LECTURE NOTES

ON

MICROPROCESSOR & MICROCONTROLLER

(INTEL-8086)

Prepared by

DR. MINU SAMANTRAY

DEPARTMENT OF ELECTRONICS & TELECOMMUNICATION ENGINEERING

TRIDENT ACADEMY OF TECHNOLOGY, BHUBANESWAR

Features of Intel 8086 microprocessor:

- Launched in 1978 by Intel Incorporation, USA.
- 16- Bit microprocessor (word length=16-bits means its ALU, internal registers, and most of the instructions are designed to work with 16-bit binary value), contains approximately 29,000 transistors.
- Supports 8/16-bits data operand.
- VLSI chip fabricated using the HMOS technology.
- Rounding out the 40- pins configuration in DIP (Dual in-line Package).
- One supply voltage, +5V and one clock input frequency up to 5 MHz (there are other two versions of the 8086, the 8086-2, which permits a clock frequency of up to 8 MHz, and the 8086-1, which can handle up to 10 MHz).
- 20 address lines, so can address up to 2^{20} (1MB) of memory.
- 16 data lines, which multiplexed with address lines.
- Includes multiprogramming features.
- Supports 8087 numeric data processor.
- Employs parallel processing to enhance the speed of execution.
- Includes memory segmentation to address 1MB of memory using 16-bit register.

Segment Memory Concept

Reason for having segmentation

As the address lines of the 8086 microprocessor is 20, so each location in the memory can be identified by 20-bit address. Whenever microprocessor accessed the memory the 20-bit address must be stored in the 20-bit register, but 8086 is designed to support 16-bit architecture, so no 20-bit registers are available in 8086 to hold the address. To avoid this problem the segmented memory is used, i.e., by using 16-bit register the whole 1MB of memory can be addressed.

Concept

In this concept the total 1MB of memory is logically divided into numbers of segments, where the maximum length of each segment is up to 64Kbytes. So to accommodate the 1MB of memory, and assuming the maximum length of 64KB of a segment, 16 segments are possible.

Considering one particular segment, each memory location is identified by two logical address components.

- Segment address: this 16-bits logical address indicates the starting address of a segment, and which is fixed for a particular segment.
- Offset address: This 16-bits logical address indicates the individual locations in that particular segment, and which varies location wise.

The offset address sometimes referred as effective address (EA). The Intel manuals tend to use the term ‘effective address’ when discussing the machine language and the term “Offset address” when discussing the assembler language.

Using the two above logical address components the 20-bits physical address can be calculated. Sometimes "displacement" is also used for calculation of physical address which is discussed in the later portion.

Physical Address Calculation

The 20-bit physical address can be calculated by using the following formula.

$$(\text{Physical Address})_{20\text{-bits}} = [(\text{Segment Address})_{16\text{-bits}} \times 16_{10} \text{ (or 10H)}] + (\text{Offset Address})_{16\text{-bits}}$$

When the physical address is calculated to get the multiplication of 16_{10} or 10H, segment address is actually shifted left 4 times to append a 0 (i.e. 0H or 0000_2) on the right hand side, then it is simply added with offset address to get the 20-bit physical address (sometimes called linear address).

For example if the logical addresses are given 2050H: 3000H, then to calculate the physical address

0010 0000 0101 0000 0000 = 20500H (Shifted version of Segment address)

0011 0000 0000 0000 = 3000H (Offset address)

0010 0011 0101 0000 0000 = 23500H (20- bits physical address/linear address)

Taking another example, if the logical address is 6789H: 5555H, to get the physical address

P.A = 67890H+5555H= 6CDE5H

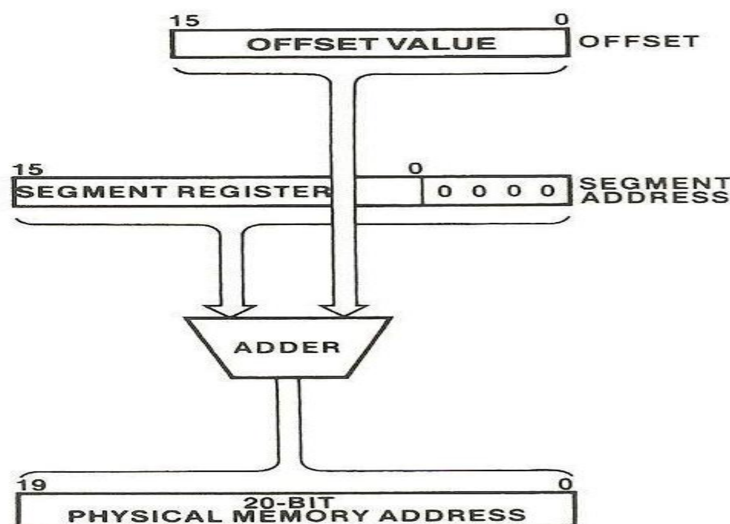
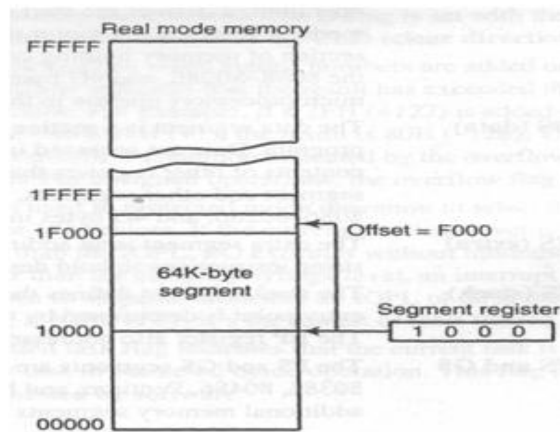


Fig 2.1(physical address calculation)



Overlapping and Non-Overlapping Segments

As we know the address bus of 8086 is 20, so it address up to 1MB OF MEMORY in the range of 00000H to FFFFFH as shown in figure 2.2.

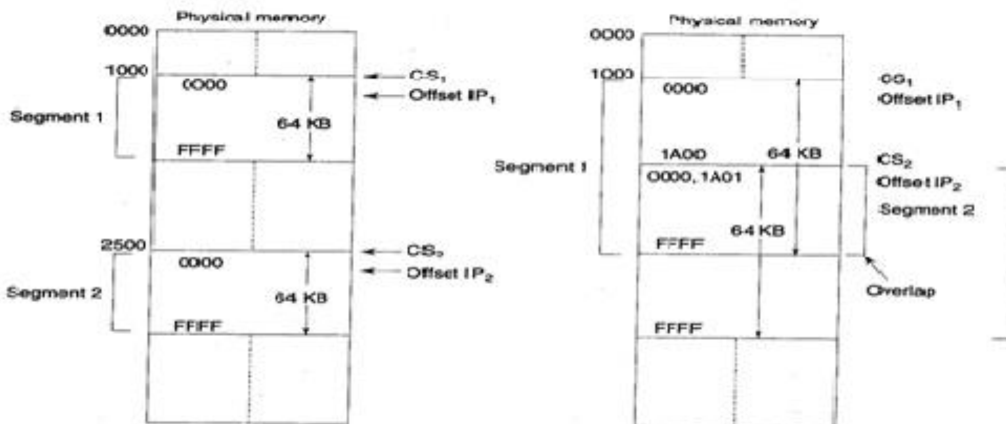


Fig 2.2 (Physical memory)

But in the segmented memory concept, the whole memory is logically divided into number of segments and each memory location is associated with two address components, segment address and logical address.

By suitable choosing the segment address some area of memory can be use as common area for more than one segment called overlapping segment. As shown in figure 2.3 one segment is started from the address 10000H of having segment address 1000H. The segment ranges from 10000H to 1FFFFH. But if another segment address is chosen 1A00H then the segment ranges from 1A000H to 29FFFH. So the area of memory ranges from 1A000H to 1FFFFH can be referred as overlapping area or overlapping segment.

If the maximum size of the segment is chosen 64KB then sixteen segments are possible. For non-overlapping segment, to express all the segments, the segment addresses should be chosen as shown in figure 2.4. The segment addresses are 0000H, 1000H, 2000H, 3000H, 4000H, 5000H, 6000H, 7000H, 8000H, 9000H, A000H, B000H, C000H, D000H, E000H, and F000H.

Advantages of using segment memory concept:

- Allow the memory capacity to be 1 MB even though the address associated with the individual instructions is only 16 bits wide.
- Allow the instructions, data, or stack portion of a program to be more than 64KB long by using more than one code, data, or stack segment.
- Facilitate the use of separate memory areas for a program, its data, and the stack.
- Permit a program and/or its data to be put into different areas of memory each time the program is executed.

Types of segment

There are four types of segments available in segmented memory of 8086.

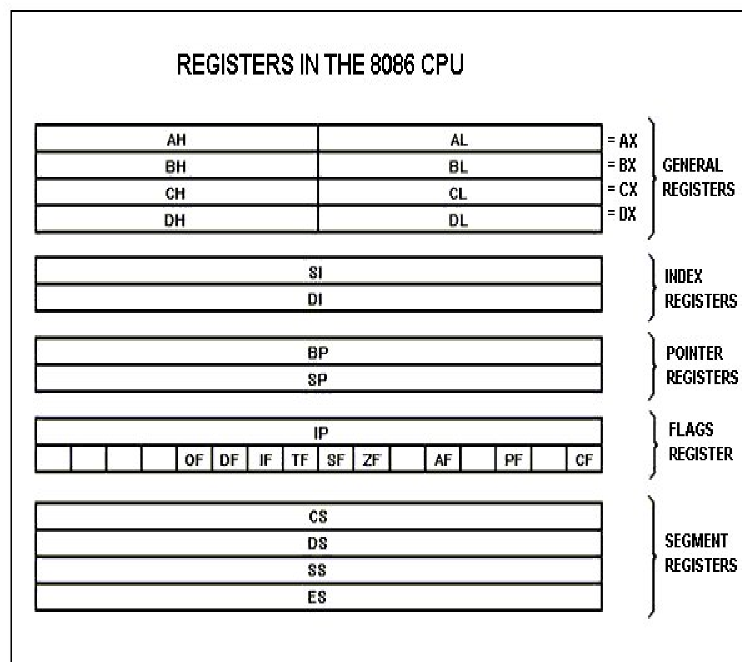
- Code Segment (CS): Used to store code or instructions or programs only.
- Data Segment (DS): Used to store data only.
- Extra segment (ES): Used to store the data only.
- Stack segment (SS): used to store stack only.

Reserved locations:

- 00000H - 003FFH are reserved for interrupt vectors. Each interrupt vector is a 32-bit pointer in format segment: offset.
- FFFF0H - FFFFFH - after RESET the processor always starts program execution at the FFFF0H address.

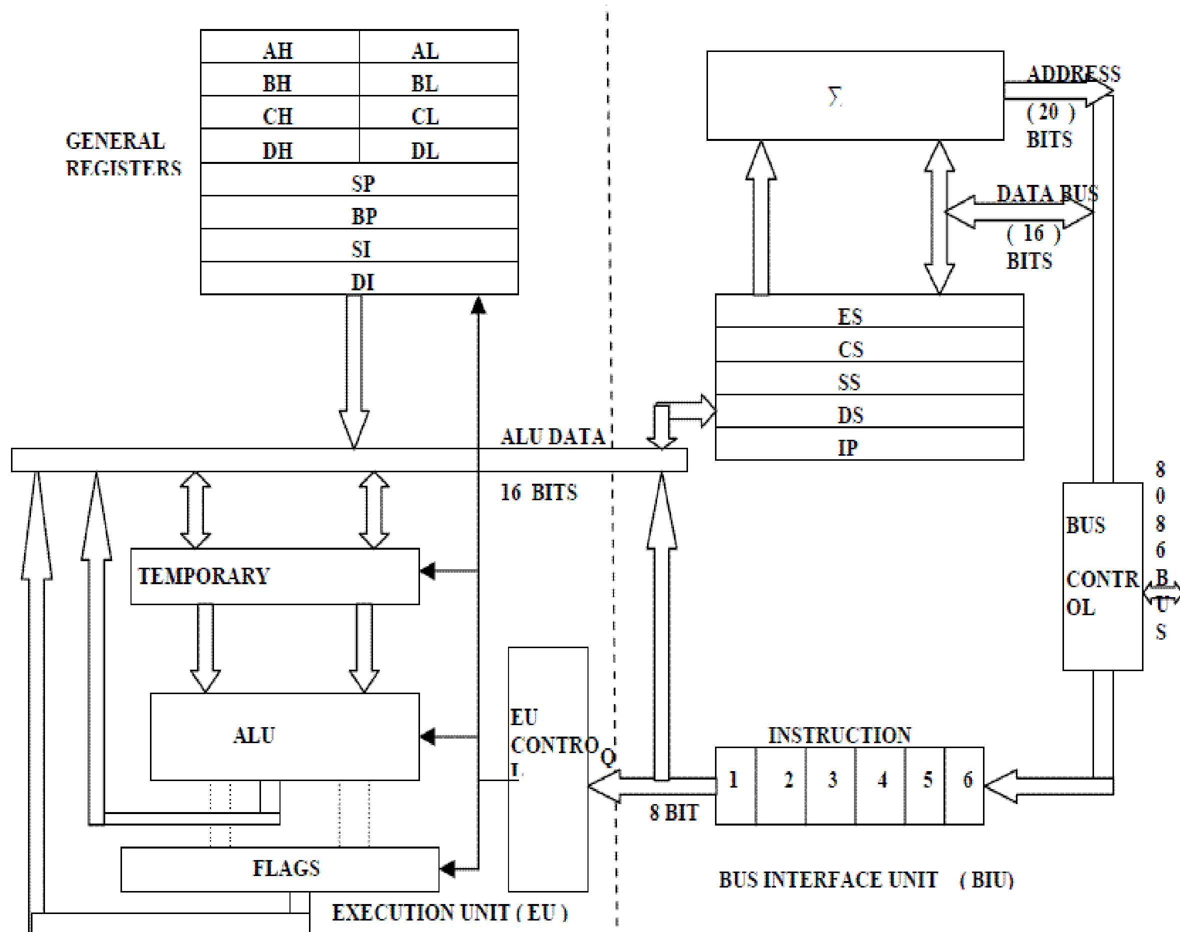
ARCHITECTURE OF 8086

Programming model of 8086



Functional Units of 8086

The internal functions of the 8086 processor are partitioned logically into two processing units. The first is the Bus Interface Unit (BIU) and the second is the Execution Unit (EU) as shown in the block diagram of Figure 1. These units can interact directly but for the most part perform as separate asynchronous operational processors.



Execution Unit (EU)

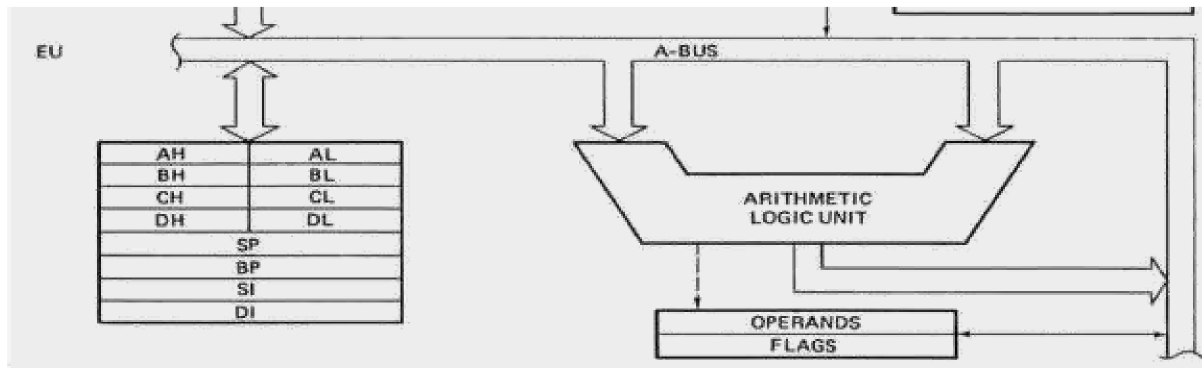
The functions of execution unit are:

- To tell BIU where to fetch the instructions or data from.
- To decode the instructions.
- To execute the instructions.

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

The EU contains the control circuitry to perform various internal operations with the help of some registers like general purpose registers (AX, BX, CX, DX), pointer registers (SP, BP) and index registers (SI, DI). A decoder in EU decodes the instruction fetched memory to

generate different internal or external control signals required to perform the operation. EU has 16-bit ALU, which can perform arithmetic and logical operations on 8-bit as well as 16-bit.



General Purpose Registers

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

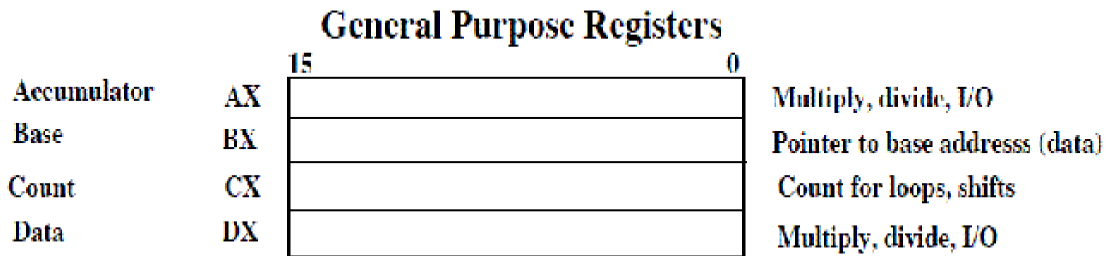
- **Base** register consists of two 8-bit registers BL and BH, which can be combined together and used as a 16-bit register BX. BL in this case contains the low-order byte of the word, and BH contains the high-order byte.

  It is the only GPR, which is used to store the offset address for the data segment i.e BX register usually contains a data pointer used for based, based indexed or register indirect addressing.

- **Count** register consists of two 8-bit registers CL and CH, which can be combined together and used as a 16-bit register CX. When combined, CL register contains the low order byte of the word, and CH contains the high order byte.
  - Count register can be used in Loop, shift/rotate instructions and as a counter in string manipulation.
  - CL can be used as 8 bit counter register whereas CX used as 16 bit counter.
  - When used along with LOOP instruction, it is decremented automatically without using the DEC instruction.

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

- In case of 32/16 bit division, it holds higher 16 bit of dividend before the division operation and stores the reminder part of the result.



Register	Purpose
AX	Word multiply, word divide, word I /O
AL	Byte multiply, byte divide, byte I/O, decimal arithmetic
AH	Byte multiply, byte divide
BX	Store address information
CX	String operation, loops
CL	Counter, Variable shift and rotate
DX	Word multiply, word divide, indirect I/O

Pointers and Index registers

Pointer registers are used to point to a particular location either in memory or stack from which the data is to be read or to which the data is to be written. Index registers are useful in implementing sophisticated addressing modes (identifying the location of the operands). The EU of 8086 has the set of pointers and index registers as shown in figure 2.5. The functions of pointers and index registers are explained as follows.

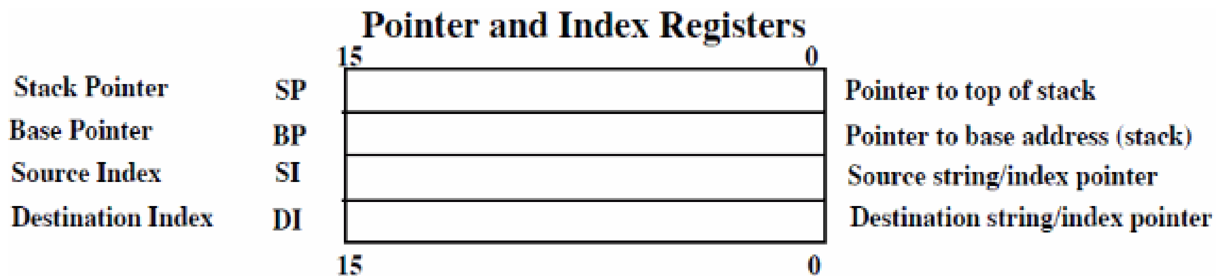
- **Stack Pointer (SP)** is a 16-bit register pointing to program stack, i.e it is used to store the offset address for the stack segment. It is also used to save the program status internally during the interrupt process and in the execution time of PUSH and POP instruction.

- **Base Pointer (BP)** is a 16-bit register pointing to data in stack segment. BP register is usually used for based, based indexed or register indirect addressing. Generally it is used with some displacement. It can be used to access data in other segments.

- **Source Index (SI)** is a 16-bit register. SI is used for indexed, based indexed and register indirect addressing, as well as a source data addresses in string manipulation instructions. When string operations are performed, the SI register points to source memory locations in the data segment which is addressed by the DS register. Thus, SI is associated with the DS in string operations.

- **Destination Index (DI)** is a 16-bit register. DI is used for indexed, based indexed and register indirect addressing, as well as a destination data addresses in string manipulation instructions. When string operations are performed, the DI register points to memory locations in the data segment which is addressed by the ES register. Thus, DI is associated with the ES in string operations.

  As SI is used the points the source location in the DS and DI is used to point the destination location in the ES in the string manipulation that is why the name is source and destination index respectively.



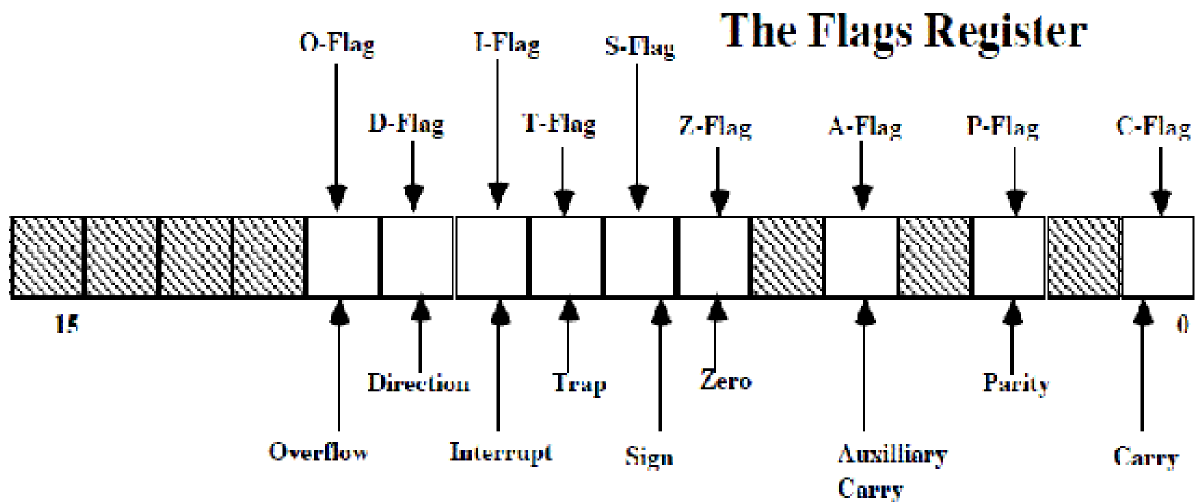
Flag Register

Flag Register contains a group of status bits called flags that indicate the status of the CPU or the result of arithmetic operations. There are two types of flags:

1. The **status flags** which reflect the result of executing an instruction. The programmer cannot set/reset these flags directly.

2. The **control flags** enable or disable certain CPU operations. The programmer can set/reset these bits to control the CPU's operation.

Nine individual bits of the status register are used as control flags (3 of them) and status flags (6 of them).The remaining 7 are not used. A flag can only take on the values 0 and 1. We say a flag is set if it has the value 1. The status flags are used to record specific characteristics of arithmetic and of logical instructions.



Control Flags: There are three control flags

1. The Direction Flag (D): Affects the direction of moving data blocks by such instructions as MOVS, CMPS and SCAS.

When DF=0, the the content of SI and DI is automatically incremented by 1 or 2 in case of any string instruction and this mode is called auto-incrementing mode.

When DF=1, the the content of SI and DI is automatically decremented by 1 or 2 in case of any string instruction and this mode is called auto-decrementing mode.

2. The Interrupt Flag (I): Dictates whether or not system interrupts can occur. Interrupts are actions initiated by hardware block such as input devices that will interrupt the normal execution of programs. The flag values are 0 = disable interrupts or 1 = enable interrupts and can be manipulated by the CLI (clear I) and STI (set I) instructions.

3. The Trap Flag (T): Determines whether or not the CPU is halted after the execution of each instruction. When this flag is set (i.e. = 1), the programmer can single step through his program to debug any errors. When this flag = 0 this feature is off.

When TF=1, after each instruction is executed, an internal interrupt is generated and the control is transfer to its vector address to execute the corresponding the ISR for showing the value of the register or memory location for debugging purpose.

Status Flags: There are six status flags

1. **The Carry Flag (C):** This flag is set when the result of an unsigned arithmetic operation is too large to fit in the destination register. This happens when there is an end carry in an addition operation or there an end borrows in a subtraction operation. A value of 1 = carry and 0 = no carry.

2. **The Overflow Flag (O):** This flag is set when the result of a signed arithmetic operation is too large to fit in the destination register (i.e. when an overflow occurs). Overflow can occur

when adding two numbers with the same sign (i.e. both positive or both negative). A value of 1 = overflow and 0 = no overflow.

3. **The Sign Flag (S):** This flag is set when the result of an arithmetic or logic operation is negative. This flag is a copy of the MSB of the result (i.e. the sign bit). A value of 1 means negative and 0 = positive.

4. **The Zero Flag (Z):** This flag is set when the result of an arithmetic or logic operation is equal to zero. A value of 1 means the result is zero and a value of 0 means the result is not zero.

5. **The Auxiliary Carry Flag (A):** This flag is set when an operation causes a carry from bit 3 to bit 4 (or a borrow from bit 4 to bit 3) of an operand. A value of 1 = carry and 0 = no carry.

6. **The Parity Flag (P):** This flags reflects the number of 1s in the result of an operation. If the number of 1s is even its value = 1 and if the number of 1s is odd then its value = 0.

Since flag register reflects the happenings inside the 8086 microprocessor, it is called the Program Status Word (PSW). The contents of PSW, accumulator and other registers are saved in the stack during the handling of interrupt. Flag registers can be summarized as follows.

Bit Mnemonic	Bit Name	Reset State	Function
OF	Overflow Flag	0	If OF is set, an arithmetic overflow has occurred.
DF	Direction Flag	0	If DF is set, string instructions are processed high address to low address. If DF is clear, strings are processed low address to high address.
IF	Interrupt Enable Flag	0	If IF is set, the CPU recognizes maskable interrupt requests. If IF is clear, maskable interrupts are ignored.
TF	Trap Flag	0	If TF is set, the processor enters single-step mode.
SF	Sign Flag	0	If SF is set, the high-order bit of the result of an operation is 1, indicating it is negative.
ZF	Zero Flag	0	If ZF is set, the result of an operation is zero.
AF	Auxiliary Flag	0	If AF is set, there has been a carry from the low nibble to the high or a borrow from the high nibble to the low nibble of an 8-bit quantity. Used in BCD operations.
PF	Parity Flag	0	If PF is set, the result of an operation has even parity.
CF	Carry Flag	0	If CF is set, there has been a carry out of, or a borrow into, the high-order bit of the result of an instruction.

Bus Interface Unit (BIU)

The bus interface unit is responsible for performing all external bus operations.
Specifically it has the following functions:

- Instruction fetching, Instruction queuing, Operand fetching and storage, Address relocation and Bus control.

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

- This queue permits pre fetch of up to six bytes of instruction code.

BIU contains Instruction queue (IQ), Segment registers (CS, DS, ES, SS), Instruction pointer (IP), and Address adder.

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

Instruction Pointer (IP) is a 16-bit register. This is a crucially important register which is used to control which instruction the CPU executes. The IP, or program counter, is used to store the memory location of the next instruction to be executed. The CPU checks the program counter to ascertain which instruction to carry out next. It then updates the program counter to point to the next instruction. Thus the program counter will always point to the next instruction to be executed.

Segment Register

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

		Segment Registers
Code Segment	CS	
Data Segment	DS	
Stack Segment	SS	
Extra Segment	ES	

Code segment (CS) is a 16-bit register containing address of 64 KB segment with processor instructions. The processor uses CS segment for all accesses to instructions referenced by instruction pointer (IP) register. CS register cannot be changed directly. The CS register is automatically updated during far jump, far call and far return instructions.

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

Data segment (DS) is a 16-bit register containing address of 64KB segment with program data. By default, the processor assumes that all data referenced by general registers (AX,

BX, CX, DX) and index register (SI, DI) is located in the data segment. DS register can be changed directly using POP and LDS instructions.

Extra segment (ES) used to hold the starting address of Extra segment. Extra segment is provided for programs that need to access a second data segment. Segment registers cannot be used in arithmetic operations.

Instruction Queue (IQ.

- The BIU uses a mechanism known as an instruction stream queue to implement pipeline architecture.
- This queue permits pre-fetch of up to six bytes of instruction code. Whenever the queue of the BIU is not full, it has room for at least two more bytes and at the same time the EU is not requesting it to read or write operands from memory, the BIU is free to look ahead in the program by prefetching the next sequential instruction.
- These prefetching instructions are held in its FIFO queue. With its 16 bit data bus, the BIU fetches two instruction bytes in a single memory cycle.
- After a byte is loaded at the input end of the queue, it automatically shifts up through the FIFO to the empty location nearest the output.
- The EU accesses the queue from the output end. It reads one instruction byte after the other from the output of the queue. If the queue is full and the EU is not requesting access to operand in memory.
- These intervals of no bus activity, which may occur between bus cycles are known as Idle state.
- If the BIU is already in the process of fetching an instruction when the EU request it to read or write operands from memory or I/O, the BIU first completes the instruction fetch bus cycle before initiating the operand read / write cycle.
- The BIU also contains a dedicated adder which is used to generate the 20bit physical address that is output on the address bus. This address is formed by adding an appended 16 bit segment address and a 16 bit offset address.
- For example**:** The physical address of the next instruction to be fetched is formed by combining the current contents of the code segment CS register and the current contents of the instruction pointer IP register.
- The BIU is also responsible for generating bus control signals such as those for memory read or write and I/O read or write.

8086 INSTRUCTION SETS

The instructions of 8086 are classified into SIX groups. They are:

1. Data transfer instructions
2. Arithmetic instructions
3. Bit manipulation instructions
4. String instructions
5. Program execution transfer instructions
6. Process control instructions

1. DATA TRANSFER INSTRUCTIONS

The DATA TRANSFER INSTRUCTIONS are those, which transfers the DATA from any one source to any one destination. The data may be of any type. They are again classified into four groups. They are:

General – purpose byte or word transfer instructions	Simple input and output port transfer instruction	Special address transfer instruction	Flag transfer instructions
MOV PUSH POP XCHG XLAT	IN OUT	LEA LDS LES	LAHF SAHF PUSHF POPF

2. ARITHMETIC INSTRUCTIONS

These instructions are those which are useful to perform Arithmetic calculations, such as addition, subtraction, multiplication and division. They are again classified into four groups. They are:

Addition instructions	Subtraction instructions	Multiplication instructions	Division instructions
ADD ADC INC AAA DAA	SUB SBB DEC NEG CMP AAS DAS	MUL IMUL AAM	DIV IDIV AAD CBW CWD

3. BIT MANIPULATION INSTRUCTIONS

These instructions are used to perform Bit wise operations.

Logical instructions	Shift instructions	Rotate instructions
NOT AND OR XOR TEST	SHL / SAL SHR SAR	ROL ROR RCL RCR

4. STRING INSTRUCTIONS

The string instructions function easily on blocks of memory. They are user friendly instructions, which help for easy program writing and execution. They can speed up the manipulating code. They are useful in array handling, tables and records.

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

5. PROGRAM EXECUTION TRANSFER INSTRUCTIONS

These instructions transfer the program control from one address to other address. (Not in a sequence). They are again classified into four groups. They are:

Unconditional transfer instructions	Conditional transfer instructions		Iteration control instructions	Interrupt instructions
CALL RET JMP	JA / JNBE JAE / JNB JB / JNAE JBE / JNA JC JE / JZ JG / JNLE JGE / JNL JL / JNGE	JLE / JNG JNC JNE / JNZ JNO JNP / JPO JNS JO JP / JPE JS	LOOP LOOPE / LOOPZ LOOPNE / LOOPNZ JCXZ	INT INTO IRET

6. PROCESS CONTROL INSTRUCTIONS

These instructions are used to change the process of the Microprocessor. They change the process with the stored information. They are again classified into two groups. They are:

Flag set / clear instructions	External hardware synchronization instructions
STC CLC CMC STD CLD STI CLI	HLT WAIT ESC LOCK NOP

DATA TRANSFER INSTRUCTION

MOV Instruction - MOV destination, source

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.

PUSH Instruction - PUSH source

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 .

POP Instruction - POP destination

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].

XCHG Instruction - Exchange XCHG destination, source

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 is not supported.

Syntax: 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.

XLAT/XLATB Instruction - Translate a byte in AL

XLAT exchanges the byte in AL register from the user table index to the table entry, addressed by BX. It transfers 16 bit information at a time. The no-operands form (XLATB) provides a "short form" of the XLAT instructions.

Example: MOV AL, [BX+AL]

SIMPLE INPUT AND OUTPUT PORT TRANSFER INSTRUCTION

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

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
```

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.
```

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

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 : (a) Port number is specified by an immediate byte constant, (0 - 255).It is also called as fixed port form. (b) Port number is provided in the DX register (0 – 65535)

Example: (a)

```
OUT 3BH, AL         ;Copy the contents of the AL to port 3Bh
OUT 2CH,AX          ;Copy the contents of the AX to port 2Ch
```

Example: (b)

```
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
```

SPECIAL ADDRESS TRANSFER INSTRUCTION

LEA Instruction - Load Effective Address

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.

Syntax – LEA register, source

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

LDS Instruction - Load register and Ds with words from memory

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.

Syntax – LDS register, memory address of first word or 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.

LES Instruction - Load register and ES with words from memory

This instruction loads a 32- bit pointer from the memory address specified to destination register and Extra Segment. The offset is placed in the destination register and the segment is placed in Extra Segment. Using this instruction, the loading of far pointers may be simplified.

Syntax – LES register, memory address of first word

FLAG TRANSFER INSTRUCTIONS

LAHF Instruction - Load Register AH From Flags

LAHF instruction copies the value of SF, ZF, AF, PF, and 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.

SAHF instruction - Store AH Register into FLAGS

SAHF instruction transfers the bits 0-7 of AH of SF, ZF, AF, PF, and CF, into the Flag register.

PUSHF Instruction - Push flag register on the stack

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

POPF Instruction - Pop word from top of stack to flag - registers.

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.

ARITHMETIC INSTRUCTIONS

ADDITION

ADD Instruction - ADD destination, source

These instructions add a number from source to a number from some destination and put the result in the specified destination. 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.

ADC Instruction - Add with carry

After performing the addition, the add with carry instruction ADC, adds the status of the carry flag into the result.

EXAMPLE:

```
ADD AL,74H          ;Add immediate number 74H to content of AL
ADC CL,BL           ;Add contents of BL plus carry 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
```

INC Instruction - Increment - INC destination

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:

```
For example, if 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.
```

AAA Instruction - ASCII Adjust after Addition

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.
```

DAA Instruction - Decimal Adjust after Addition

The contents after addition are changed from a binary value to two 4-bit binary coded decimal (BCD) digits. S, Z, AC, P, CY flags are altered to reflect the results of the operation.

If the value of the low-order 4-bits in the accumulator is greater than 9 or if AC flag is set, the instruction adds 6 to the low-order four bits.

If the value of the high-order 4-bits in the accumulator is greater than 9 or if the Carry flag is set, the instruction adds 6 to the high-order four bits.

Example:

```
MOV AL, 0Fh      ; AL = 0Fh (15)
DAA              ; AL = 15h
```

SUBTRACTION

SUB Instruction - Subtract two numbers

These instructions subtract the source number from destination number destination and put the result in the specified destination. The source and destination must be of same type , means they must be a byte location or a word location.

SBB Instruction - Subtract with borrow SBB destination, source

SBB instruction subtracts source from destination, and then subtracts 1 from source if CF flag is set and result is stored destination and it is used to set the flag.

ie.,destination = destination -(source + CF)

Example:

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

Example:

Subtracting unsigned number

Considering 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

Considering CL = 00101110 = + 46 decimal BH = 01001010= + 74 decimal

SUB CL, BH ;CL = 11100100 = - 28 decimal, CF = 1, AF, ZF =0,SF = 1 result negative

DEC Instruction - Decrement destination register or memory DEC destination.

DEC instruction subtracts one from the operand and sets the flag according to the result. DEC instruction is treated as an unsigned binary number.

Example:

Considering AX =8000h

DEC AX ;After this instruction AX = 7999h
DEC BL ; Subtract 1 from the contents of BL register

NEG Instruction - From 2's complement – NEG destination

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

Considering 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

CMP Instruction - Compare byte or word -CMP destination, source.

The CMP instruction compares the destination and source ie., it subtracts the source from destination. The result is not stored anywhere. It neglects the results, but sets the flags accordingly. This instruction is usually used before a conditional jump instruction.

Example:

```
MOV AL, 5
MOV BL, 5
CMP AL, BL                  ; AL = 5, ZF = 1 (so equal!)
```

AAS Instruction - ASCII Adjust for Subtraction

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)
```

Example:(a)

Considering AL =0011 1001 =ASCII 9 and 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
```

Example:(b)

Considering AL = 0011 0101 =ASCII 5 and 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
```

DAS Instruction - Decimal Adjust after Subtraction

This instruction corrects the result (in AL) of subtraction of two packed BCD values. The flags which modify are AF, CF, PF, SF, and ZF.

if low nibble of AL > 9 or AF = 1 then: AL = AL – 6 and AF = 1

if AL > 9Fh or CF = 1 then: AL = AL - 60h CF = 1

Example:

MOV AL, 0FFh ; AL = 0FFh (-1)
DAS ; AL = 99h, CF = 1

MULTIPLICATION INSTRUCTIONS

MUL Instruction - Multiply unsigned bytes or words-MUL source

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 .

Syntax: MUL op

Example:

Considering 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.

IMUL Instruction - Multiply signed number-IMUL source

This instruction performs a signed multiplication. There are two types of syntax for this instruction. They are:

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:

Considering 69 * 14 ; AL = 01000101 = 69 decimal ; BL = 00001110 = 14 decimal
IMUL BL AX = 03C6H = + 966 decimal ,MSB = 0 because positive result .

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

AAM Instruction - ASCII adjust after Multiplication

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)
```

DIVISION INSTRUCTIONS

DIV Instruction - Unsigned divide- Div source

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:

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

IDIV Instruction - Divide by signed byte or word IDIV source

This instruction is used to divide a signed word by a signed byte or to divide a signed double word by a signed word. If source is a byte value, AX is divided by register and the quotient is stored in AL and the remainder in AH. If source is a word value, DX:AX is divided by register, and the quotient is stored in AL and the remainder in DX.

Example:

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

AAD Instruction - ASCII adjust before Division

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:

Considering 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
```

CBW Instruction - Convert signed Byte to signed word

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: Considering AX = 00000000 10011011 = - 155 decimal

```
CBW             ; Convert signed byte in AL to signed word in AX.
                ; Result in AX = 11111111 10011011 and  = - 155 decimal
```

CWD Instruction - Convert Signed Word to - Signed Double word

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:

Considering DX = 00000000 00000000 and AX = 11110000 11000111 = - 3897 decimal

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

BIT MANIPULATION INSTRUCTIONS

LOGICAL INSTRUCTIONS

NOT Instruction - Invert each bit of operand

NOT perform the bitwise complement of operand and stores the result back into operand itself.

Syntax–NOT destination

Example :

```
NOT BX       ; Complement contents of BX register. - DX =F038h
NOT DX       ; after the instruction DX = 0FC7h
```

AND Instruction - AND corresponding bits of two operands

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 .

Example :

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

OR Instruction - Logically OR corresponding of two operands

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, otherwise the bit in the result is set to 1.

Syntax–- OR destination, source.

Examples :

OR AH, CL ;CL is OR'ed 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.

XOR Instruction - Exclusive XOR destination, source

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.

Syntax– XOR destination, source.

Example: (Numerical)

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

TEST Instruction – AND operand to update flags

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 :

Considering 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 and AL=01010001 (unchanged),PF= 0,SF= 0,
;ZF = 1 because ANDing produced is 00

SHIFT INSTRUCTIONS

SAL/SHL Instruction - Shift operand bits left, put zero in LSB(s)

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.

Syntax - SAL/AHL destination, count

Example:

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

SHR Instruction - Shift operand bits right, put zero in MSB

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

Example:(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

Example:(3)

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

SAR Instruction - Shift operand bits right, new MAB = old MSB

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.

Syntax - SAR destination, count.

Example: (1)

Considering 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

Example: (2)

Considering BH = 11110011 = - 13 decimal, CF = 1
SAR BH, 1 ;Shifted signed byte in BH to right and BH = 11111001 = -7 decimal,
CF = 1.

ROTATE INSTRUCTIONS

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

ROL instruction rotates the bits in the operand specified by oper1 towards left by the count specified in oper2. 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.

Syntax - ROL destination, count.

Example: (1)

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

Example : (2)

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

ROR Instruction - Rotate all bits of operand right, LSB to MSB

ROR instruction rotates the bits in the operand oper1 towards right by count specified in op2. The last bit rotated is copied into CF.

Syntax - ROR destination, count

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.

Example: 2

Considering 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

Considering 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
```

RCL Instruction - Rotate operand around to the left through CF

RCL instruction rotates the bits in the operand specified by oper1 towards left by the count specified in oper2.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.

Syntax - RCL destination, source.

Example: 1

```
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: 2

```
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 - Rotate operand around to the right through CF

RCR Instruction - RCR instruction rotates the bits in the operand specified by operand1 towards right by the count specified in operand2.

Syntax - RCR destination, count

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 .

Example: 2

Considering CF = 1, BL = 00111000
RCR BL, 1 ; Result: BL = 10011100, CF =0 OF = 1 because MSB is changed to 1.

ADDRESSING MODES:

The different techniques by which the operand specified in an instruction is called addressing modes.

The following addressing modes used in 8086 are specifying the operand.

1. **REGISTER ADDRESSING MODE:**

In this operands are specified through registers

e.g: MOV AL, BL

ADD CX, BX

SUB CH, DL

2. **IMMEDIATE ADDRESSING MODE:**

In this the operand are specified in the instruction immediately.

e.g: MOV AL, 05

ADD CX, 6000

SUB CH, 50

3. **MEMORY ADDRESSING MODE:**

When the data is in the memory, the address of the data must be specified. Depending upon address specification, following memory addressing modes are available

(a) **DIRECT ADDRESSING MODE**

In this the offset address is specified directly.

e.g: MOV AL, [6000]
SUB CX, [5055]

(b) **BASE REGISTER ADDRESSING MODE**

In this the offset address is specified indirectly through base register.

e.g: MOV AL, [BX]
SUB CX, [BP + 0]

(c) **INDEX ADDRESSING MODE**

In this the offset address is specified indirectly through index register.

e.g: MOV AL, [SI]
SUB CX, [DI]

(d) **BASE INDEX ADDRESSING MODE**

In this the offset address is specified indirectly through index and base register.

e.g: MOV AL, [BX + SI]
SUB CX, [BX + DI]

(e) **RELATIVE BASE ADDRESSING MODE**

In this the offset address is specified indirectly through base register with some displacement.

e.g: MOV AL, [BX + 50]
SUB CX, [BX + 6000]
ADD AL, [BD + 05]

(f) **RELATIVE INDEX ADDRESSING MODE**

In this the offset address is specified indirectly through index register with some displacement.

e.g: MOV AL, [SI + 50]
MOV BX, [DI + 0350]

(g) **INTRASEGMENT DIRECT ADDRESSING MODE**

In this the destination offset address is specified directly either direct address or through any level name.

e.g: JMP 6000
JMP START
CALL BACK

(h) **INTRASEGMENT INDIRECT ADDRESSING MODE**

In this the destination offset address is specified indirectly through register on memory.

e.g: JMP BX
JMP [BX]
CALL [SI]

(i) **INTERSEGMENT DIRECT ADDRESSING MODE**

In this the destination segment and offset address is specified directly using FAR pointer.

e.g: START PROC FAR
JMP START

BACK PROC FAR
CALL BACK

(j) **INTERSEGMENT INDIRECT ADDRESSING MODE**

In this the destination segment and offset address is specified indirectly using DWORD pointer.

e.g: JMP DWORD PTR [BX]
CALL DWORD PTR [SI]

(k) **STRING ADDRESSING MODE**

All string instruction coming under this category.

e.g: MOVSB
MOVSW
CMP SB

4. **I/O ADDRESSING MODE**

All I/O instructions are coming under this category.

e.g: IN AL, 05
OUT DX, AL
IN AL, DX

5. **IMPLIED ADDRESSING MODE:**

In this case no separate operand is used,

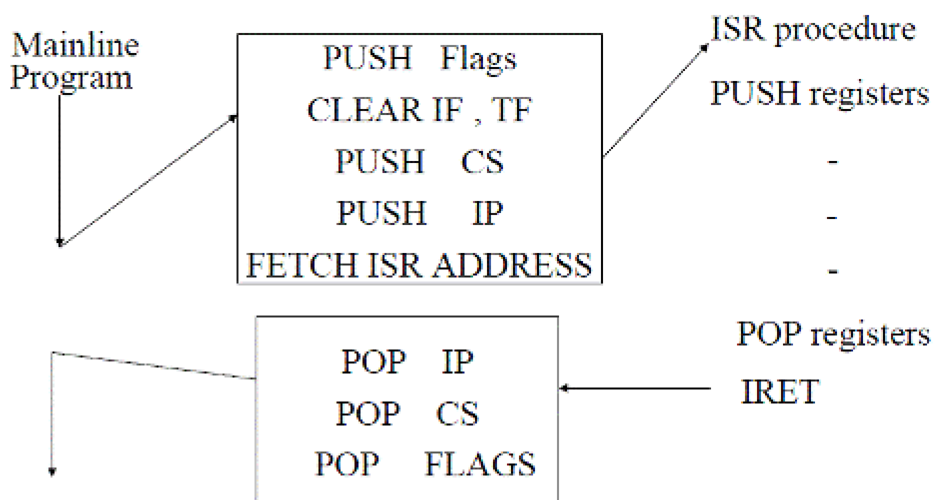
e.g: LLD
AAA
AAM
DAA
PUSHF

INTERRUPT

Definition: The meaning of 'interrupts' is to break the sequence of operation while the CPU is executing a program, on 'interrupt' breaks the normal sequence of execution of instructions, diverts its execution to some other program called Interrupt Service Routine (ISR).After executing ISR , the control is transferred back again to the main program. Interrupt processing is an alternative to polling.

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

INTERRUPT PROCESS



1. It decrements SP by 2 and pushes the flag register on the stack.
2. Disables INTR by clearing the IF.
3. It resets the TF in the flag Register.
5. It decrements SP by 2 and pushes CS on the stack.
6. It decrements SP by 2 and pushes IP on the stack.
6. Fetch the ISR address from the interrupt vector table.

Types of Interrupts: There are two types of Interrupts in 8086. They are:

i)Hardware Interrupts and

(ii)Software Interrupts

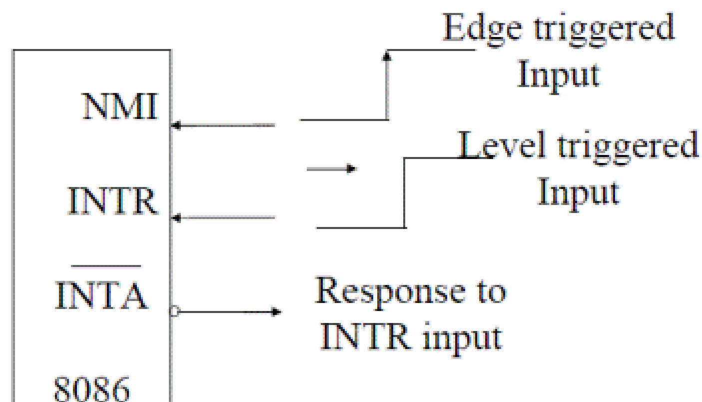
(i) Hardware Interrupts (External Interrupts). The Intel microprocessors support hardware interrupts through:

- Two pins that allow interrupt requests, INTR and NMI

- One pin that acknowledges, INTA, the interrupt requested on INTR.

INTR and NMI

- INTR is a maskable hardware interrupt. The interrupt can be enabled/disabled using STI/CLI instructions or using more complicated method of updating the FLAGS register with the help of the POPF instruction.
- When an interrupt occurs, the processor stores FLAGS register into stack, disables further interrupts, fetches from the bus one byte representing interrupt type, and jumps to interrupt processing routine address of which is stored in location 4 * <interrupt type>. Interrupt processing routine should return with the IRET instruction.
- NMI is a non-maskable interrupt. Interrupt is processed in the same way as the INTR interrupt. Interrupt type of the NMI is 2, i.e. the address of the NMI processing routine is stored in location 0008h. This interrupt has higher priority than the maskable interrupt.
- – Ex: NMI, INTR.



(ii) Software Interrupts (Internal Interrupts and Instructions) .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.

INT 00 (divide error)

- INT00 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

INT 01

- 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

INT 02 (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.

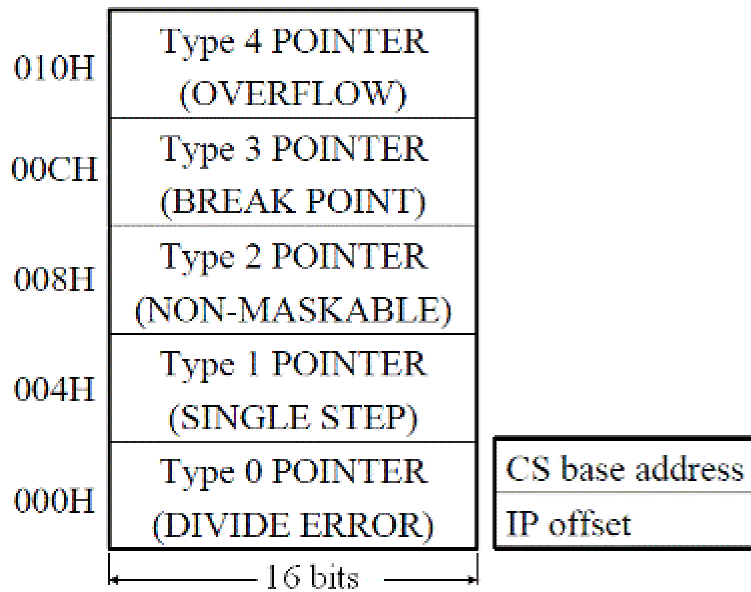
INT 03 (break point)

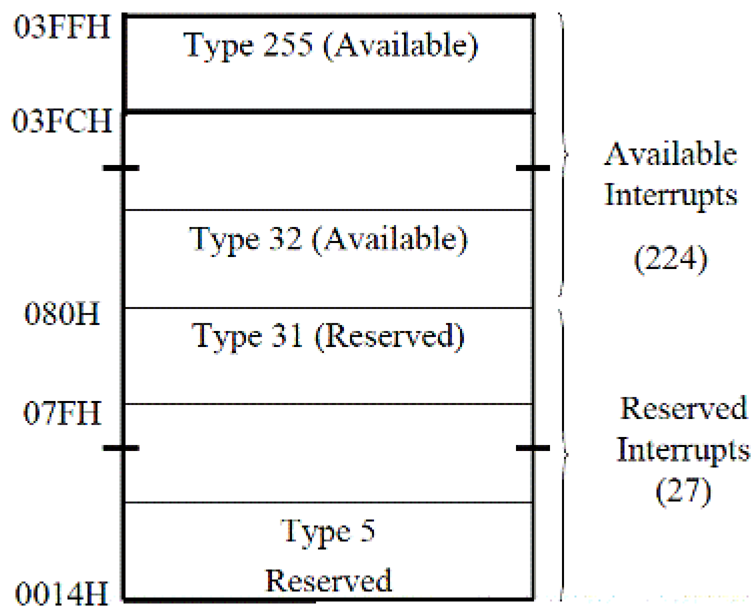
- A break point is used to examine the cpu and memory after the execution of a group of Instructions.
- It is one byte instruction whereas other instructions of the form "INT nn" are 2 byte instructions.

INT 04 (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 INT 04 if 0F = 1.
- In case where 0F = 0 , the INT 0 is not executed but is bypassed and acts as a NOP.

INTERRUPT VECTOR TABLE





INTERRUPT PRIORITY

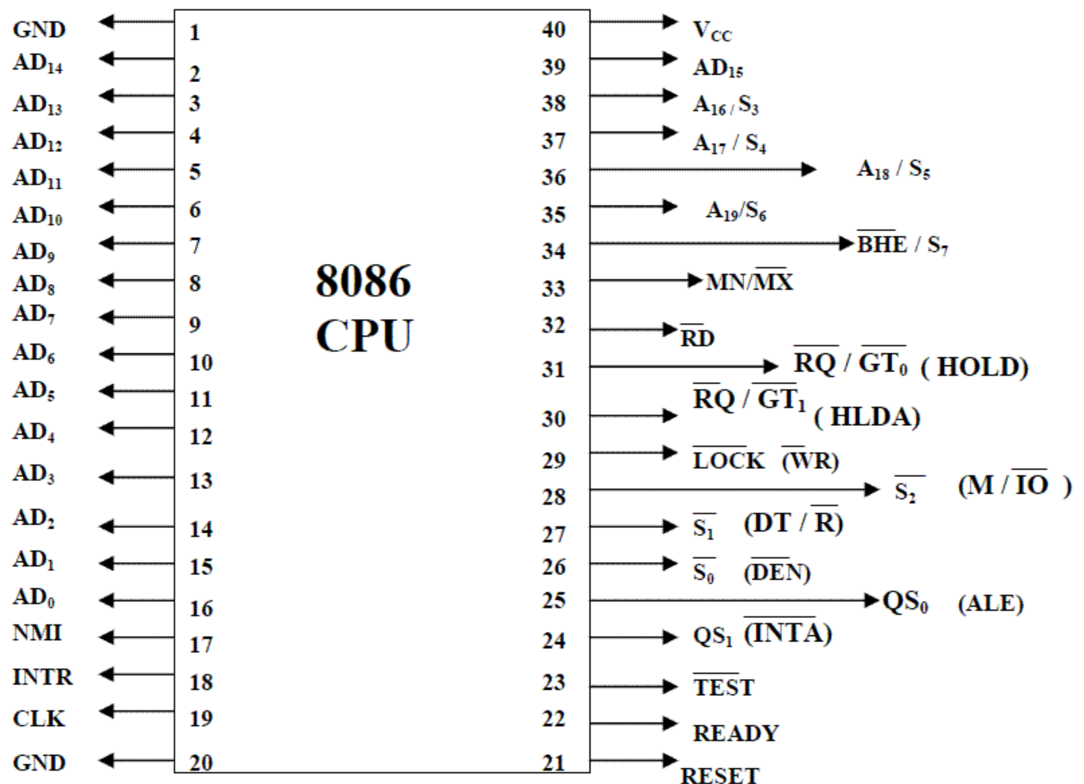
Interrupt	Priority
Divide Error, INT(n),INTO	Highest
NMI	
INTR	
Single Step	Lowest

Pin Diagram of 8086

MN/MX modes of 8086

• **Minimum mode:** The 8086 processor works in a single processor environment. All control signals for memory and I/O are generated by the microprocessor.
• **Maximum mode**: It is designed to be used when a coprocessor exists in the system. 8086 works in a multiprocessor environment. Control signals for memory and I/O are generated by an external BUS Controller.

- The Microprocessor 8086 is a 16-bit CPU available in different clock rates and packaged in a 40 pin CERDIP or plastic package.
- The 8086 operates in single processor or multiprocessor configuration to achieve high performance. The pins serve a particular function in minimum mode (single processor mode) and other function in maximum mode configuration (multiprocessor mode).
- The 8086 signals can be categorised in three groups.
  - ✓ The first are the signal having common functions in minimum as well as maximum mode.
  - ✓ The second are the signals which have special functions for minimum mode
  - ✓ The third are the signals having special functions for maximum mode.



Common Signal in both Minimum and Maximum mode

- **AD15-AD0 :** These are the time multiplexed memory I/O address and data lines.
  - ✓ Address remains on the lines during T1 state, while the data is available on the data bus during T2, T3, Tw and T4. These lines are active high and float to a tristate during interrupt acknowledge and local bus hold acknowledge cycles.
- **A19/S6,A18/S5,A17/S4, and A16/S3 :** These are the time multiplexed address and status lines.
  - ✓ During T1 these are the most significant address lines for memory operations.
  - ✓ During I/O operations, these lines are low.
  - ✓ During memory or I/O operations, status information is available on those lines for T2,T3,Tw and T4.
  - ✓ The status of the interrupt enable flag bit is updated at the beginning of each clock cycle.

- ✓ The S4 and S3 combinely indicate which segment register is presently being used for memory accesses as in below fig.
- ✓ These lines float to tri-state off during the local bus hold acknowledge. The status line S6 is always low.
- ✓ The address bit are separated from the status bit using latches controlled by the ALE signal.

S4	S3	Indication
0	0	Alternate Data
0	1	Stack
1	0	Code or None
1	1	Data
0	0	Whole word
0	1	Upper byte from or to even address
1	0	Lower byte from or to even address

- **BHE/S7 :** The bus high enable is used to indicate the transfer of data over the higher order (D15-D8) data bus as shown in table. It goes low for the data transfer over D15-D8 and is used to derive chip selects of odd address memory bank or peripherals. BHE is low during T1 for read, write and interrupt acknowledge cycles, whenever a byte is to be transferred on higher byte of data bus. The status information is available during T2, T3 and T4. The signal is active low and tristated during hold. It is low during T1 for the first pulse of the interrupt acknowledge cycle.
- **RD – Read :** This signal on low indicates the peripheral that the processor is performing memory or I/O read operation. RD is active low and shows the state for T2, T3, Tw of any read cycle. The signal remains tristate during the hold acknowledge.
- **READY :** This is the acknowledgement from the slow device or memory that they have completed the data transfer. The signal made available by the devices is synchronized by the 8284A clock generator to provide ready input to the 8086. the signal is active high.
- **INTR-Interrupt Request :** This is a triggered input. This is sampled during the last clock cycles of each instruction to determine the availability of the request. If any interrupt request is pending, the processor enters the interrupt acknowledge cycle. This can be internally masked by resulting the interrupt enable flag. This signal is active high and internally synchronized.

- **TEST :** This input is examined by a 'WAIT' instruction. If the TEST pin goes low, execution will continue, else the processor remains in an idle state. The input is synchronized internally during each clock cycle on leading edge of clock.

- **CLK- Clock Input :** The clock input provides the basic timing for processor operation and bus control activity. It's an asymmetric square wave with 33% duty cycle.

The following pin functions are for the minimum mode operation of 8086.

- **M/IO – Memory/IO :** This is a status line logically equivalent to S2 in maximum mode. When it is low, it indicates the CPU is having an I/O operation, and when it is high, it indicates that the CPU is having a memory operation. This line becomes active high in the previous T4 and remains active till final T4 of the current cycle. It is tristate during local bus "hold acknowledge ".

- **INTA – Interrupt Acknowledge :** This signal is used as a read strobe for interrupt acknowledge cycles. i.e. when it goes low, the processor has accepted the interrupt.

- **ALE – Address Latch Enable :** This output signal indicates the availability of the valid address on the address/data lines, and is connected to latch enable input of latches. This signal is active high and is never tristate.

- **DT/R – Data Transmit/Receive:** This output is used to decide the direction of data flow through the transreceivers (bidirectional buffers). When the processor sends out data, this signal is high and when the processor is receiving data, this signal is low.

- **DEN – Data Enable :** This signal indicates the availability of valid data over the address/data lines. It is used to enable the transreceivers (bidirectional buffers) to separate the data from the multiplexed address/data signal. It is active from the middle of T2 until the middle of T4. This is tristated during ' hold acknowledge' cycle.

- **HOLD, HLDA- Acknowledge :** When the HOLD line goes high, it indicates to the processor that another master is requesting the bus access. The processor, after receiving the HOLD request, issues the hold acknowledge signal on HLDA pin, in the middle of the next clock cycle after completing the current bus cycle.

- At the same time, the processor floats the local bus and control lines. When the processor detects the HOLD line low, it lowers the HLDA signal. HOLD is an asynchronous input, and is should be externally synchronized. If the DMA request is made while the CPU is performing a memory or I/O cycle, it will release the local bus during T4 provided :

  - ✓ The request occurs on or before T2 state of the current cycle.
  - ✓ The current cycle is not operating over the lower byte of a word.
  - ✓ The current cycle is not the first acknowledge of an interrupt acknowledge sequence.
  - ✓ A Lock instruction is not being executed.

The following pin functions are applicable for maximum mode operation of 8086.

- **S2, S1, and S0 – Status Lines :** These are the status lines which reflect the type of operation, being carried out by the processor. These become activity during T4 of the previous cycle and active during T1 and T2 of the current bus cycles.
- **LOCK :** This output pin indicates that other system bus master will be prevented from gaining the system bus, while the LOCK signal is low. The LOCK signal is activated by the 'LOCK' prefix instruction and remains active until the completion of the next instruction. When the CPU is executing a critical instruction which requires the system bus, the LOCK prefix instruction ensures that other processors connected in the system will not gain the control of the bus.

The 8086, while executing the prefixed instruction, asserts the bus lock signal output, which may be connected to an external bus controller. By prefetching the instruction, there is a considerable speeding up in instruction execution in 8086. This is known as **instruction pipelining**.

S2	S1	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
1	0	0	Code Access
1	0	1	Read Memory
1	1	0	Write Memory
1	1	1	Passive

- At the starting the CS:IP is loaded with the required address from which the execution is to be started. Initially, the queue will be empty an the microprocessor starts a fetch operation to bring one byte (the first byte) of instruction code, if the CS:IP address is odd or two bytes at a time, if the CS:IP address is even.
- The first byte is a complete opcode in case of some instruction (one byte opcode instruction) and is a part of opcode, in case of some instructions (two byte opcode instructions), the remaining part of code lie in second byte.
- The second byte is then decoded in continuation with the first byte to decide the instruction length and the number of subsequent bytes to be treated as instruction data. The queue is updated after every byte is read from the queue but the fetch cycle is initiated by BIU only if at least two bytes of the queue are empty and the EU may be concurrently executing the fetched instructions.
- The next byte after the instruction is completed is again the first opcode byte of the next instruction. A similar procedure is repeated till the complete execution of the program. The fetch operation of the next instruction is overlapped with the execution of the current instruction. As in the architecture, there are two separate units, namely Execution unit and Bus interface unit.

- While the execution unit is busy in executing an instruction, after it is completely decoded, the bus interface unit may be fetching the bytes of the next instruction from memory, depending upon the queue status.

QS1	QS0	Indication
0	0	No Operation
0	1	First Byte of the opcode from the queue
1	0	Empty Queue
1	1	Subsequent Byte from the Queue

- **RQ/GT0, RQ/GT1 – Request/Grant :** These pins are used by the other local bus master in maximum mode, to force the processor to release the local bus at the end of the processor current bus cycle.
- Each of the pin is bidirectional with RQ/GT0 having higher priority than RQ/GT1. RQ/GT pins have internal pull-up resistors and may be left unconnected. Request/Grant sequence is as follows:

1.A pulse of one clock wide from another bus master requests the bus access to 8086.

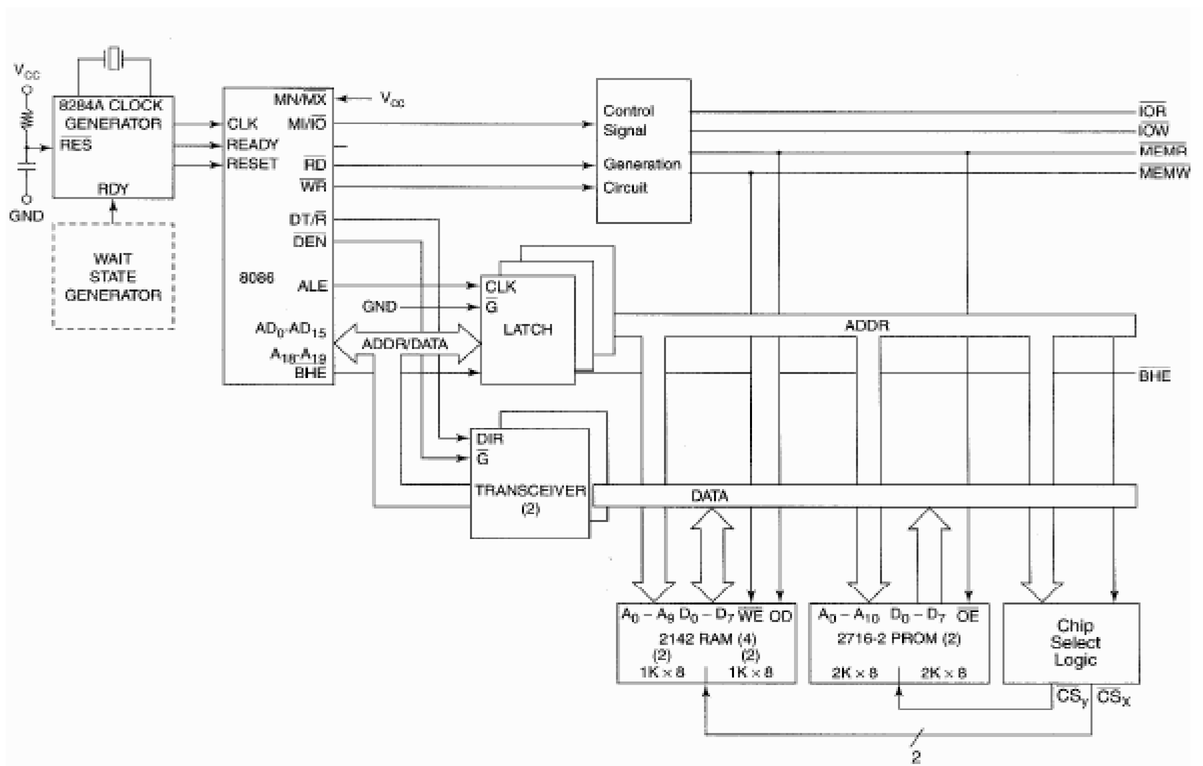
2.During T4(current) or T1(next) clock cycle, a pulse one clock wide from 8086 to the requesting master, indicates that the 8086 has allowed the local bus to float and that it will enter the 'hold acknowledge' state at next cycle. The CPU bus interface unit is likely to be disconnected from the local bus of the system.

3.A one clock wide pulse from the another master indicates to the 8086 that the hold request is about to end and the 8086 may regain control of the local bus at the next clock cycle. Thus each master to master exchange of the local bus is a sequence of 3 pulses. There must be at least one dead clock cycle after each bus exchange. The request and grant pulses are active low. For the bus request those are received while 8086 is performing memory or I/O cycle, the granting of the bus is governed by the rules as in case of HOLD and HLDA in minimum mode.

SYSTEM CONFIGURATION

MINIMUM MODE

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



- The remaining components in the system are latches, transreceivers, clock generator, memory and I/O devices. Some type of chip selection logic may be required for selecting memory or I/O devices, depending upon the address map of the system.

- Latches are generally buffered output D-type flip-flops like 74LS373 or 8282. They are used for separating the valid address from the multiplexed address/data signals and are controlled by the ALE signal generated by 8086.

- Transreceivers are the bidirectional buffers and sometimes they are called as data amplifiers. They are required to separate the valid data from the time multiplexed address/data signals.

- They are controlled by two signals namely, DEN and DT/R.

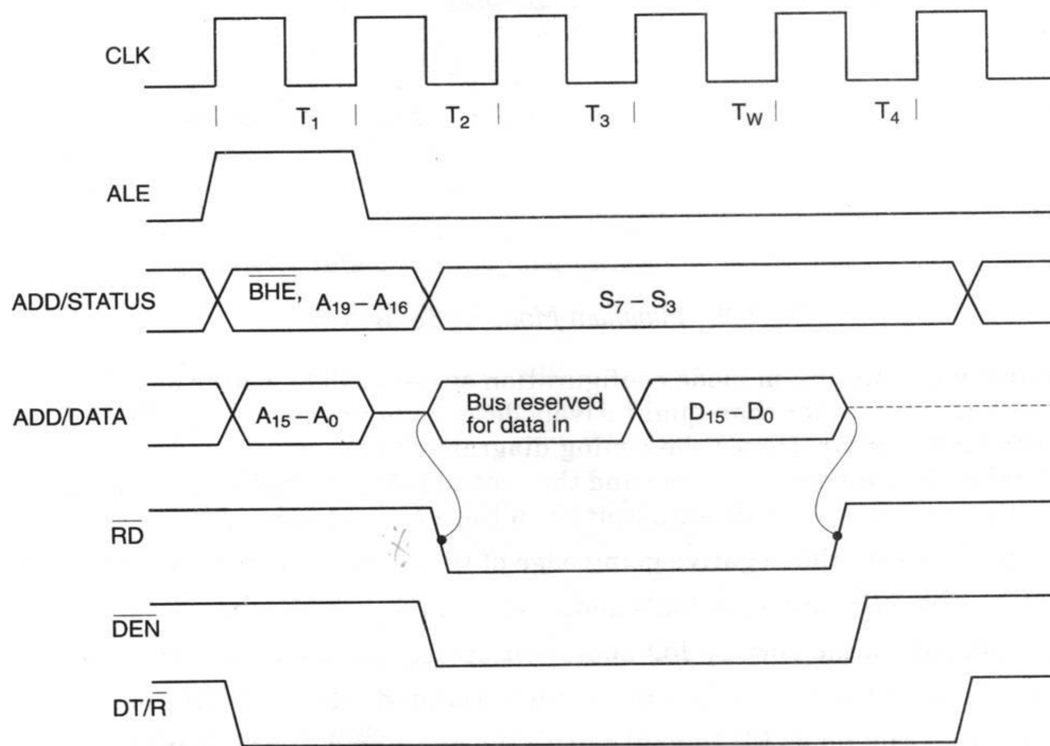
- The DEN signal indicates the direction of data, i.e. from or to the processor. The system contains memory for the monitor and users program storage.

- Usually, EPROM are used for monitor storage, while RAM for users program storage. A system may contain I/O devices.

Timing Diagram in minimum mode

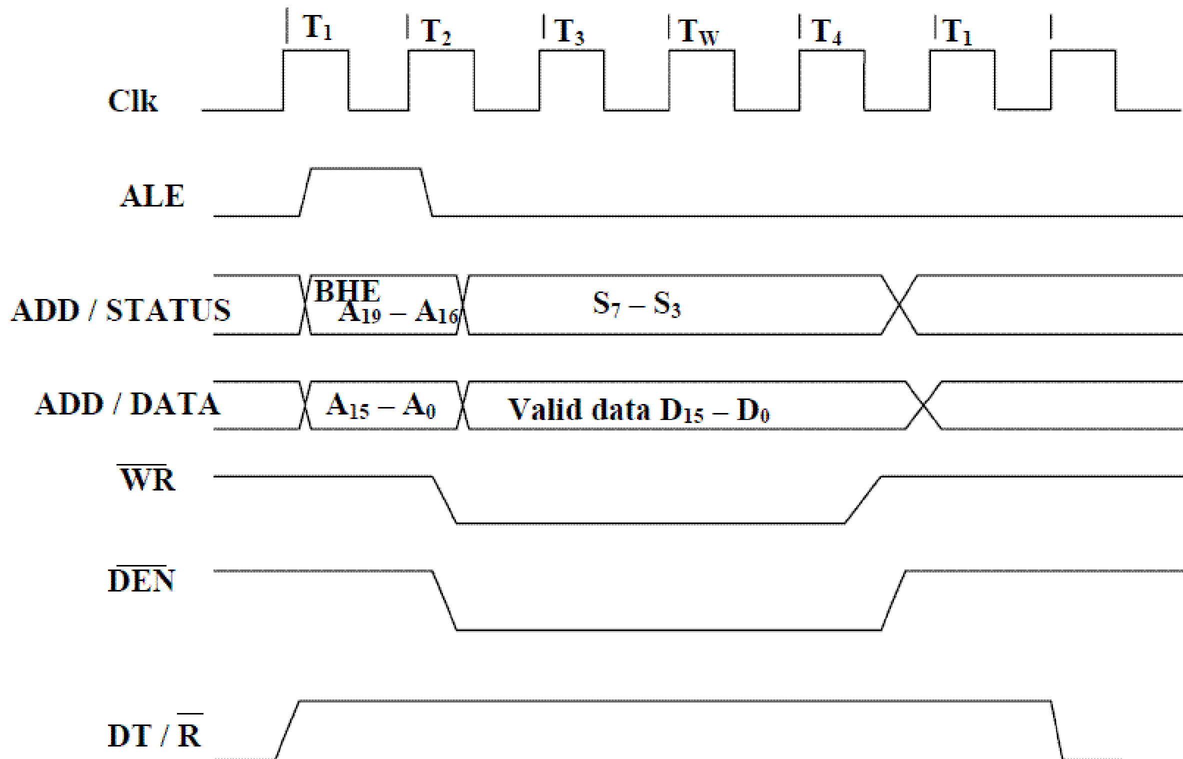
- The working of the minimum mode configuration system can be better described in terms of the timing diagrams rather than qualitatively describing the operations.
- The opcode fetch and read cycles are similar. Hence the timing diagram can be categorized in two parts, the first is the timing diagram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

Memory read



- The read cycle begins in T1 with the assertion of address latch enable (ALE) signal and also M / IO signal. During the negative going edge of this signal, the valid address is latched on the local bus.
- The BHE and A0 signals address low, high or both bytes. From T1 to T4 , the M/IO signal indicates a memory or I/O operation.
- At T2, the address is removed from the local bus and is sent to the output. The bus is then tristated. The read (RD) control signal is also activated in T2.
- The read (RD) signal causes the address device to enable its data bus drivers. After RD goes low, the valid data is available on the data bus.

Memory Write



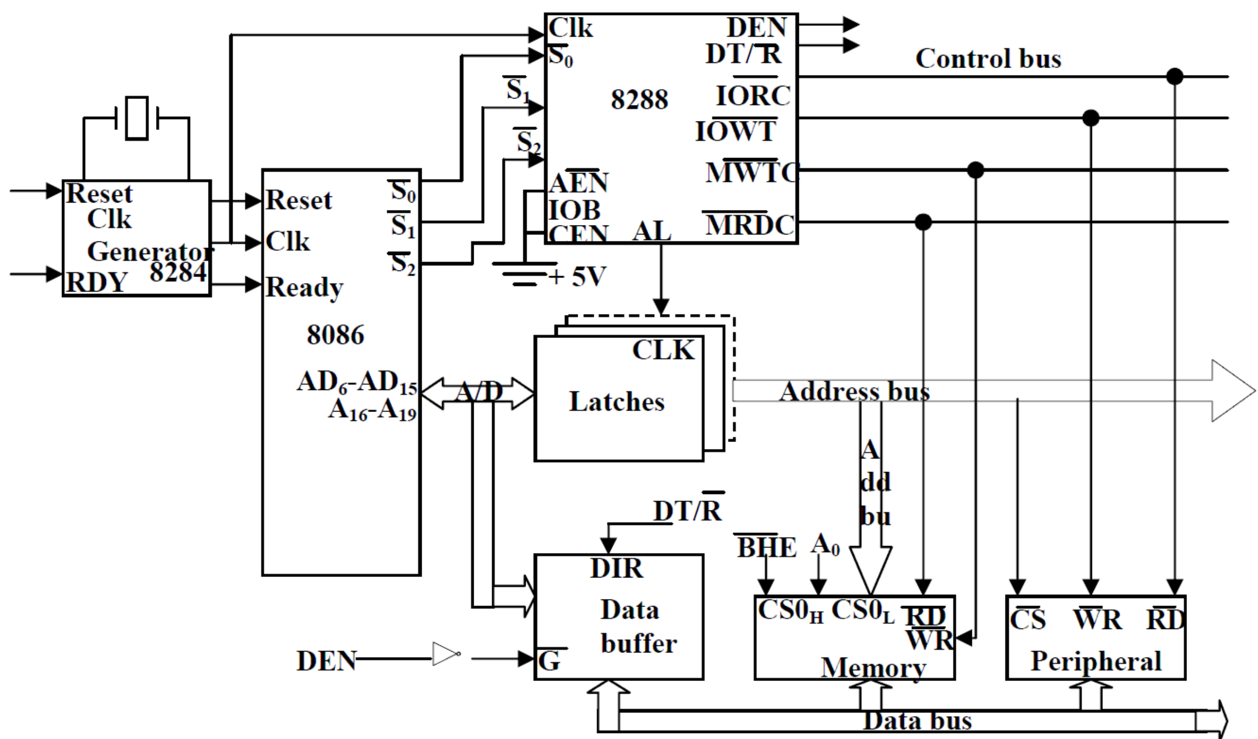
Write Cycle Timing Diagram for Minimum Mode

- The addressed device will drive the READY line high. When the processor returns the read signal to high level, the addressed device will again tristate its bus drivers.
- A write cycle also begins with the assertion of ALE and the emission of the address. The M/IO signal is again asserted to indicate a memory or I/O operation. In T2, after sending the address in T1, the processor sends the data to be written to the addressed location.
- The data remains on the bus until middle of T4 state. The WR becomes active at the beginning of T2 (unlike RD is somewhat delayed in T2 to provide time for floating).
- The BHE and A0 signals are used to select the proper byte or bytes of memory or I/O word to be read or write.
- The M/IO, RD and WR signals indicate the type of data transfer as specified in table below.

Maximum mode

- In the maximum mode, the 8086 is operated by strapping the MN/MX pin to ground.
- In this mode, the processor derives the status signal S2, S1, S0. Another chip called bus controller derives the control signal using this status information .

- In the maximum mode, there may be more than one microprocessor in the system configuration.
- The components in the system are same as in the minimum mode system.
- The basic function of the bus controller chip IC8288, is to derive control signals like RD and WR (for memory and I/O devices), DEN, DT/R, ALE etc. using the information by the processor on the status lines.
- The bus controller chip has input lines S2, S1, S0 and CLK. These inputs to 8288 are driven by CPU.
- It derives the outputs ALE, DEN, DT/R, MRDC, MWTC, AMWC, IORC, IOWC and AIOWC. The AEN, IOB and CEN pins are especially useful for multiprocessor systems.

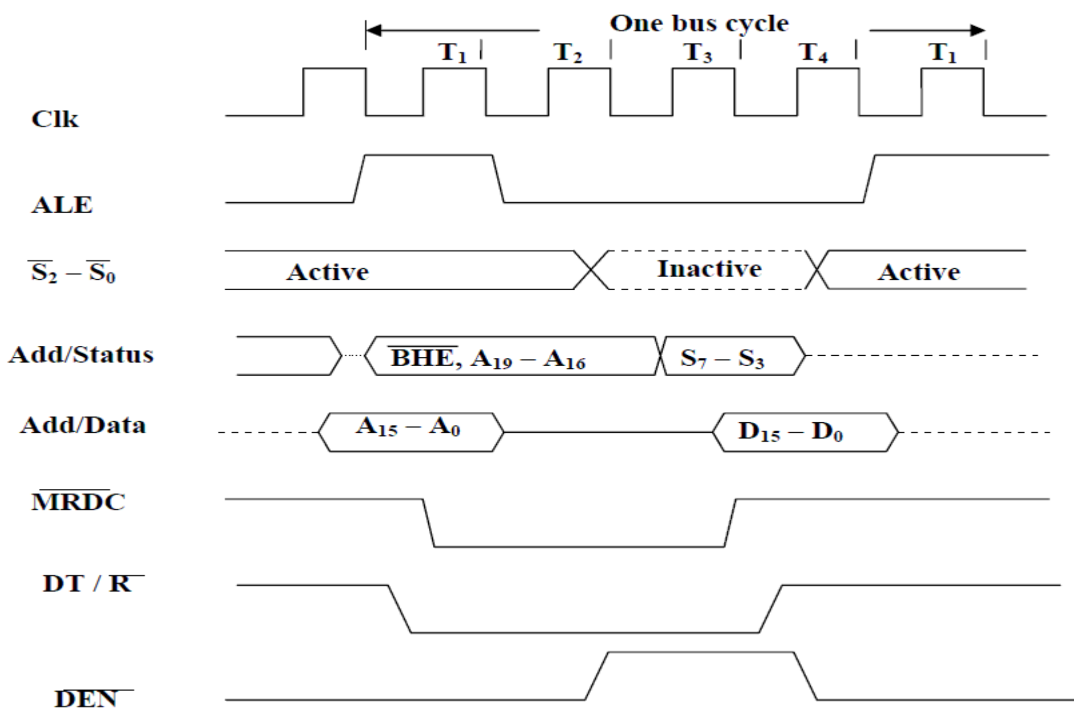


Maximum Mode 8086 System.

- AEN and IOB are generally grounded. CEN pin is usually tied to +5V. The significance of the MCE/PDEN output depends upon the status of the IOB pin.
- If IOB is grounded, it acts as master cascade enable to control cascade 8259A, else it acts as peripheral data enable used in the multiple bus configurations.
- INTA pin used to issue two interrupt acknowledge pulses to the interrupt controller or to an interrupting device.
- IORC, IOWC are I/O read command and I/O write command signals respectively.

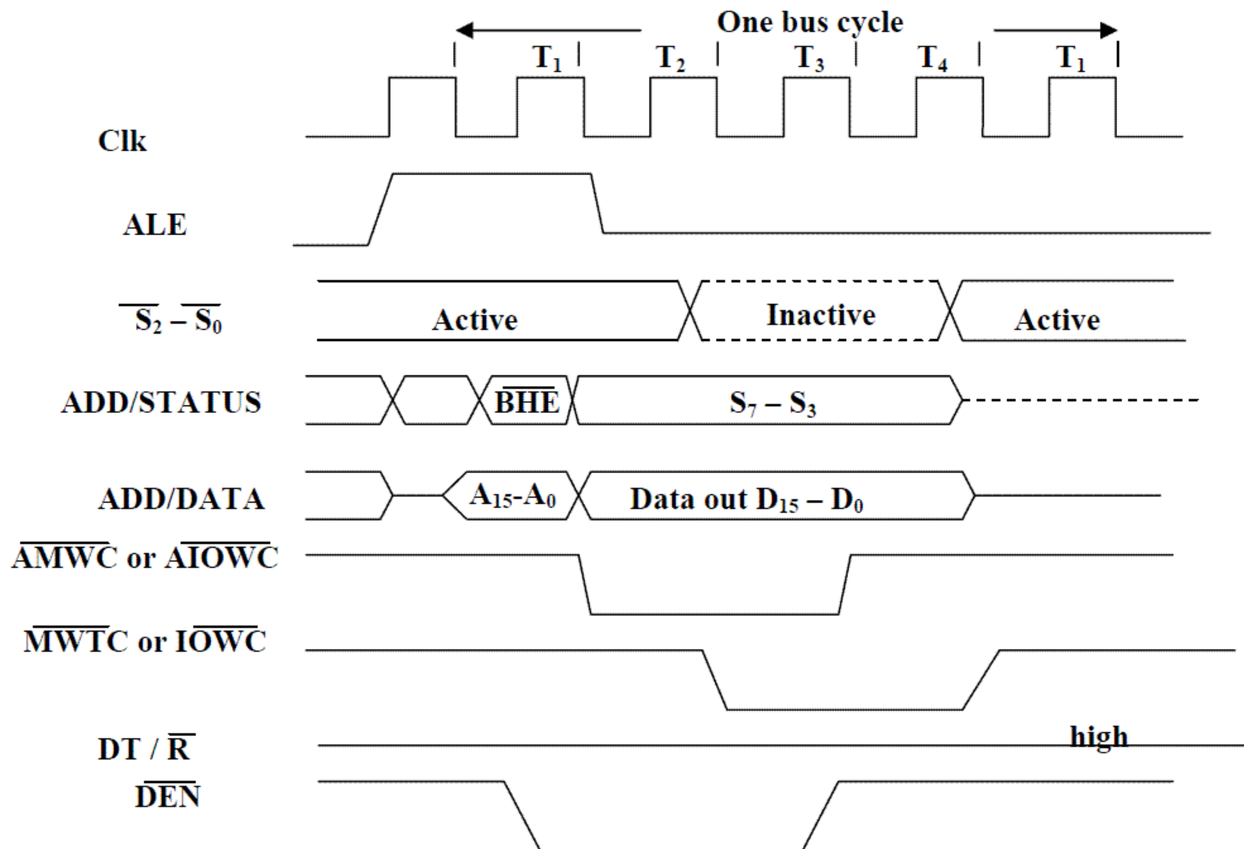
- These signals enable an IO interface to read or write the data from or to the address port.
- The MRDC, MWTC are memory read command and memory write command signals respectively and may be used as memory read or write signals.
- All these command signals instructs the memory to accept or send data from or to the bus.
- For both of these write command signals, the advanced signals namely AIOWC and AMWTC are available.
- Here the only difference between in timing diagram between minimum mode and maximum mode is the status signals used and the available control and advanced command signals.
- R0, S1, S2 are set at the beginning of bus cycle.8288 bus controller will output a pulse as on the ALE and apply a required signal to its DT / R pin during T1.
- In T2, 8288 will set DEN=1 thus enabling transceivers, and for an input it will activate MRDC or IORC. These signals are activated until T4. For an output, the AMWC or AIOWC is activated from T2 to T4 and MWTC or IOWC is activated from T3 to T4.
- The status bit S0 to S2 remains active until T3 and become passive during T3 and T4.
- If reader input is not activated before T3, wait state will be inserted between T3 and T4.

Memory read Timing Diagram



Memory Read Timing in Maximum Mode

Memory Write Timing Diagram



Memory Write Timing in Maximum mode.

Physical Memory Organization

The BIU has a combined address and data bus, commonly referred to as a time-multiplexed bus. Time multiplexing address and data information makes the most efficient use of device package pins. A system with address latching provided within the memory and I/O devices can directly connect to the address/data bus. The local bus can be demultiplexed with a single set of address latches to provide non-multiplexed address and data information to the system.

The programmer views the memory or I/O address space as a sequence of bytes. Memory space consists of 1 Mbyte, while I/O space consists of 64 Kbytes. Any byte can contain an 8-bit data element, and any two consecutive bytes can contain a 16-bit data element (identified as a word). The discussions in this section apply to both memory and I/O bus cycles. For brevity, memory bus cycles are used for examples and illustration.

The memory address space on a 16-bit data bus is physically implemented by dividing the address space into two banks of up to 512 Kbytes each (see Figure 3.21). One bank connects to the lower half of the data bus and contains even-addressed bytes (A0=0). The other bank connects to the upper half of the data bus and contains odd-addressed bytes (A0=1). Address lines A19:1 select a specific byte within each bank. A0 and Byte High Enable (BHE) determine whether one bank or both banks participate in the data transfer.

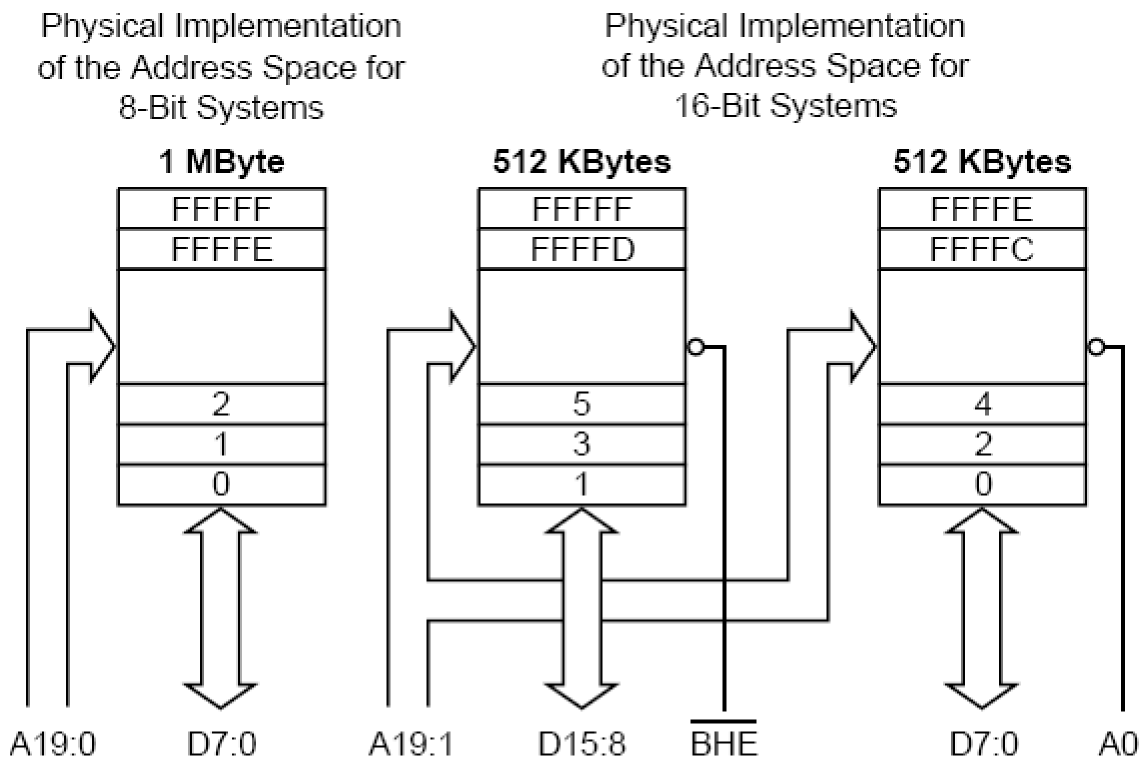


Figure 3.21 Physical Data bus Models

Byte transfers to even addresses transfer information over the lower half of the data bus (see Figure 3-22a). A0 low enables the lower bank, while BHE high disables the upper bank. The data value from the upper bank is ignored during a bus read cycle. BHE high prevents a write operation from destroying data in the upper bank. Byte transfers to odd addresses transfer information over the upper half of the data bus (see Figure 3-22b). BHE low enables the upper bank, while A0 high disables the lower bank. The data value from the lower bank is ignored during a bus read cycle. A0 high prevents a write operation from destroying data in the lower bank. To access even-addressed 16-bit words (two consecutive bytes with the least-significant byte at an even address), information is transferred over both halves of the data bus (see Figure 3-23).

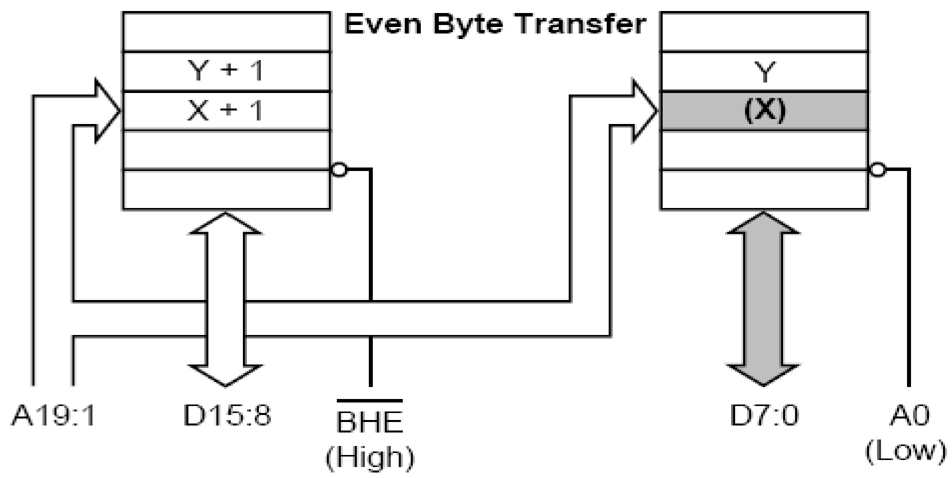


Figure 3.22a 16-bit Data Bus Byte Transfers

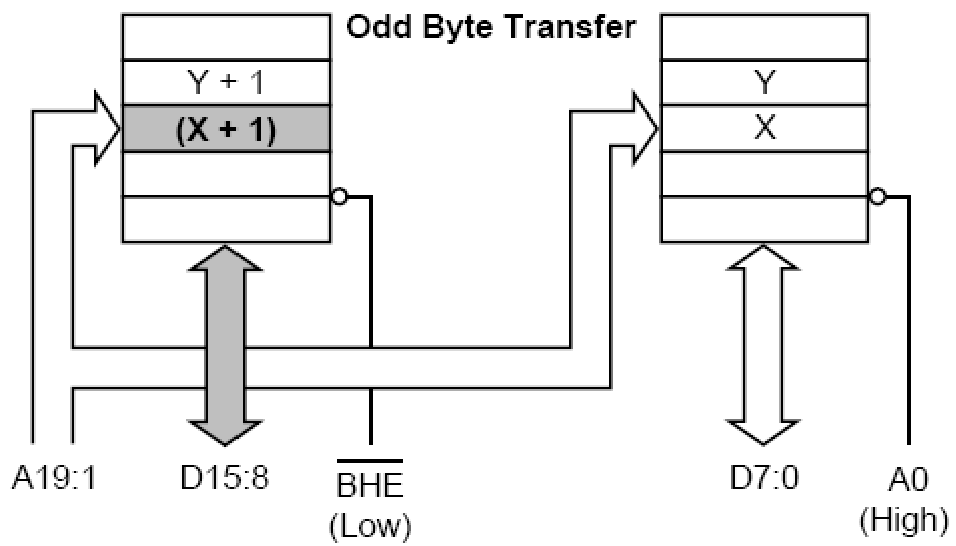


Figure 3.22b 16-bit Data Bus Byte transfers

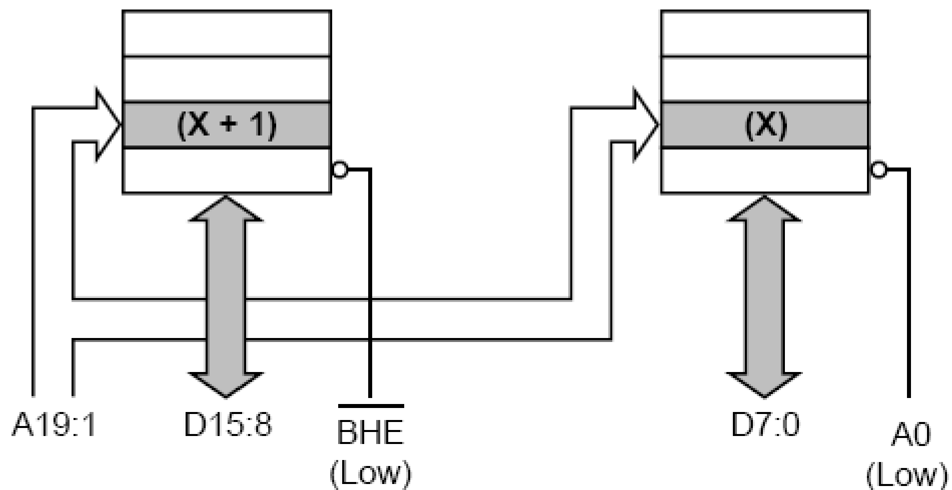


Figure 3.23 16-Bit Data Bus Even Word Transfers

A19:1 select the appropriate byte within each bank. A0 and BHE drive low to enable both banks simultaneously. Odd-addressed word accesses require the BIU to split the transfer into two byte operations (see Figure 3-24). The first operation transfers data over the upper half of the bus, while the second operation transfers data over the lower half of the bus. The BIU automatically executes the two-byte sequence whenever an odd-addressed word access is performed.

Case 1

Read/Write a byte form/to an even address – A0 will be low and BHE (Active Low) will be high – Byte is transferred to/from low bank through D0-D7

Example – MOV AH,DS:BYTE PTR[0000]

Case 2

Similar to case 1 except the word access instead of the byte access – Both A0 and BHE (Active Low) will be asserted low – Low byte of the word through D0-D7 and high byte of the word through D8-D15

Example – MOV AX,DS:WORD PTR[0000]

Case 3

Read/Wrire a byte from/to an odd address – A0 will be high and BHE (Active Low) will be asserted low – Low bank is disabled and high bank is enabled – Byte is transferred through D0-D7

Example – MOV AL,DS:BYTE PTR[0001]

Case 4

Read/Write a word from/to an odd address – 8086 requires two bus cycles – During the first machine cycle assert BHE (Active Low) as low and A0 as high – First byte is transferred through D0-D7 and the second byte is transferred through D8-D15

Example – MOV AX, DS:WORD PTR[0001H]