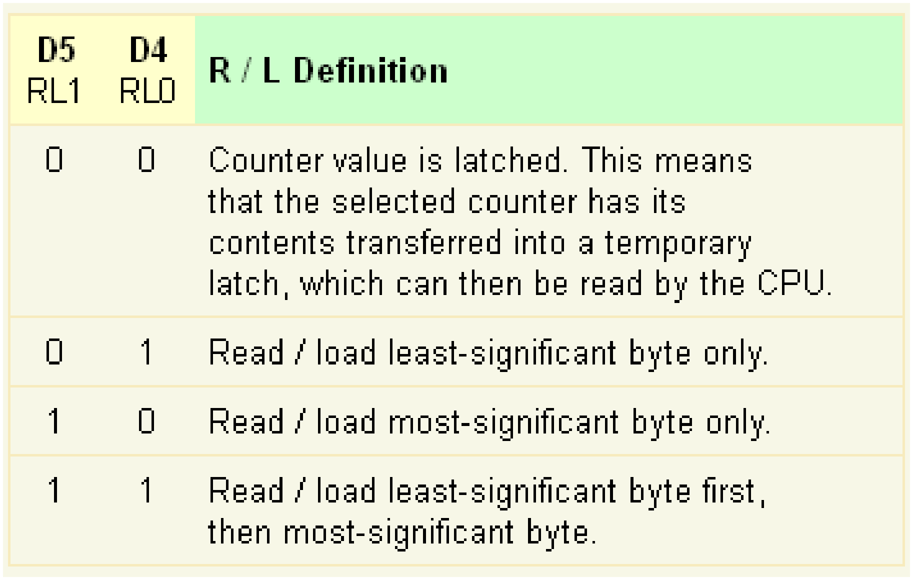
All of the operating modes for the counters are selected by writing bytes to the control register. This is the control word format.



D7 SC1	D6 SC0	Counter Select
0	0	counter 0
0	1	counter 1
1	0	counter 2
1	1	illegal value

Bits D7 and D6 are labeled SC1 and SC0. These bits select the counter to be programmed, it is necessary to define, using the control bits D7 and D6, which counter is being set up.

Once a counter is set up, it will remain that way until it is changed by another control word.



D5 RL1	D4 RL0	R / L Definition
0	0	Counter value is latched. This means that the selected counter has its contents transferred into a temporary latch, which can then be read by the CPU.
0	1	Read / load least-significant byte only.
1	0	Read / load most-significant byte only.
1	1	Read / load least-significant byte first, then most-significant byte.

Bits D5 and D4 (RL1 / RL0) of the control word shown above are defined as the read / load mode for the register that is selected by bits D7 and D6. Bits D5 and D4 define how the particular counter is to have data read from or written to it by the CPU.

These bits are defined as:

The 1st value, $00, is the *counter latch mode*. If this mode is specified, the current counter value is latched into an internal register at the time of the I/O write operation to the control register. When a read of the counter occurs, it is this latched value that is read.

If the latch mode is not used, then it is possible that the data read back may be in the process of changing while the read is occurring. This could result in invalid data being input by the CPU. To read the counter value while the counter is still in the process of counting, one must first issue a latch control word, and then issue another control word that indicates the order of the bytes to be read.

An alternative method of obtaining a stable count from the timer is to externally inhibit counting while the register is being read. To this, an external logic to the 8253 controlled by the Z80 to inhibit count during an input read operation is to connect.

Each technique has certain disadvantages. The first, the latching method, may give the CPU a reading that is "old" by several cycles, depending on the speed of the count and which byte of the counter is being read.

The second method, the external inhibiting function, requires additional hardware. In addition, it may change the overall system operation. The counters 1 and 2 of the MZ-700 are not designed with this additional hardware function. :-(but the counter 0. You can use this method for your own purposes even an amplifier is connected to the output pin of this counter.
The input to counter 0 is 1.1088MHz.

The next 3 bits of the control word are D3, D2, and D1. These bits determine the basic mode of operation for the selected counter. The mode descriptions are as follows:

D3	**D2**	**D1**	**Mode value**
M2	M1	M0	
0	0	0	mode 0: interrupt on terminal count
0	0	1	mode 1: programmable one-shot
x	1	0	mode 2: rate generator
x	1	1	mode 3: square wave generator
1	0	0	mode 4: software triggered strobe
1	0	1	mode 5: hardware triggered strobe

D0	**counts down in**
0	binary
1	BCD

The final bit **D0** of the control register determines how the register will count: The **maximum values** for the count in each count mode are 10^4 (10,000 decimal) in BCD and 2^{16} (65,536 decimal) in binary.

Modes

The following text describes all possible modes. The modes used in the MZ-700 and set by the monitor's startup are mode 0, mode 2, and mode 3.

- **Mode 0 (Interrupt on Terminal Count**)

The counter will be programmed to an initial value and afterwards counts down at a rate equal to the input clock frequency. When the count is equal to 0, the OUT pin will be a logical 1. The output will stay a logical 1 until the counter is reloaded with a new value or the same value or until a mode word is written to the device. Once the counter starts counting down, the GATE input can disable the internal counting by setting the GATE to a logical 0.

- **Mode 1 (Programmable One-Shot**)

In mode 1, the device can be setup to give an output pulse that is an integer number of clock pulses. The one-shot is triggered on the rising edge of the GATE input. If the trigger occurs during the pulse output, the 8253 will be retriggered again.

- **Mode 2 (Rate Generator**)

The counter that is programmed for mode 2 becomes a "divide by n" counter. The OUT pin of the counter goes to low for one input clock period. The time between the pulses of going low is dependent on the present count in the counter's register. I mean the time of the logical 1 pulse.
For example, suppose to get an output frequency of 1,000 Hz (Hertz), the period would be 1 / 1,000 s = 1 ms (millisecond) or 1,000 µs (microseconds). If an input clock of 1 MHz (Mega-Hertz) were applied to the clock input of the counter #0, then the counter #0 would need to be programmed to 1000 µs. This could be done in decimal or in BCD. (The period of an input clock of 1 MHz is 1 / 1,000,000 = 1 µs.)
The formula is: **$n=f_i$ divided by f_{out}.**
f_i = input clock frequency, f_{out} = output frequency, n = value to be loaded.

My example: $f_i = 1$ MHz $= 1 \times 10^6$ Hz, $f_{out} = 1$ kHz $= 1 \times 10^3$ Hz.

$n = 1 \times 10^6$ Hz / 1×10^3 Hz $= 1 \times 10^3 = 1,000$. This is the decimal value to be loaded or the hexadecimal value $03E8. The following program example uses the decimal load count.

```
B000 3E35     LD     A,$35          ; load control word
                                    ; for counter 0 mode 2
B002 3207E0  LD     ($E007),A      ; into port $E007
                                    ; for BCD count
B005 2104E0  LD     HL,$E004       ; address to the port
                                     ; of counter 0
B008 3E00     LD     A,$00
B00A 77       LD     (HL),A         ; load least significant
                                    ; byte of 1000 first
B00B 3E10     LD     A,$10
B00D 77       LD     (HL),A         ; load most significant
                                    ; byte of 1000 last
B00E 3E01     LD     A,1
B010 3208E0  LD     ($E008),A      ; start counter 0 is only
                                    ; Necessary for the MZ-700.
                                    ; Not necessary for
                                   ; counter #1 and #2
 ; The counter is now initialized and the output frequency
; will be 1000 Hz if the input frequency is 1 MHz.
```

If the count is loaded between output pulses, the present period will not be affected. A new period will occur during the next count sequence.

- **Mode 3 (Square Wave Generator**)

Mode 3 is similar to the mode 2 except that the output will be high for half the period and low for half. If the count is odd, the output will be high for (n + 1) / 2 and low for (n - 1) / 2 counts.

For example, I'll setup counter #0 for a square wave frequency of 10 kHz (kilo-Hertz), assuming the input frequency is 1 MHz.

Please refer to the formula described at mode 2.
$1 \times 10^6 / 10 \times 10^3 = 100$. This is the decimal value to be loaded or the hexadecimal value $0064. The following program example uses the binary load count.

```
B000 3E35     LD     A,$36          ; load control word
                                    ; for counter 0 mode 3
B002 3207E0  LD     ($E007),A      ; into port $E007
                                    ; for binary count
B005 2104E0  LD     HL,$E004       ; address to the port
                                    ; of counter 0
B008 3E00     LD     A,$64          ; equals to
                                    ; 100 microseconds
```

```
                                                  ; for 10,000 Hz
B00A 77         LD      (HL),A                    ; load least significant
                                                  ; byte of $0064 first
B00B 3E10       LD      A,$00
B00D 77         LD      (HL),A                    ; load most significant
                                                  ; nyte of $0064 last
B00E 3E01       LD      A,1
B010 3208E0     LD      ($E008),A                 ; start counter 0 is only
                                                  ; necessary for the MZ-700.
                                                  ; Not necessary for counter
                                                  ; #1 and #2
; The counter is now initialized and the output frequency
; will be 10 kHz if the input frequency is 1 MHz.
```

- **Mode 4 (Software Triggered Strobe**)

In this mode the programmer can set up the counter to give an output timeout starting when the register is loaded. On the terminal count, when the counter equals to 0, the output will go to a logical 0 for one clock period and then returns to a logical 1. First the mode is set, the output will be a logical 1.

- **Mode 5 (Hardware Triggered Strobe**)

In this mode the rising edge of the trigger input will start the counting of the counter. The output goes low for one clock at the terminal count. The counter is retriggerable, thus meaning that if the trigger input is taken low and then high during a count sequence, the sequence will start over.

When the external trigger input goes to a logical 1, the timer will start to time out. If the external trigger occurs again, prior to the time completing a full timeout, the timer will retrigger.

Programmable Interrupt Controller 8259A

- If we are working with an 8086, we have a problem here because the 8086 has only two interrupt inputs, NMI and INTR.
- If we save NMI for a power failure interrupt, this leaves only one interrupt for all the other applications. For applications where we have interrupts from multiple source, we use an external device called a ***priority interrupt controller*** (PIC) to the interrupt signals into a single interrupt input on the processor.

Architecture and Signal Descriptions of 8259A

- The architectural block diagram of 8259A is shown in fig1. The functional explication of each block is given in the following text in brief.
- **Interrupt Request Register (RR)**: The interrupts at IRQ input lines are handled by Interrupt Request internally. IRR stores all the interrupt request in it in order to serve them one by one on the priority basis.
- **In-Service Register (ISR)**: This stores all the interrupt requests those are being served, i.e. ISR keeps a track of the requests being served.

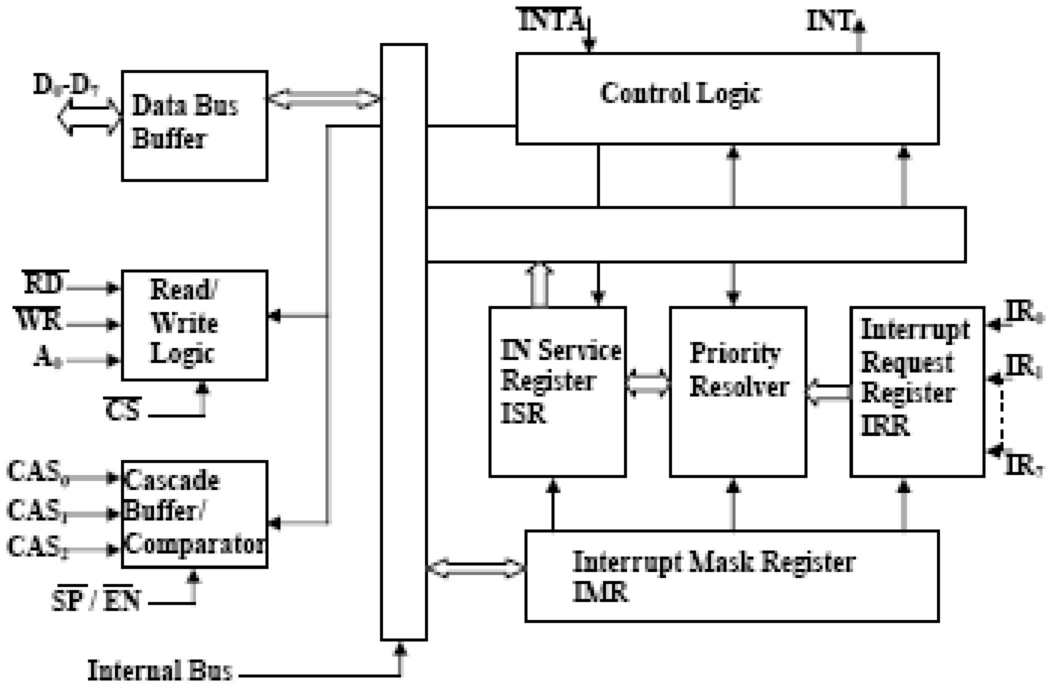


Fig:1 8259A Block Diagram

- **Priority Resolver :** This unit determines the priorities of the interrupt requests appearing simultaneously. The highest priority is selected and stored into the corresponding bit of ISR during INTA pulse. The IR0 has the highest priority while the IR7 has the lowest one, normally in fixed priority mode. The priorities however may be altered by programming the 8259A in rotating priority mode.
- **Interrupt Mask Register (IMR)** : This register stores the bits required to mask the interrupt inputs. IMR operates on IRR at the direction of the Priority Resolver.
- **Interrupt Control Logic**: This block manages the interrupt and interrupt acknowledge signals to be sent to the CPU for serving one of the eight interrupt requests. This also accepts the interrupt acknowledge (INTA) signal from CPU that causes the 8259A to release vector address on to the data bus.

- **Data Bus Buffer** : This tristate bidirectional buffer interfaces internal 8259A bus to the microprocessor system data bus. Control words, status and vector information pass through data buffer during read or write operations.
- **Read/Write Control Logic**: This circuit accepts and decodes commands from the CPU. This block also allows the status of the 8259A to be transferred on to the data bus.
- **Cascade Buffer/Comparator**: This block stores and compares the ID's all the 8259A used in system. The three I/O pins CASO-2 are outputs when the 8259A is used as a master. The same pins act as inputs when the 8259A is in slave mode. The 8259A in master mode sends the ID of the interrupting slave device on these lines. The slave thus selected, will send its preprogrammed vector address on the data bus during the next INTA pulse.
- **CS**: This is an active-low chip select signal for enabling RD and WR operations of 8259A. INTA function is independent of CS.
- **WR** : This pin is an active-low write enable input to 8259A. This enables it to accept command words from CPU.
- **RD** : This is an active-low read enable input to 8259A. A low on this line enables 8259A to release status onto the data bus of CPU.
- **D0-D7** : These pins from a bidirectional data bus that carries 8-bit data either to control word or from status word registers. This also carries interrupt vector information.
- **CAS0 – CAS2 Cascade Lines** : A signal 8259A provides eight vectored interrupts. If more interrupts are required, the 8259A is used in cascade mode. In cascade mode, a master 8259A along with eight slaves 8259A can provide upto 64 vectored interrupt lines. These three lines act as select lines for addressing the slave 8259A.
- **PS/EN** : This pin is a dual purpose pin. When the chip is used in buffered mode, it can be used as buffered enable to control buffer transreceivers. If this is not used in buffered mode then the pin is used as input to designate whether the chip is used as a master (SP =1) or slave (EN = 0).
- **INT** : This pin goes high whenever a valid interrupt request is asserted. This is used to interrupt the CPU and is connected to the interrupt input of CPU.
- **IR0 – IR7 (Interrupt requests)** :These pins act as inputs to accept interrupt request to the CPU. In edge triggered mode, an interrupt service is requested by raising an IR pin from a low to a high state and holding it high until it is acknowledged, and just by latching it to high level, if used in level triggered mode.

Pin Diagram

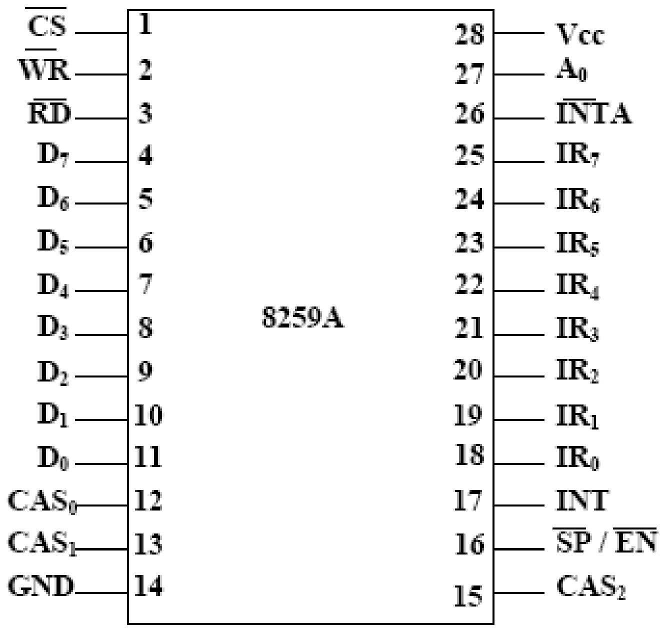


Fig : 8259 Pin Diagram

INTA (Interrupt acknowledge): This pin is an input used to strobe-in 8259A interrupt vector data on to the data bus. In conjunction with CS, WR and RD pins, this selects the different operations like, writing command words, reading status word, etc.

- The device 8259A can be interfaced with any CPU using either polling or interrupt. In polling, the CPU keeps on checking each peripheral device in sequence to ascertain if it requires any service from the CPU. If any such service request is noticed, the CPU serves the request and then goes on to the next device in sequence.
- After the entire peripheral device are scanned as above the CPU again starts from first device.
- This type of system operation results in the reduction of processing speed because most of the CPU time is consumed in polling the peripheral devices.
- In the interrupt driven method, the CPU performs the main processing task till it is interrupted by a service requesting peripheral device.
- The net processing speed of these type of systems is high because the CPU serves the peripheral only if it receives the interrupt request
- If more than one interrupt requests are received at a time, all the requesting peripherals are served one by one on priority basis.
- This method of interfacing may require additional hardware if number of peripherals to be interfaced is more than the interrupt pins available with the CPU.

Interrupt Sequence in an 8086 system

- The Interrupt sequence in an 8086-8259A system is described as follows:

1. One or more IR lines are raised high that set corresponding IRR bits.
2. 8259A resolves priority and sends an INT signal to CPU.
3. The CPU acknowledge with INTA pulse.
4. Upon receiving an INTA signal from the CPU, the highest priority ISR bit is set and the corresponding IRR bit is reset. The 8259A does not drive data during this period.
5. The 8086 will initiate a second INTA pulse. During this period 8259A releases an 8-bit pointer on to a data bus from where it is read by the CPU.
6. This completes the interrupt cycle. The ISR bit is reset at the end of the second INTA pulse if automatic end of interrupt (AEOI) mode is programmed. Otherwise ISR bit remains set until an appropriate EOI command is issued at the end of interrupt subroutine.

Command Words of 8259A

• The command words of 8259A are classified in two groups
1. Initialization command words (ICW) and
2. Operation command words (OCW).

- Initialization Command Words (ICW): Before it starts functioning, the 8259A must be initialized by writing two to four command words into the respective command word registers. These are called as initialized command words.
- If A0 = 0 and D4 = 1, the control word is recognized as ICW1. It contains the control bits for edge/level triggered mode, single/cascade mode, call address interval and whether ICW4 is required or not.
- If A0=1, the control word is recognized as ICW2. The ICW2 stores details regarding interrupt vector addresses. The initialisation sequence of 8259A is described in form of a flow chart in fig 3 below.
- The bit functions of the ICW1 and ICW2 are self explanatory as shown in fig below.

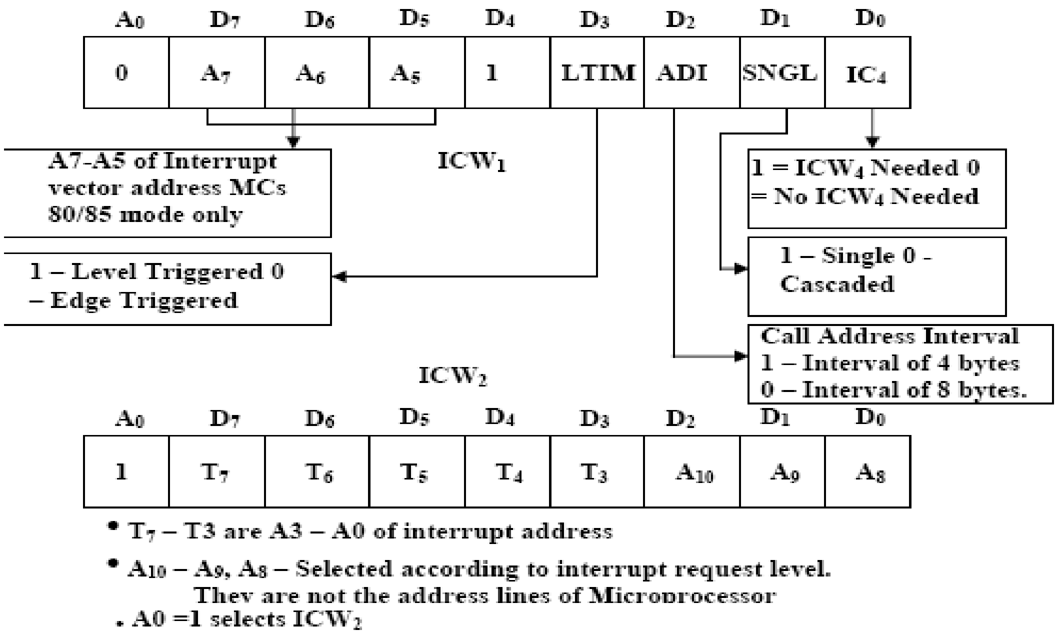

A0 D7 D6 D5 D4 D3 D2 D1 D0
0 A7 A6 A5 1 LTIM ADI SNGL IC4
ICW1
A7-A5 of Interrupt vector address MCs 80/85 mode only
1 – Level Triggered 0 – Edge Triggered
1 = ICW4 Needed 0 = No ICW4 Needed
1 – Single 0 - Cascaded
Call Address Interval 1 – Interval of 4 bytes 0 – Interval of 8 bytes.
ICW2
A0 D7 D6 D5 D4 D3 D2 D1 D0
1 T7 T6 T5 T4 T3 A10 A9 A8
• T7 – T3 are A3 – A0 of interrupt address
• A10 – A9, A8 – Selected according to interrupt request level. They are not the address lines of Microprocessor
• A0 =1 selects ICW2

Fig 4 **Fig:** ction Command Words ICW1 and

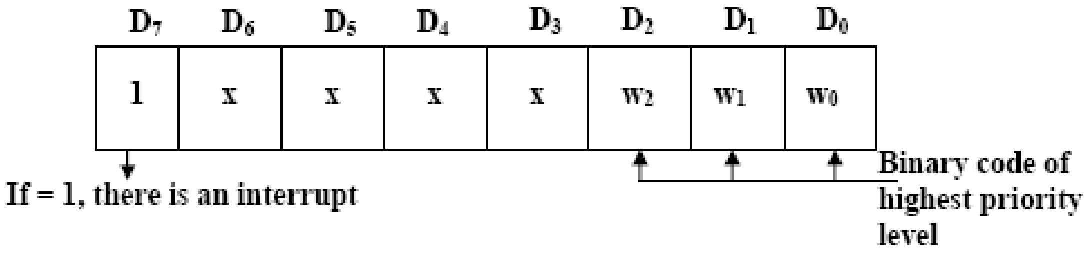
Operating Modes of 8259

- The different modes of operation of 8259A can be programmed by setting or resting the appropriate bits of the ICW or OCW as discussed previously. The different modes of operation of 8259A are explained in the following.
- **Fully Nested Mode** : This is the default mode of operation of 8259A. IR0 has the highest priority and IR7 has the lowest one. When interrupt request are noticed, the highest priority request amongst them is determined and the vector is placed on the data bus. The corresponding bit of ISR is set and remains set till the microprocessor issues an EOI command just before returning from the service routine or the AEOI bit is set.
- If the ISR (in service) bit is set, all the same or lower priority interrupts are inhibited but higher levels will generate an interrupt, that will be acknowledge only if the microprocessor interrupt enable flag IF is set. The priorities can afterwards be changed by programming the rotating priority modes.
- **End of Interrupt (EOI)** : The ISR bit can be reset either with AEOI bit of ICW1 or by EOI command, issued before returning from the interrupt service routine. There are two types of EOI commands specific and non-specific. When 8259A is operated in the modes that preserve fully nested structure, it can determine which ISR bit is to be reset on EOI.
- When non-specific EOI command is issued to 8259A it will be automatically reset the highest ISR bit out of those already set.
- When a mode that may disturb the fully nested structure is used, the 8259A is no longer able to determine the last level acknowledged. In this case a specific EOI command is issued to reset a particular ISR bit. An ISR bit that

is masked by the corresponding IMR bit, will not be cleared by non-specific EOI of 8259A, if it is in special mask mode.

- **Automatic Rotation** : This is used in the applications where all the interrupting devices are of equal priority.
- In this mode, an interrupt request IR level receives priority after it is served while the next device to be served gets the highest priority in sequence. Once all the device are served like this, the first device again receives highest priority.
- **Automatic EOI Mode** : Till AEOI=1 in ICW4, the 8259A operates in AEOI mode. In this mode, the 8259A performs a non-specific EOI operation at the trailing edge of the last INTA pulse automatically. This mode should be used only when a nested multilevel interrupt structure is not required with a single 8259A.
- **Specific Rotation** : In this mode a bottom priority level can be selected, using L2, L1 and L0 in OCW2 and R=1, SL=1, EOI=0.
- The selected bottom priority fixes other priorities. If IR5 is selected as a bottom priority, then IR5 will have least priority and IR4 will have a next higher priority. Thus IR6 will have the highest priority.
- These priorities can be changed during an EOI command by programming the rotate on specific EOI command in OCW2.
- **Specific Mask Mode**: In specific mask mode, when a mask bit is set in OCW1, it inhibits further interrupts at that level and enables interrupt from other levels, which are not masked.
- **Edge and Level Triggered Mode** : This mode decides whether the interrupt should be edge triggered or level triggered. If bit LTIM of ICW1 =0 they are edge triggered, otherwise the interrupts are level triggered.
- **Reading 8259 Status** : The status of the internal registers of 8259A can be read using this mode. The OCW3 is used to read IRR and ISR while OCW1 is used to read IMR. Reading is possible only in no polled mode.
- **Poll Command** : In polled mode of operation, the INT output of 8259A is neglected, though it functions normally, by not connecting INT output or by masking INT input of the microprocessor. The poll mode is entered by setting P=1 in OCW3.
- The 8259A is polled by using software execution by microprocessor instead of the requests on INT input. The 8259A treats the next RD pulse to the 8259A as an interrupt acknowledge. An appropriate ISR bit is set, if there is a request. The priority level is read and a data word is placed on to data bus, after RD is activated. A poll command may give more than 64 priority

levels.


D7
D6
D5
D4
D3
D2
D1
D0
1
x
x
x
x
w2
w1
w0
If = 1, there is an interrupt
Binary code of highest priority level

- **Special Fully Nested Mode** : This mode is used in more complicated system, where cascading is used and the priority has to be programmed in the master using ICW4. this is somewhat similar to the normal nested mode.
- In this mode, when an interrupt request from a certain slave is in service, this slave can further send request to the master, if the requesting device connected to the slave has higher priority than the one being currently served. In this mode, the master interrupt the CPU only when the interrupting device has a higher or the same priority than the one current being served. In normal mode, other requests than the one being served are masked out.
- When entering the interrupt service routine the software has to check whether this is the only request from the slave. This is done by sending a non-specific EOI can be sent to the master, otherwise no EOI should be sent. This mode is important, since in the absence of this mode, the slave would interrupt the master only once and hence the priorities of the slave inputs would have been disturbed.
- **Buffered Mode**: When the 83259A is used in the systems where bus driving buffers are used on data buses. The problem of enabling the buffers exists. The 8259A sends buffer enable signal on SP/ EN pin, whenever data is placed on the bus.
- **Cascade Mode:** The 8259A can be connected in a system containing one master and eight slaves (maximum) to handle up to 64 priority levels. The master controls the slaves using CAS0-CAS2 which act as chip select inputs (encoded) for slaves.
- In this mode, the slave INT outputs are connected with master IR inputs. When a slave request line is activated and acknowledged, the master will enable the slave to release the vector address during second pulse of INTA sequence.
- The cascade lines are normally low and contain slave address codes from the trailing edge of the first INTA pulse to the trailing edge of the second INTA pulse. Each 8259A in the system must be separately initialized and programmed to work in different modes. The EOI command must be issued twice, one for master and the other for the slave.
- A separate address decoder is used to activate the chip select line of each 8259A.

- Following Fig shows the details of the circuit connections of 8259A in cascade scheme.

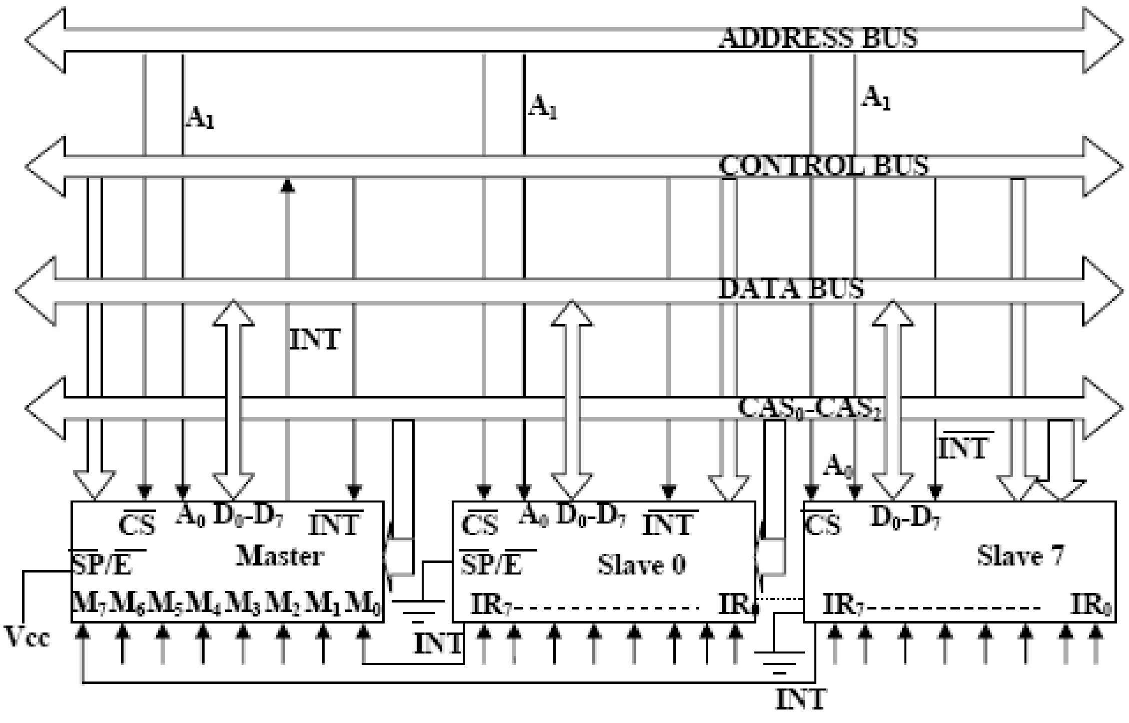


Fig : 8259A in Cascade Mode

Keyboard/Display Controller 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

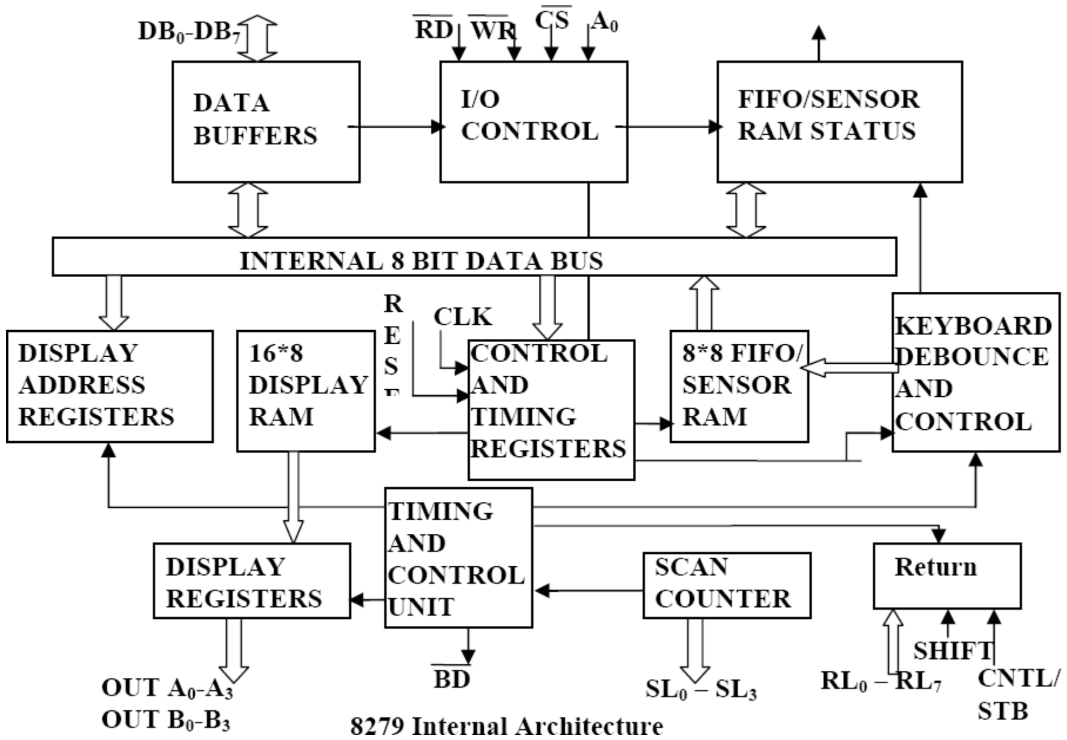
- The keyboard display controller chip 8279 provides:

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

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

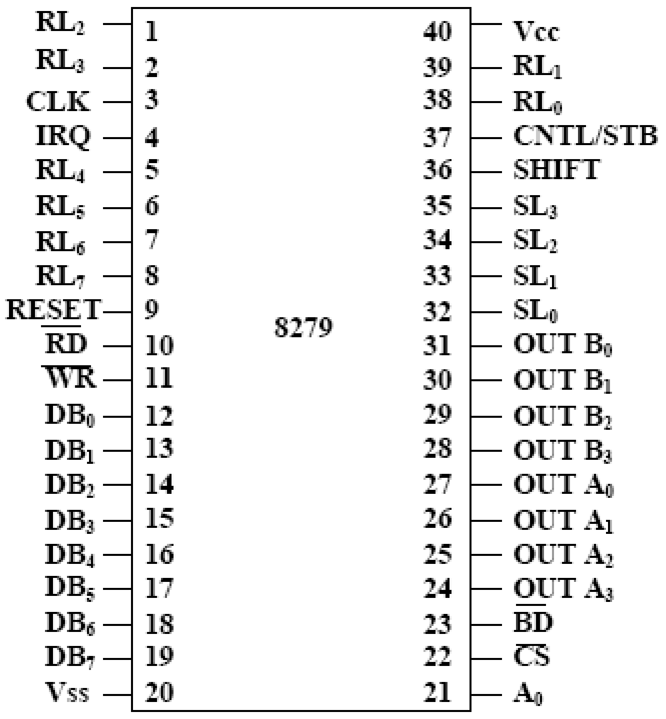
- Fig shows the functional block diagram of 8279 followed by its brief description.

- **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 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

Pin Diagram



8279 Pin Configuration

The signal description 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.
- **A0** : 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.

- **RL0 - RL7 - 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 A0 – OUT A3 and OUT B0 – OUT B3** – 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.

Command Words of 8279

- All the command words or status words are written or read with A0 = 1 and CS = 0 to or from 8279. This section describes the various command available in 8279.

a) **Keyboard Display Mode Set** – The format of the command word to select different modes of operation of 8279 is given below with its bit definitions.

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	A_0
0	0	D	D	D	K	K	K	1

D	D	Display modes
0	0	Eight 8-bit character Left entry
0	1	Sixteen 8-bit character left entry
1	0	Eight 8-bit character Right entry
1	1	Sixteen 8-bit character Right entry

K	K	K	Keyboard modes
0	0	0	Encoded Scan, 2 key lockout (Default after reset)
0	0	1	Decoded Scan, 2 key lockout
0	1	0	Encoded Scan, N- key Roll over
0	1	1	Decoded Scan, N- key Roll over
1	0	0	Encode Scan, N- key Roll over
1	0	1	Decoded Scan, N- key Roll over
1	1	0	Strobed Input Encoded Scan
1	1	1	Strobed Input Decoded Scan

b) **Programmable clock** : The clock for operation of 8279 is obtained by dividing the external clock input signal by a programmable constant called prescaler.

- PPPPP is a 5-bit binary constant. The input frequency is divided by a decimal constant ranging from 2 to 31, decided by the bits of an internal prescaler, PPPPP.

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	A_0
0	0	1	P	P	P	P	P	1

c) **Read FIFO / Sensor RAM** : The format of this command is given below.

- This word is written to set up 8279 for reading FIFO/ sensor RAM. In scanned keyboard mode, AI and AAA bits are of no use. The 8279 will automatically drive data bus for each subsequent read, in the same sequence, in which the data was entered.
- In sensor matrix mode, the bits AAA select one of the 8 rows of RAM. If AI flag is set, each successive read will be from the subsequent RAM location.

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	A_0
0	1	0	AI	X	A	A	A	1

- X – don’t care

- AI – Auto Increment Flag
- AAA – Address pointer to 8 bit FIFO RAM

d) **Read Display RAM:** This command enables a programmer to read the display RAM data. The CPU writes this command word to 8279 to prepare it for display RAM read operation. AI is auto increment flag and AAAA, the 4-bit address points to the 16-byte display RAM that is to be read. If AI=1, the address will be automatically, incremented after each read or write to the Display RAM. The same address counter is used for reading and writing.

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	A_0
0	1	1	AI	A	A	A	A	1

e) **Write Display RAM**:
AI – Auto increment Flag.
AAAA – 4 bit address for 16-bit display RAM to be written.

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	A_0
1	0	0	AI	A	A	A	A	1

f) **Display Write Inhibit/Blanking** : The IW (inhibit write flag) bits are used to mask the individual nibble as shown in the below command word. The output lines are divided into two nibbles (OUTA0 – OUTA3) and (OUTB0 – OUTB3), those can be masked by setting the corresponding IW bit to 1.

- Once a nibble is masked by setting the corresponding IW bit to 1, the entry to display RAM does not affect the nibble even though it may change the unmasked nibble. The blank display bit flags (BL) are used for blanking A and B nibbles.
  - Here D0, D2 corresponds to OUTB0 – OUTB3 while D1 and D3 corresponds to OUTA0-OUTA3 for blanking and masking.
  - If the user wants to clear the display, blank (BL) bits are available for each nibble as shown in format. Both BL bits will have to be cleared for blanking both the nibbles.

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	A_0
1	0	1	X	IW	IW	BL	BL	1

g) **Clear Display RAM** : The CD2, CD1, CD0 is a selectable blanking code to clear all the rows of the display RAM as given below. The characters A and B represents the output nibbles.

- CD2 must be 1 for enabling the clear display command. If CD2 = 0, the clear display command is invoked by setting CA=1 and maintaining

CD1, CD0 bits exactly same as above. If CF=1, FIFO status is cleared and IRQ line is pulled down.

- Also the sensor RAM pointer is set to row 0. if CA=1, this combines the effect of CD and CF bits. Here, CA represents Clear All and CF as Clear FIFO RAM.

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	A_0
1	1	0	CD_2	CD_1	CD_0	CF	CA	1

CD_2	CD_1	CD_0
1	0	X
1	1	0
1	1	1

All zeros (x don't care) AB=00

A3-A0 =2 (0010) and B3-B0=00 (0000)

All ones (AB =FF), i.e. clear RAM

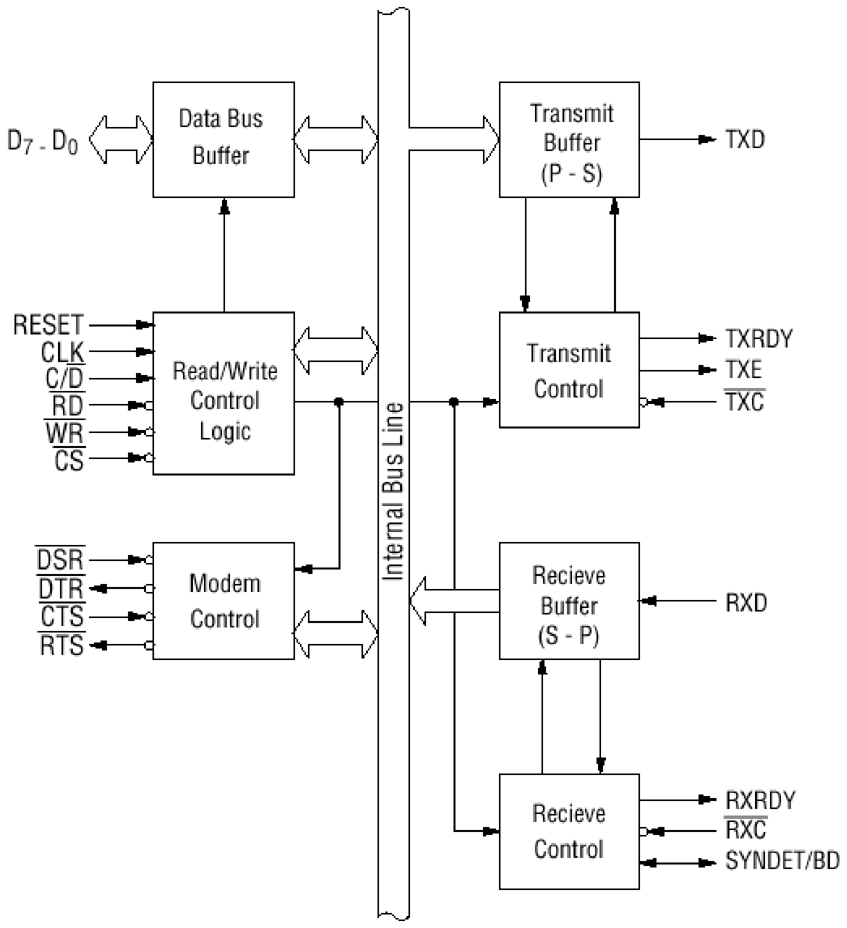
h) **End Interrupt / Error mode Set** : For the sensor matrix mode, this command lowers the IRQ line and enables further writing into the RAM. Otherwise, if a change in sensor value is detected, IRQ goes high that inhibits writing in the sensor RAM.

- For N-Key roll over mode, if the E bit is programmed to be '1', the 8279 operates in special Error mode. Details of this mode are described in scanned keyboard special error mode. **X-** don't care.

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	A_0
1	1	1	E	X	X	X	X	1

Programmable Communication Interface 8251 USART

8251 is a USART (Universal Synchronous Asynchronous Receiver Transmitter) for serial data communication. As a peripheral device of a microcomputer system, the 8251 receives parallel data from the CPU and transmits serial data after conversion. This device also receives serial data from the outside and transmits parallel data to the CPU after conversion.



Block diagram of the 8251 USART (Universal Synchronous Asynchronous Receiver Transmitter)

The 8251 functional configuration is programed by software. Operation between the 8251 and a CPU is executed by program control. Table 1 shows the operation between a CPU and the device.

$\overline{CS}$	C/$\overline{D}$	$\overline{RD}$	$\overline{WR}$	
1	×	×	×	Data Bus 3-State
0	×	1	1	Data Bus 3-State
0	1	0	1	Status → CPU
0	1	1	0	Control Word ← CPU
0	0	0	1	Data → CPU
0	0	1	0	Data ← CPU

Table 1 Operation between a CPU and 8251

Control Words

There are two types of control word.

1. Mode instruction (setting of function)

2. Command (setting of operation)

1) Mode Instruction

Mode instruction is used for setting the function of the 8251. Mode instruction will be in "wait for write" at either internal reset or external reset. That is, the writing of a control word after resetting will be recognized as a "mode instruction."

Items set by mode instruction are as follows:

• Synchronous/asynchronous mode

• Stop bit length (asynchronous mode)

• Character length

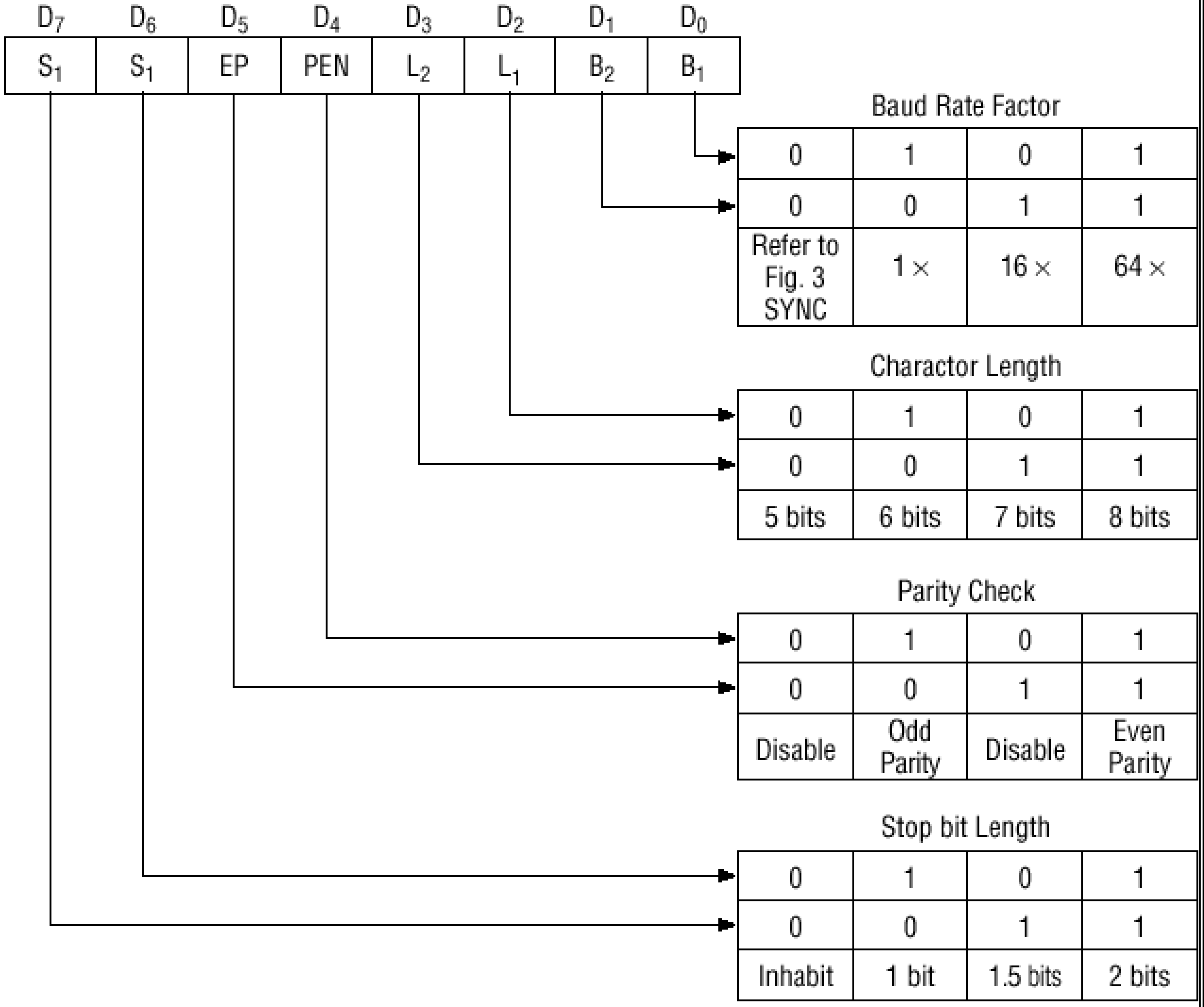
• Parity bit

• Baud rate factor (asynchronous mode)

• Internal/external synchronization (synchronous mode)

• Number of synchronous characters (Synchronous mode)

The bit configuration of mode instruction is shown in Figures 2 and 3. In the case of synchronous mode, it is necessary to write one-or two byte sync characters. If sync characters were written, a function will be set because the writing of sync characters constitutes part of mode instruction.



D_7 D_6 D_5 D_4 D_3 D_2 D_1 D_0
S_1 S_1 EP PEN L_2 L_1 B_2 B_1

Baud Rate Factor

0	1	0	1
0	0	1	1
Refer to Fig. 3 SYNC	1 ×	16 ×	64 ×

Charactor Length

0	1	0	1
0	0	1	1
5 bits	6 bits	7 bits	8 bits

Parity Check

0	1	0	1
0	0	1	1
Disable	Odd Parity	Disable	Even Parity

Stop bit Length

0	1	0	1
0	0	1	1
Inhabit	1 bit	1.5 bits	2 bits

Fig. 2 Bit Configuration of Mode Instruction (Asynchronous)

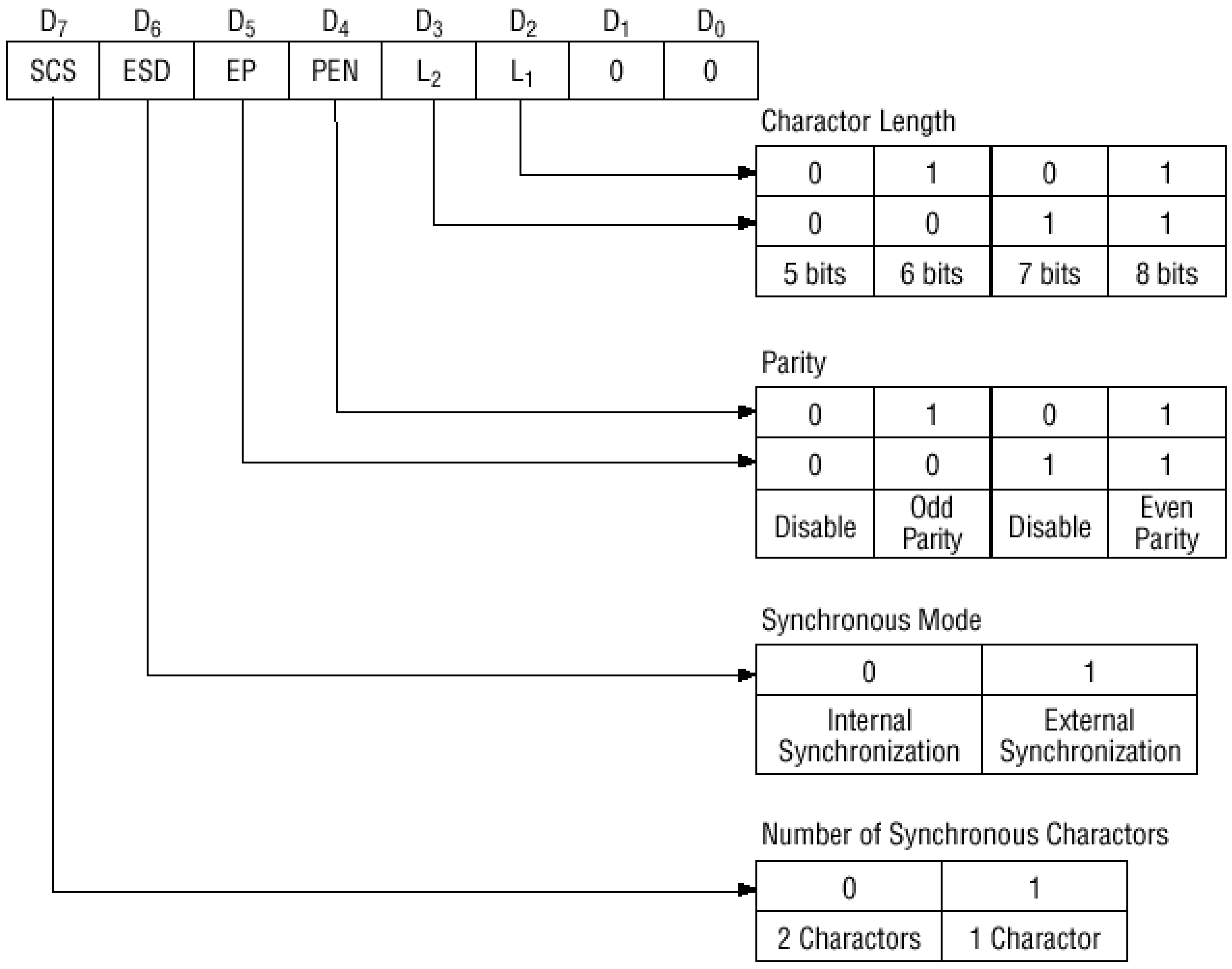


Fig. 3 Bit Configuration of Mode Instruction (Synchronous)

2) Command

Command is used for setting the operation of the 8251. It is possible to write a command whenever necessary after writing a mode instruction and sync characters.

Items to be set by command are as follows:

- Transmit Enable/Disable
- Receive Enable/Disable
- DTR, RTS Output of data.
- Resetting of error flag.

• Sending to break characters

• Internal resetting

• Hunt mode (synchronous mode)

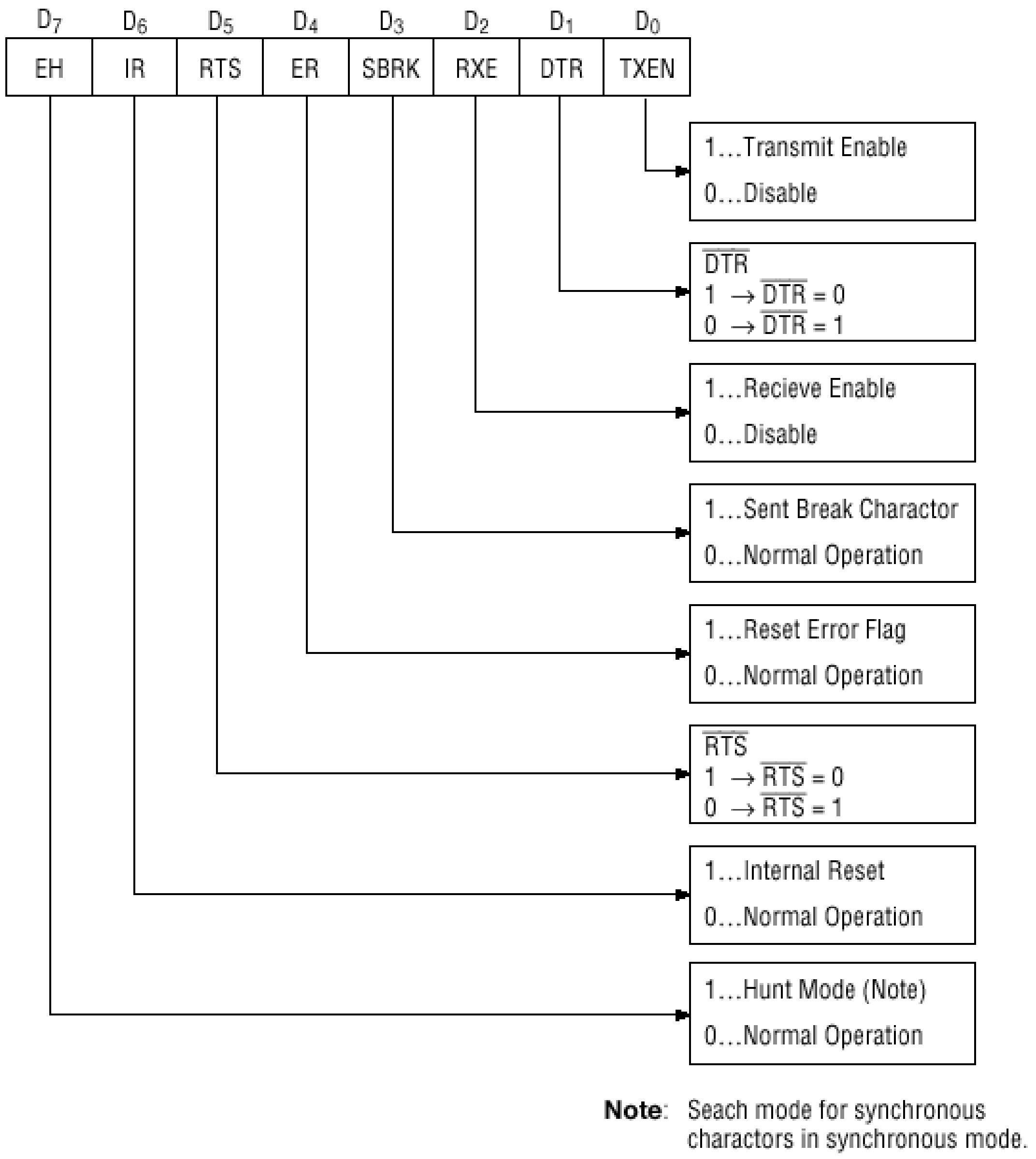
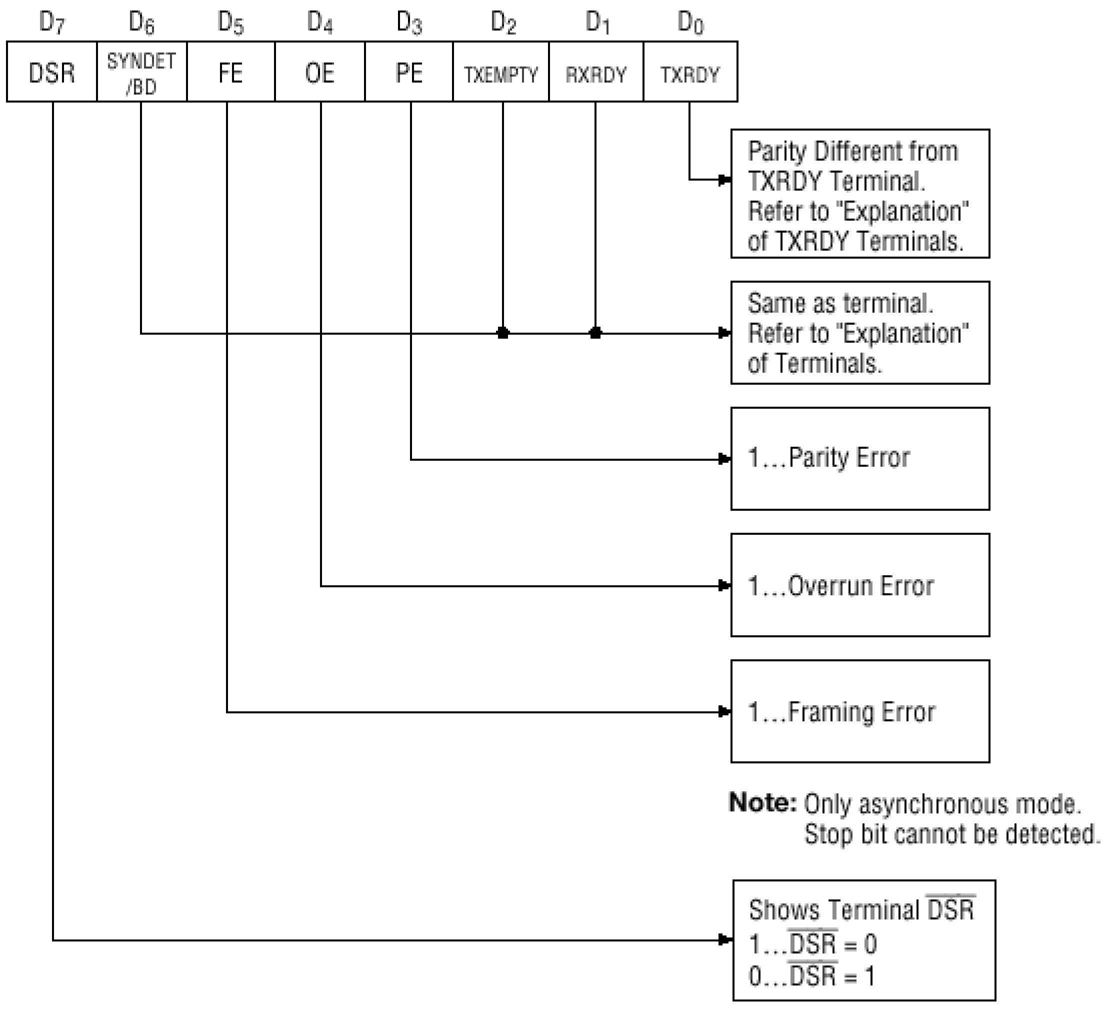


Fig. 4 Bit Configuration of Command

Status Word

It is possible to see the internal status of the 8251 by reading a status word. The bit configuration of status word is shown in Fig. 5.



D_7 D_6 D_5 D_4 D_3 D_2 D_1 D_0
DSR
SYNDET /BD
FE
OE
PE
TXEMPTY
RXRDY
TXRDY
Parity Different from TXRDY Terminal. Refer to "Explanation" of TXRDY Terminals.
Same as terminal. Refer to "Explanation" of Terminals.
1…Parity Error
1…Overrun Error
1…Framing Error
Note: Only asynchronous mode. Stop bit cannot be detected.
Shows Terminal $\overline{DSR}$
1…$\overline{DSR}$ = 0
0…$\overline{DSR}$ = 1

Fig. 5 Bit Configuration of Status Word

Pin Description

D 0 to D 7 (l/O terminal)

This is bidirectional data bus which receive control words and transmits data from the CPU and sends status words and received data to CPU.

RESET (Input terminal)

A "High" on this input forces the 8251 into "reset status." The device waits for the writing of "mode instruction." The min. reset width is six clock inputs during the operating status of CLK.

CLK (Input terminal)

CLK signal is used to generate internal device timing. CLK signal is independent of RXC or TXC. However, the frequency of CLK must be greater than 30 times the RXC and TXC at Synchronous mode and Asynchronous "x1" mode, and must be greater than 5 times at Asynchronous "x16" and "x64" mode.

WR (Input terminal)

This is the "active low" input terminal which receives a signal for writing transmit data and control words from the CPU into the 8251.

RD (Input terminal)

This is the "active low" input terminal which receives a signal for reading receive data and status words from the 8251.

C/D (Input terminal)

This is an input terminal which receives a signal for selecting data or command words and status words when the 8251 is accessed by the CPU. If C/D = low, data will be accessed. If C/D = high, command word or status word will be accessed.

CS (Input terminal)

This is the "active low" input terminal which selects the 8251 at low level when the CPU accesses. Note: The device won't be in "standby status"; only setting CS = High.

TXD (output terminal)

This is an output terminal for transmitting data from which serial-converted data is sent out. The device is in "mark status" (high level) after resetting or during a status when transmit is disabled. It is also possible to set the device in "break status" (low level) by a command.

TXRDY (output terminal)

This is an output terminal which indicates that the 8251is ready to accept a transmitted data character. But the terminal is always at low level if CTS = high or the device was set in "TX disable status" by a command. Note: TXRDY status word indicates that transmit data character is receivable, regardless of CTS or command. If the CPU writes a data character, TXRDY will be reset by the leading edge or WR signal.

TXEMPTY (Output terminal)

This is an output terminal which indicates that the 8251 has transmitted all the characters and had no data character. In "synchronous mode," the terminal is at high level, if transmit data characters are no longer remaining and sync characters are automatically transmitted. If the CPU writes a data character, TXEMPTY will be reset by the leading edge of WR signal. Note : As the transmitter is disabled by setting CTS "High" or command, data written before disable will be sent out. Then TXD and TXEMPTY will be "High". Even if a data is written after disable, that data is not sent out and TXE will be "High".After the transmitter is enabled, it sent out. (Refer to Timing Chart of Transmitter Control and Flag Timing)

TXC (Input terminal)

This is a clock input signal which determines the transfer speed of transmitted data. In "synchronous mode," the baud rate will be the same as the frequency of TXC. In "asynchronous mode", it is possible to select the baud rate factor by mode instruction. It can be 1, 1/16 or 1/64 the TXC. The falling edge of TXC sifts the serial data out of the 8251.

RXD (input terminal)

This is a terminal which receives serial data.

RXRDY (Output terminal)

This is a terminal which indicates that the 8251 contains a character that is ready to READ. If the CPU reads a data character, RXRDY will be reset by the leading edge of RD signal. Unless the CPU reads a data character before the next one is received completely, the preceding data will be lost. In such a case, an overrun error flag status word will be set.

RXC (Input terminal)

This is a clock input signal which determines the transfer speed of received data. In "synchronous mode," the baud rate is the same as the frequency of RXC. In "asynchronous mode," it is possible to select the baud rate factor by mode instruction. It can be 1, 1/16, 1/64 the RXC.

SYNDET/BD (Input or output terminal)

This is a terminal whose function changes according to mode. In "internal synchronous mode." this terminal is at high level, if sync characters are received and synchronized. If a status word is read, the terminal will be reset. In "external synchronous mode, "this is an input terminal. A "High" on this input forces the 8251 to start receiving data characters.

In "asynchronous mode," this is an output terminal which generates "high level"output upon the detection of a "break" character if receiver data contains a "low-level" space between the stop bits of two continuous characters. The terminal will be reset, if RXD is at high level. After Reset is active, the terminal will be output at low level.

DSR (Input terminal)

This is an input port for MODEM interface. The input status of the terminal can be recognized by the CPU reading status words.

DTR (Output terminal)

This is an output port for MODEM interface. It is possible to set the status of DTR by a command.

CTS (Input terminal)

This is an input terminal for MODEM interface which is used for controlling a transmit circuit. The terminal controls data transmission if the device is set in "TX Enable" status by a command. Data is transmitable if the terminal is at low level.

RTS (Output terminal)

This is an output port for MODEM interface. It is possible to set the status RTS by a command.

8527 DMA Controller

The i8527 controller has four independent channels each of which contains an address register and a counter. The counter decrements as each byte transfer occur, and forces termination of the DMA operation after the last transfer. The controller increments the address registers after each operation, so that successive data transfers are made at contiguous ascending addresses. The arbiter resolves conflicts among the channels for access to memory. Two methods have been used in this chip to make the chip useful in a variety of different applications. In one mode the channels have a fixed priority and conflicts are resolved according to the priority, for example, Channel 0 has highest priority and Channel 3 lowest. The second mode is a rotating priority scheme in which priority rankings are the four cycle shifts of 0-1-2-3, when a channel is granted access to the bus the priority ranking shifts cyclically to place the channel in the lowest priority position for the next arbitration cycle.

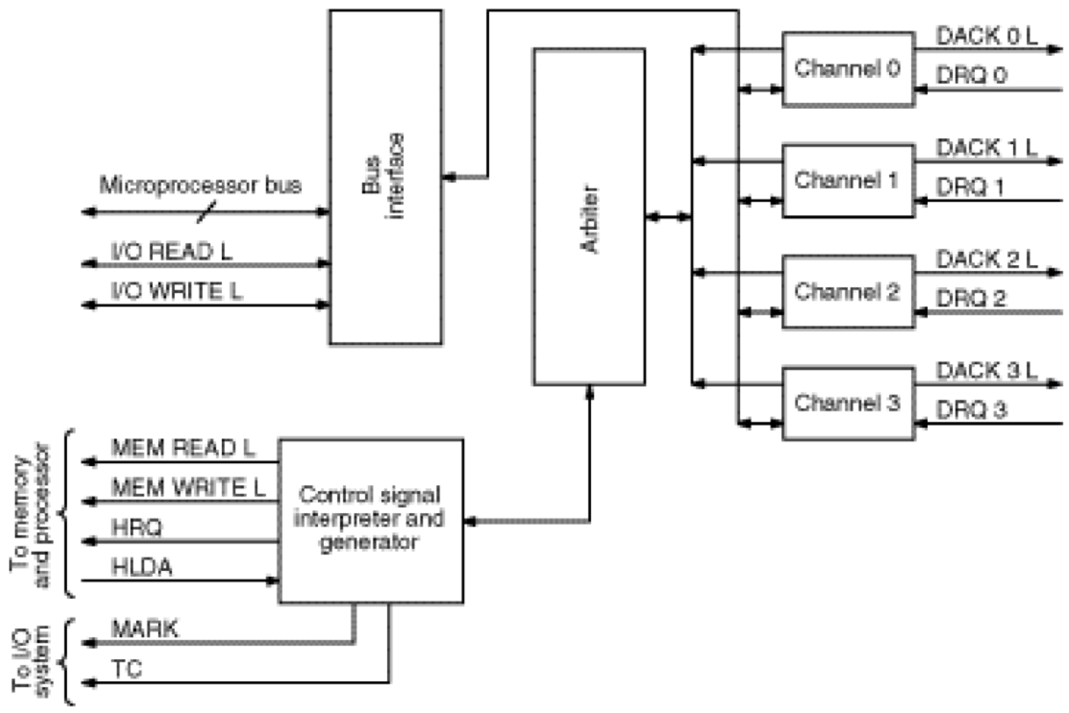


Figure 5-4: Structure of the i8527 DMA controller

The chip has four signals associated with the READ and WRITE operation. MEM READ L and MEM WRITE L are signals produced by DMA controller to exercise memory. The two signals I/O READ L and I/O WRITE L are bidirectional, they are inputs from the microprocessor when the microprocessor sends commands to the 8257 and reads back the 8257 status. During the I/O operation these signals are output from the 8257 and are functionally opposite to the memory signals. The 8257 takes control of the bus by exercising HALT (HRQ) and receives back the "go-ahead" signal on HALT ACKNOWLEDGE (HLDA).

Two signals produced by the DMA controller can be used by the I/O port to assist in controlling the transfer process. One signal TC--terminal count--is asserted during the last cycle of a DMA block. This can be used to describe a DMA mode on an I/O port or to reset the port's internal state to indicate the end of a transfer. The second--MARK--is inserted when the remaining count on a channel became a multiple of 128--providing a convenient timing signal for an external device.

Interrupts 8086

The meaning of ‘interrupts’ is to break the sequence of operation. While the CPU is executing a program, an ‘interrupt’ breaks the normal sequence of execution of instructions, diverts its execution to some other program called ***Interrupt Service*** *Routine* (TSR). After executing TSR, the control is transferred back again to the main program which was being executed at the time of interruption.

Whenever a number of devices interrupt a CPU at a lime, and if the processor is able to handle them properly, it is said to have ***multiple interrupt processing capability.***

Need for Interrupt**:** Interrupts are particularly useful when interfacing I/O devices that provide or require data at relatively low data transfer rate.

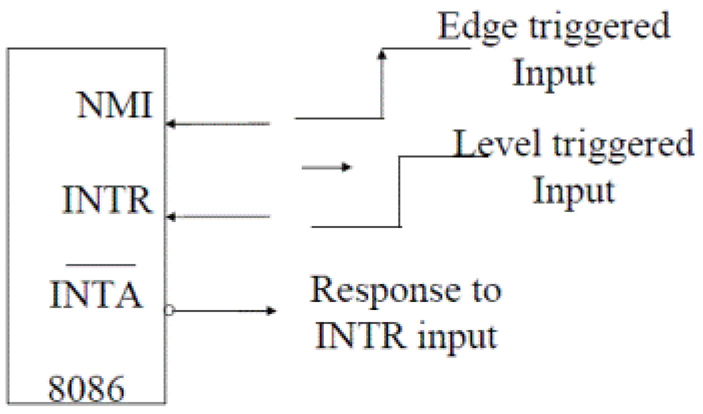
Sources of Interrupts in 8086: There are two pins for Interrupts in 8086. These are:

- *Hardware Interrupts (External Interrupts) – INTR, NMI*
- *Software Interrupts (Internal Interrupts and Instructions) – INT n instructions*

(i) ***Hardware Interrupts (External Interrupts)-***

The Intel microprocessors 8086 **support hardware interrupts** through:

- Two pins that allow interrupt requests, -INTR and NMI
- However one pin that acknowledges, INTR is INTA.



INTR and NMI

1. **INTR** is a maskable hardware interrupt. The interrupt can be enabled/disabled using STI/CLI instructions or using more complicated method of updating the Interrupt Flag (IF).
2. The 1NTR, further, is of 256 types. The INTR types may be from **00 to FFH** (or 00 to *255)*. If more than one type of INTR interrupt occurs at a time, then an external chip called programmable interrupt controller is required to handle them. The same is the case for INTR interrupt input of 8085.

- 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.

3. **NMI** is a non-maskable interrupt which means that any interrupt request at NMI input cannot be masked or disabled by any means.

- This 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 than the maskable interrupt.

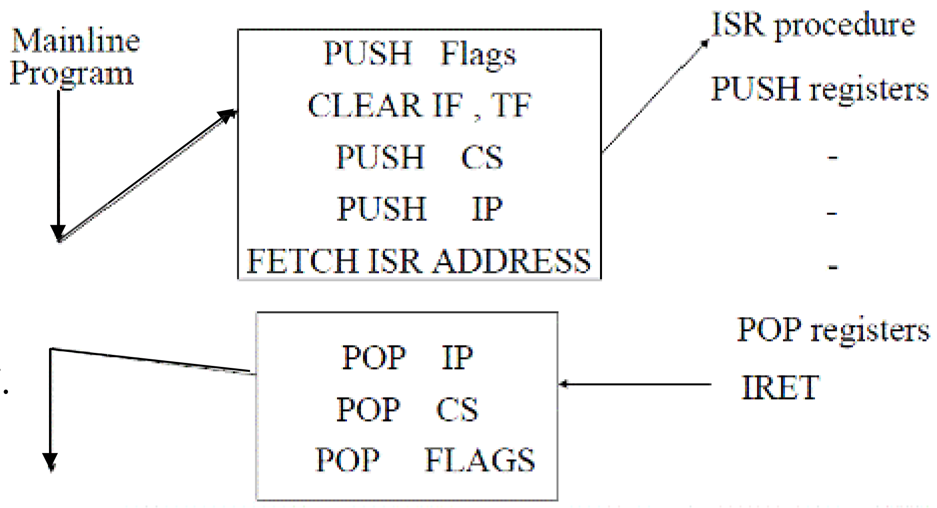
Ex: NMI, INTR.

(ii) <u>Software Interrupts (Internal Interrupts and Instructions)-</u>

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.
- **- Ex: INT n** (Software Instructions)
- **Control is provided** through:
  - IF and TF flag bits
  - IRET and IRETD

Action taken when Interrupts **ocurres**

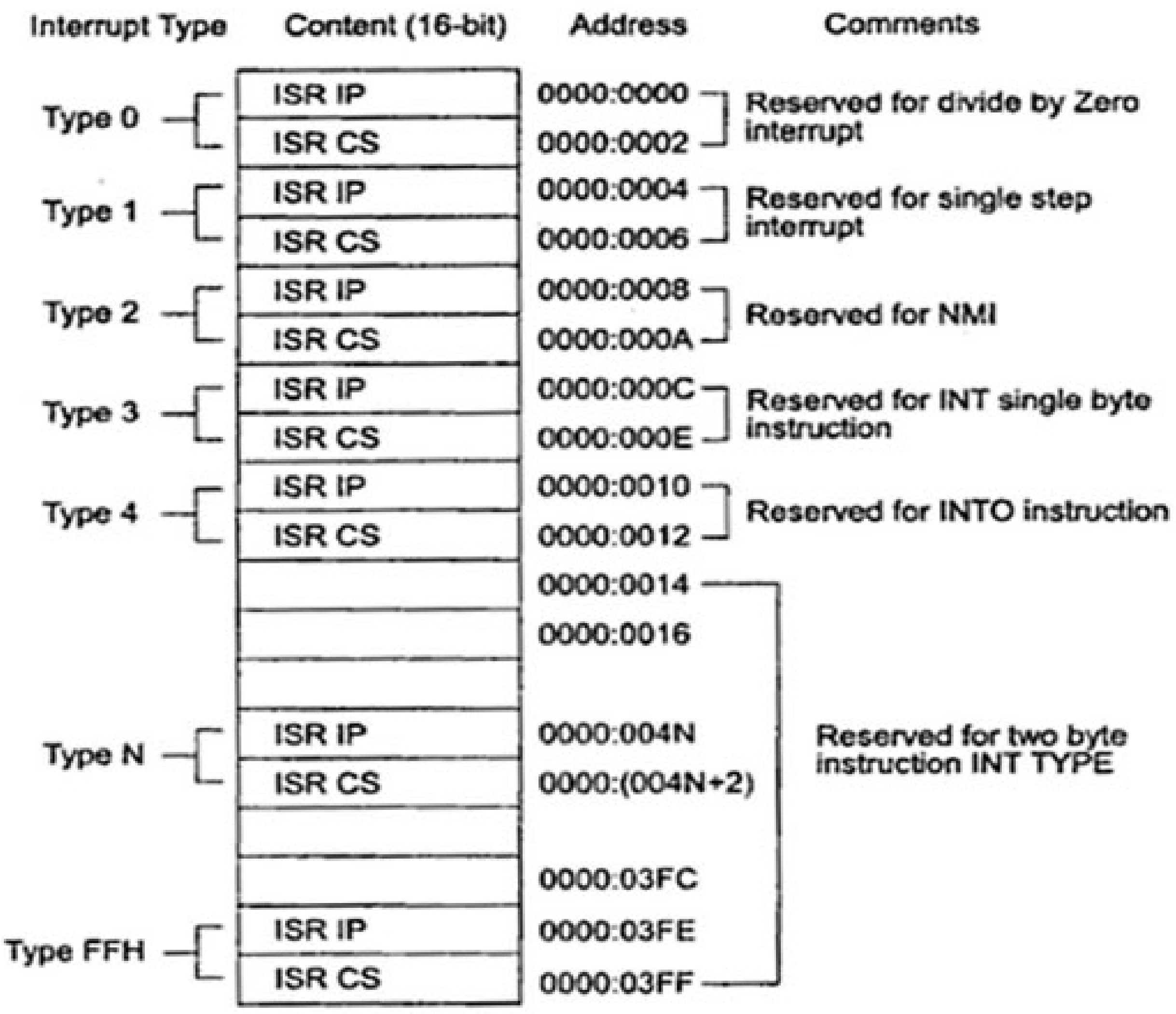


7.

decrements SP by 2 and pushes the flag register on the stack.

8. Disables INTR by clearing the IF.
9. It resets the TF in the flag Register.
10. It decrements SP by 2 and pushes CS on the stack.
11. It decrements SP by 2 and pushes IP on the stack.
12. Fetch the ISR address from the interrupt vector table.
13. After executing ISR (i.e., when IRET invoked) then IP, CS and Flag register content is popped so SP will be decremented gradually.

Interrupt Vector Table



- Every external and internal interrupt is assigned with a type (N), that is either implicit (in case of NMI, TRAP and divide by zero) or specified in the instruction INT N (in case of internal interrupts).
- In case of external interrupts, the type is passed to the processor by an external hardware like programmable interrupt controller.
- The 8086 supports a total of 256 types of the interrupts. i.e. from 00 to FFH. Each interrupt requires 4 bytes. i.e. two bytes each for IP and CS of its TSR. Thus a total of 1024 bytes are required for 256 interrupt types, hence the interrupt vector table starts at location 0000:0000 and ends at 0000:03FFH.
- The interrupt vector table contains the IP and CS of all the interrupt types stored sequentially from address 0000:0000 to 0000:03FF H.
- The interrupt type N is multiplied by 4 and the hexadecimal multiplication obtained gives the offset address in the zeroth code segment at which the IP and CS addresses of the interrupt service routine (ISR) are stored.

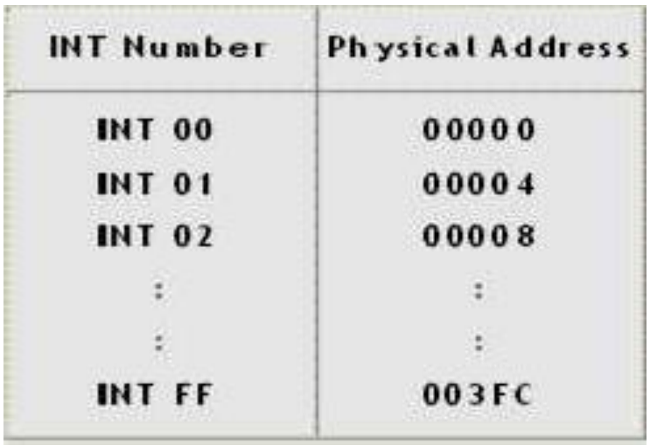
$IP = (4 \times n)_H$
$CS = (4 \times n)_H + 2$; where n-type of interrupt

- The execution automatically starts from the new CS: IP.

Interrupt Type

Type 0 – Type 4 → Intel predefined
Type 5 – Type 31 →Reserved
Type 32 – Type 255 → User defined Maskable Interrupt

Functions associated with Type 0 – Type 4 → Intel predefined



INT Number	Physical Address
INT 00	00000
INT 01	00004
INT 02	00008
:	:
:	:
INT FF	003FC

Type 0 (divide error)

- It is invoked by the microprocessor whenever there is an attempt to divide a number by zero.
- ISR is responsible for displaying the message "Divide Error" on the screen.
- IP:00000, CS:00002

Type 1 (Trap or Single step)

For single stepping the trap flag must be 1.

- After execution of each instruction, 8086 automatically jumps to 00004H to fetch 4 bytes for CS: IP of the ISR.
- The job of ISR is to dump the registers on to the screen

Type 2 (Non maskable Interrupt)

- Whenever NMI pin of the 8086 is activated by a high signal (5v), the CPU Jumps to physical memory location 00008 to fetch CS: IP of the ISR associated with NMI.

Type 3 (Break point)

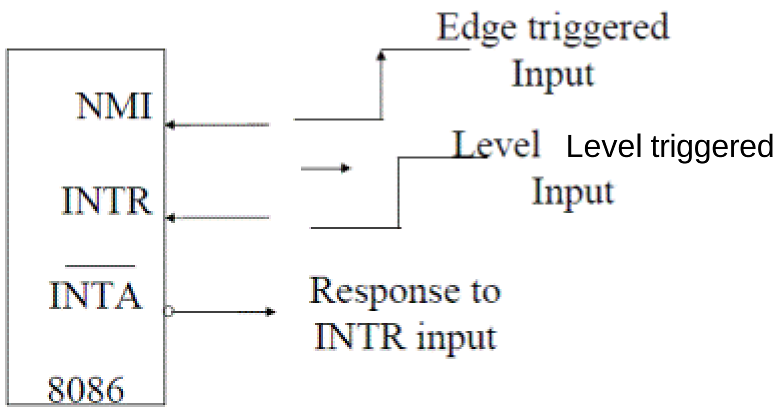
- A break point is used to examine the CPU and memory after the execution of a group of Instructions.
- It is one byte instruction whereas other instructions of the form "INT 3" are 2 byte instructions.

Type 4 (Signed number overflow)

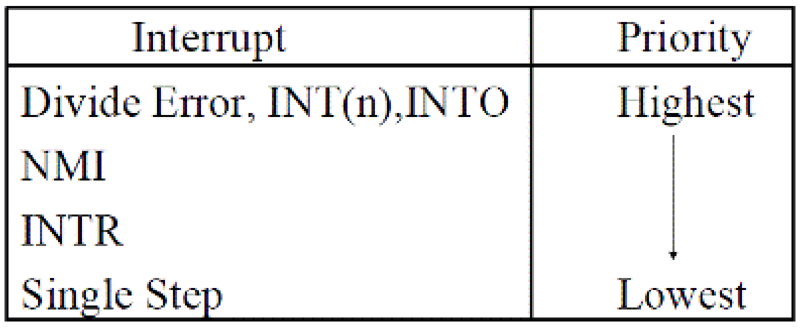
- There is an instruction associated with this INT 0 (interrupt on overflow).
- If INT 0 is placed after a signed number arithmetic as IMUL or ADD the CPU will activate Type 4 if 0F = 1.
- In case where 0F = 0 , the INT 0 is not executed but is bypassed and acts as a NOP

Performance of Hardware Interrupts

- NMI : Non maskable interrupts - TYPE 2 Interrupt
- INTR : Interrupt request - Between 20H and FFH



Interrupt Priority Structure



Interrupt	Priority
Divide Error, INT(n),INTO	Highest
NMI	
INTR	
Single Step	Lowest