Microprocessor and Assembly Language

Chapter-Five

Program Control Instructions

Outline

- The Jump Instruction
- Unconditional Jump (JMP)
- Conditional Jumps
- Controlling the Flow of the Program
- LOOP
- REPEAT
- Procedures
- CALL
- RET
- Introduction to Interrupts

The Jump Instruction

- The program control instructions direct the flow of a program and allow the flow to change.
- A change in flow often occurs after a decision made with the CMP or TEST instruction.
- The main program control instruction, the **jump (JMP)**, allows the programmer to skip sections of a program and branch to any part of the memory for the next instruction.
- It can be classified as;

  I. conditional jump and

  II. unconditional jump.
- A **conditional jump** instruction allows the programmer to make decisions based upon numerical tests.

The Jump Instruction

- The results of numerical tests are held in the flag bits, which are then tested by conditional jump instructions.

Assembly language	Tested Condition	Operation
JNE or, JNZ	Z=0	Jump if not equal or jump if not zero
JE or JZ	Z=1	Jump if equal or jump if zero
JNO	O=0	Jump if no overflow

- In this section of the text, all jump instructions are illustrated with their uses in sample programs.

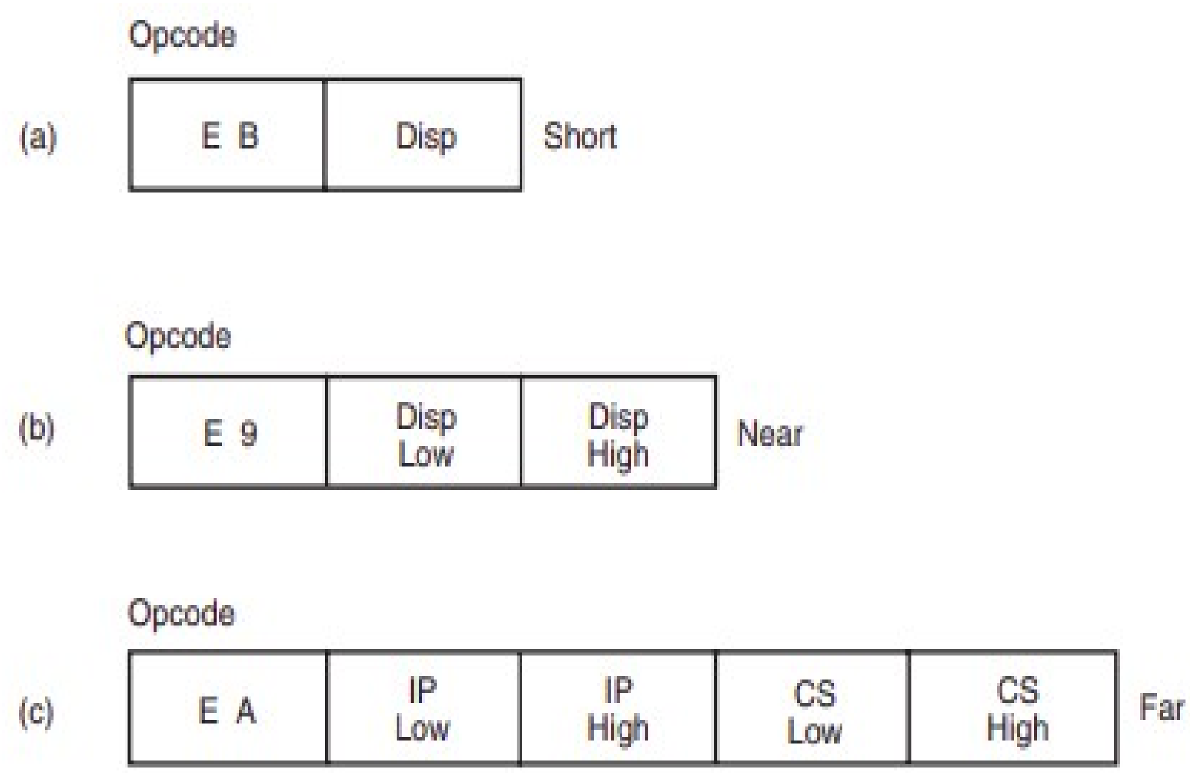
Unconditional Jump (JMP)

- It does not depend any condition or numerical tests.
- Three types of unconditional jump instructions are available:
  1. short jump,
  2. near jump, and
  3. far jump.
- The short and near jumps are often called **intrasegment jumps,** and the far jumps are often called **intersegment jumps**.
- Short jump and near jump follows a distance or displacement to jump where as far jump follows an address (segment + offset) to jump.

Unconditional Jump (JMP)

- Look below example of Short, Near and Far Jumps:

FIGURE 6–1 The three main forms of the JMP instruction. Note that Disp is either an 8- or 16-bit signed displacement or distance.



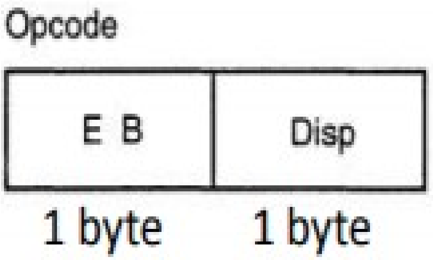
Short Jump

- Short jump is a two-byte instruction.
- They are also called relative jumps.
- Instead of a jump address, a distance, or displacement, follows the opcode.
- The short jump displacement is a distance represented by a 1-byte signed number(8-bit) whose value ranges between +128 and -128.
- It allows jumps or branches to memory location within ±128 from the address following the jump.

2^8 = 256 = +128 byte to -128 bytes

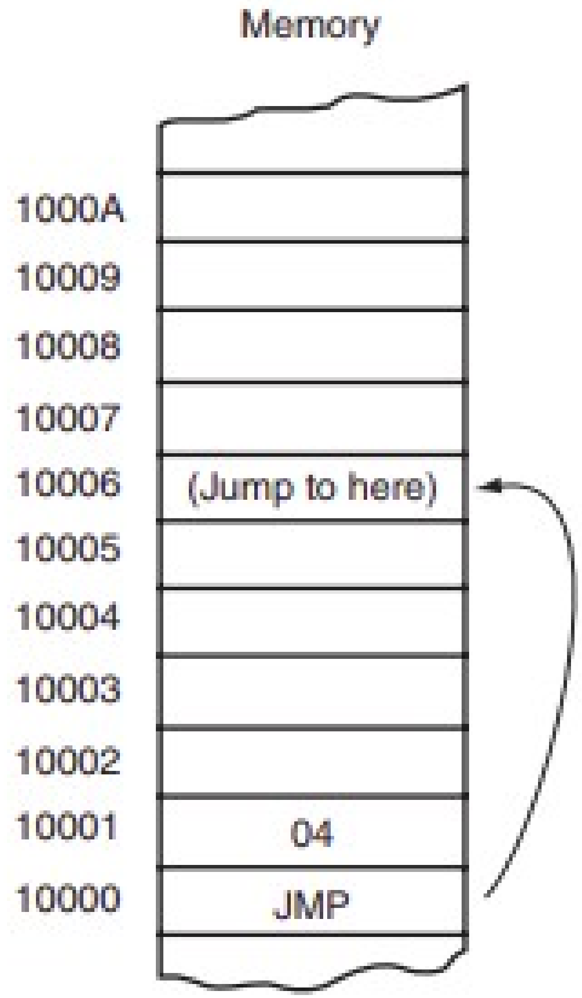
- When the microprocessor executes a short jump, the displacement is sign extended and added to the instruction pointer (IP/EIP) to generate the jump address within the current code segment.
- The short jump instruction branches to this new address for the next instruction in the program.

Example(Short Jump)



JMP disp ; here disp is 8-bit signed displacement or distance

Example:
JMP 04H



CS = 1000H
IP = 0002H
New IP = IP + 4
New IP = 0006H

Cont…..

Example: shows how short jump instructions pass control from one part of the program to another.

```
        XOR BX, BX
START:  MOV AX, 1
        ADD AX, BX
        JMP SHORT NEXT
<skipped memory locations>

NEXT:   MOV BX,AX
        JMP START
```

- Whenever a jump instruction references an address, a label normally identifies the address.
- The JMP NEXT instruction is an example; it jumps to label NEXT for the next instruction.

Cont…

- The label NEXT must be followed by a colon (NEXT:) to allow an instruction to reference it for a jump.
- If a colon does not follow a label, you cannot jump to it.
- Note that the only time a colon is used after a label is when the label is used with a jump or call instruction.
- It is very rare to use an actual hexadecimal address with any jump instruction, but the assembler supports addressing in relation to the instruction pointer by using the **$+a** displacement.
- For example, the **JMP $+2** instruction jumps over the next two memory locations (bytes) following the JMP instruction.

Near Jump

- The near jump is similar to the short jump, except that the distance is farther.
- The near jump is a 3-byte instruction that contains an opcode followed by a 2-byte signed displacement(16-bit).
- A near jump passes control to an instruction in the current code segment located within ±32K bytes from the near jump instruction.

2^{16} = 65536 = 64 Kbyte = +32 Kbyte to -32 Kbyte

- The distance is ±2G in the 80386 and above when operated in protected mode.
- I.e the 80386 through Pentium 4 processors, the displacement is 32 bits and the near jump is 5 bytes long.

2^{32}= 4G = +2G to -2G

Cont...

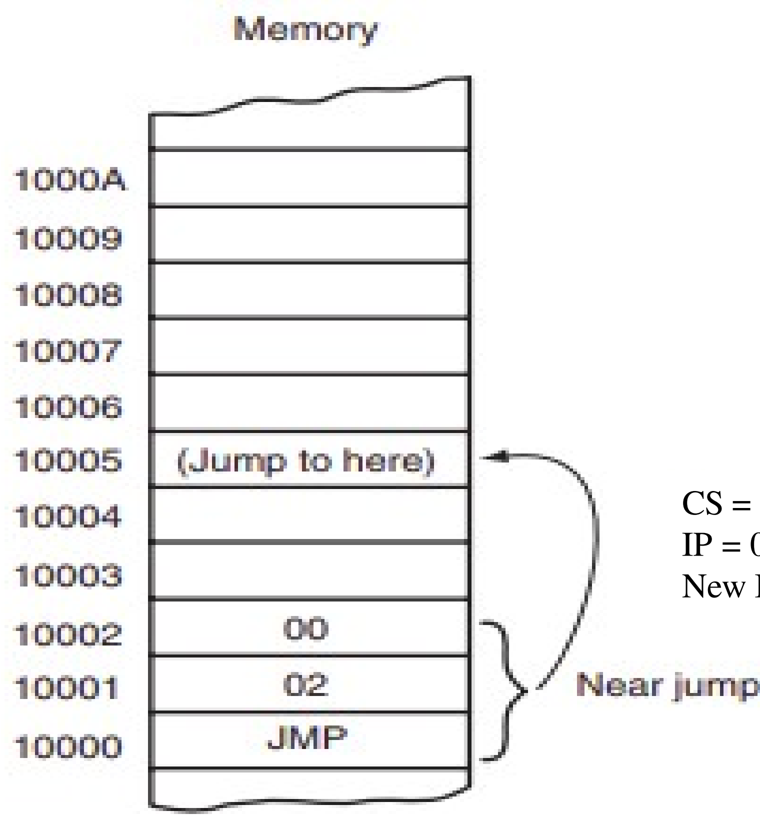
- The signed displacement adds to the instruction pointer(IP) to generate the jump address.
- Because the displacement is in the range of ±32K, near jump can jump to any memory location within the current real mode code segment.

Opcode

E 9	Disp Low	Disp High
1byte	1byte	1byte

Example:

JMP 0002H



CS = 1000H
IP = 0003H
New IP = 0005H

Cont…

- The near jump is also a relative jump.
- If the code segment moves to a new location in the memory, the distance between the jump instruction and the operand address remains the same.
- This allows a code segment to be relocated by simply moving it.
- This feature, along with the relocatable data segments, makes the Intel family of microprocessors ideal for use in a general-purpose computer system.

Example(Near Jump)

Example: below shows the same basic program that appeared in previous example, except that the jump distance is greater.

```
         XOR BX,BX
START:   MOV AX,1
         ADD AX,BX
         JMP NEXT
<skipped memory locations>

NEXT:    MOV BX,AX
         JMP START
```

Far Jump

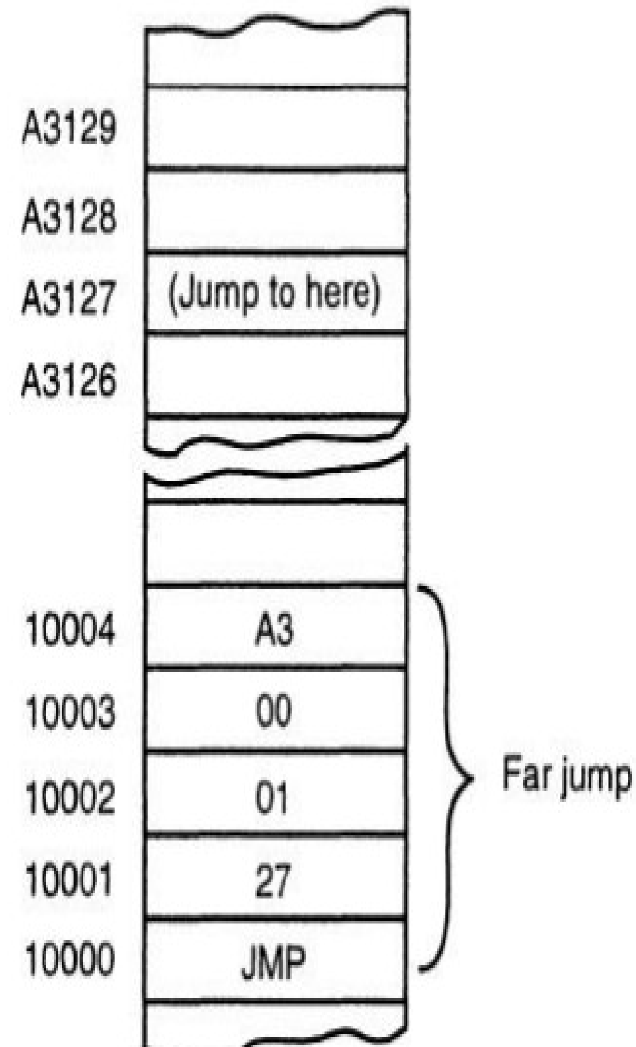
- A **far jump** instruction obtains a new segment and offset address to accomplish the jump.
- It is a 5 byte instruction.
- Bytes 2 and 3 of this 5-byte instruction contain the new offset address; bytes 4 and 5 contain the new segment address.
- If the microprocessor (80286 through the Core2) is operated in the protected mode, the segment address accesses a descriptor that contains the base address of the far jump segment.
- The offset address, which is either 16 or 32 bits, contains the offset address within the new code segment.
- It is always given with **OffsetAddress : SegmentAddress** order along with JMP instruction.

Cont...

- It allows jumps to any memory location of any memory segment. That's why far jump is called **intersegment jump**.

Opcode

E A	IP Low	IP High	CS Low	CS High



Example:

JMP 0127: A300

Jump to CSX10+IP = A300X10+0127 = A3127

Cont...

- ✓ The far jump instruction sometimes appears with the **FAR PTR** directive.
- ✓ Another way to obtain a far jump is to define a label as a **far label**.
- A label is far only if it is external to the current code segment.
- The **JMP UP** instruction in the next slides example references a far label.
- The label UP is defined as a far label by the EXTRN UP:FAR directive.
- External labels appear in programs that contain more than one program file.

Example(Far Jump)

- **Example:** lists a short program that uses a far jump instruction.

EXTERN UP: FAR

indicates UP is a far label

XOR BX,BX

START: ADD AX,1

JMP NEXT

<skipped memory locations>

NEXT: MOV BX,AX

JMP FAR PTR START

references a far label

JMP UP

Jumps with Register Operands

- The jump instruction can also use a 16- or 32-bit register as an operand.
- This automatically sets up the instruction as an **indirect jump**.
- The address of the jump is in the register specified by the jump instruction.
- **Unlike** the displacement associated with the near jump, the contents of the register are transferred directly into the instruction pointer.
- An indirect jump does not add to the instruction pointer, as with short and near jumps.
- The **JMP AX** instruction, for example, copies the contents of the AX register into the IP when the jump occurs.
- This allows a jump to any location **within the current code segment**.

Cont...

- In the 80386 and above, a **JMP EAX** instruction also jumps to any location within the current code segment;

➔ the difference is that in protected mode the code segment can be 4G bytes long, so a 32-bit offset address is needed.

Conditional Jump

- A conditional jump instruction allows the programmer to make decision based upon numerical tests.
- The conditional jump instructions are always short jump in 8086.
- Conditional jump instructions test the following **flag bits**: sign (S), zero (O), carry (C), parity (P) and overflow(O).
- If the condition under test is **true**, a branch to the label associated with the jump instruction occurs.
- If the condition is **false**, the next sequential step in the program executes.

For example, a **JC** will jump if the carry bit is set.

Conditional Jump

- Examples of some common Conditional Jump instructions:

Assembly language	Tested Condition	Operation
JNE or, JNZ	Z=0	Jump if not equal or jump if not zero
JE or JZ	Z=1	Jump if equal or jump if zero
JNO	O=0	Jump if no overflow
JNP or JPO	P=0	Jump if no parity of jump if parity odd
JP or JPE	P=1	Jump if parity or jump if parity even

Example:

```
      MOV CX, 25H
      MOV AX, 1H
      MOV BX, 4H
XXX : ADC AX, BX
      ADD BX, 3H
      DEC CX
      JNZ XXX       ; jump if result (value of CX) not zero
```

This program will execute the instruction inside XXX label(ADC and ADD) until the value of CX becomes 0.

Conditional Jump(Example2)

Assembly language	Tested Condition	Operation
JNE or, JNZ	Z=0	Jump if not equal or jump if not zero
JE or JZ	Z=1	Jump if equal or jump if zero
JNO	O=0	Jump if no overflow
JNP or JPO	P=0	Jump if no parity of jump if parity odd
JP or JPE	P=1	Jump if parity or jump if parity even

Example2:

```
TOP:    ADD CX, BX
        JC  EXIT
        JMP TOP

EXIT:   MOV CX, 00H
        MOV BX, 00H
```

This program will check the value of CF to Jump to EXIT label.

LOOP Instruction

- The loop instruction is a combination of a decrement CX and the JNZ conditional jump. So, it by default executes this two instructions.
- That is, LOOP decrement CX and check for the value of CX, if CX!=0, **it jumps to the address indicated by the label**.
- But if CX becomes 0, the program will proceeds to the next sequential instruction.
- **Example:** Assembly language program to find sum of the following series using LOOP instruction.

1+2+3+.............................+100

Solution:

No of terms=100, No of addition needed=99 (63H)

```
MOV CX, 63H
MOV  AX, 01H
MOV  BX, 02H
SUM:  ADC AX, BX
      ADD BX,01H
      LOOP  SUM
```

In **C++**,

```
Sum = 1
for(i=2, i ≤ 100, i++ {
Sum = Sum + i;
}
```

Loop Instruction(Cont...)

- As we can see from the previous example, by default loop instruction executes;
  - ✓ decrement the count register, CX and
  - ✓ Jump if not Zero, JNZ statements.

Procedures

- The **procedure** (subroutine, method, or function) is an important part of any computer system’s architecture.
- It is a group of instructions that usually performs particular task.
- It allows the same piece of code to be reused multiple times.

Advantage:

- It is a reusable section of the software that is stored in memory once, but used as often as necessary.
- It saves memory space and program development time,
- Makes it easier to develop software.

Disadvantages:

- It takes extra time to link the procedure with the main program and return from it.

Procedures Cont…..

How procedure links with main program?

✓ The CALL instruction **links** to the procedure, and the RET (return) instruction returns from the procedure.

✓ The **stack** stores the return address whenever a procedure is called during the execution of a program.

✓ The CALL instruction **pushes** the address of the instruction following the CALL (return address) on the s**tack.**

What are the specific rules for using procedures?

- A procedure **begins** with the PROC directive and **ends** with the ENDP directive.
- **Each directive** appears with the name of the procedure.
- The PROC directive is then followed by the type of procedure: NEAR (intrasegment) or FAR (intersegment).

Procedures Cont…..

- **Format of Procedure**

XXX **PROC** NEAR/FAR

…………..…………

……………………..

……………………..

RET

XXX **ENDP**

Note:

- XXX is the name of level (name of procedure) and both level name should be same.
- To call a procedure in main program write: **CALL XXX**

Procedures Cont…..

Example:

```
SUMS PROC NEAR
        ADD AX,BX
        ADD AX,CX
        ADD AX,DX
  RET
SUMS ENDP

SUMS1 PROC FAR
         ---------------
         ---------------
 RET
SUMS1 ENDP
```

Then if you want to call this procedure somewhere in your program you can use:

CALL SUMS ;for the first procedure.

CALL SUMS1 ;for the second procedure

Procedures Cont…..

- Procedures that are to be used by all software (global) should be written as **far procedures**.
- Procedures that are used by a given task (local) are normally defined as **near procedures**.
- Most procedures are near procedures.

Don't Forget!

- Once written a procedure in our program how we call it the main program?
  - ✓ We use the CALL instruction along with the procedure name.
  - ✓ That is **CALL Proc_name**.

Procedures Cont…..

CALL instruction

- The CALL instruction transfers the flow of the program to the procedure.
- The **CALL instruction** differs from the **jump instruction** because a CALL saves a return address on the stack.
- The return address returns control to the instruction that immediately follows the CALL in a program when a **RET** instruction executes.

CALL instruction Cont….

- Whenever a CALL instruction executes it:
  - ✓ Pushes the IP or, CS:IP on the stack.
  - ✓ Changes the value of IP or, CS:IP.
  - ✓ Jumps to the procedure by new IP or, CS:IP address.
- Difference between JMP and CALL instruction:
- If a JMP instruction is executed, we jump to the destination location, and the execution carries on from there, without bothering to come back later to the instruction after the JMP.
- If a CALL instruction is executed, we jump to the subroutine, and the execution carries on from there till the RET instruction is executed in the subroutine, and then we come back to the instruction after the CALL in the main program.

CALL instruction Cont….

JMP	CALL
Doesn't use stack	Uses stack
Doesn't return to the next instruction of JMP	Must return to the next instruction of CALL

- Types of CALL

  (a) Near CALL (b) Far CALL

- Difference between Near CALL and Far Call

Near CALL	Far CALL
(1) Procedure located within the same code segment (±32KB)	(1) Procedure located in the entire memory (1 MB)
(2) 3-byte instruction	(2) 5-byte instruction
(3) Only IP content is replaced by (IP±displacement)	(3) Both CS and IP contents are replaced by new CS and IP address
(4) Stack stores only return IP address (2 byte)	(4) Stack stores the return CS and IP address. (4 byte)

RET instruction

- The return (RET) instruction:

✘ removes a 16-bit number (**near return**) from the stack and places it into IP or

✘ removes a 32-bit number (**far return**) and places it into IP and CS.

- The **near** and **far return** instructions are both defined in the procedure’s PROC directive(indirectly), which automatically selects the proper return instruction.

Interrupts

- An **interrupt** is a condition that halts the microprocessor temporarily to work on a different task.
  - An interrupt is either a:
    - ✘ hardware-generated CALL (externally derived from a hardware signal) or
    - ✘ software-generated CALL (internally derived from the execution of an instruction or some other internal event) that allow normal program execution to be interrupted (**stopped**).
  - In response to an interrupt, the microprocessor stops execution of its current program and calls a procedure called **interrupt service procedure (ISP)**.

Interrupts…..

- **How an interrupt is used?**

INT nn; where nn indicates interrupt vector number

nn's value is (0 to 255 i.e. 00H to FFH).

- Each INT instruction is 2-byte long.
  - ✘ 1st byte contain **opcode** and
  - ✘ 2nd byte contains **vector type number**.
- The vector type number is 1byte in size, and hence there are:

  2^8 = 256 different interrupts.

INTO: Interrupt overflow also represented as INT 4

Interrupts…..

TABLE 6–4 Interrupt vectors defined by Intel.

Number	*Address*	*Microprocessor*	*Function*
0	0H–3H	All	Divide error
1	4H–7H	All	Single-step
2	8–BH	All	NMI pin
3	CH–FH	All	Breakpoint
4	10H–13H	All	Interrupt on overflow
5	14H–17H	80186–Core2	Bound instruction
6	18H–1BH	80186–Core2	Invalid opcode
7	1CH–1FH	80186–Core2	Coprocessor emulation
8	20H–23H	80386–Core2	Double fault
9	24H–27H	80386	Coprocessor segment overrun
A	28H–2BH	80386–Core2	Invalid task state segment
B	2CH–2FH	80386–Core2	Segment not present
C	30H–33H	80386–Core2	Stack fault
D	34H–37H	80386–Core2	General protection fault (GPF)
E	38H–3BH	80386–Core2	Page fault
F	3CH–3FH	—	Reserved
10	40H–43H	80286–Core2	Floating-point error
11	44H–47H	80486SX	Alignment check interrupt
12	48H–4BH	Pentium–Core2	Machine check exception
13–1F	4CH–7FH	—	Reserved
20–FF	80H–3FFH	—	User interrupts

Interrupt vector

- When an interrupt occurs, the CPU runs the interrupt handler. But where is this handler found? In the **interrupt vector table**.
- An **interrupt vector** is the memory address of an interrupt handler, or an index into an array called an interrupt vector table.
- It is the **4 byte** long (CS:IP) stored in the first 1024 bytes of the memory (00000H–003FFH) when the microprocessor operates in the real mode. That is,

  Since we have 256 interrupts and each of them is stored in a 4byte memory addresses, and hence needs 256 x 4 = 1024 memory addresses.

  memory address: 0 - 1023 = 00000H – 003FFH.

Interrupt vector…

- Below shows the interrupt vectors and the memory location of each vector for the real mode.
- Each vector contains a value for IP and CS that forms the address of the interrupt service procedure.
- The first 2 bytes contain the IP, and the last 2 bytes contain the CS.

In the interrupt Vector Table (IVT)

INT Number	Physical Address	Contains
INT 00	00000h	IP0:CS0
INT 01	00004h	IP1:CS1
INT 02	00008h	IP2:CS2
. . .	. . .	. . .
INT FF	003FCh	IP255:CS255

Interrupt Vectors…

- As we have said 4byte is required for storing the address of each interrupt, thus:
- The address of the interrupt vector is determined by multiplying the interrupt type number by 4.
- <u>For example</u>, **INT 100H** uses interrupt vector number 100, which appears at memory address 100 X 4 = **400**,

  but this is in decimal so we have to convert into Hexadecimal: I.e **190H**, thus the memory address of INT 100 is **190H – 193H**.

Number	Address
0	0H–3H
1	4H–7H
2	8–BH
3	CH–FH
4	10H–13H
5	14H–17H
6	18H–1BH
7	1CH–1FH
8	20H–23H
9	24H–27H
A	28H–2BH
B	2CH–2FH
C	30H–33H
D	34H–37H
E	38H–3BH
F	3CH–3FH
10	40H–43H
11	44H–47H
12	48H–4BH
13–1F	4CH–7FH
20–FF	80H–3FFH

Interrupt Vectors…

- <u>Example2</u>:Find the memory address/interrupt vector for the following instruction:

INT 10H

Solution:

- This instruction calls the interrupt service procedure whose address is stored beginning at memory location:

4 X 10 = 40 = 28H

- Therefore, INT 10H will be stored at memory addresses: **28H - 31H**.

Number	Address
0	0H–3H
1	4H–7H
2	8–BH
3	CH–FH
4	10H–13H
5	14H–17H
6	18H–1BH
7	1CH–1FH
8	20H–23H
9	24H–27H
A	28H–2BH
B	2CH–2FH
C	30H–33H
D	34H–37H
E	38H–3BH
F	3CH–3FH
10	40H–43H
11	44H–47H
12	48H–4BH
13–1F	4CH–7FH
20–FF	80H–3FFH

Quiz(5%)

1. Find the interrupt address for **INT 23H** instructions.

2. Find the Jump address for the instruction given below:

JMP 1003:3A33

1. Why do we use procedures in assembly program? Justify.

2. Differentiate between Jump and Procedure.