ADDRESSING MODES OF 8086 :-

→ Addressing mode indicates a way of locating data or operands

Types are described as follows.

(i) Immediate addressing mode :- In this Immediate data is a part of instruction and appears in the form of byte or bytes.

Eg :-
```
MOV  AX, 0005H
MOV  BL, 06H
```
0005H, 06H are Immediate data may be 8-bit or 16-bit.

ii) Direct addressing mode :- In this mode, a 16-bit memory address (offset) or an I/O address is directly specified in the instruction.

Eg :-
```
MOV  BX, [5000H]
IN   80H
```

iii) Register addressing mode :- In this mode, the data stored in a register is referred using another register

Eg :-
```
MOV  BX, AX
ADC  AL, BL
```

(iv) Register Indirect mode :- In this mode, the memory location which contains data & operands is determined in an indirect way.

Eg:- MOV AX, [BX]

v) Indexed addressing mode :- In this mode, the offset address of the operand is stored in indexed registers like (SI, DI)

Eg:- MOV AX, [SI]

MOV CX, [DI],

(vi) Register Relative mode :- In this mode, the effective address is formed by adding content of specified register with the content on instruction.

Eg:- MOV AX, 50H[BX]

∴ Effective address = 10H * DS + 50H + [BX],

vii) Based Indexed :- The effective address is formed by adding base register to the content of indexed register (SI, DI)

MOV AX, [BX][SI]

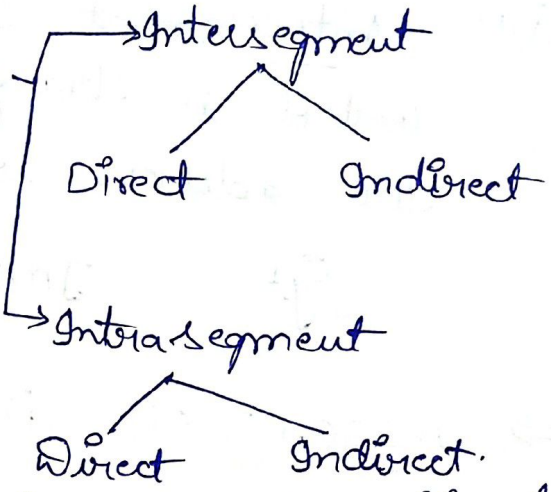
MOV [BX][DI], AX

viii) Relative Based Indexed :- The effective address is formed by adding the content, base registers and index registers

Eg:- MOV AX, 50H[BX][SI]

EA = 50H + [BX][SI]..

* control transfer instructions



* Intersegment means the transfer of location lies in different segment other than the current one.
* Intrasegment means the transfer of location lies in the same segment.

(i) Intrasegment direct mode :- In this mode, the controls are transfered to the label of same segment which is specified in the instruction as a direct displacement value.

Eg:- JMP SHORT LABEL

(ii) Intrasegment indirect mode :- In this mode the transfers occurs in the segment itself but branches are found as a content of register or memory.

Eg:- JMP [BX];

(iii) Intersegment direct :- In this the control is transfered from one segment to another specified directly in the instruction.

Eg:- JMP 5000H : 2000H.

(iv) Intersegment Indirect :- In this mode the control is transfered to different segment, but the address is specified indirectly.

Eg:- JMP [2000H].

⇒ ASSEMBLER DIRECTIVES:-

* 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.

To execute all the tasks of a program assembler needs some hints from the programmer ie., required storage for a variable or constant, name of the segments, end of file etc.,

∴ Thus some hints are given to assembler with some predefined alphabetical strings called assembler directives.

They are.

a) DB: (Define byte) :- The DB directive is used

to reserve byte or bytes of memory locations in the available memory for a given constant or variable etc.,

Eg:- Ranks DB 01H, 02H, 03H, 04H.

This indicates the assembler to reserve 4 memory locations of byte size for the list named "Ranks"

(b) DW: (Define Word) :- DW directive reserves no of memory words (16 bits) for the assigned variable or constant.

Eg:- words DW 1234H, 4567H, 3468H.

(c) DQ: (Define Quadword) :- DQ directive reserves 4 words for the specified variable or constant

(d) DT (Define Ten bytes) :- DT directive reserves 10 bytes of storage for the specified variable or constant.

e) ASSUME (assume logical segment Name) :- The ASSUME directive is used to inform the assembler, that the name of the logical segments are used in the program.

Eg:- ASSUME CS: CODE DS: DATA

It directs the assembler that machine codes are available in a segment named code hence CS segment register is loaded with address for the label CODE.

And user data is available in data segment and DS is loaded with data addresses.

f) END (End of program) :- It marks the end of an assembly language program.

ENDP: (End of Procedure) :- It is used to indicate the end of a procedure. A procedure is usually assigned with a name called 'Label'

Eg:-

```
START:                          # START is a
                                  procedure.
        .
        .
START   ENDP
```

ENDS :- (End of segment) : It marks marks the end of a logical segment.

Eg: a) Data segment

```
        .
        .
Data    ENDS
```

b)

```
ASSUME  CS: CODE        DS: DATA
CODE    SEGMENT
        .
        .
CODE    ENDS.
END
```

EVEN :- (Allign on Even memory address) :- The EVEN directive updates the location counter to next even addresses only.

Eg:-

```
        EVEN
Procedure   ROOT
        .
        .
ROOT    ENDP
```

The procedure named ROOT is to be alligned to next even addresses only.

EQU: (Equate) :— The directive EQU is used to

assign a label with a value or a symbol. That label can be used in place of a numerical value throughout the program.

Eg:-
```
LABEL     EQU   0500H
ADDITION  EQU   ADD
```

EXTRN : External and Public : Public a) The directive EXTRN informs the assembler that the name, procedure, or label declared after this directive have already been defined in some other assembly language module.
b) But before using the label as EXTRN it should be declared as public in the above module.

Eg:-
```
MODULE1   SEGMENT
          PUBLIC  FACTORIAL FAR
MODULE1   ENDS
MODULE2   SEMENT
          EXTRN   FACTORIAL FAR
MODULE2   ENDS
```

GROUP :- It is used to form logical groups of segments with similar purpose or type. That is the assembler groups all the declared segments as a group and assigns a name to it.

Eg:- PROGRAM GROUP CODE, DATA, STACK.

Here CODE & Data, stack segment forms a group with a label named program.

LABEL :- The label directive is used to assign a name to the current content of the location counter which is assigned to a program.

The label can be used for a far jump as well. As given below

Eg:- CONTINUE LABEL FAR.

LENGTH:- It is used to refer the length of a data array or a string.

MOV CX, length array.

Note:- Not available in MASM.

LOCAL:- The labels, variables, constants or procedures declared as LOCAL in a module are to be used only by that particular module.

Eg:- LOCAL a, b, Data, ARRAY, ROUTINE

NAME:- This directive is used to assign a name to an assembly language program module.

OFFSET:- (offset of a label) Offset always comes with a label (strings, labels, & procedures) to decide their offset address in the segment

Eg:- Code segment

MOV SI, OFFSET LIST

CODE ENDS.

ORG (origin):- This directive is used to direct the assembler to start the memory allocation for the particular segment i.e. block of code with a particular address

Eg:- ORG 2000H.

If no address is specified it starts from 0000H by default.

PROC (procedure):- It marks the start of a program. It also declares whether the procedure is NEAR or FAR type procedure.

Eg:- PROC NEAR
PROC FAR

PTR: (Pointer) :- Pointer is used to declare the data type of a label, variable or memory operand. This operator is always prefixed with either byte or word.

Eg:- MOV AL, BYTE PTR[SI].

(or)

MOV BX, WORD PTR[2000H].

PUBLIC :- Already discussed in EXTRN.

SEG:- The segment directive marks the starting of a logical statement. The segment is closed with a ENDS directive

Eg:- CODE SEGMENT
:
CODE ENDS

SHORT :- This operator indicates that only one byte is required for the code for the jump operation.

Syntax: JMP SHORT LABEL

TYPE :- It directs the assembler to decide the data type of the specified label.

Eg:- MOV AX, TYPE STRING.

GLOBAL :- If any label, variables, constants or procedures are declared global which means it can be used by other modules of a program.

Eg:- ROUTINE PROC GLOBAL.

'+' & '-' operators:- It indicates arithmetic addition and subtraction operators.

Eg:- MOV AL, [SI+2].
MOV BX, [BX-2]

FAR PTR :- This directive indicates that the specified label is not available within the same segment and the address is of 32-bits.

Eg:- JMP FAR PTR LABEL

NEAR PTR :- This directive indicates that the label is in the same segment and needs 16 bit of address.

Eg:- JMP NEAR PTR LABEL.

INTRODUCTION TO STACK :-

1) The stack is a block of memory that is used for temporarily storage of contents of a registers inside the CPU.

2) Stacks are accessed using SP and SS registers.

3) As we go on loading the data onto the stack the stack pointer decreases by 2. And when we retrieve it, the SP value increases by 2.

4) Stack is mostly required in case of CALL instructions

5) The process of storing data in the stack is called "pushing into" and the reverse process is called "popping off" the stack.

6) Stack is a "Last-in-first-out" data segment.

Stack structure of 8086/88 :-

The stack segment (SS) and stack pointer register (SP) together address the stack top as in following lines.

For example

SS = 5000H = 0101 0000 0000 0000

SP = 2050H = 0010 0000 0101 0000

⇒ 10H × SS ⇒ 0100 0000 0000 0000 0000

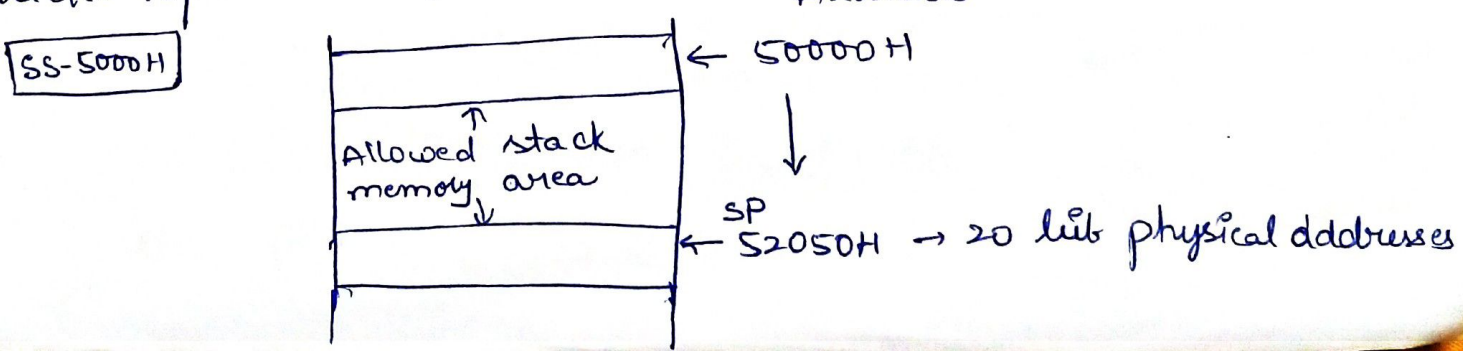
\+

SP ⇒ 0010 0000 0001 0000

0101 0010 0000 0101 0000

5 2 0 5 0

Stack-top address ⇒ 5 2 0 Addresses



* When the next push operation occurs the stack pointer will be decremented by two.

∴ SP = 52050H − 00002H

= 5204EH

* Because 2 bytes of data is loaded into the stack which occupies two memory locations in the stack.

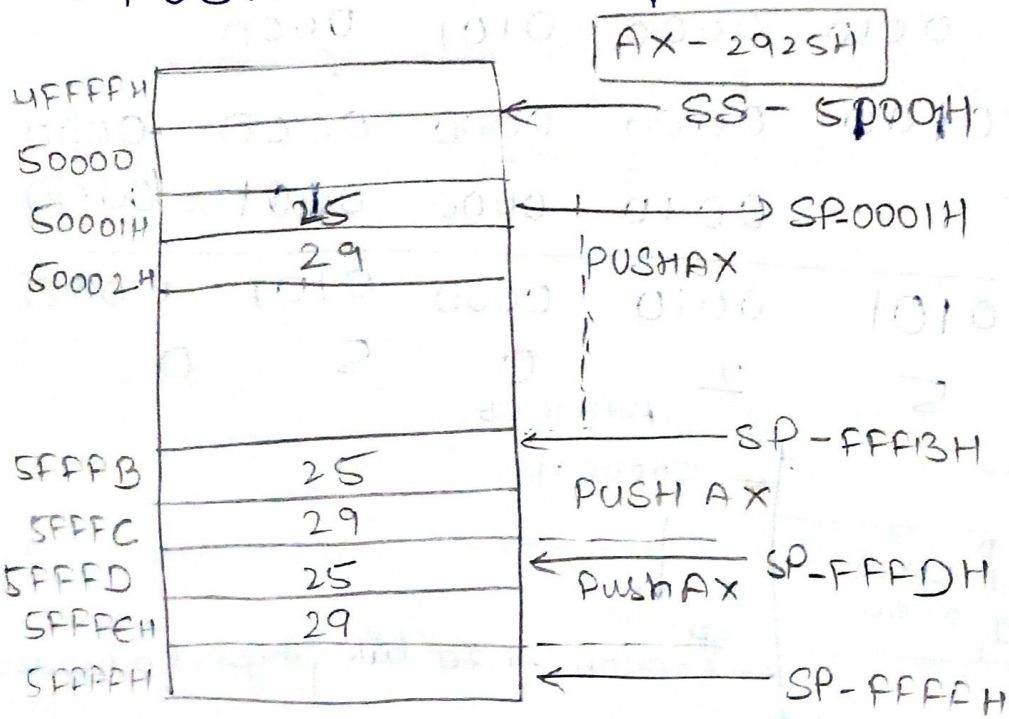
* The initial value of SP is FFFFH and for every push operation it is decremented by 2

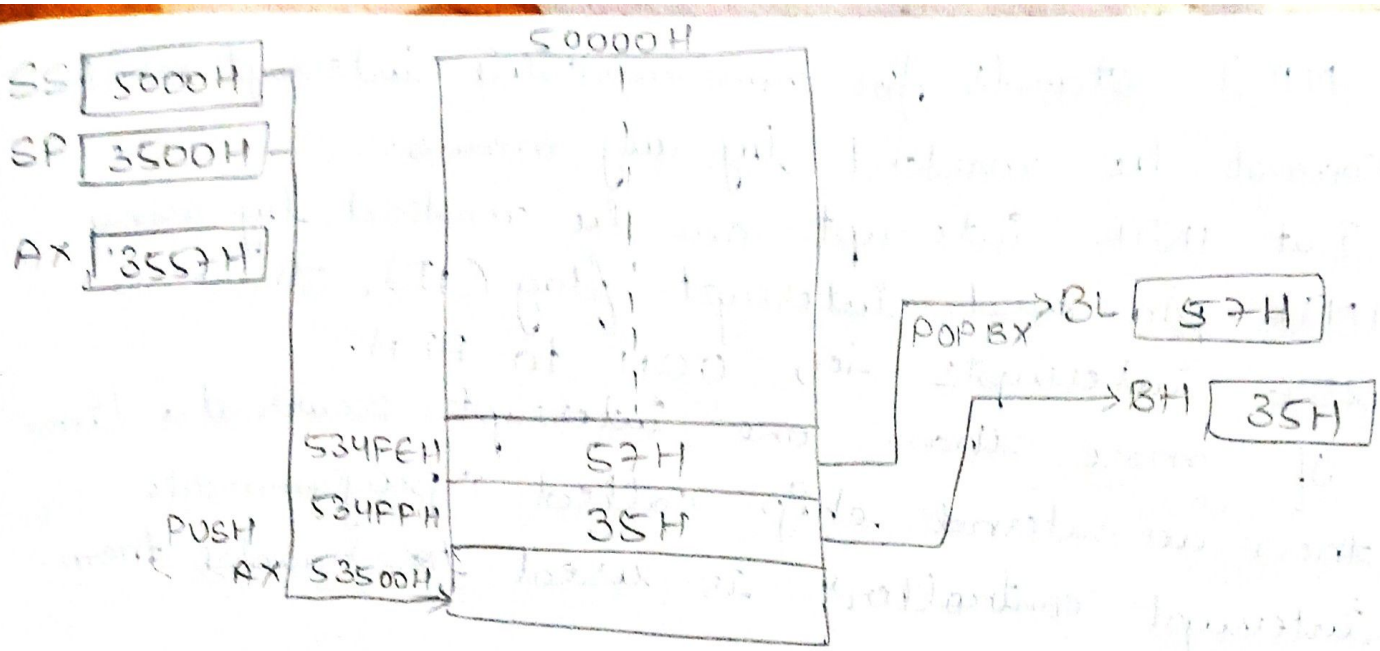
* After successive push operation if the SP (stack pointer) reaches 0000H will result in stack overflow.

* while in the case of POP operation, the stack pointer will increment by 2.

POP is used to retrieve a data from the stack.

=> PUSH AND POP operations are shown in below fig.





Effect of PUSH and POP on SP

INTERRUPTS AND INTERRUPT SERVICE ROUTINES

* The word "Interrupt" means to break the sequence of operation.
* When CPU is executing a program, an interrupt breaks the sequence of operation and diverts to some other program called "Interrupt Service Routine" (ISR)
* After executing ISR, the control is again transferred back to main program which was stopped at the time of interruption.
* When an interrupt appears within a sequence of an interrupt being executed i.e., Interrupt within Interrupt is called "Nested interrupt".
* Handling such multiple interrupts is said to have "multiple interrupt processing capability"
* In 8086, there are 2 interrupt pins "NMI & INTR pins.

* NMI stands for non-maskable interrupt which cannot be masked by any means.
* But INTR interrupt can be masked by using INTR pin and interrupt flag (IF). There are 256 interrupts ie., 00H to FFH.
* If more than one interrupt occurs at a time then an external chip called "programmable interrupt controller" is used to handle them.

INTERRUPT CYCLE OF 8086/8088 :-

Broadly there are two types of interrupts

i) External interrupt :- Interrupts which are caused by external devices like a keyboard interrupt

(ii) Internal interrupt :- Also called software interrupt for example divide by zero interrupt, overflow interrupt, overflow due to INT instructions. etc.,

→ The external interrupts occurs either by NMI or INTR pin of 8086. whereas internal interrupts like divide by zero occurs by a program software.

→ whenever it occurs the CPU first completes the execution of current instruction, IP is then incremented for next instruction

→ The CPU acknowledges the requesting device with $\overline{INTA}$ pin immediately.

→ The CPU first checks the IF flag. If IF = 1: the requested interrupt will be acknowledged with $\overline{INTA}$ pin

If IF = 0; it will ie., CPU will ignore the interrupt and continue its work.

→ Let us consider that IF = 1 and the interrupt device is acknowledged with $\overline{INTA}$ pin, then CPU first stores the next IP and CS address of the main program in the stack and then serves the requested interrupt service routine (ISR)

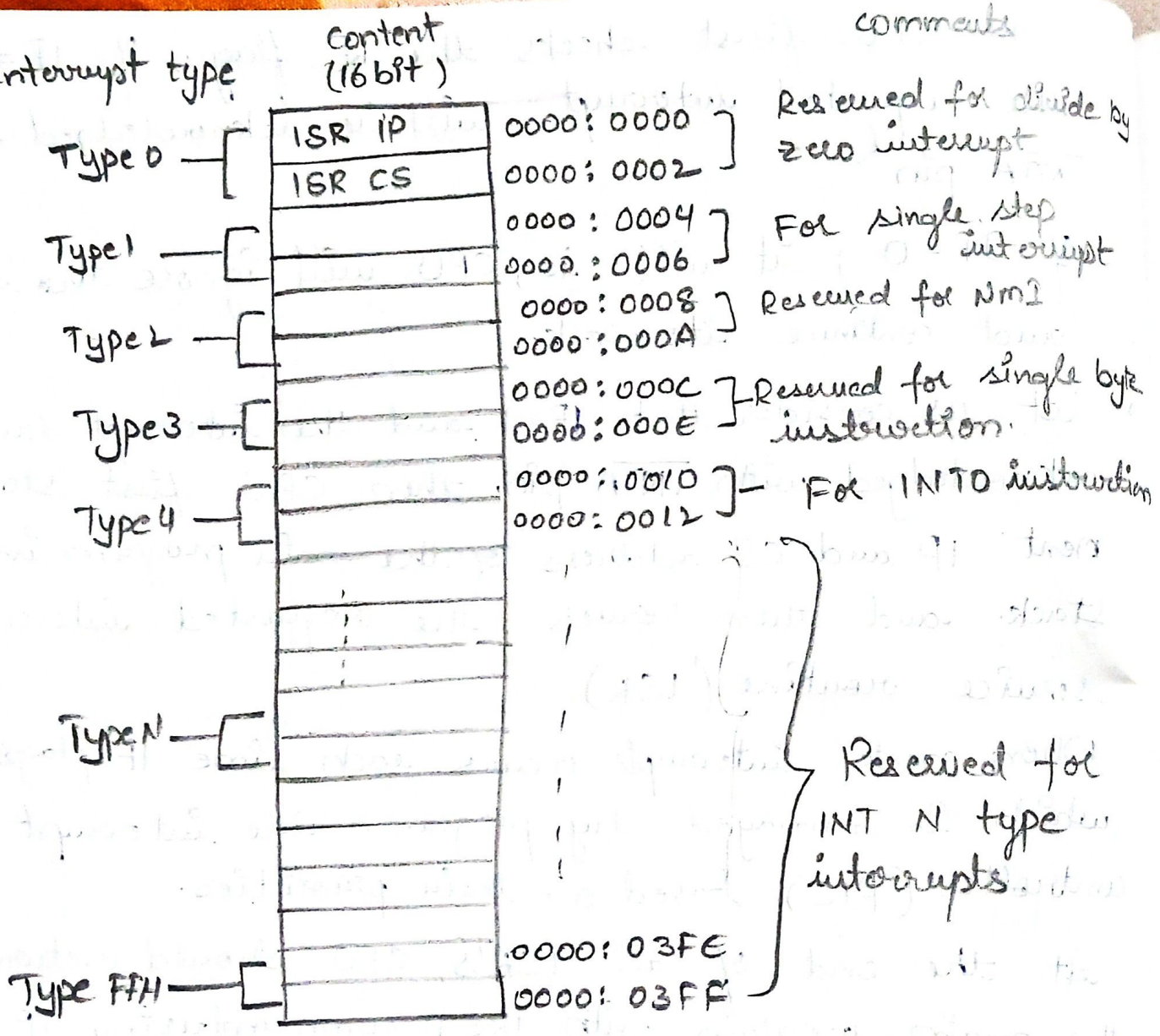
→ When nested interrupts occurs each time IF flags is set which is managed by programmable interrupt controller (PIC) based on their priorities.

→ At the end of all ISR's, CPU should return to the main program with IRET commandusing IP and CS address stored in the stack.

→ There are 256 interrupts and the set of instructions related to each interrupt is found from interrupt vector table.

→ Interrupt vector table consists of IP (offset address) & CS (segment address) related to each interrupt type from 00H to FFH

→ IP is of 16 bits which needs two address locations to reserve, whereas CS is of 16 bits which reserves 2 address locations in interrupt vector table as shown below figure.



∴ Total 1024 bytes are reserved for 256 interrupts from 0000 to 03FFH.

———— ○ ————