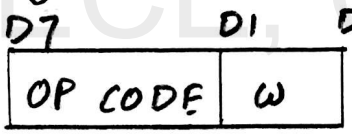
Instruction Formats :-

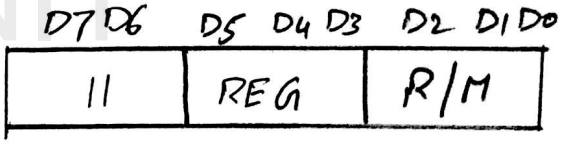
1. One byte Instruction :

This format is only one byte long and may have the implied data or register operands. The least significant 3-bits of the opcode are used for specifying the register operand, if any.

2. Register to Register :

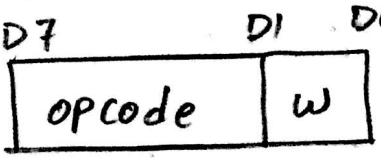
this format is 2bytes long. the first byte of the code specifies the operation code and width of the operand specified by w bit. the 2nd byte of the code shows the register operands and R/M field.

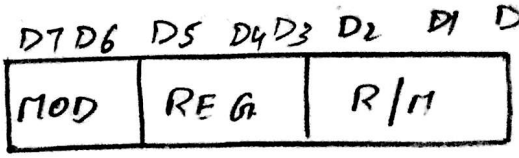




3. Register to/from Memory with no Displacement :

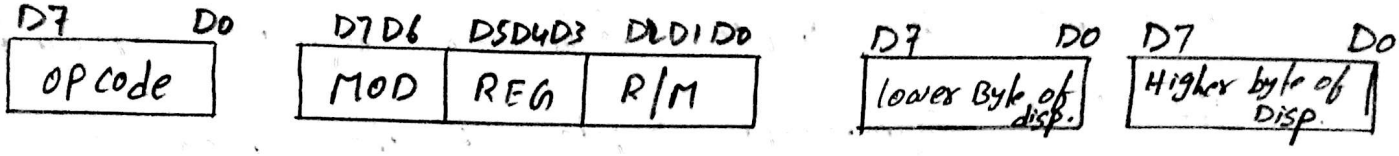
this format is also 2 bytes long and similar to the register to register format except for the MOD field.





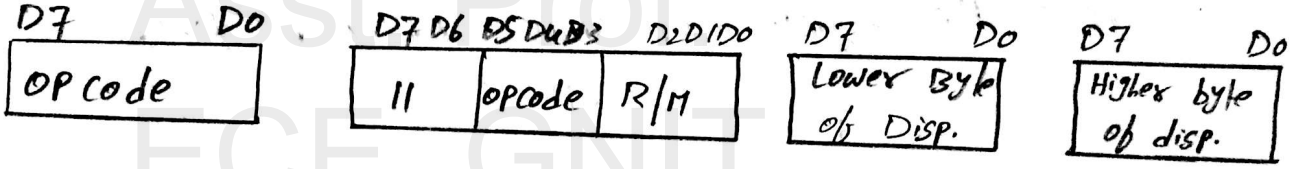
4. Register to/from memory with Displacement :

It contains one or two additional bytes for displacement along with 2-byte the format of the register to/from memory without displacement.



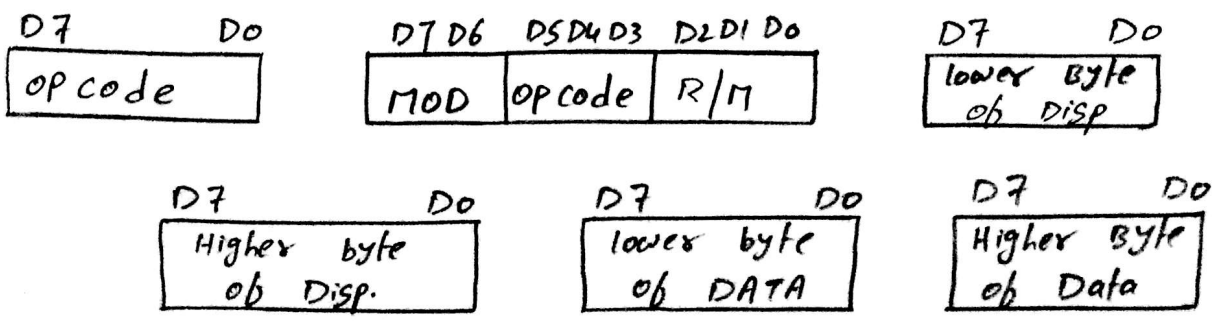
5. Immediate Operand to Register :

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.



6. Immediate operand to Memory with 16-bit Displacement

It requires 5 or 6 bytes for coding. the first 2 bytes contain the information regarding OPCODE, MOD, & R/M fields. the remaining 4 bytes contain 2 bytes of displacement & 2 bytes of data.



MACRO:-

```
Adder Macro
  Add Al, Bl
  Mov [2000], Al
endm
UP: mov Al, N1
    mov Bl, N2
    Adder
    Jmp UP
    INT 03
```

when ever we need to use a group of instructions several times through out a program, there are two ways, we can avoid having to write the group of inst. each time we want to use it. one way is to write the group of inst. as a seperate procedure. and another is MACRO.

when the repeated group of inst. is too short or not appropriate to be written as a proc. we use a Macro. A MACR is a group of inst. at the starting of our program. Every time you see a Macro name in the program, the assembler replaces it with the group of inst. defined as the macro. at the start of the prog.

Dis Advantage of using MACRO is the Assembler generates machine codes for the group of ins. each time the MACRO called. this will make the prog. take up more memory than Procedure.

using a MACRO avoids the overhead time involved in calling and returning from a procedure.

Addressing Modes of 8086 :

Addressing mode indicates a way of locating data or operands depending upon the data types used in the instruction. According to the flow of instruction execution, the instructions may be categorised as

1, sequential control flow instructions and
2, control transfer instructions.

sequential control flow instructions are the instructions which after execution, transfer control to the next instruction appearing immediately after it in program.

eg: Arithmetic, logical, data transfer ....

The control transfer instructions, ~~or the other hand~~ as the instructions which after execution, transfer control to ~~the~~ some predefined address in the program.

eg: INT, CALL, RET, RETI, & ~~JP~~ JUMP..

Addressing Modes for sequential control flow instructions:

1, **Immediate** : Here, immediate data is a part of instruction,

eg: MOV AX, 0015H : Here 0015h is immediate data, which is stored into AX register.

Effective address : No effective address

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

eg: MOV Ax, [1000H] : Here the data resides in a memory location [1000] is copied into Ax register.

Effective address : 10H * DS + offset address

= 10H * DS + 1000h

3. Register : Here, the data is stored in a register & it is refered using the particular register. All the registers except IP, may be used in this mode.

eg: MOV Ax, Bx : the content of Bx is copied into Ax register ; there is no effective address.

4. Register Indirect : the address of memory location which contains data or operand is determined in an indirect way, using the offset register, Bx .

eg: MOV AX, [Bx] : the content in the address specified by Bx register is copied in to AX register.

Before execution	After execution.
Ax : 0001H	Ax : 0020H
Bx : 1000H	Bx : 1000H
Cx : 0005H	Cx : 0005H
1000 : 20H	1000 : 20H.

Effective address = 10H * DS + Bx

5. **Indexed** : the address of memory location which contains data or operand is determined in an indirect way, using the indexed registers, it is known as Indexed addressing mode.

eg: MOV AL, [SI] (or) MOV AL, [DI]

Here, the content in the address specified by SI (or) DI is copied in to AL.

Before execution	After execution
AL : 01h	AL : 20h
SI : 1000h	SI : 1000h
1000 : 20h	1000 : 20h

Effective address = 10H * DS + SI/DI

6. **Register Relative** : Here, data is available at an effective address formed by adding an 8-bit or 16-bit displacement with the content of any one of the registers; BX, BP, SI, DI.

eg: MOV AL, 50H[BX] : Here, the contents in the address specified by [BX + 50] is copied in to AL.

Before execution	After execution
AL : 59H	AL : 19h
BX : 1000h	BX : 1000h
1000 : 40h	1000 : 40h
1050 : 19h	1050 : 19h

Effective Address = 10H * DS + BX+50h.

7. Base Indexed : Here, the effective address is formed by adding content of a base register (BX (or) BP) to the content of an index register (SI/DI).

eg: MOV AL, [BX] [SI] : Here, the content in the address specified by [BX + SI] is copied into AL register.

Effective address : 10H * DS + BX + SI

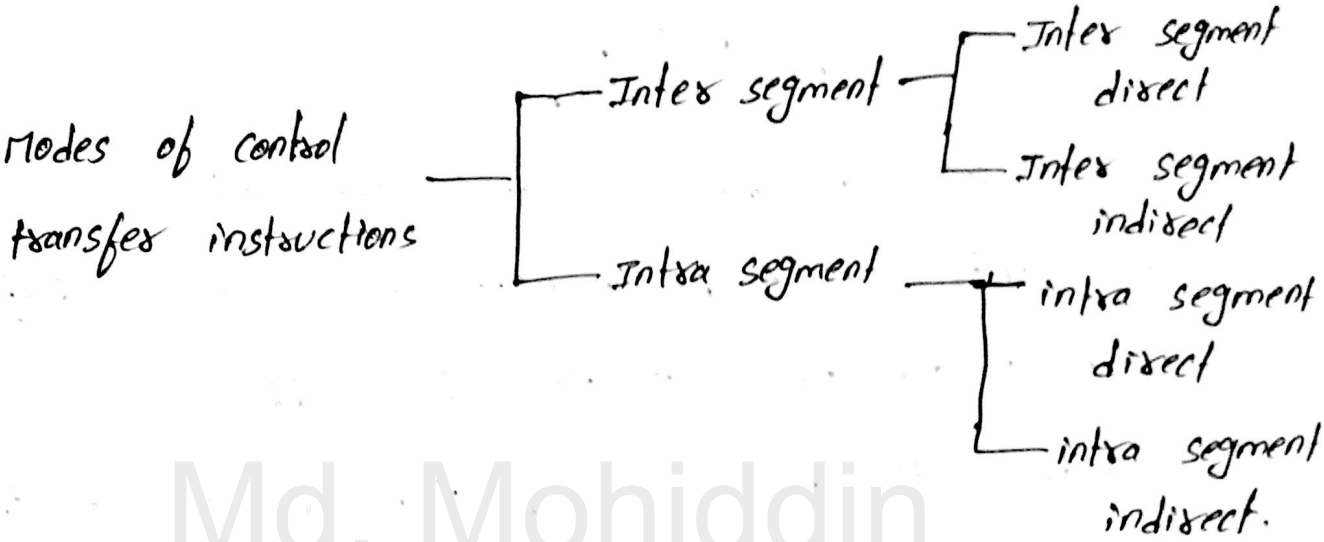
8. Relative Based Indexed : the effective address is formed by adding an 8 (or) 16-bit displacement with the sum of content of any one of the base registers (BX (or) BP) and any one of the index registers, ie

eg: MOV AL, 50H [BX] [SI] : Here, the content in the address specified by [50 + BX + SI] is copied into AL register.

Effective address : 10H * DS + 50 + BX + SI

Addressing Modes for control transfer Instructions:

For the control transfer instructions, the addressing modes depend upon whether the destination location is within the same segment or in a different one.



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

1. **Intra segment Direct mode:** Here, 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.

eg: Jump UP ;

10. Intra segment Indirect mode: Here the ~~displacement~~ Address 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.

eg: Jmp [BX];

11. Inter segment Direct: Here, the address to which the control is to be transferred is in a different segment ~~and the instruction~~ the address to which the control transfers is specified directly in the instruction

eg: Jmp 5000H : 2000H;

12. Inter segment Indirect: Here, the address to which the control is to be transferred lies in a different segment and it is passed to the instruction indirectly.

eg: Jmp [2000H].

Arithmetic Instructions:

Instruction	operation	Example.

ADD	The content of source and destination operands are added and result is stored in to destination.
	Source: immedi, direct, Register, indirect
	destinat: direct, Register, indirect.
		Add Al, Bl.
		Add Ax, [5000]
		Add Ax, 1111h.

ADC	the content of source & dest. operands are added with carry flag and result is stored in to destination.	Adc Al, Bl Al ← Al+Bl+C.F
INC	the content of specified operand is incremented by 1. (direct/Regis/indirec)	inc Al
DEC	the content of direct/Register/indirect is decremented by 1.	dec Al.
SUB	the content of destination operand is subtracted with source operand & result is stored in to destination.	sub Al, Bl Al ← Al − Bl
SBB	the content of destination ope. is subtracted with source ope. & with carry flag & result stored in destinat.	Sbb Al, Bl Al ← Al−Bl−C.F
CMP	this instruction compares the source and destination operands, result reflects in Flag register. source: imm / direc / reg / indir. desti: dir / reg / indir.	cmp Al, Bl cmp Bx, [SI]

Instruction	operation	Example.
AAA Ascii Adjust after Addition.	After addition of two ASCII numbers, to make result valid. AAA instruction must be used.	AAA
AAS Ascii Adjust AL after subtr.	After subtraction of two ASCII no. to make result valid. AAS instru. must be used.	AAS
AAM Ascii adjust after multiplicat	After Multiplicati of two ASCII no. to make result valid. AAM instr. must be used	AAM
AAD Ascii Adjust before division	Before performing Division op. of ASCII no. AAD shold write. It converts two unpacked BCD to its equ. binary.	AAD.
DAA Decimal Adjust Accumulator.	It converts the result of the addition of two packed BCD no. to a valid BCD. It affects AF, CF, PF & ZF flags.	DAA
DAS Decimal adjust after subtract.	It converts result of sub. of two packed BCD no. to a valid BCD no. It affects: AF, CF, SF, PF & ZF flags.	DAS
NEG Negate	It performs 2's compliment of specified operand, and result is stored in the same operand.	Neg Al Neg [1111h]
CBW	converts a signed byte to a signed word. It copies the sign bit of a byte to be convert in to higher byte of word. source : Al (only), dest : Ax (only).	CBW
CWD	converts a signed word to double word source : Ax (only) dest : Dx Ax (only)	CWD

instruction	operation	Example.
MUL	multiplies unsigned source & destination operands, Results stored in to Destination operant. source : Reg / direct / indirect / destination : Al (or) Ax	MUL BL Ax ← Al * BL MUL CX DxAx ← Ax * CX
IMUL	multiplies signed source with dest. operands & result stored in dest. op.	IMUL BX
DIV	It divides an unsigned word (or) double word must be Ax (or) Dx Ax with unsigned byte (or) word. source : Reg/dir/indi ; dest : Ax/Dx Ax	DIV BL Ax / BL — Q : Al, R : Ah DIV BX Dx Ax / Bx : Q : Ax, R : Dx
IDIV	It divides a signed word (or) double word with signed byte (or) word operand. source : Reg / direct / indirect destina : Ax / Dx Ax	IDIV BL

Logical Instructions :

Instruction	Operation	Example
AND	It performs logical AND oper. between source & desti operand and result stored in desti opera. source : Imm / reg / direct / indirec. desti : reg / direct / indirect.	AND Al, Bl AND [1000], cl ABND Ax, [1111]
OR	It performs logical OR operation. source : Imm / reg / direct / indir. desti : reg / dir / indirect.	OR Al, Bl.
NOT	It performs, 1's compliment of operand & result stored in same operand ; reg / dir / indir	NOT Al
XOR	It performs logical XOR operation.	XOR Ax, Bx
TEST	It performs logical compare, by ANDing each bit. Result not stores but OF, CF, SF, ZF & PF flags affected.	TEST Ax, Bx Test [0511], 06h

instruction	operation	Example
SHL/SAL logical/Arithmatic left	these ins. shift the operand bit by bit to the left and insert zero's in the newly introduced LSB's. the count must be in CL.	shl AL, 01 sAL AL, 01 shl AL, CL
SHR shift logical Right	shifts operand bit by bit to right & insert zero's in the newly introduced MSB's.	shr AL, CL
SAR shift Arithmatic right	shifts operand bit by bit to right & insert MSB in the newly introduced MSB's	SAR AL, CL
ROR Rotate right without carry	shift operand bit by bit right & insert LSB in the newly introduced MSB's.	ROR AL, CL
ROL Rotate left without carry	shift operand bit by bit left & insert MSB in the newly introduced LSB's.	ROL AL, CL.
RCR Rotate Right through carry	shifts operand bit by bit right & insert LSB in carry flag, then newly introduced MSB copied with CF.	RCR AL, CL
RCL Rotate left with carry	shift operand bit bybit left, newly introduced LSB is inserted with CF then CF is copied with MSB.	RCL AL, CL.

String Manipulation Instructions

Instruction	operation	Example
MOVSB	string of byte stored at DS:SI copied into ES:DI.	MOVSB
MOVSW	~~word~~ string of word stored at DS:SI copied into ES:DI.	MOVSW
CMPSB/W	It compares a ~~byte~~ string of byte or word stored at DS:SI with a string of byte or word stored at ES:DI.	CMPSB ; CMPSW
SCASB/W	It compares a ~~string~~ string of byte (or) word in AL (or) AX with a string of byte (or) word in ES:DI.	SCASB ; SCASW ;
LODSB/W	string of byte or word stored at DS:SI copied into AL (or) AX register.	LODSB ; LODSW
STOSB/W	string of byte (or) word stored at AL (or) AX register is copied into ES:DI.	STOSB ; STOSW.
REP	It is used as prefix for instruc. It repeats the particular instruction till counter (CX) becomes zero, & decrements CX by 1 each time & increments SI & DI register by 1 each time	REP : MOVSB

Flag Manipulation instructions :

Instruction	Operation	Example
CLC	clear carry flag	CLC
STC	set carry flag	STC
CMC	compliment carry flag	CMC
CLD	clear Directional flag	CLD
STD	set Directional flag	STD
CLI	clear Interrupt flag	CLI
~~STI~~ STI	set Interrupt flag	STI

Machine control Instructions. :

wait	wait for test i/p pin to go low
HLT	halt the processor
NOP	no operation till 4 clock cycles.
ESC	Escape to external devices like NDP
lock	Bus lock instruction Prefix

Control Transfer (or) Branching Instructions:

control transfer Instructions transfers the flow of execution of the program to a new address specified in the instruction. this type of instructions are classified in to two types:

Unconditional control Transfer Instructions:

the execution control is transferred to the specified location independent of any status (or) condition.

Instruc.	operation	Example.
Call	This instruction is used to call a subroutine. the execution control transfers to subroutine. the address of subroutine specified in the instruction directly/indirectly	Call delay
RET	this instruction returns the execution control from subroutine to Main program. the last instruction of subroutine must be RET. this instruction retrieve addresses of old CS & old IP from stack.	RET

Instruction	Operation	Example.
INT N	when an INT N instruction is executed, the type byte N is multiplied by 4 and the content of IP & CS of the interrupt service routine will be taken from vector table. ISR executes.	INT 03h.
INTO	Interrupt on overflow. This command is executes, when the overflow flag of is set.	INTO
IRET	this instructions returns the execution control from ISR to main program.	IRET
JMP	this instruction unconditionally transfers the control of execution to the specified address in the program.	Jmp up.
LOOP	this instruction unconditionally transfers the control of execution to the specified address untill the CX (counter) becomes zero.	

Conditional Branch instructions

Instruction	operation.	example.
JZ / JE	Transfer execution control if ZF=1	JZ up
JNZ / JNE	Trans. exe. ctrl if Z.F = 0	JNZ up ;
JS	Trans. exe. ctrl. if S.F :1 (-ve)	JS back;
JNS	Trans. exe. ctrl. if S.F : 0 (+ve)	JNS back;
JO	Trans. exe. ctrl. if O.F : 1	JO refer/
JNO	Trans. exe. ctrl. if OF : 0	JNO refer.
JP / JPE	Trans. exe. ctrl if P.F : 1	JP back.
JNP	Trans. exe. ctrl if P.F : 0	JNP back.
JB / JNAE / JC	Trans. exe. ctrl. if C.F : 1	JC back.
JNB / JAE / JNC	Tran. exe. ctrl. if C.F : 0	JNC up.
JBE / JNA	Tran. exe. ctrl. if C.F:1 & Z.F : 1	JBE back.
JNBE / JA	Tran. exe. ctrl. if C.F : 0 & Z.F : 0	JA up
JL / JNGE	Tr. exe. ctrl if neither S.F : 1 nor OF : 1	JL up
JNL / JGE	Tr. exe. ctrl if neither SF : 0 . nor OF : 0	JNL up.
JLE / JNG	Tr. exe. ctrl. if Z.F : 1 (or) neither SF nor OF is 1	JLE up
JNLE / JE	Tr. exe. ctrl if Z.F = 0 (or) atleast any one of S.F & OF is 1 (both not)	JE up.

Conditional Loop instructions.

Instruction	Operation	Example.
LOOPZ / LOOPE	Repeats the loop till Z.F is set.	loopz up
LOOPNZ / LOOPENE	Repeats the loop till Z.F	loopnz back

Instruction Set of 8086:

Data copy/Transfer Instructions:

Instruction	operation	Example.

MOV	Copies content ~~from~~ to destination ~~to~~ from source.
	source: Immediate, direct, Register, indirect
	destination: Register, direct, indirect
		MOV Ax, Bx ;
		MOV Ax, 1111h
		MOV [SI], Al

PUSH	copies content in to stack. SP decrement by 1 for byte ope. SP dec. by 2 for word operation.	Push Ax Push Ds Push [5000]
POP	copies content from stack to specified Register/Memory/. SP inc. by 1(or) 2.	POP Ax POP [5000].
XCHG	this instruction exchanges the contents of specified source & destination.	XCHG Ax, Bx XCHG [5000], Ax
✱ IN	this instruction used to read the i/p port. ~~source~~ destin : Al (or) Ax only ~~destination~~ source : address (or) Dx only.	IN Al, Dx IN Ax, FFFFh.
✱ OUT	this instruction is used for writing to an output port. ~~source~~ destin. : address (or) Dx only ~~destination~~ source : Al (or) Ax only	out Dx, Ax out FFF9h, Al.
XLAT	this inst. copies content in to Al from the address specified by [Al+Bx]. It uses look up tables.	XLAT

Instruction	operation	Example.
LEA Load effective Address	this ins. loads the effective address formed by ~~destination~~ Source operand in to the specified ~~source~~ destination register	LEA BX, ADR
LDS/LES Load Pointer to DS/ES	This Instruction loads DS/ES and the specified destination register with the content of memory location specified in the source.	LDS BX, 5000h. 5000:01 5001:02 5002:03 5003:04 DS:0403 BX:0201
LAHF	This instruction loads the AH register with the lower byte of flag register.	LAHF
SAHF	This instruction loads the lower byte of flag register with contents of AH.	SAHF
PUSHF	This ins. Pushes the contents of flag register in to stack. SP decremented by 2	PUSHF
POPF	This ins. load the flag register with contents of stack. SP incremented by 2	POPF

Assembler Directives : directions to the Assembler, not instructions for the 8086, for TASM, MASM.

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

eg: ar db 49h, 98h
eg: ar1 db 'THOMAS'
eg: ar2 db 10 dup(?)

2, DW : Define word : the DW directive is used to reserve word or words of memory locations in the available memory.

eg: ECE db 10 dup(0)

3, DQ : Define Qaud word : This directive is used to direct the assembler to reserve 4 words of memory for the specified variable and may initialise it with the specified values.

4, DT : Define Ten Bytes : the DT directive directs the assembler to define the specified variable requiring 10 bytes for its storage and initialise 10 bytes with the specified values.

5, ASSUME : Assume logical segment Name : The Assume directive is used to inform the assembler, the name of the logical segments to be assumed for different segments used in the program.

eg: Assume cs:code, ds:data.

6, END : End of Program : The END directive marke the end of an ALP.

7. ENDP : End of procedure : the ENDP directive is used to indicate the end of procedure.

8. ENDS : End of segment : This directive marks the end of a logical segment.

9. EVEN : Align on Even memory Address : the EVEN directive updates the location counter to the next even address.

10. EQU : equate : the directive EQU is used to assign a level with a value or a symbol.

eg: x EQU 03h

the use of this directive is just to reduce the recurrence of the numerical values or constants in a programming code.

11. EXTRN : External and public : the EXTRN directive is used to tell the assembler that the names or labels following the directive are in some other assembly module.

eg: Extrn divisor: word.

12. GROUP : Group the Related segments : the GROUP directive is used to tell the assembler to group the logical segments named after the directive into one logical group segment.

~~Assume cs~~

eg sys Group code, data, stack, Extra.

Assume cs: sys, DS: sys, ss: sys, ES: sys.

13, LABEL : label : The LABEL directive is used to assign a name to the cossent content of the location counter.

14, LENGTH : Byte length of a label : Actually this directive is not availabb in MASM. This is used to refer to the length of a data or a string.

eg: MOV CX, length arl.

15, OFFSET : offset of a label : when the assembler comes across the 'OFFSET' operator along with a label, it first computes the 16-bit displacement of a particular label, and replaces the string 'OFFSET LABEL' by the computed displacement.

eg: MOV SI, offset arl.

16, PROC : Procedure : The PROC directive marks the start of the named procedure in the statement.

17, SEG : segment of a label : The SEG operator is used to decide the segment address of a label, variable (or) procedure.

18, SEGMENT : logical segment : The segment directive marks the starting of a a logical segment.

eg: code segment

eg: data segment

19. GLOBAL : The labels, variables, constants or procedures declared 'GLOBAL' may be used by other module of the program.

eg: Global Divisor : word
→ divisor is variable of type word.

20. PUBLIC : 'PUBLIC' directive is used along with the EXTRN directive. This informs that the labels variables, or procedures declared PUBLIC may be accessed by other assembly modules.

eg: Public Divisor, Divident.

21. 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.

eg: ORG 2000h.

22. PTR : Pointer : The pointer operator is used to declare the type of a label, variable or memory operator. The operator PTR is prefixed by either Byte or word.

eg: INC [BX] — ? byte (or) word
so
eg: INC byte PTR [BX]
INC word PTR [BX]

23. TYPE : The type operator directs the assembler to decide the data type of a specified label and replaces the type label by the decided data type. for a byte-type, the assembler give a value 1, for a word-type, the assembler give a value 2,

eg
Add BX, type Arr1.