8086 Instruction Set:

The 8086 instruction set can be classified into following six groups.

→ Data Transfer Instructions
→ Arithmetic Instructions
→ Logical Instructions
→ String Manipulations
→ Control Transfer instructions
→ Processor control Instructions

Data Transfer Instructions:

These type of instructions performs data transfer of the following types:

1. General purpose byte or word transfer
2. special Address Transfer
3. Flag Transfer
4. Simple Input/output Transfer

General Purpose	Input/output
MOV - Move byte or word PUSH - Push word onto the stack POP - Pop word off the stack XCHG - Exchange byte or word XLAT - Translate byte	IN - Input byte or word OUT - output byte or word
Address Object	**Flag Transfer**
LEA - Load effective address LDS - Load pointer using DS LES - Load pointer using ES	LAHF - Load AH register from flags SAHF - store AH register in flags PUSHF - push flags onto stack POPF - pop flags off the stack

Arithmetic Instructions:

Addition
ADD - Add byte or word
ADC - Add byte or word with carry
INC - Increment byte or word by 1
AAA - ASCII adjust for addition
DAA - Decimal adjust for addition
Subtraction
SUB - ~~stabro~~ Subtract a byte or word
SBB - Subtract a byte or word with borrow
DEC - Decrement a byte or word
NEG - Negate a byte or word
CMP - compare a byte or word
AAS - ASCII adjust for subtraction
DAS - Decimal adjust for subtraction
Multiplication
MUL - Multiply byte or word unsigned
IMUL - Integer multiply byte or word
AAM - ASCII adjust for multiply.
Division
DIV - Divide byte or word unsigned
IDIV - Integer divide a byte or word
AAD - ASCII adjust for division
CBW - convert byte or word
CWD - Convert byte or word to double word

Bit Manipulation Instruction or logical instruction:

Logical	
NOT	'Not' byte or word
AND	'AND' byte or word
OR	'inclusive OR' byte or word
XOR	'exclusive OR' byte or word
TEST	'test' byte or word
Shifts	
SHL/SAL	shift logical/ arithmetic left byte or word
SHR	shift logical right byte or word
SAR	shift arithmetic right byte or word
Rotates	
ROL	Rotate left byte or word
ROR	Rotate right byte or word
RCL	Rotate through carry left byte or word
RCR	Rotate through carry right byte or word

String Instructions:

REP -	Repeat
REPE | REPZ	Repeat while equal |zero
REPNE | REPNZ	Repeat while not equal | not zero
MOVS	Move byte or word string
MOVSB | MOVSW	move byte or word string
CMPS	compare byte or word string
SCAS	Scan byte or word string
LODS	Load byte or word string
STOS	store byte or word string

string Instruction Register and flag use

SI - ~~Sou~~ Index(offset) for source string

DI - Index(offset) for destination string

CX - Repetition counter

AL|AX - scan Value

Destination for LODS

source for ~~to~~ STOS

DF - 0 = auto increment SI, DI

1 = auto decrement SI, DI

ZF - Scan/compare terminator

Program Transfer Instruction:

Unconditional Transfer	
CALL	– Call procedures
RET	– Return from procedure
JMP	– Jump
Conditional Transfer	
JA / JNBE	Jump if above / not below nor equal
JAE / JNB	Jump if above or equal / not below
JB / JNAE	Jump if below / not above nor equal
JBE / JNA	Jump if below or equal / not above
JC	Jump if carry
JE / JZ	Jump if equal / zero
JG / JNLE	Jump if greater / not less not equal
JGE / JNL	Jump if greater or equal / ~~for~~ not less
JL / JNGE	Jump if less / not greater nor equal
JLE / JNG	– Jump if equal or less / not greater
JNC	– Jump if not carry
JNE / JNZ	– Jump if not equal / not zero
JNO	– Jump if not overflow
JNP / JPO	– Jump if not parity / parity odd
JNS	– Jump if not sign
JO	– Jump if overflow
JP / JPE	– Jump if parity / parity even
JS	– Jump if sign

Iteration Control
LOOP - Loop LOOPE / LOOPZ - Loop if equal / zero LOOPNE / LOOPNZ - Loop if not equal / not zero JCXZ - ~~JCXZ~~ Jump if register CX = 0
Interrupts
INT - Interrupt INTO - Interrupt if overflow IRET - Interrupt Return

Processor Control Instructions:

Flag operations	
STC	set Carry Flag
CLC	clear carry flag
CMC	complement carry flag
STD	set direction flag
CLD	clear direction flag
STI	set Interrupt Enable flag
CLI	clear interrupt enable flag
External synchronization	
HLT	Halt until interrupt or reset
WAIT	wait for $\overline{\text{Test}}$ pin active
ESC	Escape to external processor
LOCK	Lock bus during next instruction
NO operation	
NOP	No operation

8086 Instruction Types :-

→ The 8086 instruction set can be classified into following six groups.

- Data Transfer Instructions
- Arithmetic Instructions
- Logical Instructions
- String Manipulations
- Control Transfer Instructions
- Processor Control Instructions.

Data Transfer Instructions:

- The data transfer instructions move single byte or word between memory and registers as well as between register AL or AX and I/O ports.
- The stack manipulation instructions are included in this group for transferring flag contents and for loading segment registers.

Examples: MOV, PUSH, POP, XCHG, XLAT, IN, OUT, LEA, LDS, LES, LAHF, SAHF, PUSHF, POPF

Arithmetic Instructions

Arithmetic Data format:

- 8086 Arithmetic operations can be performed on four types of numbers:
  1. ~~unsid~~ unsigned binary
  2. signed binary (Integers)
  3. unsigned packed decimal
  4. unsigned unpacked decimal.

Arithmetic Data format

- Binary numbers may be 8 bit or 16 bit long.
- Decimal members are stored in bytes.

  two digits per byte for packed decimal
  one digit per byte for unpacked decimal

- The processor always assumes that the operands specified in arithmetic instructions contain data that represent valid numbers for the type of instruction being performed

⇒ *need of adjustment of various data | result

Arithmetic Instruction and flags:

The 8086 instructions post certain characteristic of the result of the operation to six flags: These flags are

- CF carry Flag
- AF Auxiliary flag.
- SF sign flag
- ZF zero flag
- PF Parity flag
- OF Overflow flag

Most of these flags can be tested by following the arithmetic instruction with a conditional jump or INTO.

Example of arithmetic instructions:

ADD, ADC, INC, AAA, DAA, SUB, SBB, DEC, NEG, CMP, AAS, DAS, MUL, IMUL, AAM, DIV, IDIV, AAD, CBW, CWD.

Bit manipulation Instructions / Logical Instructions

The 8086 provides three groups of instructions for manipulating both bytes and words

- Logical operations
- Shifts
- Rotate

~~Bit~~ Example.

NOT	SHL / SAL	ROL
AND	SHR	ROR
OR	SAR	RCL
XOR		RCR
TEST		

String Instructions:

- Five basic operations (move, compare, load, store, and scan) on strings of bytes or words.
- → One element at a time (byte or word at a time)
- → strings of upto 64KB may be manipulated with these instruction
- A string instruction may have a source operand, a destination operand or both.
- The hardware assumes that a source string resides in the current data segment.
- Destination in extra segment.

Example

REP	MOVS	LODS
REPE / REPZ	MOVSB / MOVSW	STOS
REPNE / REPNZ	CMPS	
	SCAS	

string instruction registes and flag

SI	Index (offset) for source string
DI	Index (offset) for destination string
CX	repetition counter
AL/AX	Scan Value, Destination for LODS Source for STOS
DF	0 = auto increment of SI, DI 1 = auto decrement of SI, DI
~~ZF~~ ZF	scan / compare terminator

Program Transfer Instruction:

Program transfer instuctions have four groups

- unconditional transfer.
- conditional transfer
- Iteration control instruction
- Interrupt related instruction

Example

CALL
RET
JMP

JA | JNBE
JAE | JNB
JB | JNAE
JBE | JNA
JC
JE | JZ
JG | JNLE
JG | JNL

JL | JNGE
JLE | JNG
JNC
JNE | JNZ
JNO
JNP | JPO
JNS
JO
JP | JPE
JS

LOOP
LOOPE | LOOPZ
LOOPNE | LOOPNZ
JCXZ

INT
INTO
IRET

Processor Control Instructions

These instructions allow programs to control various CPU functions.

- To update flags
- To synchronize 8086 with external events.
- Causing processor to do nothing

8086 Instruction Format:

- The 8086 instruction size varies from one byte to six bytes in length.
- The general instruction format that most of instructions follow is

7 6 5 4 3 2	1	0	7 6	5 4 3	2 1 0		
Opcode 6 bits	D	W	MOD	REG	R/M	Lower order Displacement /Data	Higher order Displacement/ data
← 1 byte →			← 1 byte →			← 1 byte →	← 1 byte →

Opcode: The opcode field occupies 6 bits. It defines the operation to be carried out by the instruction.

D bit Register Direct Bit
The D bit specifies that the register specified in byte 2 is the source or destination operand.
* Register specified in REG field is a source or destination

D=0 (Source Register)

D=1 (Destination Register)

W-Bit (Data Size bit) defines whether the operation to be performed is an 8-bit or 16-bit data.

W=0 8 bit operation

W=1 16 bit operation

* Second byte of instruction usually specifies whether one of the operands is in memory or whether both are registers.
* Second byte specifies addressing mode.
* This byte contains — MOD ⇒ mode
REG ⇒ Register
R/M ⇒ Register or memory field

MOD Field:

MOD (2 bits)	Interpretation
0 0	Memory mode with no displacement follows except for 16 bit displacement when R/M = 110
0 1	Memory mode with 8-bit displacement
1 0	Memory mode with 16-bit displacement
1 1	Register mode (no displacement) (R/M field is treated as REG field)

REG Field: - Register field occupies 3 bits

- It defines the register for the first operand which is specified as source or destination by the D-bit.

REG codes

REG	W=0	W=1
000	AL	AX
001	CL	CX
010	DL	DX
011	BL	BX
100	AH	SP
101	CH	BP
110	DH	SI
111	BH	DI

R/M Field: - R/M field occupies 3 bits. The R/M field along with the MOD field defines the second operand.

- R/M field depends on how the MOD fild is set.

- if MOD = 11 (register to register mode) the R/M identifies the second register operand.

when MOD selects memory mode the R/M indicates how the effective address of memory operand is to be calculated.

R/M	MOD=00	MOD=01	MOD=10	Register mode MOD=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 (Direct Address)	(BP)+D8	(BP)+D16	DH	SI
111	(BX)	(BX)+D8	(BX)+D16	BH	DI

* Bytes 3 through 6 of an instruction are optional fields that normally contain the displacement value of a memory operand and / or the actual value of an intermediate constant operand.

In first byte 6 bit opcode and two bit special indicators are used

- D bit and W bit or
- S bit and W bit or
- V bit and W bit.

S bit: An 8-bit 2's complement can be extended to a 16-bit 2's complement number by making all of the bits in the higher order byte equal to the most significant bit in the lower order byte. — sign extension

S	W	operation
0	0	8-bit operation
0	1	16-bit operation with 16-bit immediate data
1	0	— undefined
1	1	16-bit operation with sign extended 8-bit immediate data

V-Bit V-bit decides the number of shifts for rotate and shift operation

if V=0 shift and rotate operation one time (or one bit)

if V=1 CL count value for number of shift or rotate operation.

Z-Bit: This bit is used as a compare bit with zero flag in conditional repeat (REP) and loop instructions

Z = 0 repeat/loop while zero flag is clear &

Z = 1 repeat/loop while zero flag is set

Example

(1) code for MOV CH, BL

This instruction transfers 8-bit content of BL to CH

8-bit so W=0

REG field (BL) is a source D=0

CH ← BL (source)

Second operand as Register so R/m field will work as reg. R/M = 101 (for CH)

MOD=11 (Both are registers, so Register to register mode.)

first operand REG= 011 (for BL)

Example 2:-

code for SUB BX, (DI)

This instruction subtracts the 16-bit content of memory location addressed by DI and DS from BX.

→ The 6 bit opcode for SUB

→ No displacement specified in the instruction, but memory access - So MOD = 00

→ DI Register is used, so R/M = 101

→ Register BX is used so REG = 011 and D = 1

→ 16-bit operation ⇒ W = 1

Types of Instruction formats:

- → One Byte Instruction
- → Two Byte Instruction
- → Three-Byte Instruction
- → Four-Byte Instruction
- → Five and Six Byte instruction

One byte Instruction Format:

→ This type of instruction are only one byte long and may have the implied data or register mode.

→ First byte - will have 6-bit opcode and two special bit indicators:

D bit and W bit (or)

S-bit and W bit (or)

V bit and W-bit

→ One byte

Opcode

implied mode

→ one byte

7 6 5 4 3	2 1 0
opcode	Reg

one byte instruction Register mode

Two-Byte Instruction Format:

→ These are two byte long

→ First byte - opcode, width (W), direction (D)

The second byte - specifies the addressing mode of the operands.
This byte has three fields :- MOD
- REG
- R/M

Opcode	D	W	MOD	REG	R/M
1 byte			1 byte		

Three-Byte Instruction Format:

→ This is three byte long.

→ This instruction format contains one additional byte for 8-bit displacement along with 2 bytes (for opcode and addressing mode)

Register to/from memory with 8 bit displacement

Opcode	D	W	MOD	REG	R/M	D8
1 Byte			1 Byte			1 byte

D8 — 8-bit displacement

<u>Example</u> Register to memory with 8-bit (Displacement)
Memory to register with 8-bit displacement)

Four Byte Instruction Format:

- This type of instructions ~~are~~ is four byte long
- Additionally used 16-bit displacement
- First byte for opcode
  Second byte for addressing mode.
  Third byte lower order displacement
  Fourth byte Higher order displacement
  ] 16 bit displacement.

Opcode	D	W	MOD	REG	R/M	lower order Displacement	High order Displ
1 Byte			1 Byte			1 Byte	1 Byte

Five and six Byte Instruction Format:

Five byte Format:

opcode	D	W	MOD	REG	R/M	Lower byte of Displ.	Higher byte of Displ.
1 Byte			1 Byte			1 Byte	1 Byte

Lower byte of Data
1 Byte

six byte Format:

opcode	D	W	MOD	REG	R/M	Low byte of Displ.	Higher byte of Displ
1 Byte			1 Byte			1 Byte	1 Byte

Lower byte of data	Higher byte of Data
1 Byte	1 Byte

Interrupts:

- An interrupt is a condition that causes the microprocessor temporarily to work on a different task and return to the previous task. Interrupt is an event or signal that request the attention of CPU.

* In interrupt method, whenever any device needs service from microprocessor, the device notifies to the processor by sending signal (called interrupt). Upon receiving ~~an signal~~ an interrupt signal, the microprocessor holds whatever it is doing and serves the corresponding device. The program associated with the interrupt is called the interrupt service routine (ISR) or interrupt handler.

- Every interrupt is assigned a type code that identifies it to CPU.
- The 8086 can handle upto 256 different interrupt types.
- A processor may be interrupted in the following ways:
    - initiated by device external to CPU
    - by software interrupt instructions
    - under certain conditions, by CPU itself.

Types of Interrupts

In general, there are two types of interrupts
- Hardware Interrupts
- Software Interrupts

Hardware Interrupts (or External Interrupts)

- Interrupt initiated by an external hardware by sending an appropriate signal to the interrupt pin of the processor is called hardware interrupt.

- The 8086 has two interrupt pins ; INTR , NMI .
- Hardware interrupts are used to handle external hardware such as keyboard.

Maskable Interrupts :

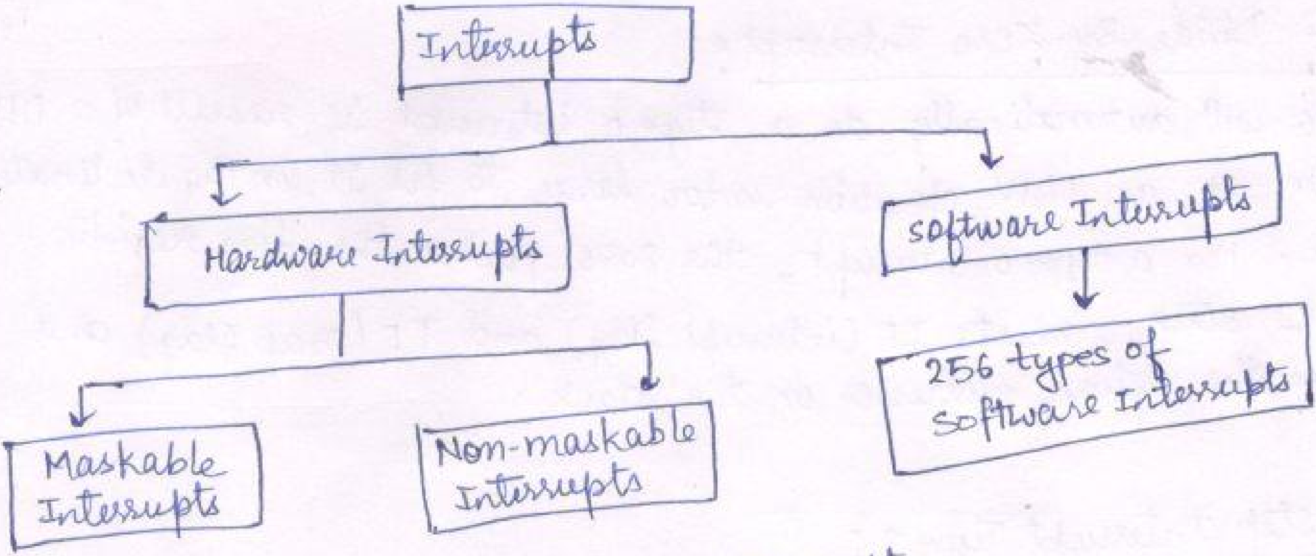
- Programming the processor to reject an interrupt is called masking or disabling the interrupt.
- Programming the processor to accept an interrupt is called unmasking or enabling.

To mask ⟹ To reject
To unmask ⟹ To accept and serve

- In 8086 the Interrupt flag (IF) can be set to to unmask or enable all hardware interrupts.
- IF (interrupt flag) is cleared to zero to mask or disable all hardware interrupt except NMI (non maskable).
- The interrupts whose request can be either accepted or rejected by the processor are called maskable interrupts.

Non-maskable Interrupts

- The interrupts whose request has to be definitely accepted (or can not be rejected) by the processor) are called non-maskable interrupts.
- Interrupts initiated through NMI pin
- Non-maskable interrupts:
  - used during power failure
  - during critical response time
  - non-recoverable hardware error
  - during memory parity error



Types of interrupts

Software Interrupts: (Internal Interrupts)

- software interrupts are the program instructions. These instructions are inserted at desired locations in a program. while running the program, if software interrupt instruction is encountered then processor initiates the interrupt.
- The 8086 processor has 256 types of software interrupt. The software interrupt instruction INT n where n is the type number in the range 0 to 255.
- Software interrupts are used by the operating system for various functions.

256 interrupts are divided into 3 groups:

- Type 0 to Type 4 interrupts (Dedicated Interrupts for fixed use)
- Type 5 to Type 31 interrupts (Reserved interrupts not used by 8086)
- Type 32 to Type 255 (Available for users called user defined Interrupts).

Type 0 - Divide-By-Zero Interrupt:

The 8086 will automatically do a type 0 interrupt if result of a DIV operation or an IDIV operation is too large to fit it in the destination register. For a type 0 interrupt, the 8086 pushes the flag register onto the stack, resets IF (interrupt flag) and TF (Trap flag) and pushes the return addresses on the stack.

Single-Step Interrupt Type 1:

→ used for executing the program in single step mode by setting TF flag.

→ In single step mode, a system will stop after it executes each instruction and wait for further direction from the user.

→ The use of single step execution feature is found in monitor and debugger programs.

Type 2: Non-maskable interrupt:

- The 8086 automatically do a type 2 interrupt response when it receives a low to high transition on its NMI pin.
- when 8086 does a type-2 interrupt, the 8086 will push the flags on the stack, reset TF and IF and push CS (code segment) value and IP (Index Pointer) register value for the next instruction on the stack.

Type 3 Breakpoint Interrupt:

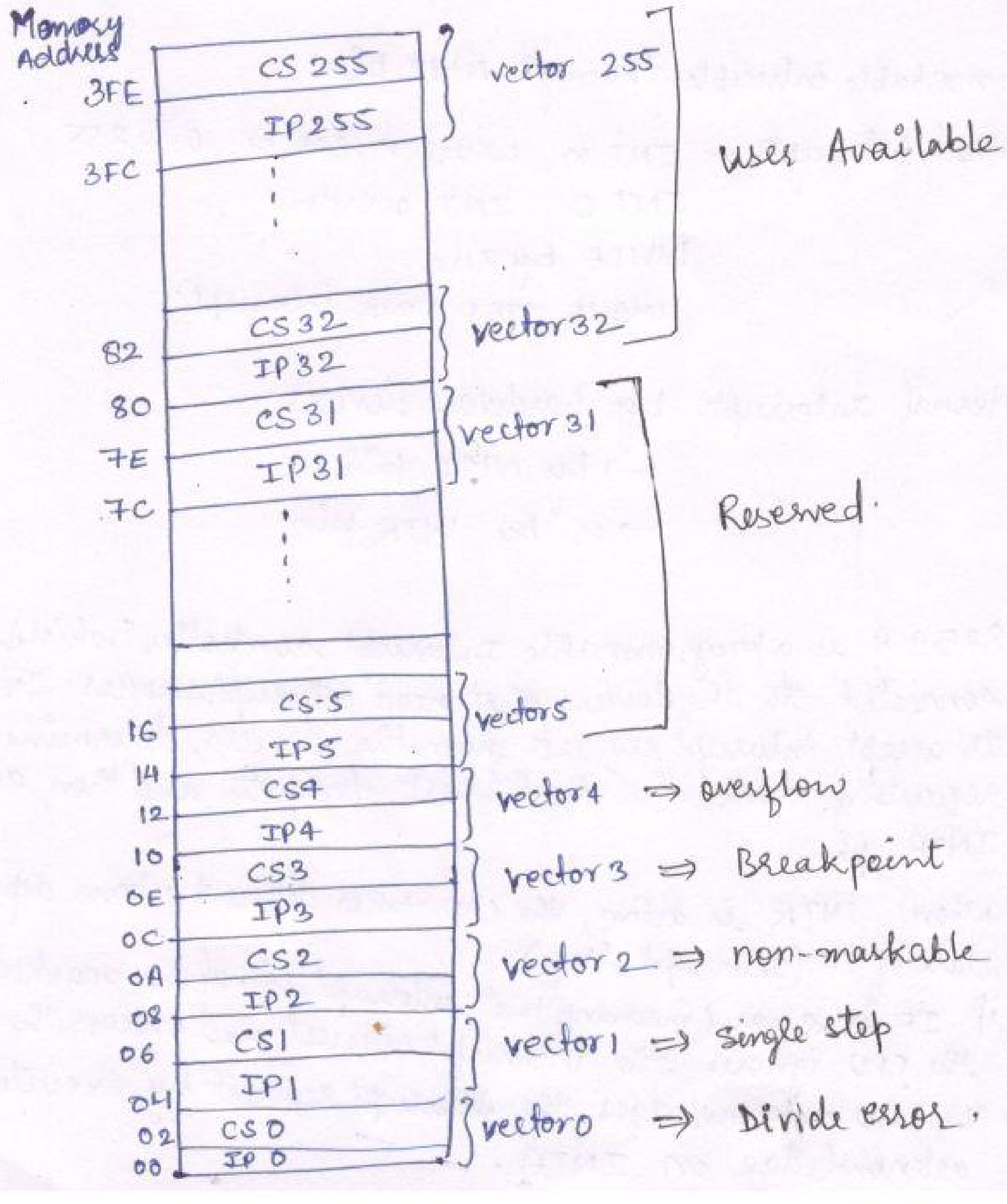
- The type 3 interrupt is produced by execution of INT3 instruction. The main use of type-3 interrupt is to implement breakpoint function in a system.

Type 4: Overflow Interrupt

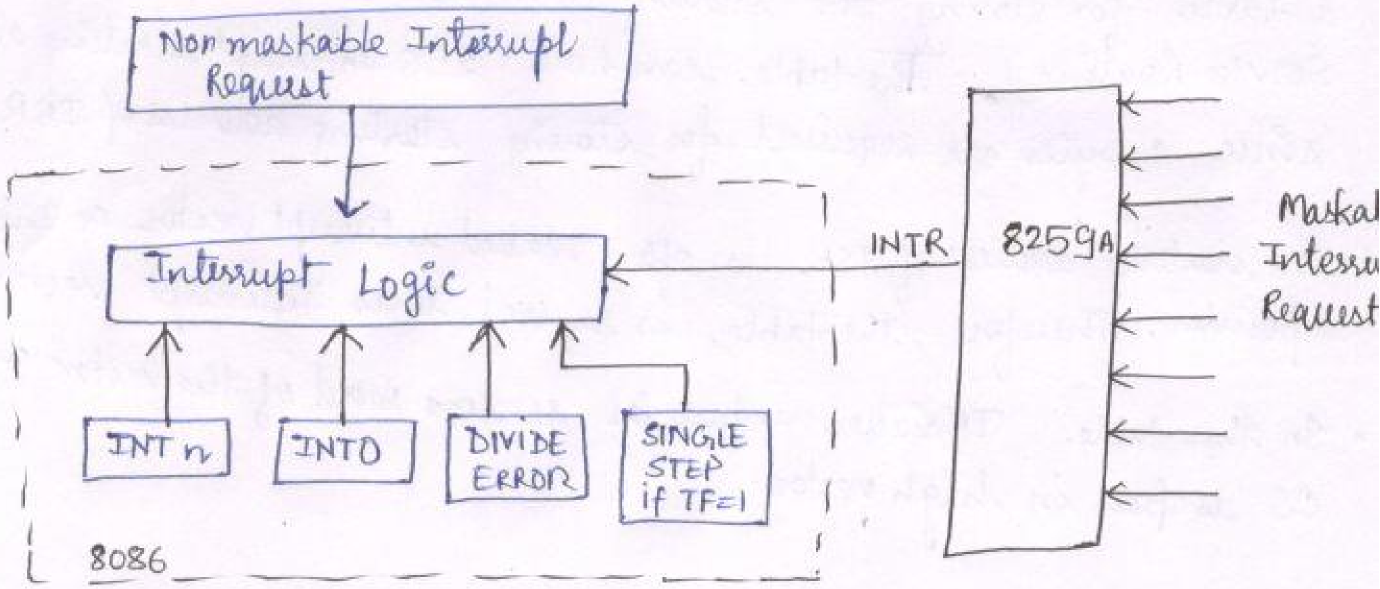
- Used to handle any overflow error.
- The 8086 overflow flag is set if the signed result of an arithmetic operation on two signed numbers is too large to be represented in the destination register.

Interrupt Vector Table

- The first 1KB memory of 8086 (00000 to 003FF) is set aside as a table for storing the starting addresses of ISR (Interrupt Service Routine). The table can hold 256 Interrupt starting address since 4-bytes are required for storing starting address of ISR.
- The starting address of ISR is often called interrupt vector or Interrupt pointer. Therefore, the table is referred to as Interrupt vector Table.
- In this table, IP value is put in as low word of the vector and CS is put in high vector.



Interrupt Sources:



→ Non maskable interrupts through NMI pin.

→ Internal interrupts – INT n where n can be 0 to 255
INT O INT overflow
DIVIDE by zero.
SINGLE STEP mode interrupt.

→ External Interrupts by hardware devices.
→ by NMI pin
→ or by INTR pin.

→ 8259 A is a programmable Interrupt controller which is in turn connected to the devices that need interrupt services. Its main job is to accept interrupt requests from the devices, determine which requesting device has the highest priority and then activate 8086 INTR pin.

when INTR is active, the CPU takes different actions depending upon the state of IF (Interrupt flag).

if IF is clear (meaning that interrupt signal are masked or disabled) the CPU ignores the interrupt request and process the next instruction.

· The CPU acknowledges the interrupt request by executing interrupt acknowledge on INTA.

ASSEMBLER DIRECTIVES

- An assembler is a program that accepts an assembly language program as input and converts it into an object module.
- An assembly language programs are composed of two types of statements:
  → Instructions
  → Assembler Directives (or Pseudo Instructions)

→ The instructions which can be translated to machine code by the assembler.

→ The assembler directive that directs the assembler during the assembly process ~~which~~ for which no machine code is generated

- Assembler directives also called pseudo instructions are into the source code along with assembly program.

- The pseudo instructions (or assembler directives do not get translated into object code (or machine code) but are used as special instructions to the assembler to perform some specific functions.

The directives control the generation of machine code and organization of the program

8086 Assembler Directives:

List of 8086 assembler Directives

DB	ASSUME	LENGTH	PUBLIC	SEGMENT	END
DD	EQU	OFFSET	SHORT	PROC	ENDS
DQ	EVEN	PTR	TYPE	MACRO	ENDP
DT	EXTRN	NAME	GLOBAL		ENDM
DW	ORG	LABEL	INCLUDE		
		GROUP			

DB (Define Byte)

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

Examples

1. PRICE DB 49H ⇒ declare byte type variable named PRICE
2. PRICE DB 49H, 98H, 20H ⇒ declare an array of 3 bytes named as PRICE and initialized
3. NAME1 DB 'ABCDEF ⇒ Declare an array of 6 bytes and initializes with ASCII code for letters.
4. TEMP DB 100 DUP(?)

⇒ declares 100 bytes named as TEMP and leave these 100 bytes unitialized.

DW (Define Word)

DW is used to define a word type variable or to reserve storage location of type word in memory.

Examples:

1. MULTI DW 4367H ⇒ declares a variable of word type with name MULTI and initialized with 4367
2. MULTI DW 100 DUP(0) ⇒ reserves an array of 100 words ~~and~~ named MULTI and initialized all the words with zero initial value.

DD (Define Double Word)

DD is used to declare a variable of type of double word.
size in memory = 4 bytes

Example: ARRAY DD 12345678H ⇒ declares a double word named array and initialize this double word with given value.

DQ (Define Quad Word):

DQ is used to declare a variable 4 words in length.
size in memory ⇒ 8 bytes. Example
BIG DQ 123456789ABCDEF0 H

DT (Define Ten Bytes)

DT is used to declare a variable 10 bytes in length.

Examples

① BCD DT 11223344556677889900H

② RESULT ~~20~~ DT 20 DUP(0) ⟹ declares an array of 20 blocks of 10 bytes each and initialized all with zero.

SEGMENT:

• This ~~segment~~ SEGMENT is used to indicate the start of a logical segment.

segment name

Example CODE SEGMENT ⟹ starts a segment named CODE

- - - - - - -

CODE ENDS ⟹ ends the segment named CODE

ENDS: This ~~dega~~ directive is used ~~to~~ with the segmentname to indicate the end of the logical segment.

ASSUME:

ASSUME tells the assembler what ~~ba~~ names have been chosen for code, data, Extra and stack segments.

Example:

ASSUME CS: Name of code segment
ASSUME DS: Name of data segment
ASSUME CS: Code1, DS: Data1.

CODE SEGMENT
MOV AX, BX
- - - -
~~ENDS~~ CODE ENDS
DATA SEGMENT

Data1 DB 10H
Data2 DW - - -
DATA ENDS

ASSUME CS:CODE, DS:DATA

it refers to the data segment

This tells the assembler that the logical segment named CODE contains the instruction statements of program

The 8086 microprocessor may have several segments. At any time 8086 works directly with only four physical segments; a code segment, a data segment, a stack segment and an extra segment.

EQU: (Equate)

EQU is used to give name to some value or symbol. Each time, the assembler finds the given name in the program, it replaces the name with the ~~vaoh~~ equated value of symbol.

Example

Num1 EQU 50H ⇒ assigns numeric value 50H to Num1

Even:

Even directive aligns even memory address. The Even directive is used to inform the assembler to increment the location counter to the next even memory address if it is not pointing to even memory location already.

Example

```
DATA SEGMENT
SUM DB 10H
EVEN
ITEMS DW 100 DUP(?)
```

⇒ data array ITEMS starts at the even memory location

ORG:

This helps in the placing the machine code in the specified location while transferring the instructions into machine code.

Example:

ORG 1000H ⇒ informs the assembler to initialize the location counter to 1000H.

EXTRN

- The EXTRN directive is used to tell the assembler that name or labels following the directive are in some other assembly module.
- For a reference to externally named variable, ~~the~~
- EXTRN DIVIDE : FAR ⇒ tells the assembler that DIVIDE is a label of type FAR in another assembly module.
- The name or labels referred to as external in one module must be declared public with the PUBLIC directive in the module in which they are defined.

LENGTH:

LENGTH is an operator, which tells the assembler to determine the number of elements in some named data item, such as string or an array.

Example   MOV CX, LENGTH STRING1 ⇒ it will determine the number of elements in STRING1 and load it into CX.

- if string was declared as a string of bytes, LENGTH will produce the number of bytes in the ~~STRING~~ string.
- if string was declared as word string, LENGTH will produce number of words in the string.

OFFSET

offset is an operator, which tells the assembler to determine the offset or displacement of a named data item (variable), a procedure from the start of the segment, which contains it.

MOV BX, OFFSET PRICES ⇒ it will determine the offset of the variable PRICES from the start of the segment in which PRICES is defined and load it into BX.

PTR (pointer) This PTR operator is used to assign a specific type of variable or to a label.

Example

INC [BX] ⇒ This instruction will not know whether to increment the byte pointed by BX or a word pointed by BX

INC BYTE PTR [BX] ⇒ increment the byte pointed by BX

INC WORD PTR [BX]

NAME: The NAME directive is used to give a specific name to each assembly module when program consisting of several module are written.

LABEL:
- An assembler assembles a section of data declarations or instruction statements, it uses a location counter to keep the track of how many byte it is from the start of the segment at any time.
- The LABEL directive is used to give name to the current value of location counter.
- The LABEL directive must be followed by a term that specifies the type you want to associate with that name.
- If label is going to be used as a destination for a jump or a call then type — NEAR (within the module or segment) / FAR (in another segment)

Example

ENTRY_POINT LABEL FAR ⇒ can jump here from another segment

NEXT: MOV AL, BL ⇒ within the segment
can not do a far jump directly to a label with a colon.

GROUP:

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

Example

SMALL_SYSTEM GROUP CODE, DATA, STACK-SEG.

PUBLIC: The PUBLIC directive is used to instruct the assembler that a specified name or label will be accessed from other module.

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

SHORT: The SHORT operator is used to tell the assembler that only 1 byte displacement is needed to code a jump instruction in the program. The destination must in the range of -128 bytes to +127 bytes from the address of the instruction after the jump.

Example JMP SHORT NEARBY_LABEL

TYPE: The TYPE operator tells the assembler to determine the type of a specified variable. The assembler actually determines the number of bytes in the type of variable.

For byte-type variable, assembler will give a value of 1
" word-type " " " " " " 2
" double word-type " " " " " " 4

GLOBAL: The GLOBAL (Declare symbols as PUBLIC or EXTRN)
- The GLOBAL directive can be used in place of PUBLIC or in place of EXTRN.
- GLOBAL directive is used to make the symbol available to other modules.

Example GLOBAL DIVISOR

INCLUDE (Include source code from file)
The directive is used to tell the assembler to insert a ~~black~~ block of source code from the named file into the current source module.

PROC

The PROC directive is used to identify the start of a procedure. After the PROC directive, the term near or the term far is used to specify the type of the procedure.

Example

DIVIDE PROC FAR

NEAR: the procedure resides in the same code segment (Local)

FAR: resides at any location in the memory. (in a segment with different name from the one that contains the instructions which calls the procedure).

```
DIVIDE  PROC  FAR   ⟹ starts procedure.
- - - - - -
- - - - - -
ENDP ~~PROC~~ DIVIDE ⟹ ends the procedure named DIVIDE
```

MACRO

- A macro is a group of instructions with a name. When a macro is invoked, the associated set of instructions is inserted in place into the source, replacing the macro name.

- A macro has a name.

Example

```
PUSHA2C  MACRO; ⟹ starts macro named PUSHA2C
PUSH AX;
PUSH BX;
PUSH CX;
ENDM  PUSHA2C; ⟹ Ends macro named PUSHA2C.
```

Procedure vs. Macro

Procedure:

- only one copy exists in the memory.
- Thus memory consumption is less.
- 'called' when required
- Execution time overhead is present because of call and return instructions

Macro:

- when macro is invoked, the corresponding text is inserted into the source. Thus multiple copies exist in the memory leading to the greater space requirement.
- The code is in-place
- There is no execution overhead because no additional call and return instructions.