3.1 Introduction

While the CPU is executing a program, an 'interrupt' breaks the normal sequence of execution of instructions, diverts its execution to some other program called Interrupt Service Routine (ISR). After executing ISR, the control is transferred back again to the main program which was being executed at the time of interruption.

Whenever a number of devices interrupt a CPU at a time, and if the processor is able to handle them properly, it is said to have multiple interrupt processing capability. For example, 8085 has five hardware interrupt pins and it is able to handle the interrupts simultaneously under the control of software. In case of 8086, there are two interrupt pins, viz. NMI and INTR. The NMI is a non-maskable interrupt input pin, which means that any interrupt request at NMI input cannot be masked, or disabled by any means. The INTR interrupt, however, may be masked using the interrupt flag (IF). The INTR, further, is of 256 types.

The INTR types may be from 00 to FFH (or 00 to 255). If more than one type of INTR interrupts occurs at a time, then an external chip called programmable interrupt controller is required to handle them. The same is the case for INTR interrupt input of 8085. Interrupt Service Routines (ISRs) are the programs to be executed by interrupting the main program execution of the CPU, after an interrupt request appears. After the execution of ISR, the main program continues its execution further from the point at which it was interrupted.
3.2 INTERRUPT CYCLE OF 8086/8088

Broadly, there are two types of interrupts. The first out of them is external interrupt and the second is internal interrupt. In external interrupt, an external device or a signal interrupts the processor from outside or, in other words, the interrupt is generated outside the processor, for example, a Keyboard Interrupt. The Internal Interrupt, on the other hand, is generated internally by the processor circuit, or by the execution of an interrupt instruction. The examples of this type are divide by zero interrupt, overflow interrupt, interrupts due to INT instructions, etc.

Suppose an external device interrupts the CPU at the interrupt pin, either NMI or INTR of 8086, while the CPU is executing an instruction of a program. The CPU first completes the execution of the current instruction. The IP is then incremented to point to the next instruction. The CPU then acknowledges the requesting device on its INTA pin immediately if it is a NMI, TRAP or Divide by Zero interrupt. If it is an INT request, the CPU checks the IF flag. If the IF is set, the interrupt request is acknowledged using the OOA pin. If the IF is not set, the interrupt requests are ignored. Note that the responses to the NMI, TRAP and Divide-by-Zero interrupt requests are independent of the IF flag. After an interrupt is acknowledged, the CPU computes the vector address from the type of the interrupt that may be passed to the interrupt structure of the CPU internally (in case of software interrupts, NMI, TRAP and Divide by Zero interrupts) or externally, i.e. from an interrupt controller in case of external interrupts. (The contents of IP and CS are next pushed to the stack. The contents of IP and CS now point to the address of the next instruction of the main program from which the execution is to be continued after executing the ISR. The PSW is also pushed to the stack). The interrupt flag (IF) is cleared. The TF is also cleared, after every response to the single step interrupt. The control is then transferred to the interrupt service routine for serving the interrupting device. The new address of ISR is found out from the interrupt vector table. The
execution of the ISR starts. If further interrupts are to be responded to during the time the first interrupt is being serviced, the IF should again be set to 1 by the ISR of the first interrupt. If the interrupt flag is not set, the subsequent interrupt signals will not be acknowledged by the processor, till the current one is completed. The programmable interrupt controller is used for managing such multiple interrupts based on their priorities. At the end of ISR the last instruction should be IRET. When the CPU executes IRET, the contents of flags, IP and CS which were saved at the start by the CALL instruction are now retrieved to the respective registers. The execution continues onwards from this address, received by IP and CS.

At the end of each instruction cycle, the 8086 checks to see if any interrupts have been requested. If an interrupt has been requested, the 8086 responds to the interrupt by stepping through the following series of major actions.

- It decrements the stack pointer by 2 and pushes the flag register on the stack.
- It disables the 8086 INTR interrupt input by clearing the interrupt flag (IF) in the flag register.
- It resets the trap flag (TF) in the flag register.
- It decrements the stack pointer by 2 and pushes the current code segment register contents on the stack.
- It decrements the stack pointer again by 2 and pushes the current instruction pointer contents on the stack.
- It does an indirect far jump to the start of the procedure you wrote to respond to the Interrupt.

Following figure summarizes these steps in diagram form. As you can see, the 8086 pushes the flag register on the stack disables the INTR input and the single-step function, and does essentially an indirect far call to the interrupt
service procedure. An IRET instruction at the end of the interrupt service
procedure returns execution to the main program.

![8086/8088 Interrupt Response](image)

We now discuss about how the 8086/88 finds out the address of an ISR. Every external and internal interrupt is assigned with a type (N), that is either
implicit (in case of NMI, TRAP and divide by zero) or specified in the instruction
INT N (in case of internal interrupts). In case of external interrupts, the type is
passed to the processor by an external hardware like programmable interrupt
controller. In the zeroth segment of physical address space, i.e. CS = 0000, Intel
has reserved 1024 locations for storing the interrupt vector table. The 8086
supports a total of 256 types of the interrupts, i.e. from 00 to FFH. Each interrupt
requires 4 bytes, i.e. two bytes each for IP and CS of its ISR. Thus a total of 1024
bytes are required for 256 interrupt types, hence the interrupt vector table starts
at location 0000:0000 and ends at 0000:03FFH. The interrupt vector table contains
the IP and CS of all the interrupt types stored sequentially from address
0000:0000 to 0000:03FFH. The interrupt type N is multiplied by 4 and the
hexadecimal multiplication obtained gives the offset address in the zeroth code
segment at which the IP and CS addresses of the interrupt service routine (ISR)
are stored. The execution automatically starts from the new CS:IP.

Following figures shows the interrupt vector table.
3.3 NON MASKABLE INTERRUPT

The processor 8086/88 has a non-maskable interrupt input pin (NMI), that has the highest priority among the external interrupts. TRAP(Single Step-Type 1) is an internal interrupt having the highest priority amongst all the interrupts except the Divide By Zero (Type 0) exception. The NMI is activated on a positive transition (low to high voltage). The assertion of the NMI interrupt is equivalent to an execution of instruction INT 02, i.e. Type 2 INTR interrupt.

The NMI pin should remain high for at least two clock cycles and is no need to be synchronized with the clock for being sensed. When NMI is activated, the current instruction being executed is completed, and then the NMI is served.
In case of string type instructions, this interrupt will be served only after the complete string has been manipulated.

Another high going edge on the NMI pin of 8086, during the period, in which the first NMI is served, triggers another response. The signal on the NMI pin must be free of logical bounces to avoid erratic NMI responses.

### 3.4 MASKABLE INTERRUPT (INTR)

The processor 8086/88 also provides a pin INTR, that has lower priority as compared to NMI. Further the priorities, within the INTR types are decided by the type of the INTR signal, that is to be passed to the processor via data bus by some external device like the programmable interrupt controller. The INTR signal is level triggered and can be masked by resetting the interrupt flag. It is internally synchronized with the high transition of CLK. For the INTR signal, to be responded to in the next instruction cycle, it must go high in the last clock cycle of the current instruction or before that. The INTR requests appearing after the last clock cycle of the current instruction will be responded to after the execution of the next instruction. The status of the pending interrupts is checked at the end of each instruction cycle.

If the IF is reset, the processor will not serve any interrupt appearing at this pin. If the IF is set, the processor is ready to respond to any INTR interrupt. Once the processor responds to an INTR signal, the IF is automatically reset. If one wants the processor to further respond to any type of INTR signal, the IF should again be set. The interrupt acknowledge sequence is as shown in Fig.
Suppose an external signal interrupts the processor and the pin LOCK goes low preventing the use of bus for any other purpose. The pin LOCK goes low at the trailing edge of the first ALE pulse that appears after the interrupt signal. The pin LOCK remains low till the start of the next machine cycle. With the trailing edge of LOCK, the OOA goes low and remains low for two clock states before returning back to the high state. It remains high till the start of the next machine cycle, i.e. next trailing edge of ALE. Then OOA again goes low, remains low for two states before returning to the high state. The first trailing edge of ALE floats the bus ADO-AD7, while the second trailing edge prepares the bus to accept the Type of the interrupt. The Type of the interrupt remains on the bus for a period of two cycles.

3.5 INTERRUPT PROGRAMMING

While programming for any type of interrupt, the programmer must, either externally or through the program, set the interrupt vector table for that type suitably with the CS and IP addresses of the interrupt service routine. The method of defining the interrupt service routine for software as well as hardware interrupt is the same. Figure shows the execution sequence in case of a software interrupt. It is assumed that the interrupt vector table is initialized suitably to point to the interrupt service routine. Figure 4.8 shows the transfer of control for the nested interrupts.

Fig: Transfer of Control during Execution of an ISR (Interrupt Service Routine)
Example Program: To read a string from the keyboard and convert the characters into upper case letters and display on the screen with enough messages displayed in between.

data segment
msgl db 'Enter the String ',0AH,0Dh,'$
msg2 db 0ah,'The String in Caps is : ','$
str db 80 dup(0)
data ends
code segment
    assume cs:code, ds:data
start: mov ax,data
    mov ds,ax
    lea dx,msg1
    mov ah,09h
    int 21h
    mov bx,offset str
    mov str[bx],0ah
    inc bx
    up:   mov ah,01
           int 21h
           cmp al,0Dh
           je stop
           cmp al,60H
           jc dwn
           sub al,20H
    dwn:  mov [bx],al
           inc bx
           jmp up
    stop:mov str[bx],'$'
           mov dx, offset msg2
           mov ah,09h
           int 21h
           mov dx,offset str
           mov ah,0ah
           int 21h
           mov ah,4ch
           int 21h
    code ends
    end start

3.6 8086 Interrupt Types

The preceding sections used the type 0 interrupt as an example of how the 8086 interrupts function. In this section we discuss in detail the different ways an 8086 can be interrupted and how the 8086 responds to each of these interrupts. We discuss these in order starting with type 0.

3.6.1 DIVIDE-BY-ZERO INTERRUPT-TYPE 0

As we described in the preceding section, the 8086 will automatically do a type 0 interrupt if the result of a DIV operation or an IDIV operation is too large to fit in the destination register. For a type 0 interrupt, the 8086 pushes the flag register on the stack, resets IF and TF and pushes the return address (CS and IP) on the stack. It then gets the CS value for the start of the interrupt-service
procedure from address 00002H in the interrupt-pointer table and tile IP value for the start of the procedure from address 00000H in the interrupt pointer-table.

Since the 8086 type 0 response is automatic and cannot be disabled in any way, you have to account for it in any program where you use the DIV or IDIV instruction. One way is to in some way make sure the result will never be too large for the result register. We first make sure the divisor is not zero and then we do the division in several steps so that the result of the division will never be too large.

Another way to account for the 8086 type 0 response is to simply write an interrupt-service procedure, which takes the desired action when an invalid division occurs. The advantage of this approach is that you don't have the overhead of a more complex division routine in your mainline program. The 8086 automatically does the checking and does the interrupt procedure only if there is a problem.

3.6.2 SINGLE-STEP INTERRUPT-TYPE 1

When you tell a system to single-step, it will execute one instruction and stop. You can then examine the contents of registers and memory locations. If they are correct, you can tell the system to go on and execute the next instruction. In other words, when in single-step mode, a system will stop after it executes each instruction and wait for further direction from you. The 8086 trap flag and type 1 interrupt response make it quite easy to implement a single-step feature in an 8086-based system.

If the 8086 trap flag is set, the 8086 will automatically do a type 1 interrupt after each instruction executes. When the 8086 does a type 1 interrupt, it pushes the flag register on the stack, resets TF and IF, and pushes the CS and IP values for the next instruction on the stack. It then gets the CS value for the start of the type 1 interrupt-service procedure from address 00006H and it gets the IP value for the start of the procedure from address 00004H.
The tasks involved in implementing single stepping are: Set the trap flag, write an interrupt-service procedure which saves all registers on the stack, where they can later be examined or perhaps displayed on the CRT and load the starting address of the type 1 interrupt-service procedure into addresses 00004H and 00006H.

The actual single-step procedure will depend very much on the system on which it is to be implemented. We do not have space here to show you the different ways to do this. We will however see how the trap flag is set or reset.

The 8086 has no instructions to directly set or reset the trap flag. These operations are done by pushing the flag register on the stack changing the trap flag bit to what you want it to be, and then popping the flag register back off the stack. Here is the instruction sequence to set the trap flag.

```
PUSHF     ; Push flags on stack
MOV BP, SP   ; Copy SP to BP for use as index
OR WORD PTR [BP+0], 0100H  ; Set TF bit
POPF      ; Restore flag register
```

To reset the trap flag, simply replace the OR instruction in the preceding sequence with the instruction AND WORD PTR [BP + 0],0FEFFH.

NOTE: We have to use [BP+0] because BP cannot be used as a pointer without a displacement.

The trap flag is reset when the 8086 does a type 1 interrupt, so the single-step mode will be disabled during the interrupt-service procedure.

### 3.6.3 NON-MASKABLE INTERRUPT-TYPE 2

The 8086 will automatically do a type 2 interrupt response when it receives a low-to-high transition on its NMI input pin. When it does a type 2 interrupt, the 8086 will push the flags on the stack, reset TF and IF and push the CS value and the IP value for the next instruction on the stack. It will then get the CS value for the start of the type 2 interrupt-service procedures from address 0000AH and the IP value for the start of the procedure from address 00008H.
The name non-maskable given to this input pin on the 8086 means that the type 2 interrupts response cannot be disabled (masked) by any program instructions. Because this input cannot be intentionally or accidentally disabled, we use it to signal the 8086 that some condition in an external system must be taken care of. We could for example have a pressure sensor on a large steam boiler connected to the NMI input. If the pressure goes above some preset limit, the sensor will send an interrupt signal to the 8086. The type 2 interrupt-service procedures for this case might turn off the fuel to the boiler open a pressure-relief valve. And sound an alarm.

Another common use of the type 2 interrupt is to save program data in case of a system power failure. Some external circuitry detects when the ac power to the system fails and sends an interrupt signal to the NMI input. Because of the large filter capacitors in most power supplies the dc system power will remain for perhaps 50 ms after the ac power is gone. This is more than enough time for type 2 interrupt-service procedures to copy program data to some RAM, which has a battery backup power supply. When the ac power returns program data can be restored from the battery-backed RAM. and the program can resume execution where it left off. A practice problem at the end of the chapter gives you a chance to write a simple procedure for this task.

3.6.4 BREAKPOINT INTERRUPT-TYPE 3

The type 3 interrupt is produced by execution of the INT 3 instruction. The main use of the type 3 interrupt is to implement a breakpoint function in a system. When you insert a breakpoint, the system executes the instructions up to the breakpoint and then goes to the breakpoint procedure. Unlike the single-step feature, which stops execution after each instruction. The breakpoint feature executes all the instructions up to the inserted breakpoint and then stops execution.
When you tell most 8086 systems to insert a breakpoint at some point in your program, they actually do it by temporarily replacing the instruction byte at that address with CCH, the 8086 code for the INT 3 instruction.

### 3.6.5 OVERFLOW INTERRUPT-TYPE 4

The 8086 overflow flag (OF) will be set if the signed result of an arithmetic operation on two signed numbers is too large to be represented in the destination register or memory location. For example, if you add the 8-bit signed number 01101100 (108 decimal) and the 8-bit signed number 01010001 (81 decimal), the result will be 10111101 (189 decimal). This would be the correct result if we were adding unsigned binary numbers. But it is not the correct signed result. For signed operations the 1 in the most significant bit of the result indicates that the result is negative and in 2's complement form.

The result, 10111101, then actually represents -67 decimal, which is obviously not the correct result for adding +108 and +89. There are two major ways to detect and respond to an overflow error in a program. One way is to put the Jump if Overflow instruction, JO, immediately after the arithmetic instruction. If the overflow flag is set as a result of the arithmetic operation, execution will jump to the address specified in the JO instruction. At this address you can put an error routine which responds to the overflow in the way you want.

The second way of detecting and responding to an overflow error is to put the Interrupt on Overflow instruction, INTO, immediately after the arithmetic instruction in the program. If the overflow flag is not set when the 8086 executes the INTO instruction, the instruction will simply function as an NOP. However, if the overflow flag is set, indicating an overflow error, the 8086 will do a type 4 interrupt after it executes the INTO instruction. When the 8086 does a type 4 interrupt, it pushes the flag register on the stack, resets TF and IF, and pushes the CS and IP values for the next instruction on the stack, it then gets the CS value
for the start of the interrupt-service procedure from address 00012H and the IP value for the procedure from address 00010H.

3.7 SOFTWARE INTERRUPTS-TYPES 0 THROUGH 255

The 8086 INT instruction can be used to cause the 8086 to do anyone of the 256 possible interrupt types. The desired interrupt type is specified as part of the instruction. The instruction INT 32, for example, will cause the 8086 to do a type 32 interrupt response. The 8086 will push the flag register on the stack, reset TF and IF, and push the CS and IP values of the next instruction on the stack. It will then get the CS and IP values for the start of the interrupt-service procedure from the interrupt-pointer table in memory. The IP value for any interrupt type is always at an address of 4 times the interrupt type, and the CS value is at a location two addresses higher. For a type 32 interrupt, then, the IP value will be put at 4 x 32 or 128 decimal (80H), and the CS value will be put at address 82H in the interrupt vector table.

Software interrupts produced by the INT instruction have many uses. In a previous section we discussed the use of the INT 3 instruction to insert breakpoints in programs for debugging. Another use of software interrupts is to test various interrupt-service procedures. You could, for example, use an INT 0 instruction to send execution to a divide-by-zero interrupt-service procedure without having to run the actual division program.

Fig: Block Diagram showing an 8259 connected to an 8086

Compiled by: Dr. Manoj V.N.V.
3.8 INTR INTERRUPTS-TYPES 0 THROUGH 255

The 8086 INTR input allows some external signal to interrupt execution of a program. Unlike the NMI input, however, INTR can be masked (disabled) so that it cannot cause an interrupt. If the interrupt flag (IF) is cleared, then the INTR input is disabled. IF can be cleared at any time with the Clear Interrupt instruction, CLI. If the interrupt flag is set, the INTR input will be enabled. IF can be set at any time with the Set Interrupt instruction, STI.

When the 8086 is reset, the interrupt flag is automatically cleared. Before the 8086 can respond to an interrupt signal on its INTR input, you have to set IF with an STI instruction. The 8086 was designed this way so that ports, timers, registers, etc., can be initialized before the INTR input is enabled. In other words, this allows you to get the 8086 ready to handle interrupts before letting an interrupt in, just as you might want to get yourself ready in the morning with a cup of coffee before turning on the telephone and having to cope with the interruptions it produces.

Remember that the interrupt flag (IF) is also automatically cleared as part of the response of an 8086 to an interrupt. This is done for two reasons. First, it prevents a signal on the INTR input from interrupting a higher-priority interrupt-service procedure in progress. However, if you want another INTR input signal to be able to interrupt a procedure in progress, you can reenable the INTR input with an STI instruction at any time.

The second reason for automatically disabling the INTR input at the start of an INTR interrupt-service procedure is to make sure that a signal on the INTR input does not cause the 8086 to interrupt itself continuously. The INTR input is activated by a high level, in other words, whenever the INTR input is high and INTR is enabled, the 8086 will be interrupted. If INTR were not disabled during the first response, the 8086 would be continuously interrupted and would never get to the actual interrupt-service procedure.
The IRET instruction at the end of an interrupt-service procedure restores the flags to the condition they were in before the procedure by popping the flag register off the stack. This will re-enable the INTR input. If a high-level signal is still present on the INTR input, it will cause the 8086 to be interrupted again. If you do not want the 8086 to be interrupted again by the same input signal, you have to use external hardware to make sure that the signal is made low again before you re-enable INTR with the STI instruction or before the IRET from the INTR service procedure.

When the 8086 responds to an INTR interrupt signal, its response is somewhat different from its response to other interrupts. The main difference is that for an INTR interrupt the interrupt type is sent to the 8086 from an external hardware device such as the 8259A priority interrupt controller as shown in Figure below. When an 8259A receives an interrupt signal on one of its IR inputs, it sends an interrupt request signal to the INTR input of the 8086. If the INTR input of the 8086 has been enabled with an STI instruction, the 8086 will respond as shown by the waveforms in Figure.

The 8086 first does two interrupt-acknowledge machine cycles as shown in Figure. The purpose of these two machine cycles is to get the interrupt type from the external device. At the start of the first interrupt-acknowledge machine cycle, the 8086 floats the data bus lines, AD0-AD15, and sends out an interrupt-acknowledge pulse on its INTA output pin. This pulse essentially tells the 8259A to "get ready." During the second interrupt-acknowledge machine cycle, the 8086 sends out another pulse on its INTA output pin. In response to this second INTA pulse,
the 8259A puts the interrupt type (number) on the lower eight lines of the data bus, where it is read by the 8086.

Once the 8086 receives the interrupt type, it pushes the flag register on the stack, clears TF and IF and pushes the CS and IP values of the next instruction on the stack. It then uses the type it read in from the external device to get the CS and IP values for the interrupt-service procedure from the interrupt-pointer table in memory. The IP value for the procedure will be put at an address equal to 4 times the type number, and the CS value will be put at an address equal to 4 times the type number plus 2, just as is done for the other interrupts.

The advantage of having an external device insert the desired interrupt type is that the external device can "funnel" interrupt signals from many sources into the INTR input pin on the 8086. When the 8086 responds with INTA pulses, the external device can send to the 8086 the interrupt type that corresponds to the source of the interrupt signal.

### 3.9 PRIORITY OF 8086 INTERRUPTS

As you read through the preceding discussions of the different interrupt types, the question that may have occurred to you is. What happens if two or more interrupts occur at the same time? The answer to this question is that the highest-priority interrupt will be serviced first and then the next-highest-priority interrupt will be serviced. Figure below shows the priorities of the 8086 interrupts as shown in the Intel data book.

Some examples will show you what these priorities actually mean. As a first example, suppose that the INTR input is enabled. the 8086 receives an INTR signal during execution of a Divide instruction, and the divide operation produces a divide-by-zero interrupt. Since the internal interrupts-such as divide error, INT, and INTO -have higher priority than INTR. the 8086 will do a divide error (type 0) interrupt response first. Part of the type 0 interrupt response is to clear IF.
This disables the INTR input and prevents the INTR signal from interrupting the higher-priority type 0 interrupt-service procedure. An IRET instruction at the end of the type 0 procedure will restore the flags to what they were before the type 0 response. This will reenable the INTR input, and the 8086 will do an INTR interrupt response. A similar sequence of operations will occur if the 8086 is executing an INT or INTO instruction and an interrupt signal arrives at the INTR input. As a second example of how this priority works, suppose that a rising-edge signal arrives at the NMI input while the 8086 is executing a DIY instruction, and that the division operation produces a divide error.

Since the 8086 checks for internal interrupts before it checks for an NMI interrupt, the 8086 will push the flags on the stack, clear TF and IF, push the return address on the stack, and go to the start of the divide error (type 0) service procedure. However, because the NMI interrupt request is not disabled, the 8086 will then do an NMI (type 2) interrupt response. In other words, the 8086 will push the flags on the stack, clear TF and IF, push the return address on the stack, and go execute the NMI interrupt-service procedure. When the 8086 finishes the NMI procedure, it will return to the divide error procedure, finish executing that procedure, and then return to the mainline program.

<table>
<thead>
<tr>
<th>INTERRUPT</th>
<th>PRIORITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIVIDE ERROR, INT n, INTO</td>
<td>Highest</td>
</tr>
<tr>
<td>NMI</td>
<td></td>
</tr>
<tr>
<td>INTR</td>
<td></td>
</tr>
<tr>
<td>SINGLE-STEP</td>
<td>Lowest</td>
</tr>
</tbody>
</table>

FIGURE Priority of 8086 interrupts. (Intel Corporation)