Intel introduced its first 4-bit microprocessor 4004 in 1971 and its 8-bit microprocessor 8008 in 1972. These microprocessors could not survive as general purpose microprocessors due to their design and performance limitations. The launch of the first general purpose 8-bit microprocessor 8080 in 1974 by Intel is considered to be the first major stepping stone towards the development of advanced microprocessors. The microprocessor 8085 followed 8080, with a few more added features to its architecture, which resulted in a functionally complete microprocessor. The main limitations of the 8-bit microprocessors were their low speed, low memory addressing capability, limited number of general purpose registers and a less powerful instruction set. All these limitations of the 8-bit microprocessors pushed the designers to build more powerful processors in terms of advanced architecture, more processing capability, larger memory addressing capability and a more powerful instruction set. The 8086 was a result of such developmental design efforts.

In the family of 16-bit microprocessors, Intel's 8086 was the first one to be launched in 1978. The introduction of the 16-bit processor was a result of the increasing demand for more powerful and high speed computational resources. The 8086 microprocessor has a much more powerful instruction set along with the architectural developments which imparts substantial programming flexibility and improvement in speed over the 8-bit microprocessors.

The peripheral chips designed earlier for 8085 were compatible with microprocessor 8086 with slight or no modifications. Though there is a considerable difference between the memory addressing techniques of 8085 and 8086, the memory interfacing technique is similar, but includes the use of a few additional signals. The clock requirements are also different as compared to 8085, but the overall minimal system organisation of 8086 is similar to that of a general 8-bit microprocessor. In this chapter, the architectures of 8086 and 8088 are discussed in adequate details along with the interfacing of the supporting chips with them to form a minimum system. The system organisation is also discussed in significant details for both the operating modes of 8086 and 8088, along with necessary timing diagrams.

may be either used for holding data, variables and intermediate results temporarily or for other purposes like a counter or for storing offset address for some particular addressing modes etc. The special purpose registers are used as segment registers, pointers, index registers or as offset storage registers for particular addressing modes. We will categorize the register set into four groups, as follows:

### 1.1.1 General Data Registers

Figure 1.1 shows the register organisation of 8086. The registers AX,BX,CX and DX are the general purpose 16-bit registers. AX is used as 16-bit *accumulator*, with the lower 8-bits of AX designated as AL and higher 8-bits as AH. AL can be used as an 8-bit accumulator for 8-bit operations. This is the most important general purpose register having multiple functions, which will be discussed later.

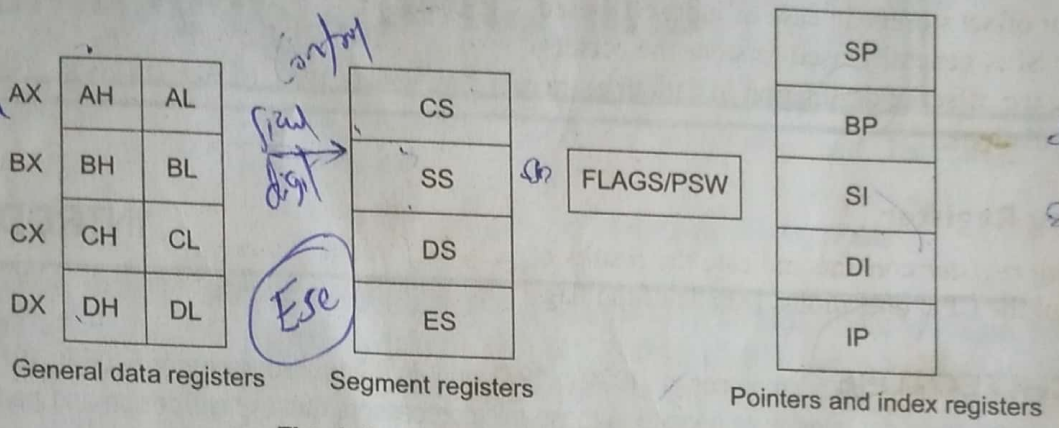



**Fig. 1.1** *Register organisation of 8086*

Usually the letters L and H specify the lower and higher bytes of a particular register. For example, CH means the higher 8-bits of the CX register and CL means the lower 8-bits of the CX register. The letter X is used to specify the complete 16-bit register. The register CX is also used as a default counter in case of string and loop instructions. The register BX is used as an offset storage for forming physical addresses in case of certain addressing modes. DX register is a general purpose register which may be used as an implicit operand or destination in case of a few instructions. The detailed uses of these registers will be more clear when we discuss the addressing modes and the instruction set of 8086.

### 1.1.2 Segment Registers

Unlike 8085, the 8086 addresses a segmented memory. The complete 1 megabyte memory, which the 8086 addresses, is divided into 16 logical segments. Each segment thus contains 64 Kbytes of memory. There are four segment registers, viz. Code Segment Register (CS), Data Segment Register (DS), Extra Segment Register (ES) and Stack Segment Register (SS). The code segment register is used for addressing a memory location in the code segment of the memory, where the executable program is stored. Similarly, the data segment register points to the data segment of the memory, where the data is resided. The extra segment also refers to a segment which essentially is another data segment of the memory. Thus, the extra segment also contains data. The stack segment register is used for addressing stack segment of memory i.e. memory which is used to store stack data. The CPU uses the stack for temporarily storing important data, e.g. the contents of the CPU registers which will be required at a later stage. The stack grows down, i.e. the data is pushed onto the stack in the memory locations with decreasing addresses. When this information will be required by the CPU, they will be popped off from the stack. While addressing any location in the memory bank, the physical address is calculated from two parts, the first is *segment address* and the second is *offset*. The segment registers contain 16-bit segment base addresses, related to different segments. Any of the pointers and index registers or BX may contain the offset of the location to be addressed. The advantage of this scheme is that

maintaining a 20-bit register for a physical address, the processor just maintains two 16-bit registers are within the word length capacity of the machine. Thus the CS, DS, SS and ES segment registers, ly, contain the segment addresses for the code, data, stack and extra segments of memory. It may hat all these segments are the logical segments. They may or may not be physically separated. In ds, a single segment may require more than one memory chip or more than one segment may be dated in a single memory chip.

### Pointers and Index Registers

ers contain offset within the particular segments. The pointers IP, BP and SP usually contain offsets code (JP), and stack (BP & SP) segments. The *index registers* are used as general purpose registers for offset storage in case of indexed, based indexed and relative based indexed addressing modes. er SI is generally used to store the offset of source data in data segment while the register DI is ore the offset of destination in data or extra segment. The index registers are particularly useful for nipulations.

### Flag Register

*flag register* contents indicate the results of computations in the ALU. It also contains some flag ntrol the CPU operations. Details of the flag register are discussed later in this chapter.

## RCHITECTURE

tecture of 8086 provides a number of improvements over 8085 architecture. It supports a 16-bit et of 16-bit registers and provides segmented memory addressing capability, a rich instruction rful interrupt structure, fetched instruction queue for overlapped fetching and execution etc. The lock diagram, shown in Fig.1.2, describes the overall organization of different units inside the

mplete architecture of 8086 can be divided into two parts (a) Bus Interface Unit (BIU) and (b) Unit (EU). *The bus interface unit contains the circuit for physical address calculations and a ng instruction byte queue (6 bytes long).* The bus interface unit makes the system's bus signals for external interfacing of the devices. In other words, this unit is responsible for establishing ns with external devices and peripherals including memory via the bus. As already stated, the 8086 a segmented memory. The complete physical address which is 20-bits long is generated using seg- offset registers, each 16-bits long.

nerating a physical address from contents of these two registers, the content of a segment register l as segment address is shifted left bit-wise four times and to this result, content of an offset register l as offset address is added, to produce a 20-bit physical address. For example, if the segment ad- 005H and the offset is 5555H, then the physical address is calculated as below:

| | | |
|---|---|---|
| Segment address | ⟶ 1005H | |
| Offset address | ⟶ 5555H | |
| Segment address | ⟶ 1005H ⟶ | 0001 0000 0000 0101 |
| Shifted by 4 bit positions | ⟶ | 0001 0000 0000 0101 0000 |
| | + | |
| Offset address | ⟶ | 0101 0101 0101 0101 |
| Physical address | ⟶ | 0001 0101 0101 1010 0101 |
| | | 1 5 5 A 5 |

instead of maintaining a 20-bit register for a physical address, the processor just maintains two 16-bit registers which are within the word length capacity of the machine. Thus the CS, DS, SS and ES segment registers, respectively, contain the segment addresses for the code, data, stack and extra segments of memory. It may be noted that all these segments are the logical segments. They may or may not be physically separated. In other words, a single segment may require more than one memory chip or more than one segment may be accommodated in a single memory chip.

### 1.1.3 Pointers and Index Registers

The pointers contain offset within the particular segments. The pointers IP, BP and SP usually contain offsets within the code (JP), and stack (BP & SP) segments. The *index registers* are used as general purpose registers as well as for offset storage in case of indexed, based indexed and relative based indexed addressing modes. The register SI is generally used to store the offset of source data in data segment while the register DI is used to store the offset of destination in data or extra segment. The index registers are particularly useful for string manipulations.

### 1.1.4 Flag Register

The 8086 *flag register* contents indicate the results of computations in the ALU. It also contains some flag bits to control the CPU operations. Details of the flag register are discussed later in this chapter.

## 1.2 ARCHITECTURE

The architecture of 8086 provides a number of improvements over 8085 architecture. It supports a 16-bit ALU, a set of 16-bit registers and provides segmented memory addressing capability, a rich instruction set, powerful interrupt structure, fetched instruction queue for overlapped fetching and execution etc. The internal block diagram, shown in Fig.1.2, describes the overall organization of different units inside the chip.

The complete architecture of 8086 can be divided into two parts (a) Bus Interface Unit (BIU) and (b) Execution Unit (EU). *The bus interface unit contains the circuit for physical address calculations and a predecoding instruction byte queue (6 bytes long).* The bus interface unit makes the system's bus signals available for external interfacing of the devices. In other words, this unit is responsible for establishing communications with external devices and peripherals including memory via the bus. As already stated, the 8086 addresses a segmented memory. The complete physical address which is 20-bits long is generated using segment and offset registers, each 16-bits long.

For generating a physical address from contents of these two registers, the content of a segment register also called as segment address is shifted left bit-wise four times and to this result, content of an offset register also called as offset address is added, to produce a 20-bit physical address. For example, if the segment address is 1005H and the offset is 5555H, then the physical address is calculated as below:

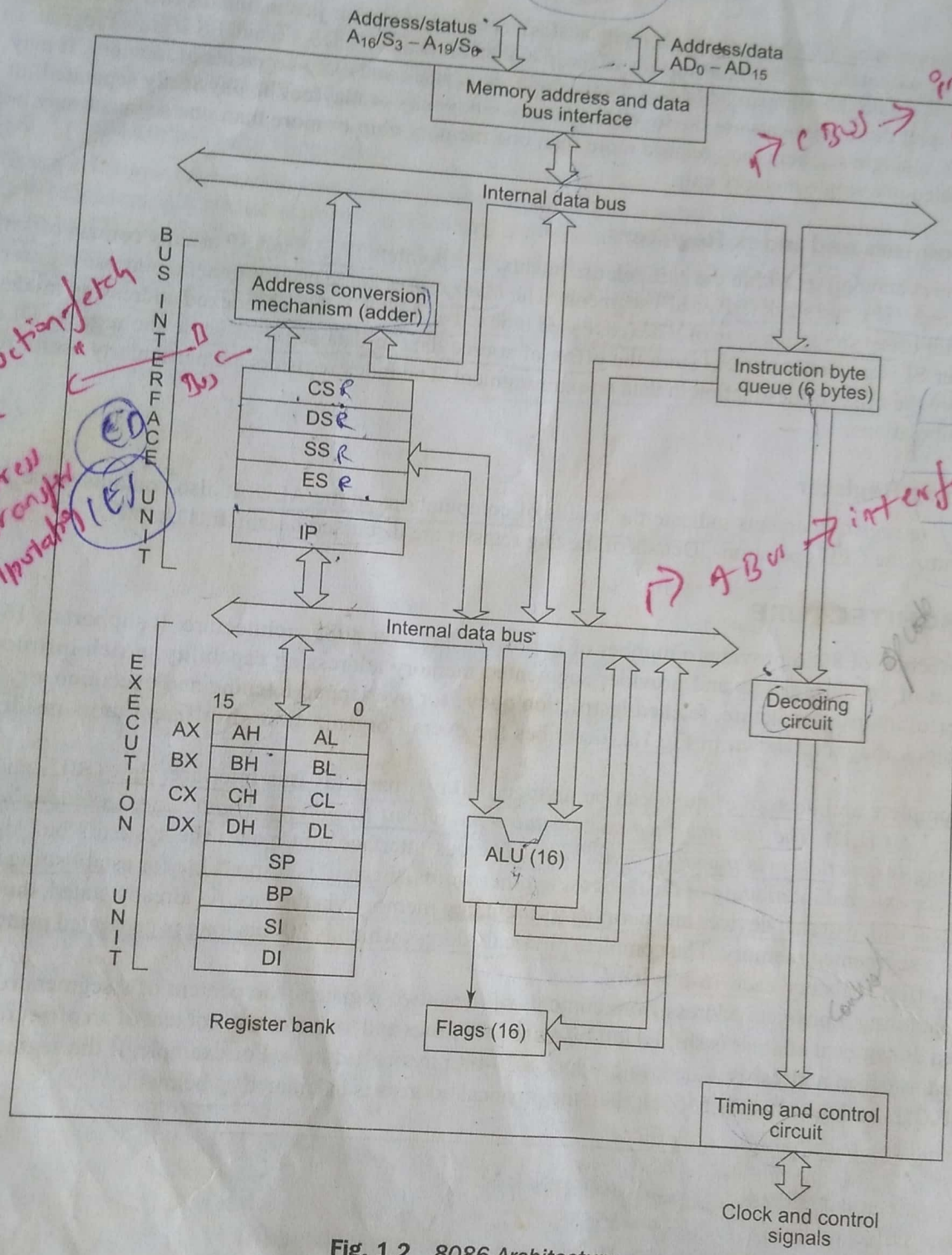



**Fig. 1.2** *8086 Architecture*

Thus, the segment addressed by the segment value 1005H can have offset values from 0000H to FFFFH within it, i.e. maximum 64K locations may be accommodated in the segment. Thus, the segment register indicates the base address of a particular segment, while the offset indicates the distance of the required memory location in the segment from the base address. Since the offset is a 16-bit number, each segment can have a maximum of 64K locations. The bus interface unit has a separate adder to perform this procedure for obtaining

a physical address while addressing memory. The segment address value is to be taken from an appropriate segment register depending upon whether code, data or stack are to be accessed, while the offset may be the content of IP, BX, SI, DI, SP, BP or an immediate 16-bit value, depending upon the addressing mode.

In case of 8085, once the opcode is fetched and decoded, the external bus remains free for some time, while the processor internally executes the instruction. This time slot is utilised in 8086 to achieve the overlapped fetch and execution cycles. While the fetched instruction is executed internally, the external bus is used to fetch the machine code of the next instruction and arrange it in a queue known as predecoded instruction byte queue. It is a 6 bytes long, first-in first-out structure. The instructions from the queue are taken for decoding sequentially. Once a byte is decoded, the queue is rearranged by pushing it out and the queue status is checked for the possibility of the next opcode fetch cycle. While the opcode is fetched by the Bus Interface Unit (BIU), the Execution Unit (EU) executes the previously decoded instruction concurrently. The BIU along with the Execution Unit (EU) thus forms a pipeline. The bus interface unit, thus manages the complete interface of execution unit with memory and I/O devices, of course, under the control of the timing and control unit.

The execution unit contains the register set of 8086 except segment registers and IP. It has a 16-bit ALU, able to perform arithmetic and logic operations. The 16-bit flag register reflects the results of execution by the ALU. The decoding unit decodes the opcode bytes issued from the instruction byte queue. The timing and control unit derives the necessary control signals to execute the instruction opcode received from the queue, depending upon the information made available by the decoding circuit. The execution unit may pass the results to the bus interface unit for storing them in memory.

### 1.2.1 Memory Segmentation

The memory in an 8086/8088 based system is organised as segmented memory. In this scheme, the complete physically available memory may be divided into a number of logical segments. Each segment is 64K bytes in size and is addressed by one of the segment registers. The 16-bit contents of the segment register actually point to the starting location of a particular segment. To address a specific memory location within a segment, we need an offset address. The offset address is also 16-bit long so that the maximum offset value can be FFFFH, and the maximum size of any segment is thus 64K locations. The physical address formation has been explained previously in Section 1.2.

To emphasize this segmented memory concept, we will consider an example of a housing colony containing say, 100 houses. The simplest method of numbering the houses will be to assign the numbers from 1 to 100 to each house sequentially. Suppose, now, if one wants to find out house number 67, then he will start from house number 1 and go on till he finds the house, numbered 67. Consider another case where the 100 houses are arranged in the $10 \times 10$ (rows $\times$ columns) pattern. In this case, to find out house number 67, one will directly go to the 6th row and then to the 7th column. In the second scheme, the efforts required for finding the same house will be too less. This second scheme in our example is analogous to the segmented memory scheme, where the addresses are specified in terms of segment addresses analogous to rows and offset addresses analogous to columns.

The CPU 8086 is able to address 1Mbytes of physical memory. The complete 1Mbytes memory can be divided into 16 segments, each of 64Kbytes size. The addresses of the segments may be assigned as 0000H to F000H respectively. The offset address values are from 0000H to FFFFH so that the physical addresses range from 00000H to FFFFFH. In the above said case, the segments are called non-overlapping segments. The non-overlapping segments are shown in Fig. 1.3(a). In some cases, however, the segments may be overlapping. Suppose a segment starts at a particular address and its maximum size can be 64Kbytes. But, if another segment starts before this 64Kbytes locations of the first segment, the two segments are said to be overlapping segments. The area of memory from the start of the second segment to the possible end of the first segment is called an overlapped segment area. Figure 1.3(b) explains the phenomenon more clearly. The locations lying in the overlapped area may be addressed by the same physical address generated from two

different sets of segment and offset addresses. The main advantages of the segmented memory scheme are as follows:

1. Allows the memory capacity to be 1Mbytes although the actual addresses to be handled are of 16-bit size
2. Allows the placing of code, data and stack portions of the same program in different parts (segments) of memory, for data and code protection
3. 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.

In the Overlapped Area Locations Physical Address = $CS_1 + IP_1 = CS_2 + IP_2$, where '+' indicates the procedure of physical address formation.

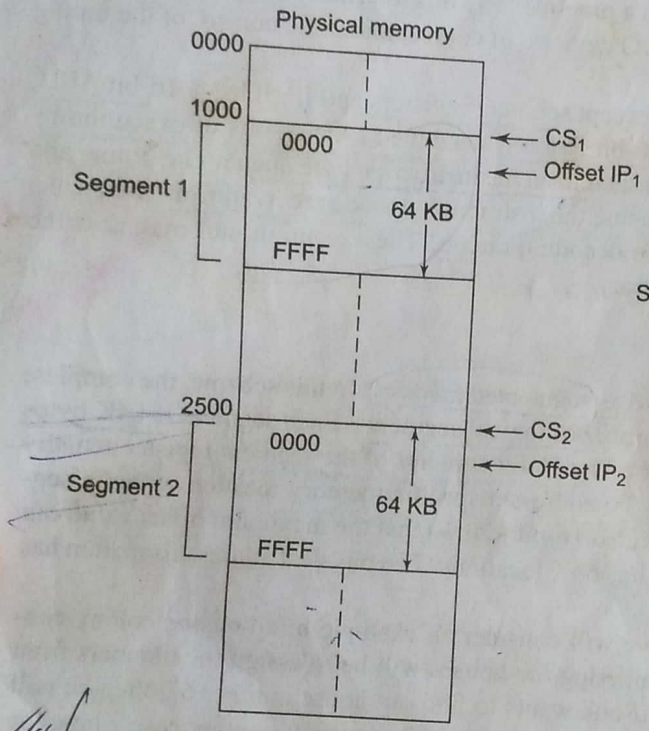



Fig. 1.3(a) *Non-overlapping Segments*

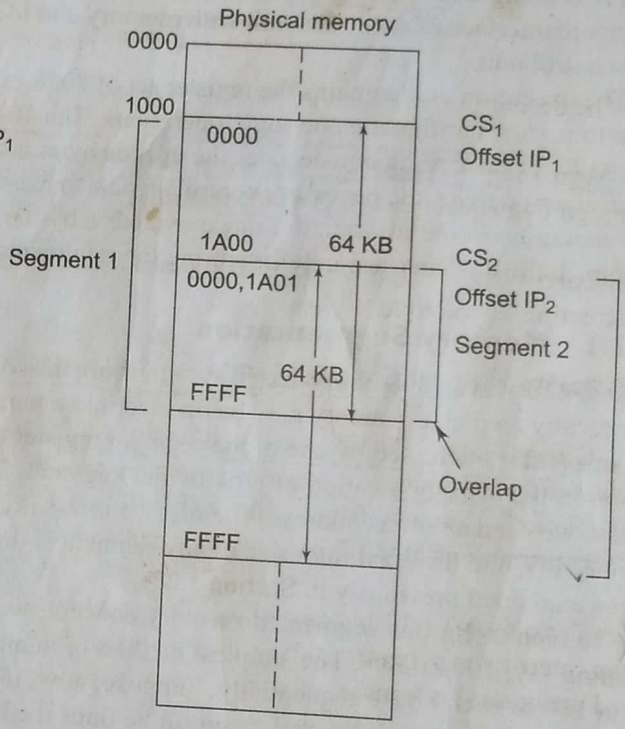



Fig. 1.3(b) *Overlapping Segments*

## 1.2.2 Flag Register

8086 has a 16-bit flag register which is divided into two parts, viz. (a) *condition code or status flags* and (b) *machine control flags*. The ***condition code flag register*** is the lower byte of the 16-bit flag register along with the *overflow flag*. This flag is identical to the 8085 flag register, with an additional overflow flag, which is not present in 8085. This part of the flag register of 8086 reflects the results of the operations performed by ALU. The *control flag register* is the higher byte of the flag register of 8086. It contains three flags, viz. *direction flag* (D), *interrupt flag* (I) and *trap flag* (T).

The complete bit configuration of 8086 flag register is shown in Fig. 1.4.

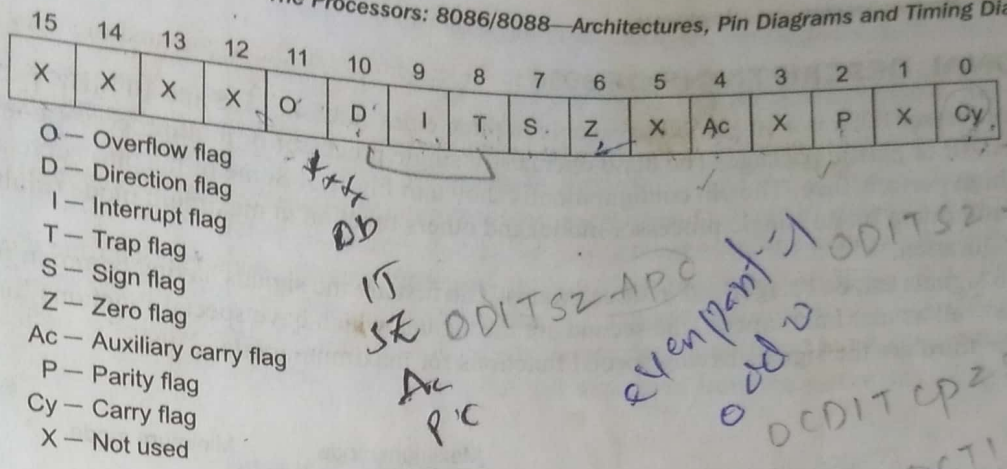



**Fig. 1.4** *Flag Register of 8086*

The description of each flag bit is as follows:

**S-Sign Flag** This flag is set when the result of any computation is negative. For signed computations, the sign flag equals the MSB of the result.

**Z-Zero Flag** This flag is set if the result of the computation or comparison performed by the previous instruction/instructions is zero.

**P-Parity Flag** This flag is set to 1 if the lower byte of the result contains even number of 1s.

**C-Carry Flag** 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'. Some other instructions also affect or use this flag and will be discussed later in this text.

**T-Trap Flag** 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.

**I-Interrupt Flag** If this flag is set, the maskable interrupts are recognised by the CPU, otherwise they are ignored.

**D-Direction Flag** 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. *autoincrementing mode*. Otherwise, the string is processed from the highest address towards the lowest address, i.e. *autodecrementing mode*. We will describe string manipulations later in chapter 2 in more detail.

**AC-Auxiliary Carry Flag** This is set if there is a carry from the lowest nibble, i.e. bit three, during addition or borrow for the lowest nibble, i.e. bit three, during subtraction.

**O-Overflow Flag** 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. For example, in case of the addition of two signed numbers, if the result overflows into the sign bit, i.e. the result is of more than 7-bits in size in case of 8-bit signed operations and more than 15-bits in size in case of 16-bit signed operations, then the overflow flag will be set.

## 1.3 SIGNAL DESCRIPTIONS OF 8086

The microprocessor 8086 is a 16-bit CPU available in three clock rates, i.e. 5, 8 and 10 MHz, packaged in a 40 pin CERDIP or plastic package. The 8086 operates in single processor or multiprocessor configurations to achieve high performance. The pin configuration is shown in Fig. 1.5. Some of the pins serve a particular function in minimum mode (single processor mode) and others function in maximum mode (multiprocessor mode) configuration.

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

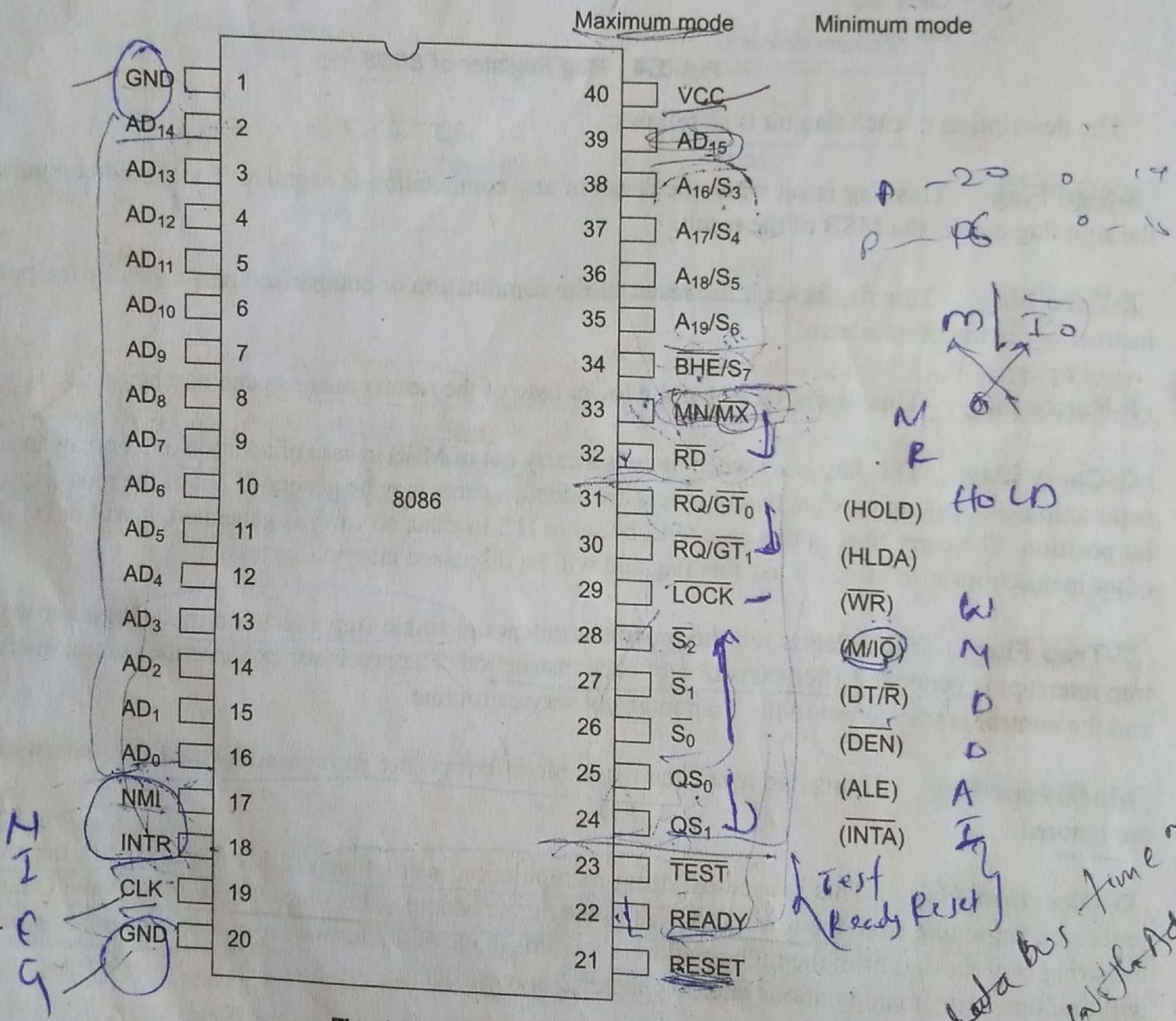



**Fig. 1.5** *Pin Configuration of 8086*

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

**$AD_{15}$-$AD_0$** These are the time multiplexed memory I/O address and data lines. Address remains on the lines during $T_1$ state, while the data is available on the data bus during $T_2, T_3, T_w$ and $T_4$. Here $T_1, T_2, T_3, T_4$ and

clock states of a machine cycle. $T_w$ is a wait state. These lines are active high and float to a tristate during interrupt acknowledge and local bus hold acknowledge cycles.

**$A_{19}/S_6, A_{18}/S_5, A_{17}/S_4, A_{16}/S_3$.** These are the time multiplexed address and status lines. During $T_1$, 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 $T_2, T_3, T_w$ and $T_4$. The status of the interrupt enable flag bit(displayed on $S_5$) is updated at the beginning of each clock cycle. The $S_4$ and $S_3$ together indicate which segment register is presently being used for memory accesses, as shown in Table 1.1. These lines float to tri-state off (tristated) during the local bus hold acknowledge. The status line $S_6$ is always low (logical). The address bits are separated from the status bits using latches controlled by the ALE signal.

**Table 1.1** *Status*

| $S_4$ | $S_3$ | *Indications* |
|---|---|---|
| 0 | 0 | Alternate Data |
| 0 | 1 | Stack |
| 1 | 0 | Code or none |
| 1 | 1 | Data |

**$\overline{BHE}/S_7$-Bus High Enable/Status** The bus high enable signal is used to indicate the transfer of data over the higher order ($D_{15}$–$D_8$) data bus as shown in Table 1.2. It goes low for the data transfers over $D_{15}$–$D_8$ and is used to derive chip selects of odd address memory bank or peripherals. $\overline{BHE}$ is low during $T_1$ for read, write and interrupt acknowledge cycles, whenever a byte is to be transferred on the higher byte of the data bus. The status information is available during $T_2$, $T_3$ and $T_4$. The signal is active low and is tristated during 'hold'. It is low during $T_1$ for the first pulse of the interrupt acknowledge cycle. $S_7$ is not currently used.

**Table 1.2** *Bus High Enable and $A_0$*

| $\overline{BHE}$ | $A_0$ | *Indication* |
|---|---|---|
| 0 | 0 | Whole word (2 bytes) |
| 0 | 1 | Upper byte from or to odd address. |
| 1 | 0 | Lower byte from or to even address |
| 1 | 1 | None |

**$\overline{RD}$ -Read** Read signal, when low, indicates the peripherals that the processor is performing a memory or I/O read operation. $\overline{RD}$ is active low and shows the state for $T_2$, $T_3$, $T_w$ of any read cycle. It remains tristated during the 'hold acknowledge'.

**INTR-Interrupt Request** This is a level triggered input. This is sampled during the last clock cycle 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 resetting the interrupt enable flag. This signal is active high and internally synchronized.

**$\overline{\text{TEST}}$** This input is examined by a 'WAIT' instruction. If the $\overline{\text{TEST}}$ input 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.

**NMI-Non-maskable Interrupt** This is an edge-triggered input which causes a Type2 interrupt. The NMI is not maskable internally by software. A transition from low to high initiates the interrupt response at the end of the current instruction. This input is internally synchronized.

**RESET** This input causes the processor to terminate the current activity and start execution from FFFF0H. The signal is active high and must be active for at least four clock cycles. It restarts execution when the RESET returns low. RESET is also internally synchronised.

**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 range of frequency for different 8086 versions is from 5MHz to 10MHz.

**Vcc** +5V power supply for the operation of the internal circuit.

**GND** ground for the internal circuit.

**MN/$\overline{\text{MX}}$** The logic level at this pin decides whether the processor is to operate in either minimum (single processor) or maximum (multiprocessor) mode.

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

**M/$\overline{\text{IO}}$-Memory/IO** This is a status line logically equivalent to $\overline{S_2}$ in the 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 in the previous $T_4$ and remains active till final $T_4$ of the current cycle. It is tristated during local bus "hold acknowledge".

**$\overline{\text{INTA}}$-Interrupt Acknowledge** This signal is used as a read strobe for interrupt acknowledge cycles. In other words, when it goes low, it means that the processor has accepted the interrupt. It is active low during $T_2$, $T_3$ and $T_w$ of each interrupt acknowledge cycle.

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

**DT/$\overline{\text{R}}$-Data Transmit/Receive** This output is used to decide the direction of data flow through the transreceivers (bidirectional buffers). When the processor sends out data, this signal is high and when the processor is receiving data, this signal is low. Logically, this is equivalent to $\overline{S}_1$ in maximum mode. Its timing is the same as M/$\overline{\text{IO}}$. This is tristated during 'hold acknowledge'.

**$\overline{\text{DEN}}$-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 $T_2$ until the middle of $T_4$. $\overline{\text{DEN}}$ is tristated during 'hold acknowledge' cycle.

**HOLD,HLDA-Hold/Hold 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 (instruction) 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 it 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 $T_4$ provided:

1. The request occurs on or before $T_2$ state of the current cycle
2. The current cycle is not operating over the lower byte of a word (or operating on an odd address)
3. The current cycle is not the first acknowledge of an interrupt acknowledge sequence
4. A Lock instruction is not being executed.

So far we have presented the pin descriptions of 8086 in minimum mode.
The following pin functions are applicable for maximum mode operation of 8086.

**$\overline{S}_2$, $\overline{S}_1$, $\overline{S}_0$-Status Lines** These are the status lines which indicate the type of operation, being carried out by the processor. These become active during $T_4$ of the previous cycle and remain active during $T_1$ and $T_2$ of the current bus cycle. The status lines return to passive state during $T_3$ of the current bus cycle so that they may again become active for the next bus cycle during $T_4$. Any change in these lines during $T_3$ indicates the starting of a new cycle, and return to passive state indicates end of the bus cycle. These status lines are encoded in Table 1.3.

**Table 1.3**

| $\overline{S}_2$ | $\overline{S}_1$ | $\overline{S}_0$ | *Indication* |
|---|---|---|---|
| 0 | 0 | 0 | Interrupt acknowledge |
| 0 | 0 | 1 | Read I/O port |
| 0 | 1 | 0 | Write I/O port |
| 0 | 1 | 1 | Halt |
| 1 | 0 | 0 | Code access |
| 1 | 0 | 1 | Read memory |
| 1 | 1 | 0 | Write memory |
| 1 | 1 | 1 | Passive |

**$\overline{\text{LOCK}}$** This output pin indicates that other system bus masters will be prevented from gaining the system bus, while the $\overline{\text{LOCK}}$ signal is low. The $\overline{\text{LOCK}}$ signal is activated by the 'LOCK' prefix instruction and remains active until the completion of the next instruction. This floats to tri-state off during "hold acknowledge". When the CPU is executing a critical instruction which requires the system bus, the LOCK prefix instruction ensures that other processors connected in the system will not gain the control of the bus. The 8086, while executing the prefixed instruction, asserts the bus lock signal output, which may be connected to an external bus controller.

**$QS_1$, $QS_0$-Queue Status** These lines give information about the status of the code-prefetch queue. These are active during the CLK cycle after which the queue operation is performed. These are encoded as shown in Table 1.4.

**Table 1.4**

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

This modification in a simple fetch and execute architecture of a conventional microprocessor offers an added advantage of *pipelined processing* of the instructions. The 8086 architecture has a 6-byte instruction prefetch queue. Thus even the largest (6-bytes) instruction can be prefetched from the memory and stored in the prefetch queue. This results in a faster execution of the instructions. In 8085, an instruction (opcode and operand) is fetched, decoded and executed and only after the execution of this instruction, the next one is fetched. By prefetching the instruction, there is a considerable speeding up in instruction execution in 8086. This scheme is known as *instruction pipelining*.

In the beginning, the CS:IP is loaded with the required address from which the execution is to be started. Initially, the queue will be empty and 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 instructions (one byte opcode instruction) and it is a part of opcode, in case of other instructions (two byte long opcode instructions), the remaining part of opcode may lie in the second byte. But invariably the first byte of an instruction is an opcode. These opcodes along with data are fetched and arranged in the queue. When the first byte from the queue goes for decoding and interpretation, one byte in the queue becomes empty and subsequently the queue is updated. The microprocessor does not perform the next fetch operation till at least two bytes of the instruction queue are emptied. The instruction execution cycle is never broken for fetch operation. After decoding the first byte, the decoding circuit decides whether the instruction is of single opcode byte or double opcode byte. If it is single opcode byte , the next bytes are treated as data bytes depending upon the decoded instruction length, otherwise, the next byte in the queue is treated as the second byte of the instruction opcode. 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 main point to be noted here is, that the fetch operation of the next instruction is overlapped with the execution of the current instruction. As shown in the architecture, there are two separate units, namely, the execution unit and the 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. Figure 1.6 explains the queue operation.

**$\overline{RQ}/\overline{GT_0}$, $\overline{RQ}/\overline{GT_1}$ -Request/Grant** These pins are used by other local bus masters, in maximum mode, to force the processor to release the local bus at the end of the processor's current bus cycle. Each of the pins is bidirectional with $\overline{RQ}/\overline{GT_0}$ having higher priority than $\overline{RQ}/\overline{GT_1}$. $\overline{RQ}/\overline{GT}$ pins have internal pull-up resistors and may be left unconnected. The request/grant sequence is as follows:

1. A pulse one clock wide from another bus master requests the bus access to 8086.
2. During $T_4$ (current) or $T_1$ (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"

...the 8086 may regain control of the local bus at the next clock ...

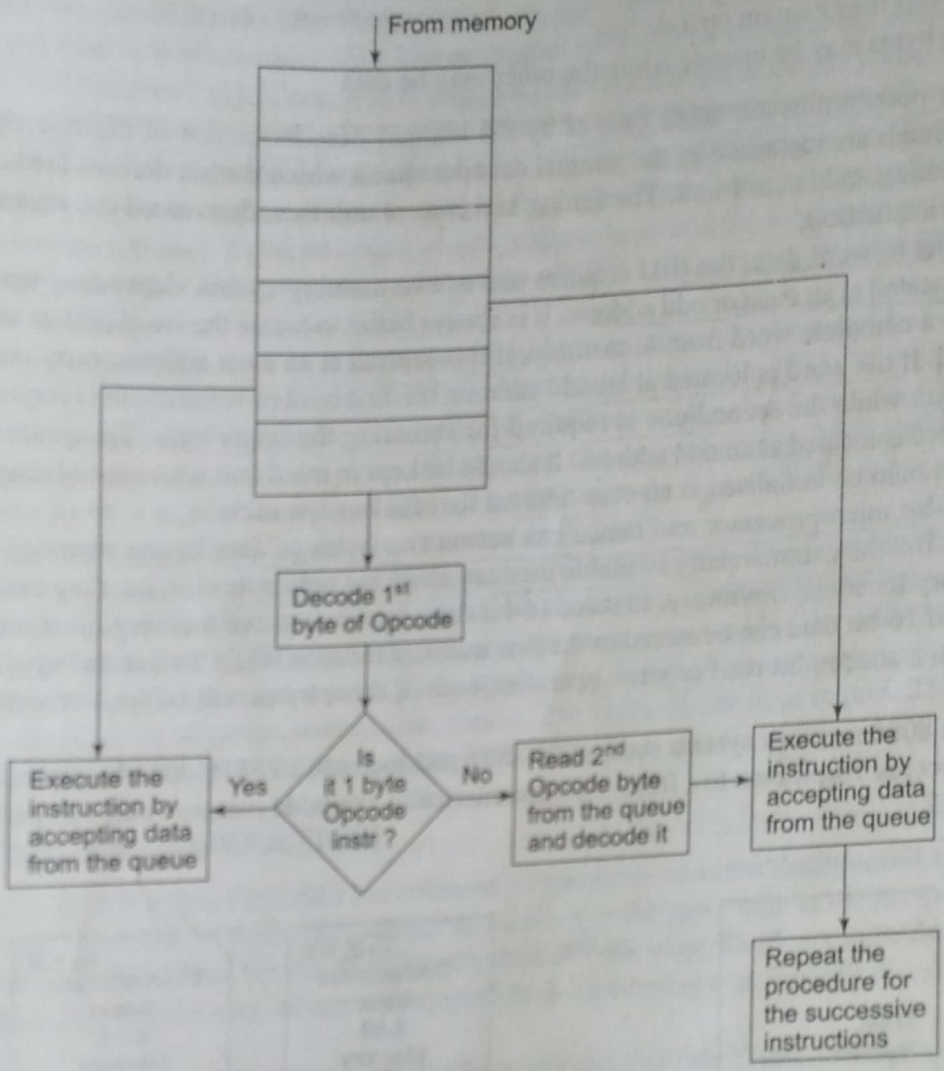



**Fig. 1.6** *The Queue Operation*

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 requests those are received while 8086 is performing memory or I/O cycle, the granting of the bus is governed by the rules as discussed in case of HOLD, and HLDA in the minimum mode.

Until now, we have described the architecture and pin configuration of 8086. In the next section, we will study some operational features of 8086 based systems.

## 1.4 PHYSICAL MEMORY ORGANISATION

In an 8086 based system, the 1Mbytes memory is physically organised as an odd bank and an even bank, each of 512 Kbytes, addressed in parallel by the processor. Byte data with an even address is transferred on $D_7 - D_0$, while the byte data with an odd address is transferred on $D_{15} - D_8$ buss lines. The processor provides two enable signals, $\overline{BHE}$ and $A_0$ for selection of either even or odd or both the banks. The instruction

stream is fetched from memory as words and is addressed internally by the processor as necessary. In other words, if the processor fetches a word (consecutive two bytes) from memory, there are different possibilities, like:

1. Both the bytes may be data operands
2. Both the bytes may contain opcode bits
3. One of the bytes may be opcode while the other may be data

All the above possibilities are taken care of by the internal decoder circuit of the microprocessor. The opcodes and operands are identified by the internal decoder circuit which further derives the signals those act as input to the timing and control unit. The timing and control unit then derives all the signals required for execution of the instruction.

While referring to word data, the BIU requires one or two memory cycles, depending upon whether the starting byte is located at an even or odd address. It is always better to locate the word data at an even address. To read or write a complete word from/to memory, if it is located at an even address, only one read or write cycle is required. If the word is located at an odd address, the first read or write cycle is required for accessing the lower byte while the second one is required for accessing the upper byte. Thus, two bus cycles are required, if a word is located at an odd address. It should be kept in mind that while initialising the structures like stack they should be initialised at an even address for efficient operation.

8086 is a 16-bit microprocessor and hence can access two bytes of data in one memory or I/O read or write operation. But the commercially available memory chips are only byte size, i.e. they can store only one byte in a memory location. Obviously, to store 16-bit data, two successive memory locations are used and the lower byte of 16-bit data can be stored in the first memory location while the second byte is stored in the next location. In a sixteen bit read or write operation both of these bytes will be read or written in a single machine cycle.

A map of an 8086 memory system starts at 00000H and ends at FFFFFH. 8086 being a 16-bit processor is expected to access 16-bit data to / from 8-bit commerically available memory chips in parallel, as shown below in Fig. 1.7.

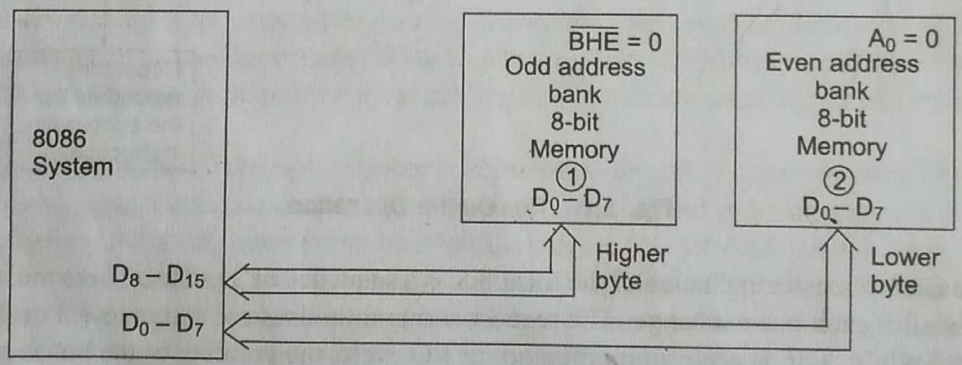



**Fig. 1.7** *Physical Memory Organisation*

Thus, bits $D_0$–$D_7$ of a 16-bit data will be transferred over $D_0$–$D_7$ (lower byte) of 16-bit data bus to / from 8-bit memory (2) and bit $D_8$–$D_{15}$ of the 16-bit data will be transferred over $D_8$–$D_{15}$ (higher byte) of the 16-bit data bus of the microprocessor, to / from 8-bit memory (1). Thus to achieve 16-bit data transfer using 8-bit memories, in parallel, the map of the complete system byte memory addresses will obviously be divided into the two memory banks as shown in Fig. 1.7.

16-bit data is stored at the first address of the map 00000H and it is to be transferred
roprocessor bus so 00000H must be in 8-bit memory (2). Higher byte of the 16-bit
xt address 00001H; it is to be transferred over $D_8$–$D_{15}$ of the microprocessor bus
must be in 8-bit memory (1) of Fig. 1.7. On similar lines, for the next 16-Bit data
immediately after the previous one, the lower byte will be stored at the next address
in 8-bit memory (2) while the higher byte will be stored at the next address 00003H
nemory (1). Thus, if it is imagined that the complete memory map of 8086 is filled
e lower bytes ($D_0$–$D_7$) will be stored in the 8-bit memory bank (2) and all the higher
e stored in the 8-bit memory bank (1). Consequently, it can be observed that all the
stored at even addresses and all the higher bytes have to be stored at odd addresses.
y bank (1) will be called an odd address bank and the 8-bit memory bank (2) will be
s bank. The complete memory map of 8086 system is thus divided into even and odd
s. )

a 16-bit data to / from memory, both of these banks must be selected for the
owever, to maintain an upward compatibility with 8085, 8086 must be able to
erations. In which case, two possibilities arise; the first being 8-bit operation
bank, i.e. with an even address and the second one is 8-bit operation with odd
nk, i.e. with an odd address. The two signals $A_0$ and $\overline{BHE}$ solve the problem of
riate memory banks as presented in Table 1.2.

in memory are reserved for specific CPU operations. The locations from FFFF0H to
d for operations including jump to initialisation programme and I/O-processor initialisa-
00000H to 003FFH are reserved for *interrupt vector table*. The interrupt structure pro-
l of 256 interrupt vectors. The vectors, i.e. CS and IP for each interrupt routine requires
t in the interrupt vector table. Hence, 256 types of interrupt require 256 C 4 = 03FFH
or the complete interrupt vector table.

## . BUS OPERATION

bined address and data bus commonly referred to as a time multiplexed address and data
on behind multiplexing address and data over the same pins is the maximum utilisation
nd it facilitates the use of 40 pin standard DIP package. The bus can be demultiplexed us-
nd transreceivers, whenever required. In the following text, we will discuss a general bus

processor bus cycles consist of at least four clock cycles. These are referred to as $T_1$, $T_2$,$T_3$
s is transmitted by the processor during $T_1$. It is present on the bus only for one cycle. During
cle, the bus is tristated for changing the direction of bus for the following data read cycle.
kes place during $T_3$ and $T_4$. In case, an addressed device is slow and shows 'NOT READY'
es $T_w$ are inserted between $T_3$ and $T_4$. These clock states during wait period are called *idle*
*ates* ($T_w$) or *inactive states*. The processor uses these cycles for internal housekeeping. The
able (ALE) signal is emitted during $T_1$ by the processor (minimum mode) or the bus control-
de) depending upon the status of the MN/$\overline{MX}$ input. The negative edge of this ALE pulse is
e address and the data or status information. In maximum mode, the status lines $\overline{S}_0$, $\overline{S}_1$ and $\overline{S}_2$
te the type of operation as discussed in the signal description section of this chapter. Status bits
iplexed with higher order address bits and the $\overline{BHE}$ signal. Address is valid during $T_1$ while
to $S_7$ are valid during $T_2$ through $T_4$. Figure 1.8 shows a general bus operation cycle of 8086.

lower bytes have to be stored at even addresses and all the higher bytes have to be stored at odd addresses. Thus, the 8-bit memory bank (1) will be called an odd address bank and the 8-bit memory bank (2) will be called an even address bank. The complete memory map of 8086 system is thus divided into even and odd address memory banks.

If 8086 transfers a 16-bit data to / from memory, both of these banks must be selected for the 16-bit operation. However, to maintain an upward compatibility with 8085, 8086 must be able to implement 8-bit operations. In which case, two possibilities arise; the first being 8-bit operation with even memory bank, i.e. with an even address and the second one is 8-bit operation with odd address memory bank, i.e. with an odd address. The two signals $A_0$ and $\overline{BHE}$ solve the problem of selection of appropriate memory banks as presented in Table 1.2.

Certain locations in memory are reserved for specific CPU operations. The locations from FFFF0H to FFFFFH are reserved for operations including jump to initialisation programme and I/O-processor initialisation. The locations 00000H to 003FFH are reserved for *interrupt vector table*. The interrupt structure provides space for a total of 256 interrupt vectors. The vectors, i.e. CS and IP for each interrupt routine requires 4 bytes for storing it in the interrupt vector table. Hence, 256 types of interrupt require 256 C 4 = 03FFH (1Kbyte) locations for the complete interrupt vector table.

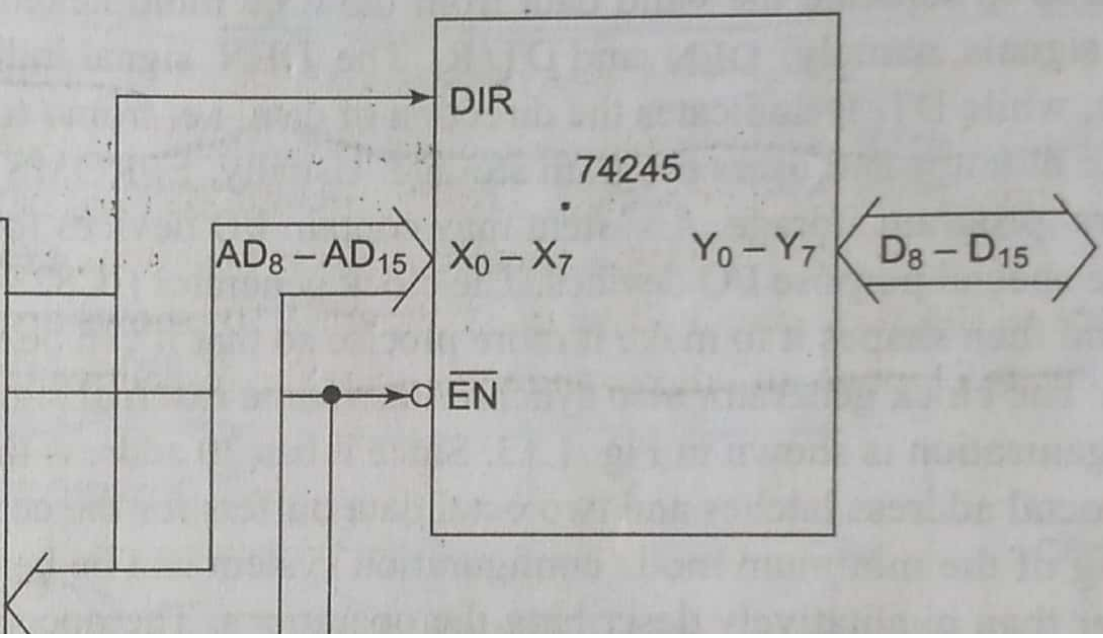
## 1.5 GENERAL BUS OPERATION

The 8086 has a combined address and data bus commonly referred to as a time multiplexed address and data bus. The main reason behind multiplexing address and data over the same pins is the maximum utilisation of processor pins and it facilitates the use of 40 pin standard DIP package. The bus can be demultiplexed using a few latches and transreceivers, whenever required. In the following text, we will discuss a general bus operation cycle.

Basically, all the processor bus cycles consist of at least four clock cycles. These are referred to as $T_1$, $T_2$, $T_3$ and $T_4$. The address is transmitted by the processor during $T_1$. It is present on the bus only for one cycle. During $T_2$, i.e. the next cycle, the bus is tristated for changing the direction of bus for the following data read cycle. The data transfer takes place during $T_3$ and $T_4$. In case, an addressed device is slow and shows 'NOT READY' status the wait states $T_w$ are inserted between $T_3$ and $T_4$. These clock states during wait period are called *idle states* ($T_i$), *wait states* ($T_w$) or *inactive states*. The processor uses these cycles for internal housekeeping. The Address Latch Enable (ALE) signal is emitted during $T_1$ by the processor (minimum mode) or the bus controller (maximum mode) depending upon the status of the MN/$\overline{MX}$ input. The negative edge of this ALE pulse is used to separate the address and the data or status information. In maximum mode, the status lines $\overline{S}_0$, $\overline{S}_1$ and $\overline{S}_2$ are used to indicate the type of operation as discussed in the signal description section of this chapter. Status bits $\overline{S}_3$ to $\overline{S}_7$ are multiplexed with higher order address bits and the $\overline{BHE}$ signal. Address is valid during $T_1$ while the status bits $S_3$ to $S_7$ are valid during $T_2$ through $T_4$. Figure 1.8 shows a general bus operation cycle of 8086.

bus in the form of $AD_0$–$AD_{15}$. The data can be separated from the address
al buffers 74245. It may be noted that the data can either be transferred
from memory to microprocessor in case of write or read operations re-
ers are required for deriving the data bus. The signals $\overline{DNE}$ and DT / $\overline{R}$
bus and the direction of the data, i.e. to / from the microprocessor. They
able) and direction pins of the buffers as indicated below in Fig. 1.11.
the data is available on the multiplexed bus and both the buffers (74245)
DIR pin goes high the data available at X pins of 74245 are transferred to
microprocessor to either memory or IO device (write operation). If DIR
pins of 74245 is transferred to X pins, i.e. data is received by micropro-
read operation).

ne available control signals $\overline{RD}$, $\overline{WR}$ and M / $\overline{IO}$ in case of minimum
al logic circuit may be used as shown in Fig. 1.12 (a) and Fig. 1.12 (b).
peration a chip bus controller derives all the control signals using status



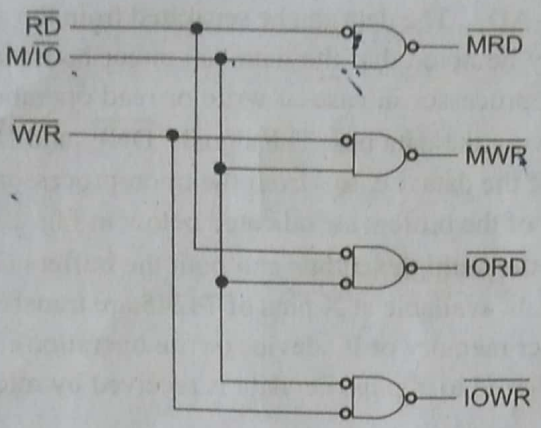



Fig. 1.12 (a) *Deriving 8086 Control Signals*

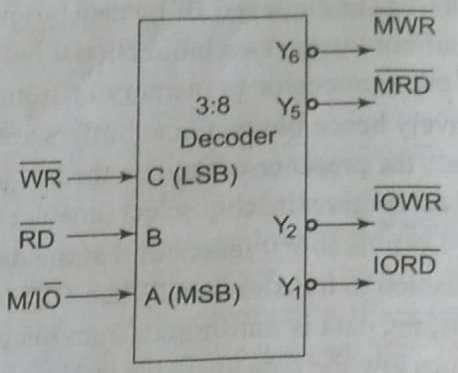



Fig. 1.12 (b) *Deriving 8086 Control Signals*

The latches are generally buffered output D-type flip-flops, like, 74LS373 or 8282. They are used for separating the valid address from the multiplexed address/data signals and are controlled by the ALE signal generated by 8086. Transreceivers are the bidirectional buffers and sometimes they are called data amplifiers. They are required to separate the valid data from the time multiplexed address/data signal. They are controlled by two signals, namely, $\overline{DEN}$ and DT/$\overline{R}$. The $\overline{DEN}$ signal indicates that the valid data is available on the data bus, while DT/$\overline{R}$ indicates the direction of data, i.e. from / to the processor. The system contains memory for the monitor and users program storage. Usually, EPROMS are used for monitor storage, while RAMs for users' program storage. A system may contain I/O devices for communication with the processor as well as some special purpose I/O devices. The clock generator (IC8284) generates the clock from the crystal oscillator and then shapes it to make it more precise so that it can be used as an accurate timing reference for the system. The clock generator also synchronizes some external signals with the system clock. The general system organisation is shown in Fig. 1.13. Since it has 20 address lines and 16 data lines, the 8086 CPU requires three octal address latches and two octal data buffers for the complete address and data separation.

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

The read cycle begins in $T_1$ with the assertion of the Address Latch Enable (ALE) signal and M/$\overline{IO}$ signal. During the negative going edge of this signal, the valid address is latched on the local bus. The $\overline{BHE}$ and $A_0$ signals address low, high or both bytes. From $T_1$ to $T_4$, the M/$\overline{IO}$ signal indicates a memory or I/O operation. At $T_2$, the address is removed from the local bus and is sent to the output. The bus is then tristated. The Read ($\overline{RD}$) control signal is also activated in $T_2$. This signal causes the addressed device to enable its data bus drivers. After $\overline{RD}$ goes low, the valid data is available on the data bus. The addressed device will drive the READY line high. When the processor returns the read signal to high level, the addressed device will again tristate its bus drivers. CS logic indicates chip select logic and 'e' and 'O' sufixes indicate even and odd address memory banks.

A write cycle also begins with the assertion of ALE and the emission of the address. The M/$\overline{IO}$ signal is again asserted to indicate a memory or I/O operation. In $T_2$, after sending the address in $T_1$, the processor sends the data to be written to the addressed location. The data remains on the bus until the middle of $T_4$ state. The $\overline{WR}$ becomes active at the beginning of $T_2$ (unlike $\overline{RD}$ is somewhat delayed in $T_2$ to provide time for floating).

The $\overline{BHE}$ and $A_0$ signals are used to select the proper byte or bytes of memory or I/O word to be read or written as already discussed in the signal description section of this chapter.

The M/$\overline{IO}$, $\overline{RD}$ and $\overline{WR}$ signals indicate the types of data transfer as specified in Table 1.5.

**Table 1.5**

| $M/\overline{IO}$ | $\overline{RD}$ | $\overline{DEN}$ | *Transfer Type* |
|---|---|---|---|
| 0 | 0 | 1 | I/O read |
| 0 | 1 | 0 | I/O write |
| 1 | 0 | 1 | Memory read |
| 1 | 1 | 0 | Memory write |

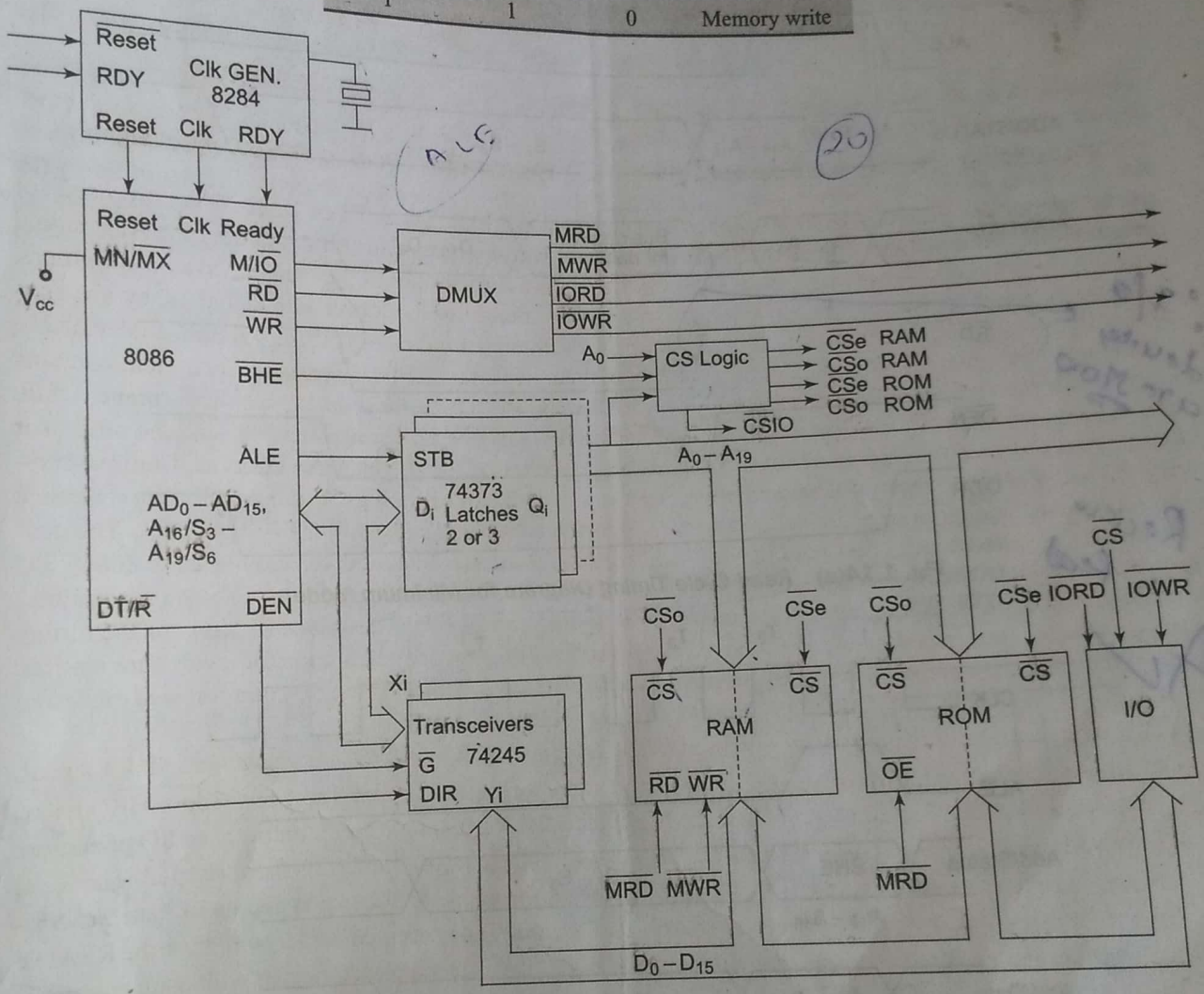



**Fig. 1.13** *Minimum Mode 8086 System*

Figure 1.14(a) shows the read cycle while Fig. 1.14(b) shows the write cycle.

### 1.8.1 HOLD Response Sequence

The HOLD pin is checked at the end of each bus cycle. If it is received active by the processor before $T_4$ of the previous cycle or during $T_1$ state of the current cycle, the CPU activates HLDA in the next clock cycle and for the succeeding bus cycles, the bus will be given to another requesting master. The control of the bus is not regained by the processor until the requesting master does not drop the HOLD pin low. When the request

is dropped by the requesting master, the HLDA is dropped by the processor at the trailing edge of the next clock, as shown in Fig. 1.14 (c). The other conditions have already been discussed in the signal description section for the HOLD and HLDA signals.

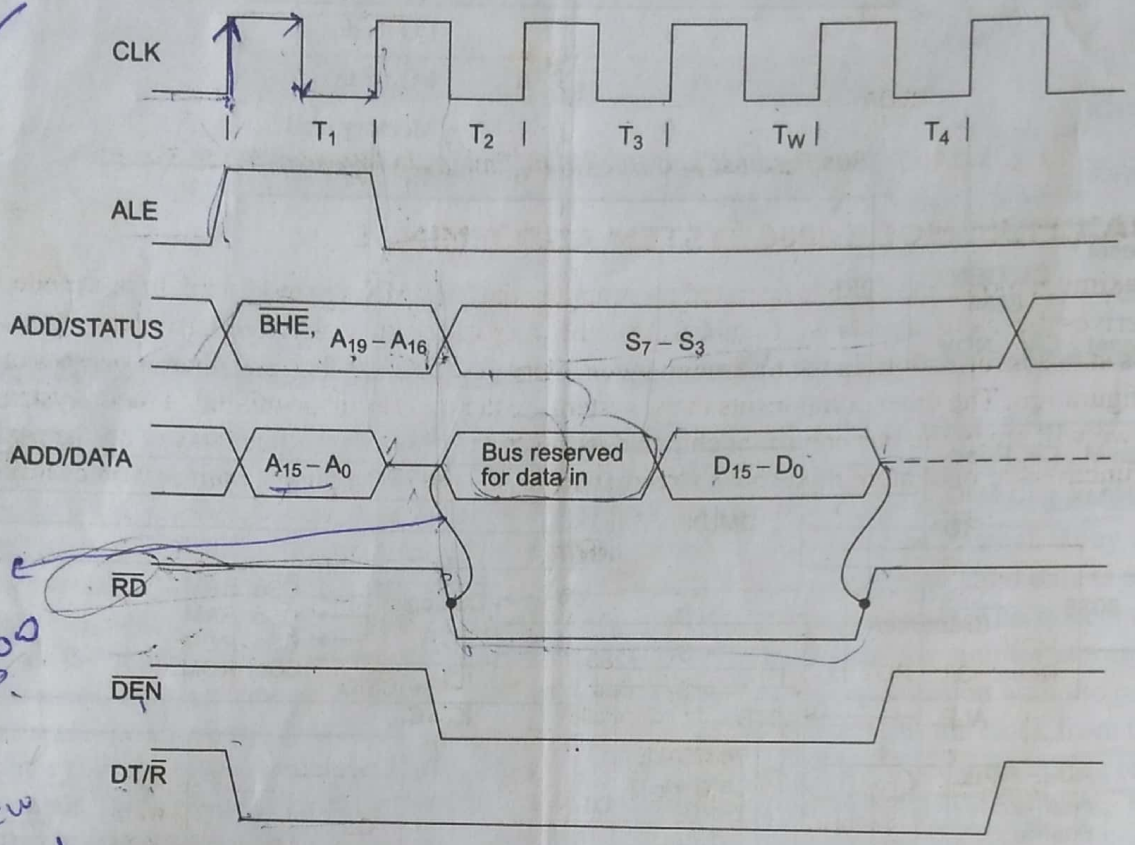



**Fig. 1.14(a)** *Read Cycle Timing Diagram for Minimum Mode*

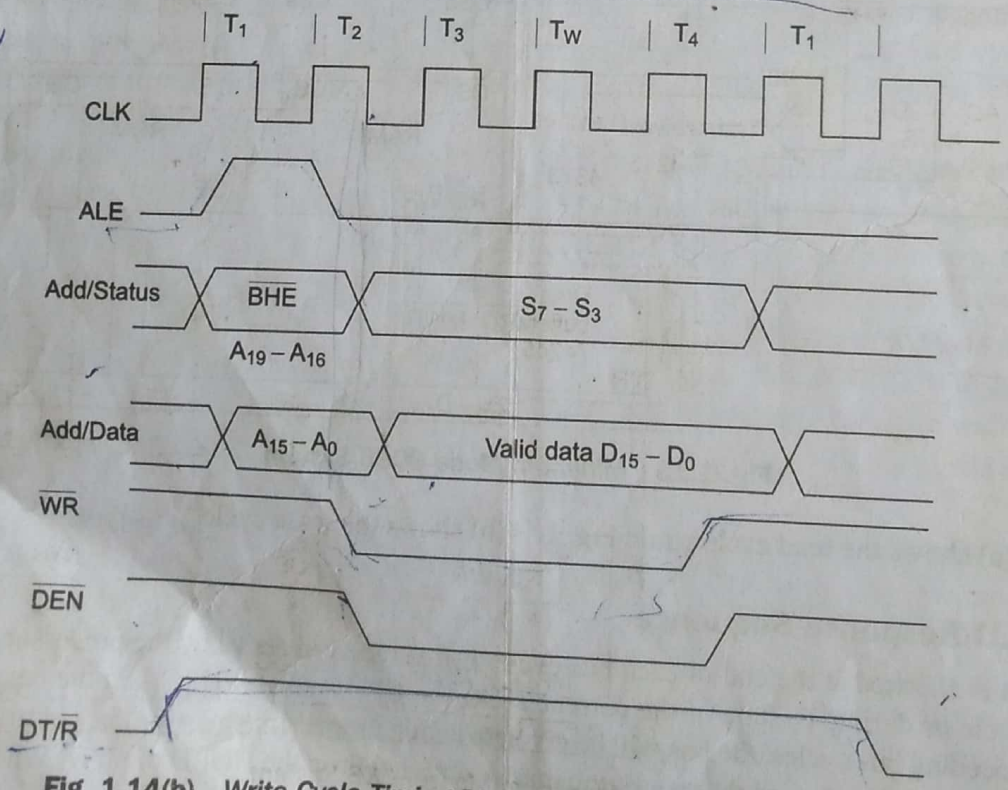



**Fig. 1.14(b)** *Write Cycle Timing Diagram for Minimum Mode Operation*

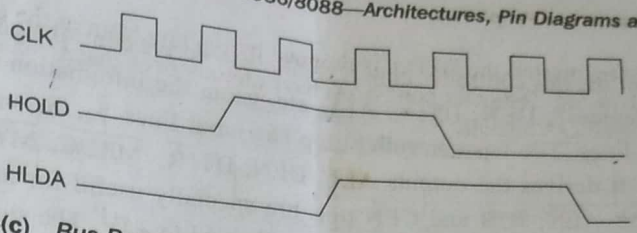



Fig. 1.14(c) *Bus Request and Bus Grant Timings in Minimum Mode System*

## 1.9 MAXIMUM MODE 8086 SYSTEM AND TIMINGS

In the maximum mode, the 8086 is operated by strapping the MN/$\overline{MX}$ pin to ground. In this mode, the processor derives the status signals $\overline{S}_2$, $S_1$ and $\overline{S}_0$. Another chip called bus controller derives the control signals using this status information. In the maximum mode, there may be more than one microprocessor in the system configuration. The other components in the system are the same as in the minimum mode system. In this section, we will study the bus controller chip and its functions in brief. The functions of all the pins having special functions in maximum mode have already been discussed in the pin diagram section of this chapter.

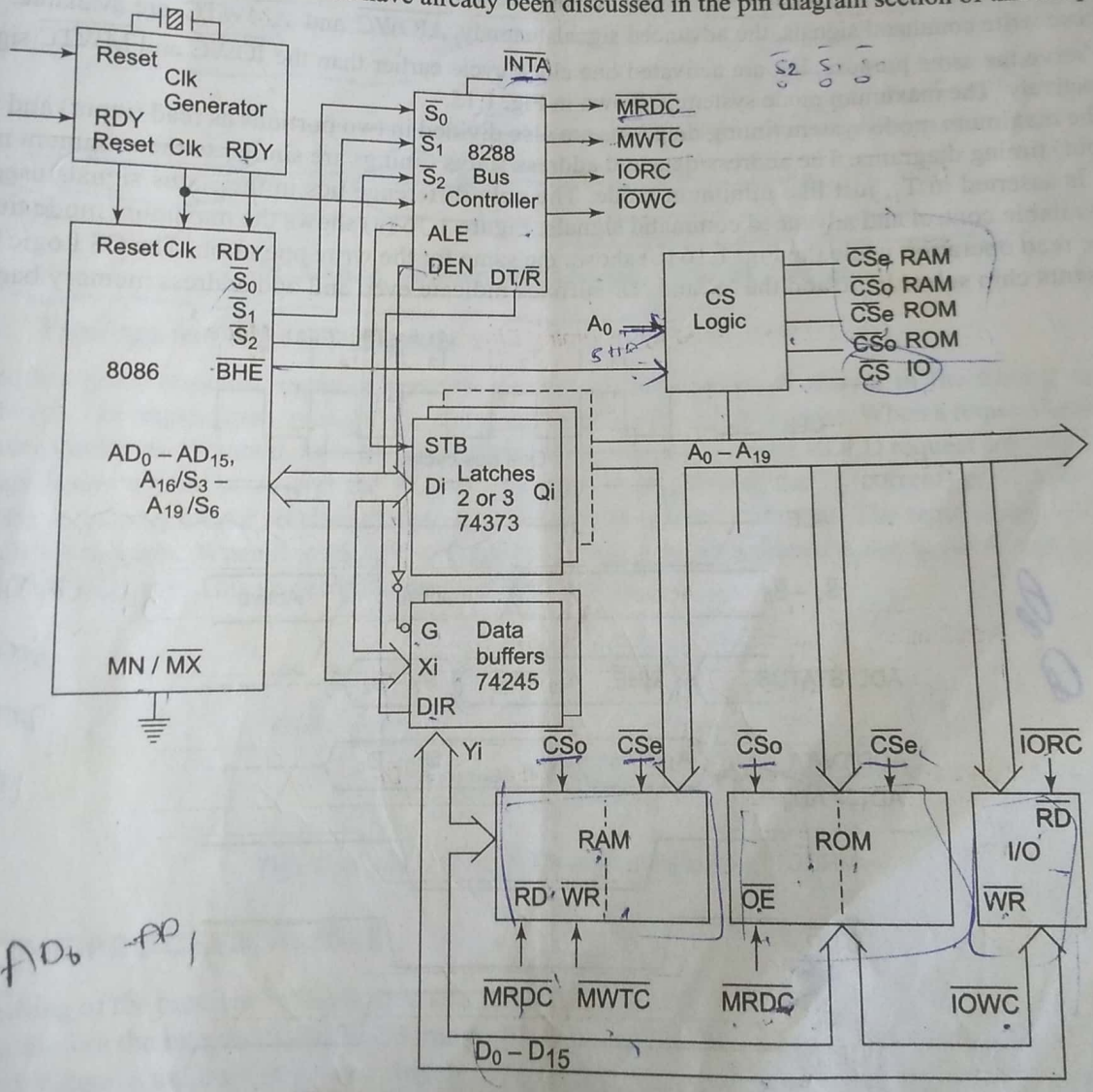



Fig. 1.15 *Maximum Mode 8086 System*

The basic functions of the bus controller chip IC8288, is to derive control signals like $\overline{RD}$ and $\overline{WR}$ (for memory and I/O devices), $\overline{DEN}$, DT/$\overline{R}$, ALE, etc. using the information made available by the processor on the status lines. The bus controller chip has input lines $\overline{S}_2$, $\overline{S}_1$ and $\overline{S}_0$ and CLK, which are driven by the CPU. It derives the outputs ALE, DEN, DT/$\overline{R}$, $\overline{MRDC}$, $\overline{MWTC}$, $\overline{AMWC}$, $\overline{IORC}$, $\overline{IOWC}$ and $\overline{AIOWC}$. The $\overline{AEN}$, IOB and CEN pins are specially useful for multiprocessor systems. $\overline{AEN}$ and IOB are generally grounded. CEN pin is usually tied to +5V. The significance of the MCE/$\overline{PDEN}$ output depends upon the status of the IOB pin. If IOB is grounded, it acts as master cascade enable to control cascaded 8259A, else it acts as peripheral data enable used in the multiple bus configurations. $\overline{INTA}$ pin is used to issue two interrupt acknowledge pulses to the interrupt controller or to an interrupting device.

$\overline{IORC}$, $\overline{IOWC}$ are I/O read command and I/O write command signals respectively. These signals enable an IO interface to read or write the data from or to the addressed port. The $\overline{MRDC}$, $\overline{MWTC}$ are memory read command and memory write command signals respectively and may be used as memory read and write signals. All these command signals instruct the memory to accept or send data to or from the bus. For both of these write command signals, the advanced signals namely $\overline{AIOWC}$ and $\overline{AMWTC}$ are available. They also serve the same purpose, but are activated one clock cycle earlier than the $\overline{IOWC}$ and $\overline{MWTC}$ signals, respectively. The maximum mode system is shown in Fig. 1.15.

The maximum mode system timing diagrams are also divided in two portions as read (input) and write (output) timing diagrams. The address/data and address/status timings are similar to the minimum mode. ALE is asserted in $T_1$, just like minimum mode. The only difference lies in the status signals used and the available control and advanced command signals. Figure 1.16 (a) shows the maximum mode timings for the read operation while the Fig. 1.16 (b) shows the same for the write operation. The CS Logic block represents chip select logic and the 'e' and 'O' suffixes indicate even and odd address memory bank.

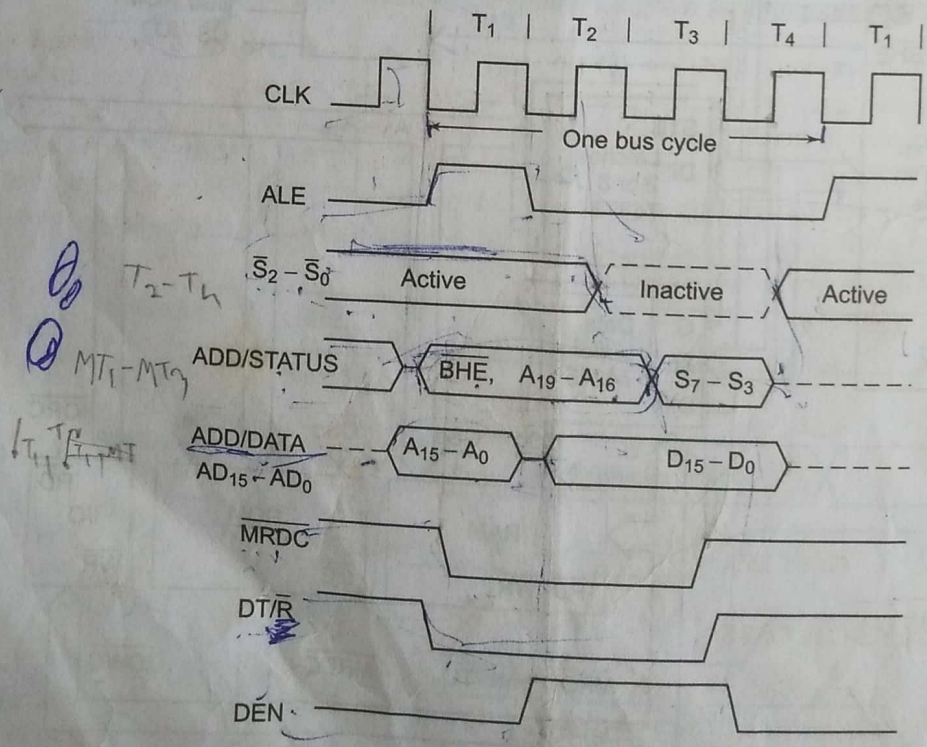



**Fig. 1.16 (a)** *Memory Read Timing in Maximum Mode*

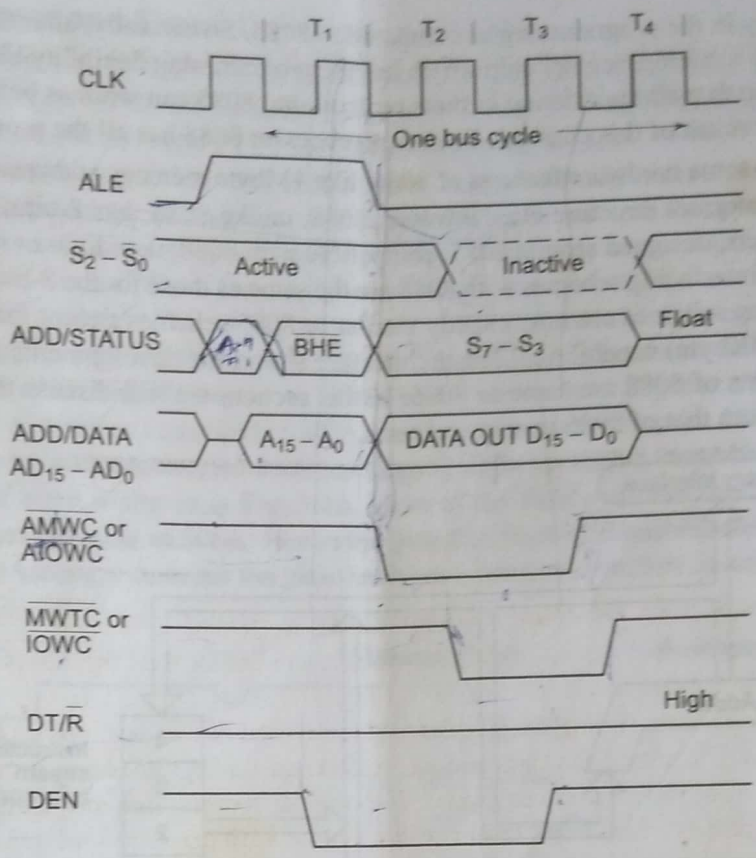



**Fig. 1.16(b)** *Memory Write Timing in Maximum Mode*

### 1.9.1 Timings for $\overline{RQ}/\overline{GT}$ Signals

The request/grant response sequence contains a series of three pulses as shown in the timing diagram Fig.1.16 (c). The request/grant pins are checked at each rising pulse of clock input. When a request is detected and if the conditions discussed in pin diagram section of this chapter for valid HOLD request are satisfied, the processor issues a grant pulse over the $\overline{RQ}/\overline{GT}_0$ pin immediately during the $T_4$ (current) or $T_1$ (next) state. When the requesting master receives this pulse, it accepts the control of the bus. The requesting master uses the bus till it requires. When it is ready to relinquish the bus, it sends a release pulse to the processor (host) using the $\overline{RQ}/\overline{GT}$ pin. This sequence is shown in Fig. 1.16 (c).

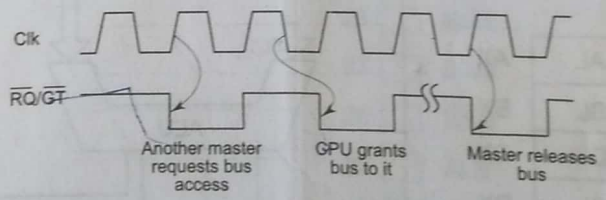



**Fig. 1.16 (c)** *$\overline{RQ}/\overline{GT}$ Timings in Maximum Mode*

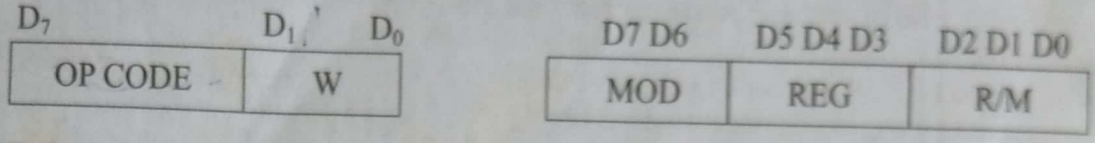
## 1.10 THE PROCESSOR 8088

The launching of the processor 8086 is seen as a remarkable step in the development of high speed computing machines. Before the introduction of 8086, most of the circuits required for the different applications in computing and industrial control fields were already designed around the 8-bit processor 8085. The 8086 imparted

In Chapter 1, we have discussed the 8086/8088 architecture, pin diagrams and timing diagrams of read and write cycles. This chapter aims at introducing the readers with the general instruction formats, different addressing modes supported by 8086/8088 along with 8086/8088 instruction set. Further, a few important and frequently used assembler directives and operators have also been discussed. Thus this chapter creates a background for 'assembly language programming using 8086/8088'. A number of assemblers are available for programming with 8086/8088. Each of them has slightly different syntax, directives and operators. However, most of them work on similar principles. The directives and operators considered here are available with MASM (Microsoft MACRO ASSEMBLER).

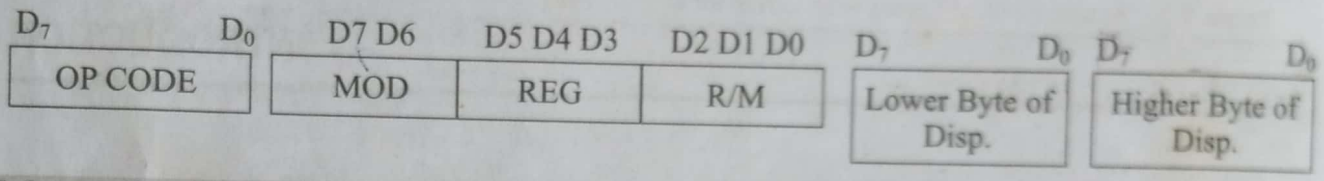
The register represented by the REG field is one of the operands. The R/M field specifies another register or memory location, i.e. the other operand.

**3. Register to/from Memory with no Displacement** This format is also 2 bytes long and similar to the register to register format except for the MOD field as shown.

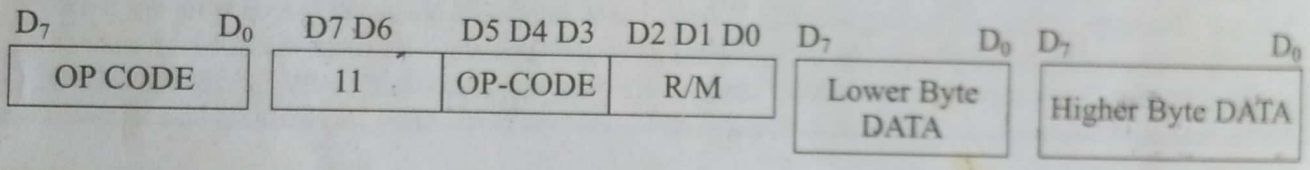



The MOD field shows the mode of addressing. The MOD, R/M, REG and the W fields are decided in Table 2.2.

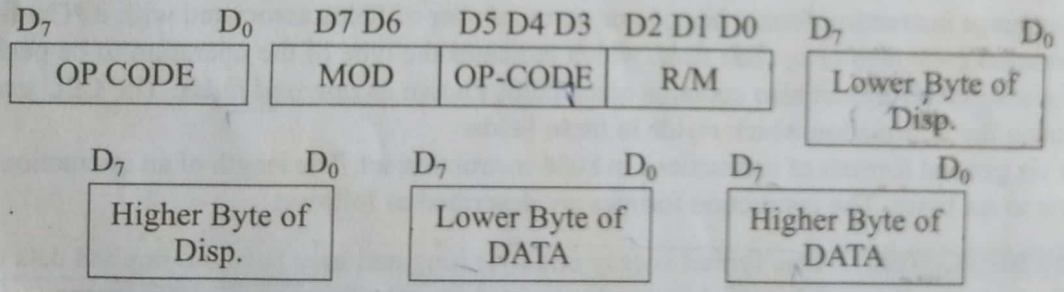
**4. Register to/from Memory with Displacement** This type of instruction format contains one or two additional bytes for displacement along with 2-byte the format of the register to/from memory without displacement. The format is as shown below.




**5. Immediate Operand to Register** In this format, the first byte as well as the 3-bits from the second byte which are used for REG field in case of register to register format are used for opcode. It also contains one or two bytes of immediate data. The complete instruction format is as shown below.




**6. Immediate Operand to Memory with 16-bit Displacement** This type of instruction format requires 5 or 6 bytes for coding. The first 2 bytes contain the information regarding OPCODE, MOD, and R/M fields. The remaining 4 bytes contain 2 bytes of displacement and 2 bytes of data as shown.



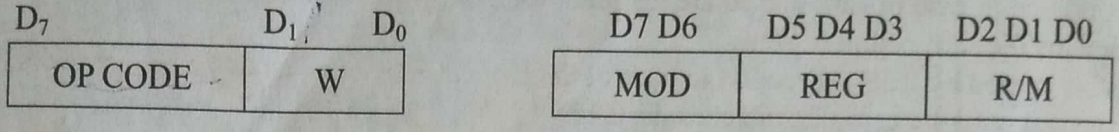

The opcode usually appears in the first byte, but in a few instructions, a register destination is in the first byte and few other instructions may have their 3-bits of opcode in the second byte. The opcodes have the single bit indicators. Their definitions and significances are given as follows:

**W-bit** This indicates whether the instruction is to operate over an 8-bit or 16-bit data/operands. If W bit is 0, the operand is of 8-bits and if W is 1, the operand is of 16-bits.

**D-bit** This is valid in case of double operand instructions. One of the operands must be a register specified by the REG field. The register specified by REG is source operand if D = 0, else, it is a destination operand.

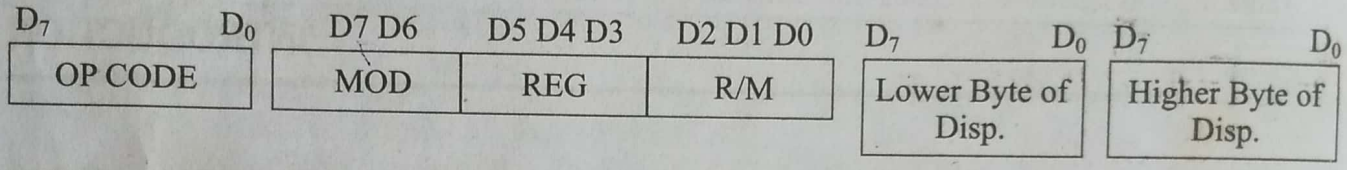
The register represented by the REG field is one of the operands. The R/M field specifies another register or memory location, i.e. the other operand.

**3. Register to/from Memory with no Displacement** This format is also 2 bytes long and similar to the register to register format except for the MOD field as shown.

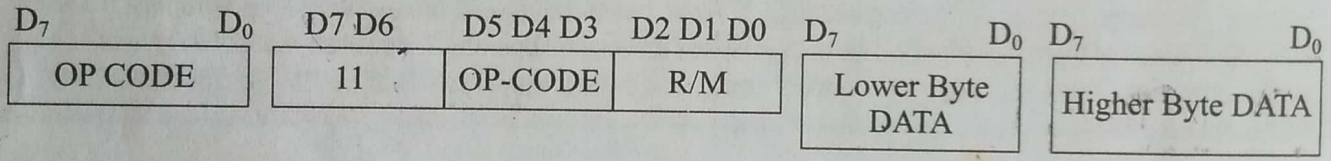



The MOD field shows the mode of addressing. The MOD, R/M, REG and the W fields are decided in Table 2.2.

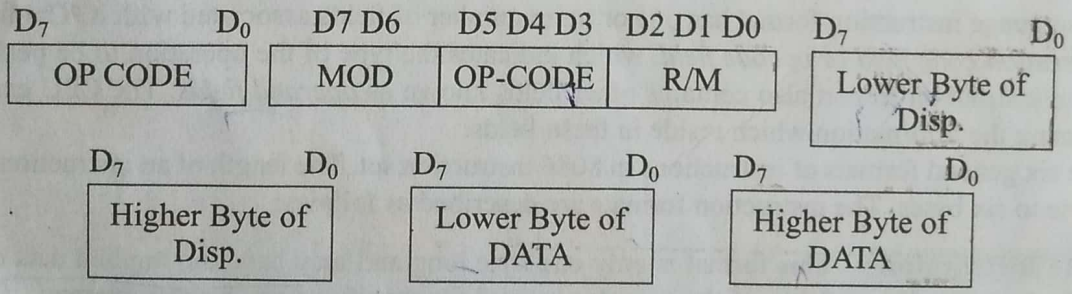
**4. Register to/from Memory with Displacement** This type of instruction format contains one or two additional bytes for displacement along with 2-byte the format of the register to/from memory without displacement. The format is as shown below.




**5. Immediate Operand to Register** In this format, the first byte as well as the 3-bits from the second byte which are used for REG field in case of register to register format are used for opcode. It also contains one or two bytes of immediate data. The complete instruction format is as shown below.




**6. Immediate Operand to Memory with 16-bit Displacement** This type of instruction format requires 5 or 6 bytes for coding. The first 2 bytes contain the information regarding OPCODE, MOD, and R/M fields. The remaining 4 bytes contain 2 bytes of displacement and 2 bytes of data as shown.




The opcode usually appears in the first byte, but in a few instructions, a register destination is in the first byte and few other instructions may have their 3-bits of opcode in the second byte. The opcodes have the single bit indicators. Their definitions and significances are given as follows:

**W-bit** This indicates whether the instruction is to operate over an 8-bit or 16-bit data/operands. If W bit is 0, the operand is of 8-bits and if W is 1, the operand is of 16-bits.

**D-bit** This is valid in case of double operand instructions. One of the operands must be a register specified by the REG field. The register specified by REG is source operand if D = 0, else, it is a destination operand.

**Table 2.2** *Addressing Modes and the Corresponding MOD, REG and R/M Fields*

| Operands | Memory Operands: No Displacement | Memory Operands: Displacement 8-bit | Memory Operands: Displacement 16-bit | Register Operands | |
|---|---|---|---|---|---|
| MOD | 00 | 01 | 10 | 11 | |
| R/M | | | | W = 0 | W = 1 |
| 000 | (BX) + (SI) | (BX) + (SI) + D8 | (BX) + (SI) + D16 | AL | AX |
| 001 | (BX) + (DI) | (BX) + (DI) + D8 | (BX) + (DI) + D16 | CL | CX |
| 010 | (BP) + (SI) | (BP) + (SI) + D8 | (BP) + (SI) + D16 | DL | DX |
| 011 | (BP) + (DI) | (BP) + (DI) + D8 | (BP) + (DI) + D16 | BL | BX |
| 100 | (SI) | (SI) + D8 | (SI) + D16 | AH | SP |
| 101 | (DI) | (DI) + D8 | (DI) + D16 | CH | BP |
| 110 | D16 | (BP) + D8 | (BP) + D16 | DH | SI |
| 111 | (BX) | (BX) + D8 | (BX) + D16 | BH | DI |

*Note:* 1. D8 and D16 represent 8 and 16 bit displacements respectively.
2. The default segment for the addressing modes using BP and SP is SS. For all other addressing modes the default segments are DS or ES.

DS is the default data segment register when a data is to be referred as an operand. CS is the default code segment register for storing program codes (executable codes). SS is the default segment register for the stack data accesses and operations. ES is the default segment register for the destination data storage. All the segments available (defined in a particular program) can be read or written as data segments by newly defining the data segment as required. There is no physical difference in the memory structure or no physical separation between the segment areas. They may or may not overlap with each other. Chapter 3 on 'Assembly Language Programming' explains the coding procedure of the instructions with suitable examples.

## 2.2 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 categorised 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 or the 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 and 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.

| | | | | | |
|---|---|---|---|---|---|
| 000 | (BX) + (SI) | (BX) + (SI) + D8 | (BX) + (SI) + D16 | AX | AX |
| 001 | (BX) + (DI) | (BX) + (DI) + D8 | (BX) + (DI) + D16 | CX | CX |
| 010 | (BP) + (SI) | (BP) + (SI) + D8 | (BP) + (SI) + D16 | DX | DX |
| 011 | (BP) + (DI) | (BP) + (DI) + D8 | (BP) + (DI) + D16 | BX | BX |
| 100 | (SI) | (SI) + D8 | (SI) + D16 | SP | SP |
| 101 | (DI) | (DI) + D8 | (DI) + D16 | BP | BP |
| 110 | D16 | (BP) + D8 | (BP) + D16 | SI | SI |
| 111 | (BX) | (BX) + D8 | (BX) + D16 | DI | DI |

*Note* 1. D8 and D16 represent 8 and 16 bit displacements respectively.
2. The default segment for the addressing modes using BP and SP is SS. For all other addressing modes the default segments are DS or ES.

DS is the default data segment register when a data is to be referred as an operand. CS is the default code segment register for storing program codes (executable codes). SS is the default segment register for the stack data accesses and operations. ES is the default segment register for the destination data storage. All the segments available (defined in a particular program) can be read or written as data segments by newly defining the data segment as required. There is no physical difference in the memory structure or no physical separation between the segment areas. They may or may not overlap with each other. Chapter 3 on 'Assembly Language Programming' explains the coding procedure of the instructions with suitable examples.

## 2.2 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 categorised 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 or the address somehow specified in the instruction, after their execution.* For example, JMP, CALL, RET and JUMP instructions fall under this category.

The addressing modes for sequential and 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.

**Table 2.2** *Addressing Modes and the Corresponding MOD, REG and R/M Fields*

| *Operands* | *Memory Operands* | | | *Register Operands* | |
|---|---|---|---|---|---|
| | *No Displacement* | *Displacement 8-bit* | *Displacement 16-bit* | | |
| *MOD* / *R/M* | *00* | *01* | *10* | *11* | |
| | | | | *W = 0* | *W = 1* |
| 000 | (BX) + (SI) | (BX) + (SI) + D8 | (BX) + (SI) + D16 | AL | AX |
| 001 | (BX) + (DI) | (BX) + (DI) + D8 | (BX) + (DI) + D16 | CL | CX |
| 010 | (BP) + (SI) | (BP) + (SI) + D8 | (BP) + (SI) + D16 | DL | DX |
| 011 | (BP) + (DI) | (BP) + (DI) + D8 | (BP) + (DI) + D16 | BL | BX |
| 100 | (SI) | (SI) + D8 | (SI) + D16 | AH | SP |
| 101 | (DI) | (DI) + D8 | (DI) + D16 | CH | BP |
| 110 | D16 | (BP) + D8 | (BP) + D16 | DH | SI |
| 111 | (BX) | (BX) + D8 | (BX) + D16 | BH | DI |

*Note:* 1. D8 and D16 represent 8 and 16 bit displacements respectively.
2. The default segment for the addressing modes using BP and SP is SS. For all other addressing modes the default segments are DS or ES.

DS is the default data segment register when a data is to be referred as an operand. CS is the default code segment register for storing program codes (executable codes). SS is the default segment register for the stack data accesses and operations. ES is the default segment register for the destination data storage. All the segments available (defined in a particular program) can be read or written as data segments by newly defining the data segment as required. There is no physical difference in the memory structure or no physical separation between the segment areas. They may or may not overlap with each other. Chapter 3 on 'Assembly Language Programming' explains the coding procedure of the instructions with suitable examples.

## 2.2 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 categorised 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 or the 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 and 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 2.1**

```
MOV AX, 0005H
MOV BL, 06H
```

In the above examples 0005H and 06H are 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) or an IO address is directly specified in the instruction as a part of it.

**Example 2.2**

```
MOV AX, [5000H]
IN 80H
```

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. In the second instruction 80H is IO address.

**3. Register** In the 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 2.3**

```
MOV BX, AX.
ADC AL, BL
```

The operands in these instructions are provided in registers BX, AX and AL, BL respectively.

**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 register. 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 2.4**

```
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 is the default segment for index registers SI and DI. In case of string instructions DS and ES are default segments for SI and DI respectively. This mode is a special case of the above discussed register indirect addressing mode.

**Example 2.5**

```
MOV AX, [SI]
MOV CX, [DI]
```

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]. The content of address 10H*DS+[SI] will be transferred into register CX.

**Example 2.6**

```
MOV AX, 50H[BX]
MOV 10H[SI], DX
```

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

**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 2.7**

```
MOV AX, [BX] [SI]
MOV [BX] [DI], AX
```

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 or 16-bit displacement with the sum of contents of any one of the base registers (BX or BP) and any one of the index registers, in a default segment.

**Example 2.8**

```
MOV AX,50H [BX][SI]
ADD 50H [BX] [SI], BP
```

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 10H*DS+[BX]+[SI]+50H. The second instruction adds content of B with memory location of which offset is given by adding 50H of content of BX and SI. The result is stered in the memory location.

For the control transfer instructions, the addressing modes depend upon whether the destination location is within the same segment or in 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. intersegment and intrasegment 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 intersegment mode. If the destination location lies in the same segment, the mode is called intrasegment mode.*

Figure 2.1 shows the modes for control transfer instructions.

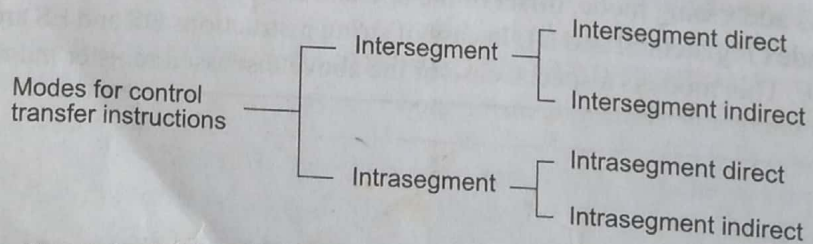



**Fig. 2.1** *Addressing Modes for Control Transfer Instructions*

**9. Intrasegment 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. $-128 < d < +127$), we term it as *short jump* and if it is of 16 bits (i.e. $-32768 < d < +32767$), it is termed as *long jump*.

**Example 2.9**

```
JMP SHORT LABEL; LABEL lies within -128 TO +127 from the current IP
content.
```

Thus SHORT LABEL is 8-bit signed displacement.

A 16-bit target address of a label indicates that it lies within – 32768 to + 32767. But a problem arises when one requires a forward jump at a relative address greater than 32767 or backward jump at relative address – 32768; in the same segment. Suppose current contents of IP are 5000H then a forward jump may be allowed at all the displacement DISP so that IP + DISP = FFFFH or DISP = FFFF – 5000 = AFFFH. Thus forward jumps may be allowed for all 16-bit displacement values from 0000H to AFFFH. If displacement exceeds AFFFH i.e. from B000H to FFFFH, then all such jumps will be treated as backward jumps. All such jumps are called NEAR PTR jumps and coded as below.

```
JMP NEAR PTR LABEL
```

**10. Intrasegment 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.

**Example 2.10**

```
JMP [BX]; Jump to effective address stored in BX.
                    JMP [ BX + 5000H ]
```

**11. Intersegment Direct** 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.

**Example 2.11**

```
JPM 5000H : 2000H;
Jump to effective address 2000H in segment 5000H.
```

**12. Intersegment Indirect** 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.

**Example 2.12**

```
JMP [2000H];
```

Jump to an address in the other segment specified at effective address 2000H in DS, that points to the memory block as said above.

**Forming the Effective Addresses** The following examples explain forming of the effective addresses in the different modes.

**Example 2.13**

The contents of different registers are given below. Form effective addresses for different addressing modes.

Offset (displacement) = 5000H

[AX]-1000H, [BX]-2000H, [SI]-3000H, [DI]-4000H, [BP]-5000H,
[SP]-6000H, [CS]-0000H, [DS]-1000H, [SS]-2000H, [IP]-7000H.

Shifting a number four times is equivalent to multiplying it by $16_D$ or $10_H$.

(i) Direct addressing mode

```
                        MOV AX, [5000H]
DS:OFFSET ⇔  1000H: 5000H
   10H* DS ⇔   10000
     Offset ⇔ + 5000
              ---------
              15000H - Effective address
```

(ii) Register indirect

```
                        MOV AX, [BX]
   DS:BX ⇔  1000H:2000H
 10H*DS ⇔    10000
    [BX] ⇔  + 2000
            ----------
            12000H - Effective address
```

(iii) Register relative

```
                        MOV AX, 5000 [BX]
DS: [5000 + BX]
 10H*DS ⇔    10000
   Offset ⇔ + 5000
    [BX] ⇔  + 2000
            ----------
            17000H - Effective address
```

(iv) Based indexed

```
                        MOV AX, [BX] [SI]
DS:[BX + SI]
 10H*DS ⇔    10000
    [BX] ⇔  + 2000
    [SI] ⇔  + 3000
            ----------
            15000H - Effective address
```

(v) Relative based indexed

```
                        MOV AX, 5000 [BX] [SI]
DS: [BX + SI + 5000]
 10H*DS ⇔    10000
    [BX] ⇔  + 2000
    [SI] ⇔  + 3000
   Offset ⇔ + 5000
            ----------
            1A000 - effective address
```

Below, we present examples of address formation in control transfer instructions.

**Example 2.14**

Suppose our main program resides in the code segment where CS = 1000H. The main program calls a subroutine which resides in the same code segment. The base register contains offset of the subroutine, i.e. BX = 0050H. Since the offset is specified indirectly, as the content of BX, this is indirect addressing. The instruction CALL [BX] calls the subroutine located at an address 10H*CS + [BX] = 10050H, i.e. in the same code segment. Since the control goes to the subroutine which resides in the same segment, this is an example of intrasegment indirect addressing mode.

**Example 2.15**

Let us now assume that the subroutine resides in another code segment, where CS = 2000H. Now CALL 2000H:0050H is an example of intersegment direct addressing mode, since the control now goes to different segment and the address is directly specified in the instruction. In this case, the address of the subroutine is 20050H.

## 2.3 INSTRUCTION SET OF 8086/8088

The 8086/8088 instructions are categorised into the following main types. This section explains the function of each of the instructions with suitable examples wherever necessary.

(i) **Data Copy/Transfer Instructions** These types of instructions are used to transfer data from source operand to destination operand. All the store, move, load, exchange, input and output instructions belong to this category.

(ii) **Arithmetic and Logical Instructions** All the instructions performing arithmetic, logical, increment, decrement, compare and scan instructions belong to this category.

(iii) **Branch Instructions** These instructions transfer control of execution to the specified address. All the call, jump, interrupt and return instructions belong to this class.

(iv) **Loop Instructions** If these instructions have REP prefix with CX used as count register, they can be used to implement unconditional and conditional loops. The LOOP, LOOPNZ and LOOPZ instructions belong to this category. These are useful to implement different loop structures.

(v) **Machine Control Instructions** These instructions control the machine status. NOP, HLT, WAIT and LOCK instructions belong to this class.

(vi) **Flag Manipulation Instructions** All the instructions which directly affect the flag register, come under this group of instructions. Instructions like CLD,STD,CLI,STI, etc. belong to this category of instructions.

(vii) **Shift and Rotate Instructions** These instructions involve the bitwise shifting or rotation in either direction with or without a count in CX.

(viii) **String Instructions** These instructions involve various string manipulation operations like load, move, scan, compare, store, etc. These instructions are only to be operated upon the strings.

### 2.3.1 Data Copy/Transfer Instructions

**MOV: Move** This data transfer instruction transfers data from one register/memory location to another register/memory location. The source may be any one of the segment registers or other general or special purpose registers or a memory location and, another register or memory location may act as destination.

However, in case of immediate addressing mode, a segment register cannot be a destination register. In other words, direct loading of the segment registers with immediate data is not permitted. To load the segment registers with immediate data, one will have to load any general purpose register with the data and then it will have to be moved to that particular segment register. The following example instructions explain the fact.

**Example 2.16**

Load DS with 5000H.

1. MOV DS, 5000H; Not permitted (invalid)
   Thus to transfer an immediate data into the segment register, the correct procedure is given below.
2. MOV AX, 5000H
   MOV DS, AX
   It may be noted here that both the source and destination operands cannot be memory locations (except for string instructions). Other MOV instruction examples are given below with the corresponding addressing modes.
3. MOV AX, 5000H; Immediate
4. MOV AX, BX; Register
5. MOV AX, [SI]; Indirect
6. MOV AX, [2000H]; Direct
7. MOV AX, 50H[BX]; Based relative, 50H Displacement

**PUSH: Push to Stack** This instruction pushes the contents of the specified register/memory location on to the stack. The stack pointer is decremented by 2, after each execution of the instruction. The actual current stack-top is always occupied by the previously pushed data. Hence, the push operation decrements SP by two and then stores the two byte contents of the operand onto the stack. The higher byte is pushed first and then the lower byte. Thus out of the two decremented stack addresses the higher byte occupies the higher address and the lower byte occupies the lower address.

The actual operation takes place as given below SS : SP points to the stack top of 8086 system as shown in Fig. 2.2 and AH, AL contains data to be pushed.

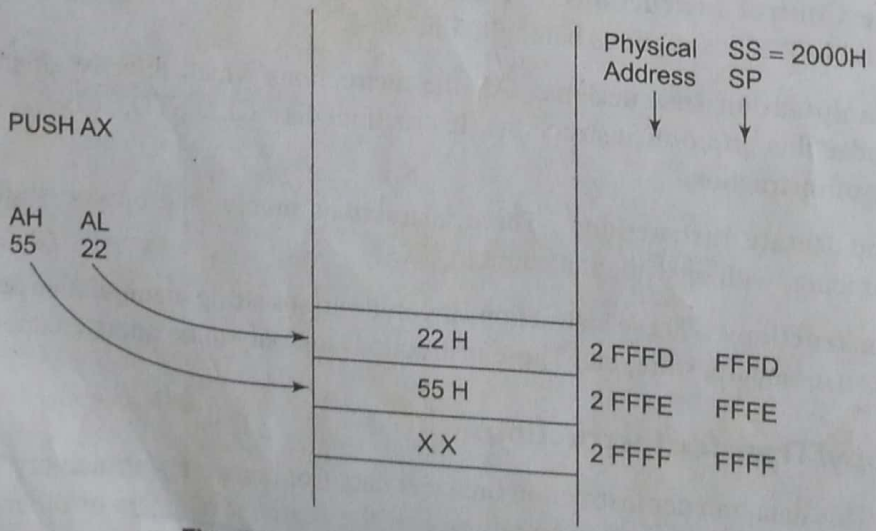



Fig. 2.2 Pushing Data to Stack Memory

**Example 2.17**

```
1. PUSH AX
2. PUSH DS
3. PUSH [5000H]; Content of location 5000H and 5001H in DS are pushed onto the stack
```

**POP: Pop from Stack** This instruction when executed, loads the specified register/memory location with the contents of the memory location of which the address is formed using the current stack segment and stack pointer as usual. The stack pointer is incremented by 2. The POP instruction serves exactly opposite to the PUSH instruction.

16-bit contents of current stack top are poped into the specified operand as follows.
The sequence of operation is as below.

1. Contents of stack top memory location is stored in AL and SP is incremented by one
2. Further contents of memory location pointed to by SP are copied to AH and SP is again incremented by 1

Effectively SP is incremented by 2 and points to next stack top.
The examples of these instructions are shown as follows:

**Example 2.18**

```
1.POP     AX
2.POP     DS
3.POP     [5000H]
```

**XCHG: Exchange** This instruction exchanges the contents of the specified source and destination operands, which may be registers or one of them may be a memory location. However, exchange of contents of two memory locations is not permitted. Immediate data is also not allowed in these instructions. The examples are as follows:

**Example 2.19**

```
1. XCHG  [5000H], AX     ; This instruction exchanges data between AX and a
                         ; memory location [5000H] in the data segment.
2. XCHG  BX, AX          ; This instruction exchanges data between AX and BX.
```

**IN: Input the Port** This instruction is used for reading an input port. The address of the input port may be specified in the instruction directly or indirectly. AL and AX are the allowed destinations for 8 and 16-bit input operations. DX is the only register (implicit) which is allowed to carry the port address. If the port address is of 16 bits it must be in DX. The examples are given as shown:

**Example 2.20**

```
1. IN AL,03H             ; This instruction reads data from an 8-bit port whose address
                         ; is 03H and stores it in AL.
2. IN AX, DX             ; This instruction reads data from a 16-bit port whose
                         ; address is in DX (implicit) and stores it in AX.
3. MOV DX,0800H          ; The 16-bit address is taken in DX.
   IN AX, DX             ; Read the content of the port in AX.
```

**OUT: Output to the Port** This instruction is used for writing to an output port. The address of the output port may be specified in the instruction directly or implicitly in DX. Contents of AX or AL are transferred to a directly or indirectly addressed port after execution of this instruction. The data to an odd addressed port is transferred on $D_8$-$D_{15}$ while that to an even addressed port is transferred on $D_0$-$D_7$. The registers AL and AX are the allowed source operands for 8-bit and 16-bit operations respectively. If the port address is of 16 bits it must be in DX. The examples are given as shown:

**Example 2.21**

```
1. OUT   03H,AL          ; This sends data available in AL to a port whose
                         ; address is 03H.
2. OUT   DX, AX          ; This sends data available in AX to a port whose
                         ; address is specified implicitly in DX.
3. MOV DX, 0300H         ; The 16-bit port address is taken in DX.
   OUT DX, AX            ; Write the content of AX to a port of which address is in DX.
```

**XLAT: Translate** The translate instruction is used for finding out the codes in case of code conversion problems, using look up table technique. We will explain this instruction with the aid of the following example.

Suppose, a hexadecimal key pad having 16 keys from 0 to F is interfaced with 8086 using 8255. Whenever a key is pressed, the code of that key (0 to F) is returned in AL. For displaying the number corresponding to the pressed key on the 7-segment display device, it is required that the 7-segment code corresponding to the key pressed is found out and sent to the display port. This translation from the code of the key pressed to the corresponding 7-segment code is performed using XLAT instruction.

For this purpose, one is required to prepare a look up table of codes, starting from an offset say 2000H, and store the 7-segment codes for 0 to F at the locations 2000H to 200FH sequentially. For executing the

**Example 2.22**

```
MOV AX, SEG TABLE    ; Address of the segment containing look-up-table
MOV DS,AX            ; is transferred in DS
MOV AL, CODE         ; Code of the pressed key is transferred in AL
MOV BX, OFFSET TABLE; Offset of the code look-up-table in BX
XLAT                 ; Find the equivalent code and store in AL
```

| *Mnemonics & Description* | *Instruction Code* | | | |
|---|---|---|---|---|
| **Data Transfer** | | | | |
| MOV = Move | 76543210 | 76543210 | 76543210 | 76543210 |
| Register/Memory to/from Register | 100010 dw | mod reg r/m | | |
| Immediate to Register/Memory | 1100011 w | mod 000 r/m | data | data if w = 1 |
| Immediate to Register | 1011 w reg | data | data if w = 1 | |
| Memory to Accumulator | 1010000 w | addr-low | addr-high | |
| Accumulator to Memory | 1010001 w | addr-low | addr-high | |
| Register/Memory to Segment Register | 10001110 | mod 0 reg r/m | | |
| Segment Register to Register/Memory | 10001100 | mod 0 reg r/m | | |
| **PUSH = Push:** | | | | |
| Register/Memory | 11111111 | mod 110 r/m | | |
| Register | 01010 reg | | | |
| Segment Register | 000 reg 110 | | | |
| **POP = Pop:** | | | | |
| Register/Memory | 10001111 | mod 000 r/m | | |
| Register | 01011 reg | | | |
| Segment Register | 000 reg 111 | | | |
| **XCHG = Exchange** | | | | |
| Register/Memory with Register | 1000011 w | mod reg r/m | | |
| Register with Accumulator | 10010 reg | | | |
| **IN = Input from:** | | | | |
| Fixed Port | 1110010 w | port | | |
| Variable Port | 1110110 w | | | |
| **OUT = Output to** | | | | |
| Fixed Port | 1110011 w | port | | |
| Variable Port | 1110111 w | | | |
| **XLAT** = Translate Byte to AL | 11010111 | | | |
| **LEA** = Load EA to Register | 10001101 | mod reg r/m | | |
| **LDS** = Load Pointer to DS | 11000101 | mod reg r/m | | |
| **LES** = Load Pointer to ES | 11000100 | mod reg r/m | | |
| **LAHF** = Load AH with Flags | 10011111 | | | |
| **SAHF** = Store AH into Flags | 10011110 | | | |
| **PUSHF** = Push Flags | 10011100 | | | |
| **POPF** = Pop Flags | 10011101 | | | |
| **ARITHMETIC** | **76543210** | **76543210** | **76543210** | **76543210** |
| **ADD = Add:** | | | | |
| Reg/Memory with Register to Either | 000000 dw | mod reg r/m | | |
| Immediate to Register/Memory | 100000 sw | mod 000 r/m | data | data if s w = 01 |

| Mnemonics & Description | Instruction Code | | | |
|---|---|---|---|---|
| Immediate to Accumulator | 0000010 w | data | data if w = 1 | |
| **ADC = Add with Carry:** | | | | |
| Reg/Memory with Register to Either | 000100 dw | mod reg r/m | | |
| Immediate to Register/Memory | 100000 sw | mod 010 r/m | data | data if s w = 01 |
| Immediate to Accumulator | 0001010 w | data | data if w = 1 | |
| **INC = Increment:** | | | | |
| Register/Memory | 1111111 w | mod 000 r/m | | |
| Register | 01000 reg | | | |
| **AAA** = ASCII Adjust for Addition | 00110111 | | | |
| **DAA** = Decimal Adjust for Addition | 00100111 | | | |
| **SUB** = Subtract | | | | |
| Reg/Memory and Register to Either | 001010 dw | mod reg r/m | | |
| Immediate from Register/Memory | 100000 sw | mod 101 r/m | data | data if s w = 01 |
| Immediate from Accumulator | 0010110 w | data | data if w = 1 | |
| **SBB = Subtract with Borrow** | | | | |
| Reg/Memory and Register to Either | 000110 dw | mod reg r/m | | |
| Immediate from Register/Memory | 100000 sw | mod 011 r/m | data | data if s w = 01 |
| Immediate from accumulator | 0001110 w | data | data if w = 1 | |
| **DEC = Decrement:** | | | | |
| Register/Memory | 1111111 w | mod 001 r/m | | |
| Register | 01001 reg | | | |
| **NEG** = Change sign | 1111011 w | mod 011 r/m | | |
| **CMP = Compare:** | | | | |
| Register/Memory and Register | 001110 dw | mod reg r/m | | |
| Immediate with Register/Memory | 100000 sw | mod 111 r/m | data | data if s w = 01 |
| Immediate with Accumulator | 0011110 w | data | data if w = 1 | |
| **AAS** = ASCII Adjust for Subtract | 00111111 | | | |
| **DAS** = Decimal Adjust for Subtract | 00101111 | | | |
| **MUL** = Multiply (Unsigned) | 1111011 w | mod 100 r/m | | |
| **IMUL** = Integer Multiply (Signed) | 1111011 w | mod 101 r/m | | |
| **AAM** = ASCII Adjust Multiply | 11010100 | 00001010 | | |
| **DIV** = Divide (Unsigned) | 1111011 w | mod 110 r/m | | |
| **IDIV** = Integer Divide (Signed) | 1111011 w | mod 111 r/m | | |
| **AAD** = ASCII Adjust for Divide | 11010101 | 00001010 | | |
| **CBW** = Convert Byte to Word | 10011000 | | | |
| **CWD** = Convert Word to Double Word | 10011001 | | | |
| **LOGICAL** | 76543210 | 76543210 | 76543210 | 76543210 |
| **NOT** = Invert | 1111011 w | mod 010 r/m | | |
| **SHL/SAL** = Shift Logical/Arithmetic Left | 110100 v w | mod 100 r/m | | |
| **SHR** = Shift Logical Right | 110100 v w | mod 101 r/m | | |
| **SAR** = Shift Arithemtic Right | 110100 v w | mod 111 r/m | | |
| **ROL** = Rotate Left | 110100 v w | mod 000 r/m | | |
| **ROR** = Rotate Right | 110100 v w | mod 001 r/m | | |
| **RCL** = Rotate Through Carry Flag Left | 110100 v w | mod 010 r/m | | |
| **RCR** = Rotate Through Carry Right | 110100 v w | mod 011 r/m | | |
| **AND = And:** | | | | |
| Reg/Memory and Register to Either | 001000 dw | mod reg r/m | | |
| Immediate to Register/Memory | 1000000 w | mod 100 r/m | data | data if w = 1 |
| Immediate to Accumulator | 0010010 w | data | data if w = 1 | |
| **TEST = And Function to Flags, No Result:** | | | | |
| Register/Memory and Register | 1000010 w | mod reg r/m | | |
| Immediate Data and Register/Memory | 1111011 w | mod 000 r/m | data | data if w = 1 |
| Immediate Data and Accumulator | 1010100 w | data | data if w = 1 | |
| **OR = Or:** | | | | |
| Reg/Memory and Register to Either | 000010 dw | mod reg r/m | | |
| Immediate to Register/Memory | 1000000 w | mod 001 r/m | data | data if w = 1 |
| Immediate to Accumulator | 0000110 w | data | data if w = 1 | |

| Mnemonics & Description | Instruction Code | | | |
|---|---|---|---|---|
| **XOR = Exclusive or:** | | | | |
| Reg/Memory and Register to Either | 001100 dw | mod reg r/m | data | data if w = 1 |
| Immediate to Register/Memory | 1000000 w | mod 110 r/m | data if w = 1 | |
| Immediate to Accumulator | 0011010 w | data | | |
| **STRING MANIPULATIONS** | | | | |
| **REP** = Repeat | 1111001 z | | | |
| **MOVS** = Move Byte/Word | 1010010 w | | | |
| **CMPS** = Compare Byte/Word | 1010011 w | | | |
| **SCAS** = Scan Byte/Word | 1010111 w | | | |
| **LODS** = Load byte/Wd to AL/AX | 1010110 w | | | |
| **STOS** = Stor Byte/Wd from AL/A | 1010101 w | | | |
| **CONTROL TRANSFER** | | | | |
| **CALL = Call:** | | | | |
| Direct Within Segment | 11101000 | disp-low | disp-high | |
| Indirect Within Segment | 11111111 | mod 010 r/m | | |
| Direct Intersegment | 10011010 | offset-low | offset-high | |
| | | seg-low | seg-high | |
| | 76543210 | 76543210 | 76543210 | |
| Indirect Intersegment | 11111111 | mod 011 r/m | | |
| **JMP = Unconditional Jump:** | | | | |
| Direct Within Segment | 11101001 | disp-low | disp-high | |
| Direct Within Segment-short | 11101011 | disp | | |
| Indirect Within Segment | 11111111 | mod 100 r/m | | |
| Direct Intersegment | 11101010 | offset-low | offset-high | |
| | | seg-low | seg-high | |
| Indirect Intersegment | 11111111 | mod 101 r/m | | |
| **RET = Return from CALL:** | | | | |
| Within Segment | 11000011 | | | |
| Within Seg Adding Immediate to SP | 11000010 | data-low | data-high | |
| Intersegment | 11001011 | | | |
| Intersegment Adding Immediate to SP | 11001010 | data-low | data-high | |
| **JE/JZ** = Jump on Equal/Zero | 01110100 | disp | | |
| **JL/JNGE** = Jump on Less/Not Greater or Equal | 011111100 | disp | | |
| **JLE/JNG** = Jump on Less or Equal/Not Greater | 01111110 | disp | | |
| **JB/JNAE** = Jump on Below/Not Above or Equal | 01110010 | disp | | |
| **JBE/JNA** = Jump on Below or Equal/Not Above | 01110110 | disp | | |
| **JP/JPE** = Jump on Parity/Parity Even | 01111010 | disp | | |
| **JO** = Jump on Overflow | 01110000 | disp | | |
| **JS** = Jump on Sign | 01111000 | disp | | |
| **JNE/JNZ** = Jump on Not Equal/Not Zero | 01110101 | disp | | |
| **JNL/JGE** = Jump on Not Less/Greater or Equal | 01111101 | disp | | |
| **JNLE/JG** = Jump on Not Less or Equal/Greater | 01111111 | disp | | |
| **JNB/JAE** = Jump on Not Below/Above or Equal | 01110011 | disp | | |
| **JNBE/JA** = Jump on Not Below or Equal/Above | 01110111 | disp | | |
| **JNP/JPO** = Jump on Not Par/Par Odd | 01111011 | disp | | |
| **JNO** = Jump on Not Overflow | 01110001 | disp | | |
| **JNS** = Jump on Not Sign | 01111001 | disp | | |
| **LOOP** = Loop CX Times | 11100010 | disp | | |
| **LOOPZ/LOOPE** = Loop While Zero/ | 11100001 | disp | | |

| Mnemonics & Description | | Instruction Code |
|---|---|---|
| Equal | | |
| **LOOPNZ/LOOPNE** = Loop While Not Zero/Equal | 11100000 | disp |
| **JCXZ** = Jump on CX Zero | 11100011 | disp |
| **INT** = **Interrupt** | | |
| **Type Specified** | 11001101 | type |
| Type 3 | 11001100 | |
| **INTO** = Interrupt on Overflow | 11001110 | |
| **IRET** = Interrupt Return | 11001111 | |
| | 76543210 | 76543210 |
| **PROCESSOR CONTROL** | | |
| **CLC** = Clear Carry | 11111000 | |
| **CMC** = Complement Carry | 11110101 | |
| **STC** = Set Carry | 11111001 | |
| **CLD** = Clear Direction | 11111100 | |
| **STD** = Set Direction | 11111101 | |
| **CLI** = Clear Interrupt | 11111010 | |
| **STI** = Set Interrupt | 11111011 | |
| **HLT** = Halt | 11110100 | |
| **WAIT** = Wait | 10011011 | |
| **ESC** = Escape (to External Device) | 11011xxx | mod xxx r/m |
| **LOCK** = Bus Lock Prefix | 11110000 | |

*The v, w, d, s and z bits and the mod, reg, r/m fields are discussed in the addressing modes' section.

**Fig. 2.4** *8086/8088 Instruction Set Summary*

**LEA: Load Effective Address** The load effective address instruction loads the effective address formed by destination operand into the specified source register. This instruction is more useful for assembly language rather than for machine language. The examples are given below.

**Example 2.23**

```
LEA BX,ADR      ; Effective address of Label ADR i.e. offset of ADR will be
                  transferred to Reg ; BX.
LEA SI, ADR[Bx]; offset of Label ADR will be added to content of Bx to form effective
                ; address and it will be loaded in SI
```

**LDS/LES: Load Pointer to DS/ES** This instruction loads the DS or ES register and the specified destination register in the instruction with the content of memory location specified as source in the instruction. The example in Fig. 2.5 explains the operation.

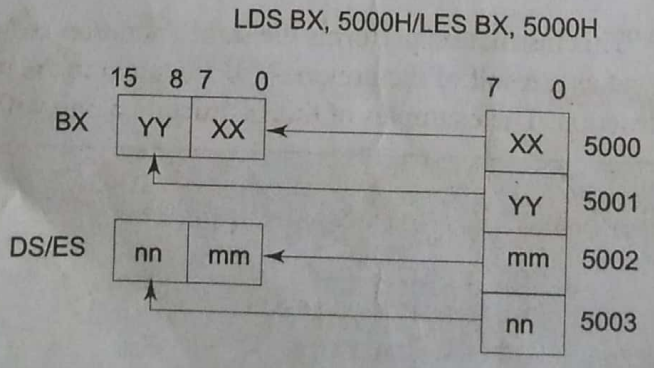



**Fig. 2.5** *LDS/LES Instruction Execution*

**LAHF : Load AH from Lower Byte of Flag** This instruction loads the AH register with the lower byte of the flag register. This command may be used to observe the status of all the condition code flags (except over flow) at a time.

**SAHF: Store AH to Lower Byte of Flag Register** This instruction sets or resets the condition code flags (except overflow) in the lower byte of the flag register depending upon the corresponding bit positions in AH. If a bit in AH is 1, the flag corresponding to the bit position is set, else it is reset.

**PUSHF: Push Flags to Stack** The push flag instruction pushes the flag register on to the stack; first the upper byte and then the lower byte is pushed on to it. The SP is decremented by 2, for each push operation. The general operation of this instruction is similar to the PUSH operation.

**POPF: Pop Flags from Stack** The pop flags instruction loads the flag register completely (both bytes) from the word contents of the memory location currently addressed by SP and SS. The SP is incremented by 2 for each pop operation.

Figure 2.4 shows the data sheet for the hand coding of all the 8086 instructions. The MOD and R/M fields are to be decided as already described in this chapter. This type of instructions do not affect any flags.

### 2.3.2 Arithmetic Instructions

These instructions perform the arithmetic operations, like addition, subtraction, multiplication and division along with the respective ASCII and decimal adjust instructions. The increment and decrement operations also belong to this type of instructions. The 8086/8088 instructions falling under this category are discussed below in significant details. The arithmetic instructions affect all the condition code flags. The operands are either the registers or memory locations or immediate data depending upon the addressing mode.

**ADD: Add** This instruction adds an immediate data or contents of a memory location specified in the instruction or a register (source) to the contents of another register (destination) or memory location. The result is in the destination operand. However, both the source and destination operands cannot be memory operands. That means memory to memory addition is not possible. Also the contents of the segment registers cannot be added using this instruction. All the condition code flags are affected, depending upon the result. The examples of this instruction are given along with the corresponding modes.

**Example 2.24**

```
1.ADD AX, 0100H        Immediate
2.ADD AX, BX           Register
3.ADD AX, [SI]         Register indirect
4.ADD AX, [5000H]      Direct
5.ADD [5000H], 0100H   Immediate
6.ADD 0100H            Destination AX (implicit)
```

**ADC: Add with Carry** This instruction performs the same operation as ADD instruction, but adds the carry flag bit (which may be set as a result of the previous calculations) to the result. All the condition code flags are affected by this instruction. The examples of this instruction along with the modes are as follows:

**Example 2.25**

```
1.ADC 0100H            Immediate (AX implicit)
2.ADC AX, BX           Register
3.ADC AX, [SI]         Register indirect
4.ADC AX, [5000H]      Direct
5.ADC [5000H], 0100H   Immediate
```

**INC: Increment** This instruction increases the contents of the specified register or memory location by 1. All the condition code flags are affected except the carry flag CF. This instruction adds 1 to the contents of the operand. Immediate data cannot be operand of this instruction. The examples of this instruction are as follows:

**Example 2.26**

```
1. INC    AX        Register
2. INC    [BX]      Register indirect
3. INC    .[5000H]  Direct
```

**DEC: Decrement** The decrement instruction subtracts 1 from the contents of the specified register or memory location. All the condition code flags, except the carry flag, are affected depending upon the result. Immediate data cannot be operand of the instruction. The examples of this instruction are as follows:

**Example 2.27**

```
1.DEC     AX        Register
2.DEC     [5000H]   Direct
```

**SUB: Subtract** The subtract instruction subtracts the source operand from the destination operand and the result is left in the destination operand. Source operand may be a register, memory location or immediate data and the destination operand may be a register or a memory location, but source and destination operands both must not be memory operands. Destination operand can not be an immediate data. All the condition code flags are affected by this instruction. The examples of this instruction along with the addressing modes are as follows:

**Example 2.28**

```
1.SUB     AX, 0100H       Immediate [destination AX]
2.SUB     AX, BX          Register
3.SUB     AX, [5000H]     Direct
4.SUB     [5000H], 0100   Immediate
```

**SBB: Subtract with Borrow** The subtract with borrow instruction subtracts the source operand and the borrow flag (CF) which may reflect the result of the previous calculations, from the destination operand. Subtraction with borrow, here means subtracting 1 from the subtraction obtained by SUB, if carry (borrow) flag is set.

The result is stored in the destination operand. All the flags are affected (Condition code) by this instruction. The examples of this instruction are as follows:

**Example 2.29**

```
1.SBB     AX, 0100H       Immediate [destination AX]
2.SBB     AX, BX          Register
3.SBB     AX, [5000H]     Direct
4.SBB     [5000H], 0100   Immediate
```

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

For comparison, it subtracts the source operand from the destination operand but does not store the result anywhere. The flags are affected depending upon the result of the subtraction. If both of the operands are equal, zero flag is set. If the source operand is greater than the destination operand, carry flag is set or else, carry flag is reset. The examples of this instruction are as follows:

**Example 2.30**

```
1.CMP BX, 0100H          Immediate
2.CMP AX, 0100H          Immediate
3.CMP [5000H], 0100H     Direct
4.CMP BX, [SI]           Register indirect
5.CMP BX, CX             Register
```

**AAA: ASCII Adjust After Addition** The AAA instruction is executed after an ADD instruction that adds two ASCII coded operands to give a byte of result in AL. The AAA instruction converts the resulting contents of AL to unpacked decimal digits. After the addition, the AAA instruction examines the lower 4 bits of AL to check whether it contains a valid BCD number in the range 0 to 9. If it is between 0 to 9 and AF is zero, AAA sets the 4 high order bits of AL to 0. The AH must be cleared before addition. If the lower digit of AL is between 0 to 9 and AF is set, 06 is added to AL. The upper 4 bits of AL are cleared and AH is incremented by one. If the value in the lower nibble of AL is greater than 9 then the AL is incremented by 06, AH is incremented by 1, the AF and CF flags are set to 1, and the higher 4 bits of AL are cleared to 0. The remaining flags are unaffected. The AH is modified as sum of previous contents (usually 00) and the carry from the adjustment, as shown in Fig. 2.6. This instruction does not give exact ASCII codes of the sum, but they can be obtained by adding 3030H to AX.

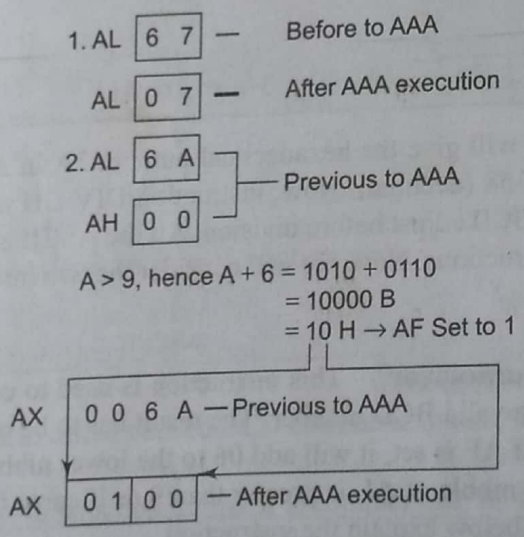



**Fig. 2.6** *ASCII Adjust after Addition Instruction*

**AAS: ASCII Adjust AL after Subtraction** AAS instruction corrects the result in AL register after subtracting two unpacked ASCII operands. The result is in unpacked decimal format. If the lower 4 bits of AL register are greater than 9 or if the AF flag is 1, the AL is decremented by 6 and AH register is decremented by 1, the CF and AF are set to 1. Otherwise, the CF and AF are set to 0, the result needs no correction. As a result, the upper nibble of AL is 00 and the lower nibble may be any number from 0 to 9. The procedure is similar to the AAA instruction except for the subtraction of 06 from AL. AH is modified as difference of the previous contents (usually zero) of AH and the borrow for adjustment.

**AAM : ASCII Adjust after Multiplication** This instruction, after execution, converts the product available in AL into unpacked BCD format. The AAM—ASCII Adjust After Multiplication—instruction follows a multiplication instruction that multiplies two unpacked BCD operands, i.e. higher nibbles of the multiplication operands should be O. The multiplication of such operands is carried out using MUl instruction. Obviously the result of multiplication is available in AX. The following AAM instruction replaces content of AH by tens of the decimal multiplication and AL by singles of the decimal multiplication.

**Example 2.31**

```
MOV AL, 04  ;  AL ← 04
MOV BL, 09  ;  BL ← 09
MUL BL      ;  AH-AL ← 24H (9 × 4)
AAM         ;  AH ← 03
            ;  AL ← 06
```

**AAD: ASCII Adjust before Division** Though the names of these two instructions (AAM and AAD) appear to be similar, there is a lot of difference between their functions. The AAD instruction converts two unpacked BCD digits in AH and AL to the equivalent binary number in AL. This adjustment must be made before dividing the two unpacked BCD digits in AX by an unpacked BCD byte. PF, SF, ZF are modified while AF, CF, OF are undefined, after the execution of the instruction AAD. The example explains the execution of the instruction. In the instruction sequence, this instruction appears before DIV instruction unlike AAM appears after MUL. Let AX contains 0508 unpacked BCD for 58 decimal, and DH contains 02H.

```
(ii) AL = 73      CL = 29
     ADD AL, CL   ; AL ← AL + CL
                  ; AL ← 73 + 29
                  ; AL ← 9C
     DAA          ; AL ← 02 and CF = 1
                    AL = 7 3
                      +
                  CL = 2 9
                    ------
                       9 C
                     + 6
                    ------
                       A 2
                     + 6 0
                    ------
                  CF = 1  0 2  in AL
```

The instruction DAA affects AF, CF, PF, and ZF flags. The OF is undefined.

**DAS: Decimal Adjust after Subtraction** This instruction converts the result of subtraction of two packed BCD numbers to a valid BCD number. The subtraction has to be in AL only. If the lower nibble of AL is greater than 9, this instruction will subtract 06 from lower nibble of AL. If the result of subtraction sets the carry flag or if upper nibble is greater than 9, it subtracts 60H from AL. This instruction modifies the AF, CF, SF, PF and ZF flags. The OF is undefined after DAS instruction. The examples are as follows:

**Example 2.34**

```
(i) AL = 75       BH = 46
    SUB AL, BH    ; AL ← 2 F = (AL) - (BH)
                  ; AF = 1
    DAS           ; AL ← 2 9 (as F > 9, F - 6 = 9)
(ii) AL = 38      CH = 6 1
    SUB AL, CH    ; AL ← D 7  CF = 1 (borrow)
    DAS           ; AL ← 7 7  (as D > 9, D - 6 = 7)
                  ; CF = 1 (borrow)
```

DAA and DAS instructions are also called packed BCD arithmetic instructions.

**NEG: Negate** The negate instruction forms 2's complement of the specified destination in the instruction. For obtaining 2's complement, it subtracts the contents of destination from zero. The result is stored back in the destination operand which may be a register or a memory location. If OF is set, it indicates that the operation could not be completed successfully. This instruction affects all the condition code flags.

**MUL: Unsigned Multiplication Byte or Word** This instruction multiplies an unsigned byte or word by the contents of AL. The unsigned byte or word may be in any one of the general purpose registers or memory locations. The most significant word of the result is stored in DX, while the least significant word of the result is stored in AX. All the flags are modified depending upon the result. The example instructions are as shown. Immediate operand is not allowed in this instruction. If the most significant byte or word of the result is '0' CF and OF both will be set.

**Example 2.35**

```
1. MUL BH                ; (AX) ← (AL) × (BH)
2. MUL CX                ; (DX) (AX) ← (AX) × (CX)
3. MUL WORD PTR [SI]     ; (DX) (AX) ← (AX) × ([SI])
```

**IMUL: Signed Multiplication** This instruction multiplies a signed byte in source operand by a signed byte in AL or a signed word in source operand by a signed word in AX. The source can be a general purpose register, memory operand, index register or base register, but it cannot be an immediate data. In case of 32-bit results, the higher order word (MSW) is stored in DX and the lower order word is stored in AX. The AF, PF, SF, and ZF flags are undefined after IMUL. If AH and DX contain parts of 16 and 32-bit result respectively, CF and OF both will be set. The AL and AX are the implicit operands in case of 8 bits and 16 bits multiplications respectively. The unused higher bits of the result are filled by sign bit and CF, AF are cleared. The example instructions are given as follows:

**Example 2.36**

```
1. IMUL    BH
2. IMUL    CX
3. IMUL    [SI]
```

**CBW: Convert Signed Byte to Word** This instruction converts a signed byte to a signed word. In other words, it copies the sign bit of a byte to be converted to all the bits in the higher byte of the result word. The byte to be converted must be in AL. The result will be in AX. It does not affect any flag.

**CWD: Convert Signed Word to Double Word** This instruction copies the sign bit of AX to all the bits of the DX register. This operation is to be done before signed division. It does not affect any flag.

**DIV: Unsigned Division** This instruction performs unsigned division. It divides an unsigned word or double word by a 16-bit or 8-bit operand. The dividend must be in AX for 16-bit operation and divisor may be specified using any one of the addressing modes except immediate. The result will be in AL (quotient) while AH will contain the remainder. If the result is too big to fit in AL, type 0 (divide by zero) and an interrupt is generated. In case of a double word dividend (32-bit), the higher word should be in DX and lower word should be in AX. The divisor may be specified as already explained. The quotient and the remainder, in this case, will be in AX and DX respectively. This instruction does not affect any flag.

**IDIV: Signed Division** This instruction performs the same operation as the DIV instruction, but with signed operands. The results are stored similarly as in case of DIV instruction in both cases of word and double word divisions. The results will also be signed numbers. The operands are also specified in the same way as DIV instruction. Divide by 0 interrupt is generated, if the result is too big to fit in AX (16-bit dividend operation) or AX and DX (32-bit dividend operation). All the flags are undefined after IDIV instruction.

### 2.3.3 Logical Instructions

These type of instructions are used for carrying out the bit by bit shift, rotate, or basic logical operations. All the condition code flags are affected depending upon the result. Basic logical operations available with 8086 instruction set are AND, OR, NOT, and XOR. The instruction for each of these operations are discussed as follows.

**AND: Logical AND** This instruction bit by bit ANDs the source operand that may be an immediate, a register or a memory location to the destination operand that may be a register or a memory location. The result is stored in the destination operand. At least one of the operands should be a register or a memory operand. Both the operands cannot be memory locations or immediate operands. An immediate operand cannot be a destination operand. The examples of this instruction are as follows:

**Example 2.37**

```
1.AND AX, 0008H
2.AND AX, BX
3.AND AX, [5000H]
4.AND [5000H], DX
```

If the content of AX is 3F0FH, the first example instruction will carry out the operation as given below. The result 3F9FH will be stored in the AX register.

```
  0 0 1 1      1 1 1 1      0 0 0 0      1 1 1 1      = 3F0F H [AX]
  ↓ ↓ ↓ ↓      ↓ ↓ ↓ ↓      ↓ ↓ ↓ ↓      ↓ ↓ ↓ ↓        AND
  0 0 0 0      0 0 0 0      0 0 0 0      1 0 0 0      = 0008 H
  ---------    ---------    ---------    ---------    ----------
  0 0 0 0      0 0 0 0      0 0 0 0      1 0 0 0      = 0008 H [AX]
```

The result 0008H will be in AX.

**OR: Logical OR** The OR instruction carries out the OR operation in the same way as described in case of the AND operation. The limitations on source and destination operands are also the same as in case of AND operation. The examples are as follows:

**Example 2.38**

```
1.OR AX, 0098H
2.OR AX, BX
3.OR AX, [5000H]
4.OR [5000H], 0008H
```

The contents of AX are say 3F0FH, then the first example instruction will be carried out as given below.

```
  0 0 1 1      1 1 1 1      0 0 0 0      1 1 1 1      = 3F0F H
  ↓ ↓ ↓ ↓      ↓ ↓ ↓ ↓      ↓ ↓ ↓ ↓      ↓ ↓ ↓ ↓        OR
  0 0 0 0      0 0 0 0      1 0 0 1      1 0 0 0      = 0098 H
  ---------    ---------    ---------    ---------    ----------
  0 0 1 1      1 1 1 1      1 0 0 1      1 1 1 1      = 3F9F H
```

Thus the result 3F9FH will be stored in the AX register.

**NOT: Logical Invert** The NOT instruction complements (inverts) the contents of an operand register or a memory location, bit by bit. The examples are as follows:

**Example 2.39**

```
NOT AX
NOT [5000H]
```

If the content of AX is 200FH, the first example instruction will be executed as shown.

```
  AX     = 0 0 1 0      0 0 0 0      0 0 0 0      1 1 1 1
invert     ↓ ↓ ↓ ↓      ↓ ↓ ↓ ↓      ↓ ↓ ↓ ↓      ↓ ↓ ↓ ↓
           1 1 0 1      1 1 1 1      1 1 1 1      0 0 0 0
```

| Result in AX = | D | F | F | 0 |
|---|---|---|---|---|

The result DFF0H will be stored in the destination register AX.

**XOR: Logical Exclusive OR** The XOR operation is again carried out in a similar way to the AND and OR operation. The constraints on the operands are also similar. The XOR operation gives a high output, when the 2 input bits are dissimilar. Otherwise, the output is zero. The example instructions are as follows:

**Example 2.40**

```
1.XOR   AX, 0098H
2.XOR   AX, BX
3.XOR   AX, [5000H]
```

If the content of AX is 3F0FH, then the first example instruction will be executed as explained. The result 3F97H will be stored in AX.

```
AX = 3F0FH =    0 0 1 1    1 1 1 1    0 0 0 0    1 1 1 1
      XOR       ↓ ↓ ↓ ↓    ↓ ↓ ↓ ↓    ↓ ↓ ↓ ↓    ↓ ↓ ↓ ↓
   0098H =      0 0 0 0    0 0 0 0    1 0 0 1    1 0 0 0
AX = Result =   0 0 1 1    1 1 1 1    1 0 0 1    0 1 1 1
        = 3F97H
```

**TEST: Logical Compare Instruction** The TEST instruction performs a bit by bit logical AND operation on the two operands. Each bit of the result is then set to 1, if the corresponding bits of both operands are 1, else the result bit is reset to 0. The result of this ANDing operation is not available for further use, but flags are affected. The affected flags are OF, CF, SF, ZF and PF. The operands may be registers, memory or immediate data. The examples of this instruction are as follows:

**Example 2.41**

```
1.TEST    AX, BX
2.TEST    [0500], 06H
3.TEST    [BX] [DI], CX
```

**SHL/SAL: Shift Logical/Arithmetic Left** These instructions shift the operand word or byte bit by bit to the left and insert zeros in the newly introduced least significant bits. In case of all the SHIFT and ROTATE instructions, the count is either 1 or specified by register CL. The operand may reside in a register or a memory location but cannot be an immediate data. All flags are affected depending upon the result. Figure 2.7 explains the execution of this instruction. It is to be noted here that the shift operation is through carry flag.


**Fig. 2.7** *Execution of SHL/SAL Instruction*

**SHR: Shift Logical Right** This instruction performs bit-wise right shifts on the operand word or byte that may reside in a register or a memory location, by the specified count in the instruction and inserts zeros in the shifted positions. The result is stored in the destination operand. Figure 2.8 explains execution of this instruction. This instruction shifts the operand through the carry flag.

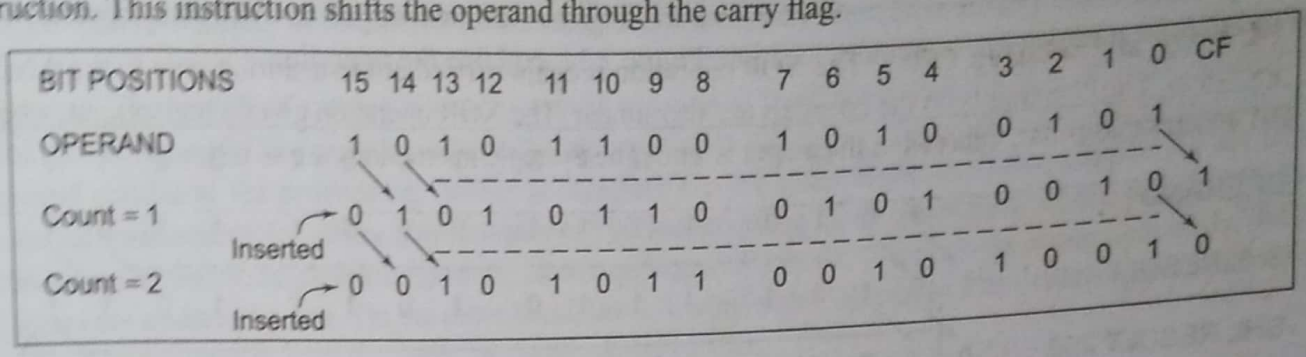



**Fig. 2.8** *Execution of SHR Instruction*

**SAR: Shift Arithmetic Right** This instruction performs right shifts on the operand word or byte, that may be a register or a memory location by the specified count in the instruction. It inserts the most significant bit of the operand in the newly inserted positions. The result is stored in the destination operand. Figure 2.9 explains execution of the instruction. All the condition code flags are affected. This shift operation shifts the operand through the carry flag.

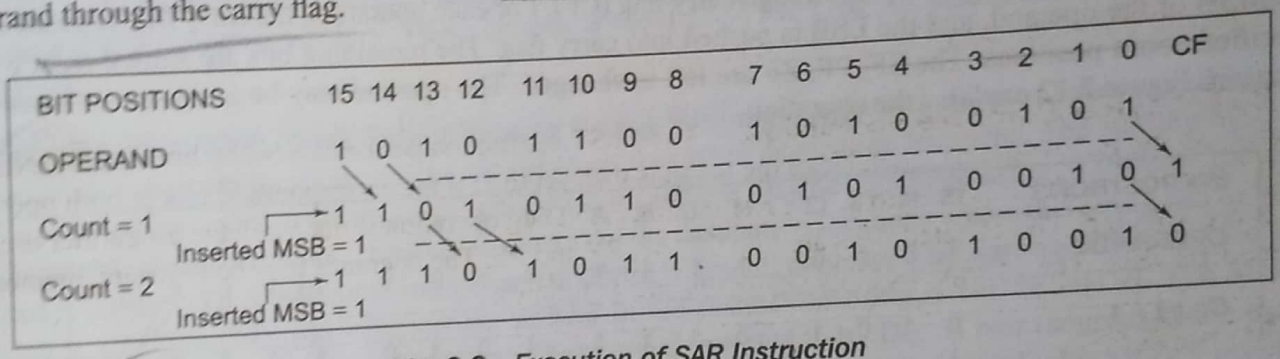
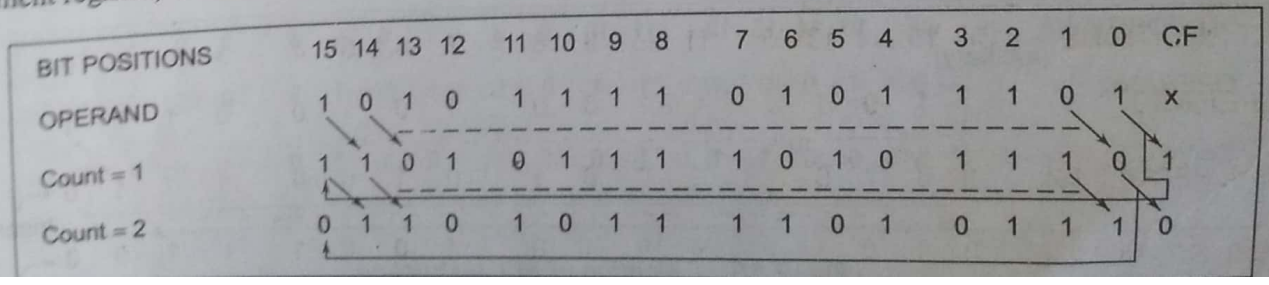



**Fig. 2.9** *Execution of SAR Instruction*

Immediate operand is not allowed in any of the shift instructions.

**ROR: Rotate Right without Carry** This instruction rotates the contents of the destination operand to the right (bit-wise) either by one or by the count specified in CL, excluding carry. The least significant bit is pushed into the carry flag and simultaneously it is transferred into the most significant bit position at each operation. The remaining bits are shifted right by the specified positions. The PF, SF, and ZF flags are left unchanged by the rotate operation. The operand may be a register or a memory location but it cannot be an immediate operand. Figure 2.10 explains the operation. The destination operand may be a register (except a segment register) or a memory location.



**ROL: Rotate Left without Carry** This instruction rotates the content of the destination operand to the left by the specified count (bit-wise) excluding carry. The most significant bit is pushed into the carry flag as well as the least significant bit position at each operation. The remaining bits are shifted left subsequently by the specified count positions. The PF, SF, and ZF flags are left unchanged in this rotate operation. The operand may be a register or a memory location. Figure 2.11 explains the operation.

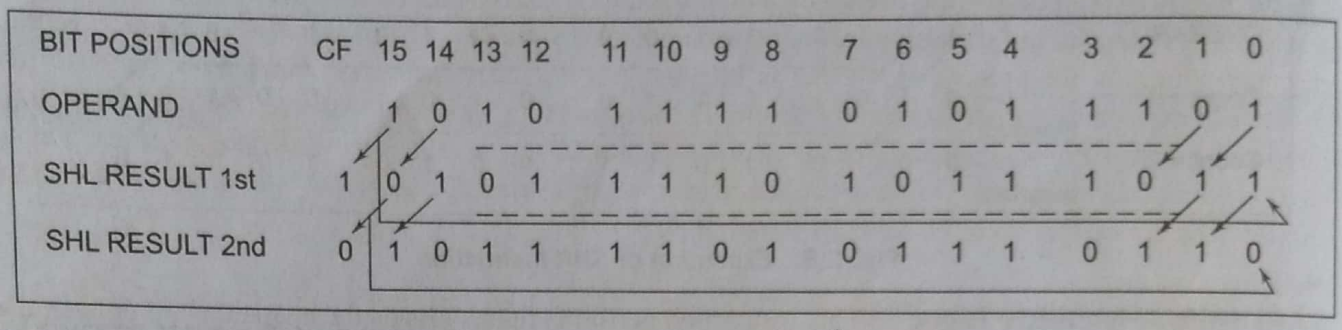



**Fig. 2.11** *Execution of ROL Instruction*

**RCR: Rotate Right through Carry** This instruction rotates the contents (bit-wise) of the destination operand right by the specified count through carry flag (CF). For each operation, the carry flag is pushed into the MSB of the operand, and the LSB is pushed into carry flag. The remaining bits are shifted right by the specified count positions. The SF, PF, ZF are left unchanged. The operand may be a register or a memory location. Figure 2.12 explains the operation.

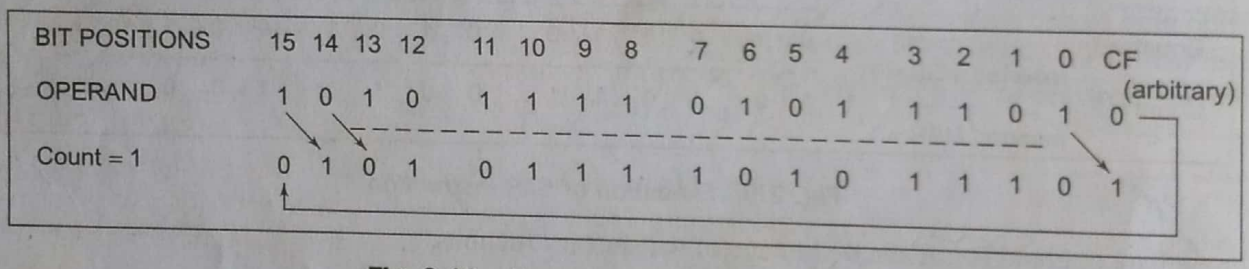



**Fig. 2.12** *Execution of RCR Instruction*

**RCL: Rotate Left through Carry** This instruction rotates (bit-wise) the contents of the destination operand left by the specified count through the carry flag (CF). For each operation, the carry flag is pushed into LSB and the MSB of the operand is pushed into carry flag. The remaining bits are shifted left by the specified positions. The SF, PF, ZF are left unchanged. The operand may be a register or a memory location. Figure 2.13 explains the operation.

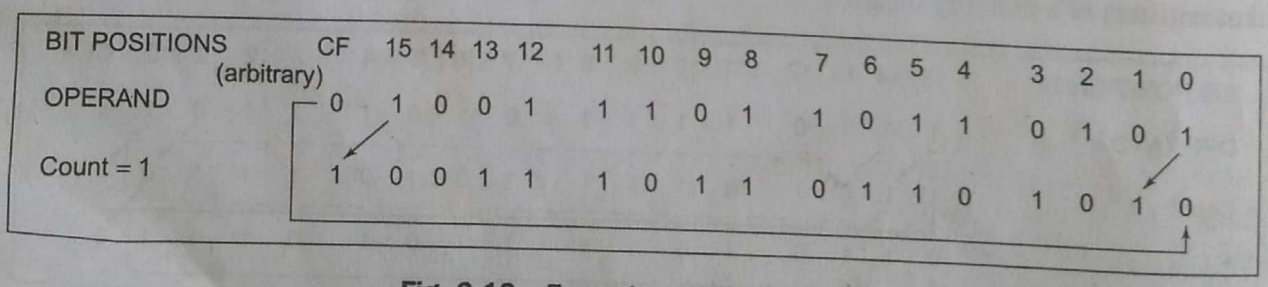



**Fig. 2.13** *Execution of RCL Instruction*

The count for rotation or shifting is either 1 or is specified using register CL, in case of all the shift and rotate instructions.

### 2.3.4 String Manipulation Instructions

A series of data bytes or words available in memory at consecutive locations, to be referred to collectively or individually, are called as *byte strings* or *word strings*. For example, a string of characters may be located in consecutive memory locations, where each character may be represented by its ASCII equivalent. For referring to a string, two parameters are required, (a) starting or end address of the string and (b) length of the string. The length of a string is usually stored as count in the CX register. In case of 8085, similar structures can be set up by the pointer and counter arrangements which may be modified at each iteration, till the required condition for proceeding further is satisfied. On the other hand, the 8086 supports a set of more powerful instructions for string manipulations. The incrementing or decrementing of the pointer, in case of 8086 string instructions, depends upon the Direction Flag (DF) status. If it is a byte string operation, the index registers are updated by one. On the other hand, if it is a word string operation, the index registers are updated by two. The counter in both the cases, is decremented by one.

**REP: Repeat Instruction Prefix** This instruction is used as a prefix to other instructions. The instruction to which the REP prefix is provided, is executed repeatedly until the CX register becomes zero (at each iteration CX is automatically decremented by one). When CX becomes zero, the execution proceeds to the next instruction in sequence. There are two more options of the REP instruction. The first is REPE/REPZ, i.e. repeat operation while equal/zero. The second is REPNE/REPNZ allows for repeating the operation while not equal/not zero. These options are used for CMPS, SCAS instructions only, as instruction prefixes.

**MOVSB/MOVSW: Move String Byte or String Word** Suppose a string of bytes stored in a set of consecutive memory locations is to be moved to another set of destination locations. The starting byte of the source string is located in the memory location whose address may be computed using SI (Source Index) and DS (Data Segment) contents. The starting address of the destination locations where this string has to be relocated is given by DI (Destination Index) and ES (Extra Segment) contents. The starting address of the source string is 10H*DS+[SI], while the starting address of the destination string is 10H*ES+[DI]. The MOVSB/MOVSW instruction thus, moves a string of bytes/words pointed to by DS: SI pair (source) to the memory location pointed to by ES: DI pair (destination). The REP instruction prefix is used with MOVS instruction to repeat it by a value given in the counter (CX). The length of the byte string or word string must be stored in CX register. No flags are affected by this instruction.

After the MOVS instruction is executed once, the index registers are automatically updated and CX is decremented. The incrementing or decrementing of the pointers, i.e. SI and DI depend upon the direction flag DF. If DF is 0, the index registers are incremented, otherwise, they are decremented, in case of all the string manipulation instructions. The following string of instructions explain the execution of the MOVS instruction.

**Example 2.42**

```
MOV AX, 5000H    ; Source segment address is 5000h
MOV DS, AX       ; Load it to DS
MOV AX, 6000H    ; Destination segment address is 6000h
MOV ES, AX       ; Load it to ES
MOV CX, 0FFH     ; Move length of the string to counter register CX
MOV SI, 1000H    ; Source index address 1000H is moved to SI
MOV DI, 2000H    ; Destination index address 2000H is moved to DI
CLD              ; Clear DF, i.e. set autoincrement mode
REP MOVSB        ; Move 0FFH string bytes from source address to destination
```

**CMPS: Compare String Byte or String Word** The CMPS instruction can be used to compare two strings of bytes or words. The length of the string must be stored in the register CX. If both the byte or word strings are equal, zero flag is set. The flags are affected in the same way as CMP instruction. The DS:SI and ES:DI point to the two strings.The REP instruction prefix is used to repeat the operation till CX(counter) becomes zero or the condition specified by the REP prefix is false.

The following string of instructions explain the instruction. The comparison of the string starts from initial byte or word of the string, after each comparison the index registers are updated depending upon the direction flag and the counter is decremented. This byte by byte or word by word comparison continues till a mismatch is found. When, a mismatch is found, the carry and zero flags are modified appropriately and the execution proceeds further.

**Example 2.43**

```
MOV AX, SEG1                ; Segment address of STRING1, i.e. SEG1 is moved
                              to AX
MOV DS, AX                  ; Load it to DS
MOV AX, SEG2                ; Segment address of STRING2, i.e. SEG2 is moved
                              to AX
MOV ES, AX                  ; Load it to ES
MOV SI, OFFSET STRING1      ; Offset of STRING1 is moved to SI
MOV DI, OFFSET STRING2      ; Offset of STRING2 is moved to DI
MOV CX, 010H                ; Length of the string is moved to CX
CLD                         ; Clear DF, i.e. set autoincrement mode
REPE CMPSW                  ; Compare 010H words of STRING1 and
                            ; STRING2, while they are equal, If a mismatch is found,
                            ; modify the flags and proceed with further execution
```

If both strings are completely equal, i.e. CX becomes zero, the ZF is set, otherwise, ZF is reset.

**SCAS: Scan String Byte or String Word** This instruction scans a string of bytes or words for an operand byte or word specified in the register AL or AX. The string is pointed to by ES:DI register pair. The length of the string is stored in CX. The DF controls the mode for scanning of the string, as stated in case of MOVSB instruction. Whenever a match to the specified operand, is found in the string, execution stops and the zero flag is set. If no match is found, the zero flag is reset. The REPNE prefix is used with the SCAS instruction. The pointers and counters are updated automatically, till a match is found. The following string of instructions elaborates the use of SCAS instruction.

**Example 2.44**

```
 MOV AX,SEG     ; Segment address of the string, i.e. SEG is moved to AX
 MOV ES,AX      ; Load it to ES
 MOV DI,OFFSET  ; String offset, i.e. OFFSET is moved to DI
 MOV CX,010H    ; Length of the string is moved to CX
 MOV AX,WORD    ; The word to be scanned for, i.e. WORD is in AL
 CLD            ; Clear DF
REPNE SCASW     ; Scan the 010H bytes of the string, till a match to
                ; WORD is found
```

This string of instructions finds out, if it contains WORD. If the WORD is found in the word string, before CX becomes zero, the ZF is set, otherwise the ZF is reset. The scanning will continue till a match is found. Once a match is found the execution of the programme proceeds further.

**LODS: Load String Byte or String Word** The LODS instruction loads the AL/AX register by the content of a string pointed to by DS:SI register pair. The SI is modified automatically depending upon DF. The DF plays exactly the same role as in case of MOVSB/MOVSW instruction. If it is a byte transfer(LODSB), the SI is modified by one and if it is a word transfer(LODSW), the SI is modified by two. No other flags are affected by this instruction.

**STOS: Store String Byte or String Word** The STOS instruction stores the AL/AX register contents to a location in the string pointed by ES: DI register pair. The DI is modified accordingly. No flags are affected by this instruction.

The direction flag controls the string instruction execution. The source index SI and destination index DI are modified after each iteration automatically. If DF = 1, then the execution follows autodecrement mode. In this mode, SI and DI are decremented automatically after each iteration (by 1 or 2 depending upon byte or word operations). Hence, in autodecrementing mode, the strings are referred to by their ending addresses. If DF = 0, then the execution follows autoincrement mode. In this mode, SI and DI are incremented automatically (by 1 or 2 depending upon byte or word operation) after each iteration, hence the strings, in this case, are referred to by their starting addresses. Chapter 3 on assembly language programming explains the use of some of these instructions in assembly language programs.

### 2.3.5 Control Transfer or Branching Instructions

The control transfer instructions transfer the flow of execution of the program to a new address specified in the instruction directly or indirectly. When this type of instruction is executed, the CS and IP registers get loaded with new values of CS and IP corresponding to the location where the flow of execution is going to be transferred. Depending upon the addressing modes specified in Chapter 1, the CS may or may not be modified. This type of instructions are classified in two types:

**Unconditional Control Transfer (Branch) Instructions** In case of unconditional control transfer instructions, the execution control is transferred to the specified location independent of any status or condition. The CS and IP are unconditionally modified to the new CS and IP.

**Conditional Control Transfer (Branch) Instructions** In the conditional control transfer instruc-

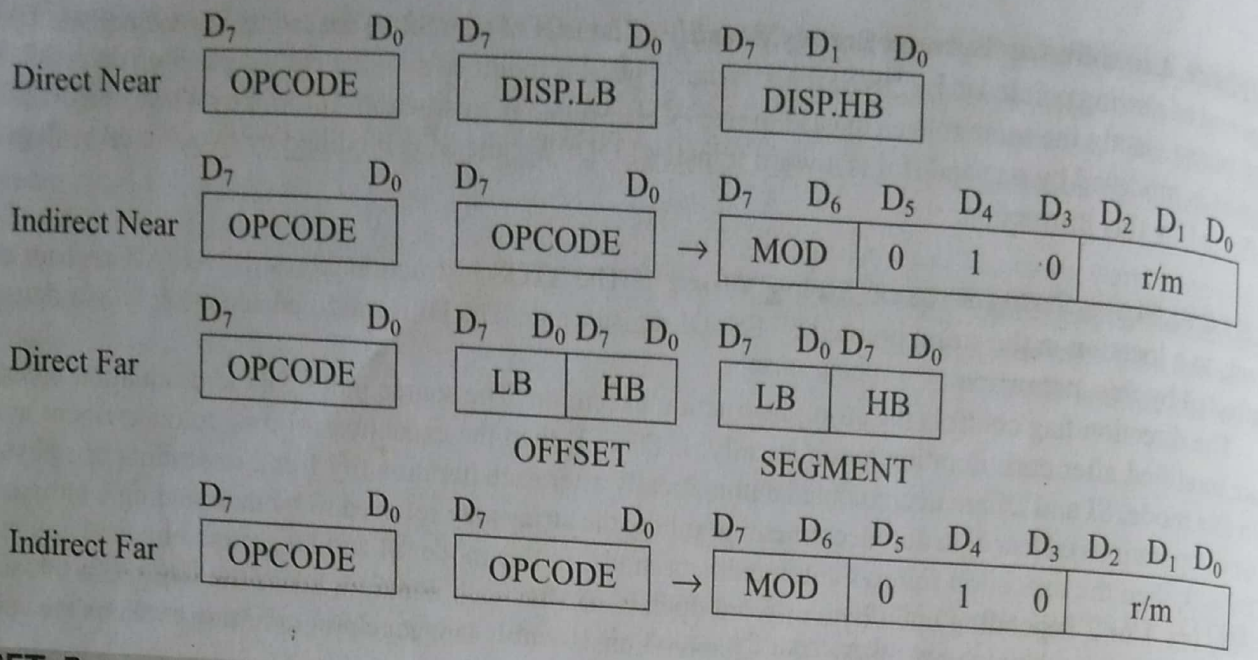
**LODS: Load String Byte or String Word** The LODS instruction loads the AL/AX register by the content of a string pointed to by DS:SI register pair. The SI is modified automatically depending upon DF. The DF plays exactly the same role as in case of MOVSB/MOVSW instruction. If it is a byte transfer(LODSB), the SI is modified by one and if it is a word transfer(LODSW), the SI is modified by two. No other flags are affected by this instruction.

**STOS: Store String Byte or String Word** The STOS instruction stores the AL/AX register contents to a location in the string pointed by ES: DI register pair. The DI is modified accordingly. No flags are affected by this instruction.

The direction flag controls the string instruction execution. The source index SI and destination index DI are modified after each iteration automatically. If DF = 1, then the execution follows autodecrement mode. In this mode, SI and DI are decremented automatically after each iteration (by 1 or 2 depending upon byte or word operations). Hence, in autodecrementing mode, the strings are referred to by their ending addresses. If DF = 0, then the execution follows autoincrement mode. In this mode, SI and DI are incremented automatically (by 1 or 2 depending upon byte or word operation) after each iteration, hence the strings, in this case, are referred to by their starting addresses. Chapter 3 on assembly language programming explains the use of some of these instructions in assembly language programs.

### 2.3.5 Control Transfer or Branching Instructions

The control transfer instructions transfer the flow of execution of the program to a new address specified in the instruction directly or indirectly. When this type of instruction is executed, the CS and IP registers get loaded with new values of CS and IP corresponding to the location where the flow of execution is going to be




**RET: Return from the Procedure** At each CALL instruction, the IP and CS of the next instruction is pushed onto stack, before the control is transferred to the procedure. At the end of the procedure, the RET instruction must be executed. When it is executed, the previously stored content of IP and CS along with flags are retrieved into the CS, IP and flag registers from the stack and the execution of the main program continues further. The procedure may be a near or a far procedure. In case of a FAR procedure, the current contents of SP points to IP and CS at the time of return. While in case of a NEAR procedure, it points to only IP. Depending upon the type of procedure and the SP contents, the RET instruction is of four types.

1. Return within segment
2. Return within segment adding 16-bit immediate displacement to the SP contents.
3. Return intersegment
4. Return intersegment adding 16-bit immediate displacement to the SP contents.

**INT N: Interrupt Type N** In the interrupt structure of 8086/8088, 256 interrupts are defined corresponding to the types from 00H to FFH. When an INT N instruction is executed, the TYPE byte N is multiplied by 4 and the contents of IP and CS of the interrupt service routine will be taken from the hexadecimal multiplication (N´4) as offset address and 0000 as segment address. In other words, the multiplication of type N by 4 (offset) points to a memory block in 0000 segment, which contains the IP and CS values of the interrupt service routine. For the execution of this instruction, the IF must be enabled.

**Example 2.45**

Thus the instruction INT 20H will find out the address of the interrupt service routine as follows:

```
INT      20H
Type* 4 = 20 * 4 = 80H
```

Pointer to IP and CS of the ISR is 0000 : 0080 H

Figure 2.14 shows the arrangement of CS and IP addresses of the ISR in the interrupt vector table.

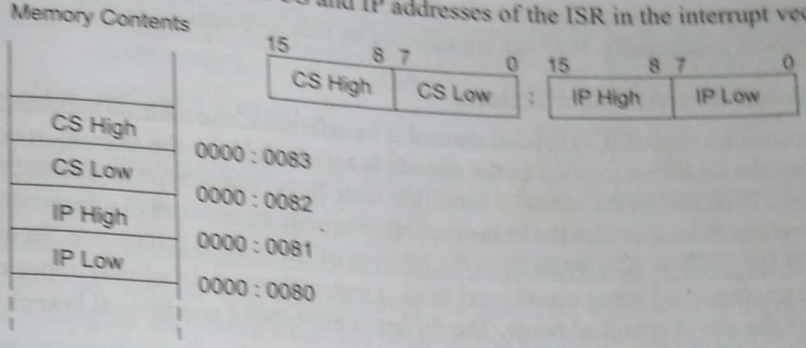



**Fig. 2.14** *Contents of IVT*

**INTO: Interrupt on Overflow** This command is executed, when the overflow flag OF is set. The new contents of IP and CS are taken from the address 0000:0010 as explained in INT type instruction. This is equivalent to a Type 4 interrupt instruction.

**JMP: Unconditional Jump** This instruction unconditionally transfers the control of execution to the specified address using an 8-bit or 16-bit displacement (intrasegment relative, short or long) or CS: IP (intersegment direct far). No flags are affected by this instruction. Corresponding to the methods of specifying jump addresses, the JUMP instruction may have the following three formats. For other JMP types the reader may refer to the following datasheet.

```
JUMP DISP 8-bit                                  Intrasegment, relative, short
                                                 jump

JUMP | DISP.16-bit (LB) | DISP.16-bit (HB) |     Intrasegment, relative, short
                                                 jump

JUMP | IP (LB) | IP (HB) | CS (LB) | S (HB) |    Intrasegment, direct, far jump
```

**IRET: Return from ISR** When an interrupt service routine is to be called, before transferring control to it, the IP, CS and flag register are stored on to the stack to indicate the location from where the execution is to be continued, after the ISR is executed. So, at the end of each ISR, when IRET is executed, the values of IP, CS and flags are retrieved from the stack to continue the execution of the main program. The stack is modified accordingly.

**LOOP: Loop Unconditionally** This instruction executes the part of the program from the label or address specified in the instruction up to the loop instruction, CX number of times. The following sequence explains the execution. At each iteration, CX is decremented automatically. In other words, this instruction implements DECREMENT COUNTER and JUMP IF NOT ZERO structure.

**Example 2.46**

```
        MOV   CX, 0005   ; Number of times in CX
        MOV   BX, 0FF7H  ; Data to BX
Label : MOV   AX, CODE1
        OR    BX, AX
        AND   DX, AX
        Loop  Label
```

The execution proceeds in sequence, after the loop is executed, CX number of times. If CX is already 00H, the execution continues sequentially. No flags are affected by this instruction.

### 2.3.7 Conditional Branch Instructions

When these instructions are executed, execution control is transferred to the address specified relatively in the instruction, provided the condition implicit in the opcode is satisfied. If not the execution continues sequentially. The conditions, here, means the status of condition code flags. These type of instructions do not affect any flag. The address has to be specified in the instruction relatively in terms of displacement which must lie within −80H to 7FH (or −128 to 127) bytes from the address of the branch instruction. In other words, only short jumps can be implemented using conditional branch instructions. A label may represent the displacement, if it lies within the above specified range. The different 8086/8088 conditional branch instructions and their operations are listed in Table 2.3.

**Table 2.3** *Conditional Branch Instructions*

| | *Mnemonic* | *Displacement* | *Operation* |
|---|---|---|---|
| 1. | JZ/JE | Label | Transfer execution control to address 'Label', if ZF=1. |
| 2. | JNZ/JNE | Label | Transfer execution control to address 'Label', if ZF=0. |
| 3. | JS | Label | Transfer execution control to address 'Label', if SF=1. |
| 4. | JNS | Label | Transfer execution control to address 'Label', if SF=0. |
| 5. | JO | Label | Transfer execution control to address 'Label', if OF=1. |
| 6. | JNO | Label | Transfer execution control to address 'Label', if OF=0. |
| 7. | JP/JPE | Label | Transfer execution control to address 'Label', if PF=1. |
| 8. | JNP | Label | Transfer execution control to address 'Label', if PF=0. |
| 9. | JB/JNAE/JC | Label | Transfer execution control to address 'Label', if CF=1. |
| 10. | JNB/JAE/JNC | Label | Transfer execution control to address 'Label', if CF=0. |
| 11. | JBE/JNA | Label | Transfer execution control to address 'Label', if CF=1 or ZF=1. |
| 12. | JNBE/JA | Label | Transfer execution control to address 'Label', if CF=0 or ZF=0. |
| 13. | JL/JNGE | Label | Transfer execution control to address 'Label', if neither SF=1 nor OF=1. |
| 14. | JNL/JGE | Label | Transfer execution control to address 'Label', if neither SF=0 nor OF=0. |
| 15. | JLE/JNC | Label | Transfer execution control to address 'Label', if ZF=1 or neither SF nor OF is 1. |
| 16. | JNLE/JE | Label | Transfer execution control to address 'Label', if ZF=0 or at least any one of SF and OF is 1(Both SF and OF are not 0). |

While the remaining instructions can be used for unsigned binary operations, the last four instructions are used in case of decisions based on signed binary number operations. The terms above and below are generally used for unsigned numbers, while the terms less and greater are used for signed numbers. A conditional jump instruction, that does not check status flags for condition testing, is given as follows:

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

The conditional LOOP instructions are given in Table 2.4 with their meanings. These instructions may be used for implementing structures like DO_WHILE, REPEAT_UNTIL, etc.

**Table 2.4** *Conditional Loop Instructions*

| *Mnemonic* | *Displacement* | *Operation* |
|---|---|---|
| LOOPZ/LOOPE (Loop while ZF = 1; equal) | Label | Loop through a sequence of instructions from 'Label' while ZF=1 and CX ¹ 0. |
| LOOPNZ/LOOPENE (Loop while ZF = 0; not equal) | Label | Loop through a sequence of instructions from 'Label' while ZF=0 and CX ¹ 0. |

These instructions will be clear with programming practice. This topic aims at introducing them to the readers. Of course, examples are quoted wherever possible, but the JUMP and the LOOP instructions require a sequence of instructions for explanations and they will be emphasized more in Chapter 3.

### 2.3.8 Flag Manipulation and Processor Control Instructions

These instructions control the functioning of the available hardware inside the processor chip. These are categorized into two types; (a) flag manipulation instructions and (b) machine control instructions. *The flag manipulation instructions directly modify some of the flags of 8086. The machine control instructions control the bus usage and execution*. The flag manipulation instructions and their functions are listed in Table 2.5.

**Table 2.5** *Flag Manipulation Instructions*

| | | |
|---|---|---|
| CLC | - | Clear carry flag |
| CMC | - | Complement carry flag |
| STC | - | Set carry flag |
| CLD | - | Clear direction flag |
| STD | - | Set direction flag |
| CLI | - | Clear interrupt flag |
| STI | - | Set interrupt flag |

These instructions modify the Carry (CF), Direction (DF) and Interrupt (IF) flags directly. The DF and IF, which may be modified using the flag manipulation instructions, further control the processor operation; like interrupt responses and autoincrement or autodecrement modes. Thus, the respective instructions may also be called machine or processor control instructions. The other flags can be modified using POPF and SAHF instructions, which are termed as data transfer instructions, in this text. No direct instructions, are available for modifying the status flags except carry flag.

The machine control instructions supported by 8086 and 8088 are listed in Table 2.6 along with their functions. They do not require any operand.

**Table 2.6** *Machine Control Instructions*

| | | |
|---|---|---|
| WAIT | - | Wait for Test input pin to go low |
| | | Halt the processor |

As explained in Chapter 1, after executing the HLT instruction, the processor enters the halt state. The two ways to pull it out of the halt state are to reset the processor or to interrupt it. When NOP instruction is executed, the processor does not perform any operation till 4 clock cycles, except for incrementing the IP by one. It then continues with further execution after 4 clock cycles. ESC instruction when executed, frees the bus for an external master like a coprocessor or peripheral devices. The LOCK prefix may appear with another instruction. When it is executed, the bus access is not allowed for another master till the lock prefixed instruction is executed completely. This instruction is used in case of programming for multiprocessor systems. The WAIT instruction when executed, holds the operation of processor with the current status till the logic level on the $\overline{\text{TEST}}$ pin goes low. The processor goes on inserting WAIT states in the instruction cycle, till the $\overline{\text{TEST}}$ pin goes low. Once the $\overline{\text{TEST}}$ pin goes low, it continues further execution.

## 2.4 ASSEMBLER DIRECTIVES AND OPERATORS

The main advantage of machine language programming is that the memory control is directly in the hands of the programmer enabling him to manage the memory of the system more efficiently. However, there are more disadvantages. The programming, coding and resource management techniques are tedious. As the programmer has to consider all these functions, the chances of human errors are more. To understand the programs one has to have a thorough technical knowledge of the processor architecture and instruction set.

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

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

Another type of hint which helps the assembler to assign a particular constant with a label or initialise particular memory locations or labels with constants is an *operator*. In fact, the operators perform the arithmetic and logical tasks unlike directives that just direct the assembler to correctly interpret the program to code it appropriately. The following directives are commonly used in the assembly language programming practice using Microsoft Macro Assembler or Turbo Assembler. The directives and operators are discussed here but their meanings and uses will be more clear in Chapter 3 on assembly language programming techniques.

**DB: Define Byte** The DB directive is used to reserve byte or bytes of memory locations in the available memory. While preparing the EXE file, this directive directs the assembler to allocate the specified number of memory bytes to the said data type that may be a constant, variable, string, etc. Another option of this directive also initialises the reserved memory bytes with the ASCII codes of the characters specified as a string. The following examples show how the DB directive is used for different purposes.

**Example 2.47**

```
RANKS    DB 01H, 02H, 03H, 04H
```

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

```
MESSAGE DB 'GOOD MORNING'
```

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

```
VALUE    DB  50H
```

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

**DW: Define Word** The DW directive serves the same purposes as the DB directive, but it now makes the assembler reserve the number of memory words (16-bit) instead of bytes. Some examples are given to explain this directive.

**Example 2.48**

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

This makes the assembler reserve four words in memory (8 bytes), and initialize the words with the specified values in the statements. During initialisation, the lower bytes are stored at the lower memory addresses, while the upper bytes are stored at the higher addresses. Another option of the DW directive is explained with the DUP operator.

```
  WDATA    DW 5 DUP (6666H)
```

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

**DQ: Define Quadword** This directive is used to direct the assembler to reserve 4 words (8 bytes) of memory for the specified variable and may initialise it with the specified values.

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

**ASSUME: Assume Logical Segment Name** The ASSUME directive is used to inform the assemble, the names of the logicals segments to be assumed for different segments used in the program. In the assembly language program, each segment is given a name. For example, the code segment may be given the name CODE, data segment may be given the name DATA etc. The statement ASSUME CS : CODE directs the assembler that the machine codes are available in a segment named CODE, and hence the CS register is to be loaded with the address (segment) allotted by the operating system for the label CODE, while loading. Similary, ASSUME DS : DATA indicates to the assembler that the data items related to the program, are available in a logical segment named DATA, and the DS register is to be initialised by the segment address

value decided by the operating system for the data segment, while loading. It then considers the segment DATA as a default data segment for each memory operation, related to the data and the segment CODE as a source segment for the machine codes of the program. The ASSUME statement is a must at the starting of each assembly language program, without which a message 'CODE/DATA EMITTED WITHOUT SEGMENT' may be issued by an assembler.

**END: END of Program** The END directive marks the end of an assembly language program. When the assembler comes across this END directive, it ignores the source lines available later on. Hence, it should be ensured that the END statement should be the last statement in the file and should not appear in between. Also, no useful program statement should lie in the file, after the END statement.

**ENDP: END of Procedure** In assembly language programming, the subroutines are called procedures. They may be independent program modules which return particular results or values to the calling programs. The ENDP directive is used to indicate the end of a procedure. A procedure is usually assigned a name, i.e. label. To mark the end of a particular procedure, the name of the procedure, i.e. label may appear as a prefix with the directive ENDP. The statements, appearing in the same module but after the ENDP directive, are neglected from that procedure. The structure given below explains the use of ENDP.

```
PROCEDURE STAR
      ⋮
STAR ENDP
```

**ENDS: END of Segment** This directive marks the end of a logical segment. The logical segments are assigned with the names using the ASSUME directive. The names appear with the ENDS directive as prefixes to mark the end of those particular segments. Whatever are the contents of the segments, they should appear in the program before ENDS. Any statement appearing after ENDS will be neglected from the segment. The structure shown below explains the fact more clearly.

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

The above structure represents a simple program containing two segments named DATA and CODE. The data related to the program must lie between the DATA SEGMENT and DATA ENDS statements. Similary, all the executable instructions must lie between CODE SEGMENT and CODE ENDS statements.

**EVEN: Align on Even Memory Address** The assembler, while starting the assembling procedure of any program, initialises a location counter and goes on updating it, as the assembly proceeds. It goes on assigning the available addresses, i.e. the contents of the location counter, sequentially to the program variables, constants and modules as per their requirements, in the sequence in which they appear in the program. The EVEN directive updates the location counter to the next even address, if the current location counter contents are not even, and assigns the following routine or variable or constant to that address. The structure given below explains the directive.

```
        EVEN
PROCEDURE       ROOT
                  :
        ROOT    ENDP
```

The above structure shows a procedure ROOT that is to be aligned at an even address. The assembler will start assembling the main program calling ROOT. When the assembler comes across the directive EVEN, it checks the contents of the location counter. If it is odd, it is updated to the next even value and then the ROOT procedure is assigned to that address, i.e. the updated contents of the location counter. If the content of the location counter is already even, then the ROOT procedure will be assigned with the same address.

**EQU: Equate** The directive EQU is used to assign a label with a value or a symbol. The use of this directive is just to reduce the recurrence of the numerical values or constants in a program code. The recurring value is assigned with a label, and that label is used in place of that numerical value, throughout the program. While assembling, whenever the assembler comes across the label, it substitutes the numerical value for that label and finds out the equivalent code. Using the EQU directive, even an instruction mnemonic can be assigned with a label, which can then be used in the program in place of that mnemonic. Suppose, a numerical constant which appears in a program ten times. If that constant is to be changed at a later time, one will have to make the correction 10 times. This may lead to human errors, because it is possible that a human programmer may miss one of those corrections. This will result in the generation of wrong codes. If the EQU directive is used to assign the value with a label that can be used in place of each recurrence of that constant, only one change in the EQU statement will give the correct and modified code. The examples given below show the syntax.

**Example 2.49**

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

The first statement assigns the constant 500H with the label LABEL, while the second statement assigns another label ADDITION with mnemonic ADD.

**EXTRN: External and PUBLIC: Public** The directive EXTRN informs the assembler that the names, procedures and labels declared after this directive have already been defined in some other assembly language modules. While in the other module, where the names, procedures and labels actually appear, they must be declared public, using the PUBLIC directive. If one wants to call a procedure FACTORIAL appearing in MODULE1 from MODULE 2; in MODULE1, it must be declared PUBLIC using the statement PUBLIC FACTORIAL and in module 2, it must be declared external using the declaration EXTRN FACTORIAL. The statement of declaration EXTRN must be accompained by the SEGMENT and ENDS directives of the MODULE 1, before it is called in MOBULE 2. Thus the MODULE1 and MODULE 2 must have the following declarations.

```
MODULE1     SEGMENT
PUBLIC      FACTORIAL FAR
MODULE1     ENDS
MODULE2     SEGMENT
EXTRN       FACTORIAL FAR
MODULE2     ENDS
```

**GROUP: Group the Related Segments** This directive is used to form logical groups of segments with similar purpose or type. This directive is used to inform the assembler to form a logical group of the following segment names. The assembler passes an information to the linker/loader to form the code such that the group declared segments or operands must lie within a 64Kbyte memory segment. Thus all such segments and labels can be addressed using the same segment base.

```
PROGRAM GROUP CODE, DATA, STACK
```

The above statement directs the loader/linker to prepare an EXE file such that CODE, DATA and STACK segment must lie within a 64kbyte memory segment that is named as PROGRAM. Now, for the ASSUME statement, one can use the label PROGRAM rather than CODE, DATA and STACK as shown.

```
ASSUME CS: PROGRAM, DS: PROGRAM, SS: PROGRAM
```

**LABEL: Label** The Label directive is used to assign a name to the current content of the location counter. When the assembly process starts, the assembler initialises a location counter to keep track of memory locations assigned to the program. As the program assembly proceeds, the contents of the location counter are updated. During the assembly process, whenever the assembler comes across the LABEL directive, it assigns the declared label with the current contents of the location counter. The type of the label must be specified, i.e. whether it is a NEAR or a FAR label, BYTE or WORD label, etc.

A LABEL directive may be used to make a FAR jump as shown below. A FAR jump cannot be made at a normal label with a colon. The label CONTINUE can be used for a FAR jump, if the program contains the following statement.

```
CONTINUE  LABEL FAR
```

The LABEL directive can be used to refer to the data segment along with the data type, byte or word as shown.

```
DATA          SEGMENT
DATAS DB 50H DUP (?)
DATA-LAST LABEL BYTE FAR
DATA ENDS
```

After reserving 50H locations for DATAS, the next location will be assigned a label DATA-LAST and its type will be byte and far.

**LENGTH: Byte Length of a Label** This directive is not available in MASM. This is used to refer to the length of a data array or a string.

```
MOV CX, LENGTH ARRAY
```

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

**LOCAL** The lables, variables, constants or procedures declared LOCAL in a module are to be used only by that particular module. After some time, some other module may declare a particular data type LOCAL, which was previously declared LOCAL by an other module or modules. Thus the same label may serve different purposes for different modules of a program. With a single declaration statement, a number of variables can be declared local, as shown.

```
LOCAL a, b, DATA, ARRAY, ROUTINE
```

**NAME: Logical Name of a Module** The NAME directive is used to assign a name to an assembly language program module. The module, may now be referred to by its declared name. The names, if selected to be suggestive, may point out the functions of the different modules and hence may help in the documentation.

**OFFSET: Offset of a Label** When the assembler comes across the OFFSET operator along with a label, it first computes the 16-bit displacement (also called as offset interchangebly) of the particular label, and replaces the string 'OFFSET LABEL' by the computed displacement. This operator is used with arrays, strings, lables and procedures to decide their offsets in their default segments. The segment may also be decided by another operator of similar type, viz, SEG. Its most common use is in the case of the indirect, indexed, based indexed or other addressing techniques of similar types, used to refer to the memory indirectly. The examples of this operator are as follows:

**Example 2.50**

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

**ORG : Origin** The ORG directive directs the assembler to start the memory allotment for the particular segment, block or code from the declared address in the ORG statement. While starting the assembly process for a module, the assembler initialises a location counter to keep track of the allotted addresses for the module. If the ORG statement is not written in the program, the location counter is initialised to 0000. If an ORG 200H statement is present at the starting of the code segment of that module, then the code will start from 200H address in code segment. In other words, the location counter will get initialised to the address 0200H instead of 0000H. Thus, the code for different modules and segments can be located in the available memory as required by the programmer. The ORG directive can even be used with data segments similarly.

**PROC: Procedure** The PROC directive marks the start of a named procedure in the statement. Also, the types NEAR or FAR specify the type of the procedure, i.e whether it is to be called by the main program located within 64K of physical memory or not. For example, the statement RESULT PROC NEAR marks the start of a routine RESULT, which is to be called by a program located in the same segment of memory. The FAR directive is used for the procedures to be called by the programs located in different segments of memory. The example statements are as follows:

**Example 2.52**

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

In case of JMP instructions, the PTR operator is used to specify the type of the jump, i.e. near or far, as explained in the examples given below.

```
JMP NEAR PTR [BX]-NEAR Jump
JMP FAR PTR [BX]-FAR Jump.
```

**PUBLIC** As already discussed, the PUBLIC directive is used along with the EXTRN directive. This informs the assembler that the labels, variables, constants, or procedures declared PUBLIC may be accessed by other assembly modules to form their codes, but while using the PUBLIC declared lables, variables, constants or procedures the user must declare them externals using the EXTRN directive. On the other hand, the data types declared EXTRN in a module of the program, may be declared PUBLIC in at least any one of the other modules of the same program. (Refer to the explanation on EXTRN directive to get the clear idea of PUBLIC.)

**SEG: Segment of a Label** The SEG operator is used to decide the segment address of the label, variable, or procedure and substitutes the segment base address in place of "SEG" label. The example given below explains the use of SEG operator.

**Example 2.53**

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

**SEGMENT: Logical Segment** The SEGMENT directive marks the starting of a logical segment. The started segment is also assigned a name, i.e. label, by this statement. The SEGMENT and ENDS directive must bracket each logical segment of a program. In some cases, the segment may be assigned a type like PUBLIC (i.e. can be used by other modules of the program while linking) or GLOBAL (can be accessed by any other modules). The program structure given below explains the use of the SEGMENT directive.

```
EXE.CODE SEGMENT GLOBAL; Start of Segment named EXE.CODE,
                       ; that can be accessed by any other module.
EXE.CODE ENDS          ; END of EXE.CODE logical segment.
```

**SHORT** The SHORT operator indicates to the assembler that only one byte is required to code the displacement for a jump (i.e. displacement is within –128 to +127 bytes from the address of the byte next to the jump opcode). This method of specifying the jump address saves the memory. Otherwise, the assembler may reserve two bytes for the displacement. The syntax of the statement is as given below.

```
JMP SHORT LABEL
```

**TYPE** The TYPE operator directs the assembler to decide the data type of the specified label and replaces the 'TYPE' label by the decided data type. For the word type variable, the data type is 2, for double word type, it is 4, and for byte type, it is 1. Suppose, the STRING is a word array. The instruction MOV AX, TYPE STRING moves the value 0002H in AX.

**GLOBAL** The labels, variables, constants or procedures declared GLOBAL may be used by other modules of the program. Once a variable is declared GLOBAL, it can be used by any module in the program. The following statement declares the procedure ROUTINE as a global label.

```
ROUTINE PROC     GLOBAL
```

**'+ & –' Operators** These operators represent arithmetic addition and subtraction respectively and are typically used to add or subtract displacements (8 or 16 bit) to base or index registers or stack or base pointers as given in the example:

**Example 2.54**

```
MOV AL, [ SI +2 ]
MOV DX, [ BX – 5 ]
MOV BX, [ OFFSET LABEL + 10 H ]
MOV AX, [ BX + 9I ]
```

**FAR PTR** This directive indicates the assembler that the label following FAR PTR is not available within the same segment and the address of the label is of 32-bits i.e. 2 bytes offset followed by 2 bytes segment address.

**Example 2.55**

```
JMP FAR PTR LABEL
CALL FAR PTR ROUTINE
```

Both the above instructions indicate to the assembles that the target address is going to require four bytes; Lower byte of offset, higher byte of offset, lower byte of segment and higher byte of segment; indicating intersegment addressing mode.

**NEAR PTR** This directive indicates that the label following NEAR PTR is in the same segment and needs only 16 bit i.e. 2 byte offset to address it.

**Example 2.56**

```
JMP NEAR PTR LABEL
CALL NEAR PTR ROUTINE
```

If a label is not preceded by NEAR PTR or FAR PTR, then it is by default considered a NEAR PTR label and two bytes are reserved by the assembler for its address during the process of assembling.

## 2.5 Dos and Don'ts While Using Instructions

1) The logic required for implementing a program must be visualized very clearly. There may be many methods or alternative logics to implement a program. A simple-to-implement method must be selected.
2) Express the selected logic in terms of a flowchart or algorithm.
3) Identify the most appropriate instructions to implement the algorithmic steps of the logic.
4) Arrange them in logical sequence to solve the problem.
5) Use more efficient instructions in terms of length in bytes and execution time for implementing the logic. For example, one can use INC AL instead of ADD AL, O1H.
6) Simulate the program on paper assuming a few possible sets of inputs.
7) Provide comments with each instruction regarding its role in the overall implementation of the logic.
8) Remove unnecessary or redundant instructions from the program.
9) If the simulation in 6) goes wrong, repeat from Step 3.

### Don'ts While Using Instructions

1) Don't use any instruction, till its operation is very clear to you. You must be convinced about its role in implementing the logic.
2) Don't use unnecessarily complex addressing modes and instructions, until there is no simpler alternative. Indirect addressing modes must be used conservatively.
3) Don't use mismatching size operands in any instruction until it is demanded by the operation.

```
MOV AX, DL          ; Not allowed.
DIV BL              ; Divide DX-AX (32 bit)
                    ; by BL (8 bit) is allowed.
```

4) Don't use both the operands in an instruction as memory operands. Only one memory operand can be specified in one instruction.

```
MOV [SI], [DI]      ; Not allowed.
MOV AX, [SI]                ; Allowed.
```

5) Don't use immediate 8-bit or 16-bit operand as a destination operand. Destination operand must be a storage element like a register or a memory location. The immediate operand can only be a source operand.

```
MOV 55H, AL         ; Not allowed.
ADD 5779H, AX       ; Not allowed.
ADD AX, 5779H       ; Allowed.
```

6) Both the operands of an instruction can't be immediate operands.
7) Don't use stack operations unnecessarily. Stack operations must be used very carefully.
8) Don't write very big continuous single program for an application. Rather divide the big task in small modules and write modular programmes using subroutines or interrupt service routines if required.
9) Don't use segment registers as operands for arithmatic or logical instructions. It is not allowed.