Chapter 3 Assembly Language Programming with 8086

UNIT - III

Assembly Language Programming with 8086- Machine level programs, Machine coding the programs, Programming with an assembler, Assembly Language example programs.

Stack structure of 8086, Interrupts and Interrupt service routines, Interrupt cycle of 8086, Interrupt programming, Passing parameters to procedures, Macros, Timings and Delays.

Lecture Number	Topic to be covered	Student Learning Outcomes	URLs
1.	Machine level programs	Student will be able to write machine level programs.	http://www.nptel.ac.in/courses/Webcourse-contents/IISc-BANG/Microprocessors%20and%20Microcontrollers/pdf/, https://www.youtube.com/watch?v=HXYhBCpDoVc
2.	Machine coding the programs	Student will be able to cod e the ALP for various problems.	http://www.nptel.ac.in/courses/Webcourse-contents/IISc-BANG/Microprocessors%20and%20Microcontrollers/pdf/,
3.	Programming with an assembler, Assembly Language example programs	Student will be able to work with Assembler.	http://www.nptel.ac.in/courses/Webcourse-contents/IISc-BANG/Microprocessors%20and%20Microcontrollers/pdf/
4.	Stack structure of 8086	Student will be able to explain functioning of stack for 8086 processor.	https://www.youtube.com/watch?v=d-2Peb3pCBg
5.	Interrupts and Interrupt service routines	Student will be able to differentiate different types of interrupts.	https://www.youtube.com/watch?v=C02weCM9yWA http://www.dauniv.ac.in/downloads/EmbsysRevEd_PPTs/
6.	Interrupt cycle of 8086	Student will be able to demonstrate interrupt cycle.	https://www.youtube.com/watch?v=C02weCM9yWA http://www.dauniv.ac.in/downloads/EmbsysRevEd_PPTs/
7.	Interrupt programming	Student will be able to develop various ALP programs using an interrupt service routines	https://www.youtube.com/watch?v=C02weCM9yWA http://www.dauniv.ac.in/downloads/EmbsysRevEd_PPTs/
8.	Passing parameters to procedures	Student will be able to explain parameter passing techniques.	https://www.youtube.com/watch?v=K4YMXyRcWaI
9.	Macros, Timings and Delays.	Student will be able to analyze the timing delays in 8086 processor.	https://www.youtube.com/watch?v=K4YMXyRcWaI

Practical Inferences

Develop/ Write Assembly Language Programs to solve various problems.

Notes:

3. Assembly language program with 8086

3.1. A Few Machine Level Programs:

- A few machine level programming examples, rather, instruction sequences are presented for comparing the 8085 programming with that of 8086.
- This programs are in the form of instructions sequences just like 8085 programs.
- These may even be hand coded, entered byte by byte and executed on an 8086 based system due to complex instruction set of 8086 and its tedious opcode conversion procedure, most of the programmers prefer to use assemblers.

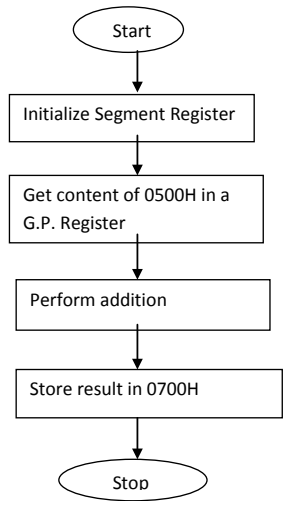
Example 3.1

Write a program to add a data byte located at offset 0500H in 2000H segment to another data byte available at 0600H in the same segment and store the result at 0700H in the same segment.

Solution:

The flow chart for this problem may be drawn as shown below.

```
MOV  AX.2000H : Initializing DS with value
MOV  DS,AX    : 2000H
MOV  AX,[500H] : Get first data byte from 0500H
                 Offset
ADD  AX,[600H] : Add this to the second byte
                 From 0600H
MOV [700H], AX  : Store AX in 0700H (result).
HLT             : Stop
```



- The above instruction sequence is quite straightforward. As the immediate data cannot be loaded into a segment register, the data is transferred to one of the general purpose registers, say AX, and then the register content is moved to the segment register DS.
- Thus the data segment register DS contains 2000H.

- ❖ The instruction MOV AX,[500H] signifies that the contents of the particular location, whose offset is specified in the brackets with the segment pointed to by DS as segment register,is to be moved to AX.
- ❖ The MOV [0700H],AX instruction moves the contents of the register AX to an offset 0700H in DS (DS=2000H).
- ❖ The opcode in the first option is only of 2 bytes, while the second option will have 4 bytes of opcode. Thus the second option will require more memory and execution time.
- ❖ The immediate data byte 05H is added to the content of 0600H using the ADD instruction. The result will be in the destination operand 0600H.
- ❖ This is next stored at the location 0700H. In case of the 8086/8088 instruction set, there is no instruction for the direct transfer of data from the memory source operand to the memory destination operand except, the string instructions.
- ❖ Hence the result of addition which is present at 0600H,should be moved to any one of the general purpose registers, except BX and CX, otherwise the contents of CX and BX will be changed. We have selected DX for this purpose.

3.2 Machine Coding the Programs:

- ❖ Most of the instructions either have specific opcodes or they can be decided only by setting the S,V,W,D,REG,MOD and R/M fields suitably but the critical point is the calculation of jump addresses for intersegment branch instructions.

Example 3.1.2:

MOV BL, CL

For hand coding this instruction, we will have to first note down the following features:

(i) It fits in the register/Memory to/from register format.

(ii) It is an 8-bit operation.

(iii) BL is the destination register and CL is the source register.

Now from the feature (i) the opcode format is given as shown.

D7	D6	D5	D4	D3	D2	D1	D0	D7D6	D5D4D3	D2D1D0
1	0	0	0	1	0	d	w	[MOD]	[REG]	[R/M]

If d=1, then transfer of data is to the register shown by the REG field, i.e., the destination is a register (REG).

Id d=0, the source is a register shown by the REG field.

It is an operation, hence w bit is 0. If it had been a 16-bit operation, the w bit would have been 1.

Refer to addressing modes, to search the REG to REG addressing in it, i.e., the last column with MOD 11. According to the data sheet when MOD is 11, the R/M field is treated as a REG field. The REG field is used for source register and the R/M field is used for the destination register, if d is 0. If d=1, the REG field is used for destination and the R/M field is used to indicate source.

Now the complete machine code of this instruction comes out to be

MOV BL, CL

D7		D0	D7		D0
Code	d	w	MOD	REG	R/M
1 0 0 0 1 0	w	0	1 1	0 0 1	0 1 1

3.2.1 Finding out Machine Code for Conditional JUMP (Intrasegment) Instructions:

- To each of the conditional jump instructions, the first byte of the opcode is fixed and the jump displacement must be less than or equal to 127(D) bytes and greater than or equal to -128(D).
- The following example explains how to find the displacement.
- The displacement is an 8-bit signed number. If it is positive, it indicates a forward jump, otherwise it indicates a backward jump.
- The following example is a sequence of instructions rather than a single instruction to elaborate the procedure of the calculation of positive displacement for a forward jump.

Example 3.1.2.1

```
2000 , 01        XOR AX,BX
2002,03          JNZ OK
2004             NOP
2005             NOP
2006 , 7, 8, 9   ADD BX, 05H
200A             OK : HLT
```

- The above sequence shows that the programmer wants a conditional jump to label OK, if the zero flag is not set.
- For finding out the displacement corresponding to the label OK, subtract the address of the jump instruction (2002H), from the address of label (200AH).
- The required displacement is 200AH – 2002H = 08H. The 08H is the displacement for the forward jump.

Let us find out the displacement for a backward jump. Consider the following sequence of instructions.

Example 3.1.2.2

2000, 01, 02	MOV CL, 05H
2003	Repeat : INC AX
2004	DEC CL
2005, 2006	JNZ Repeat

- For finding out the backward displacement, subtract the address of the label (repeat) from the address of the jump instruction. Complement the subtraction.
- The lower byte gives the displacement .
- In the above example, the signed displacement for the JNZ instruction comes out to be (2005H – 2003H = 02, complement – FDH) The magnitude of the displacement must be less than or equal to 127(D).
- The MSB of the displacement decides whether it is a forward or backward jump. If it is 1, it is a backward jump or else it is a forward jump.

A similar procedure is used to find the displacement for intra segment short calls.

3.2.2 Finding out Machine Code for Unconditional JUMP Intra segment:

- For this instruction there are again two types of jump, i.e., short jump and long jump.
- The displacement calculation procedures are again the same as given in case of the conditional jump.
- The only new thing here is that, the displacement may be beyond -+127(D).
- This type of jump is called the long jump. The method of calculation of the displacement is again similar to that for short jump.

3.2.3 Finding out Machine Code for Inter segment Direct jump:

- This type of instruction is used to make a jump directly to the address lying in another segment. The opcode itself specifies the new offset and the segment of jump address, directly.

Example 3.1.2.3

JUMP 2000 : 5000

This instruction implies a jump to a memory location in another code segment with CS = 2000H and offset = 5000H. The code formation is as shown:

Code	1110 1010	0000 0000	0101 0000	0000 0000	0010 0000
Formation	Opcode	Offset LB	Offset HB	Seg. LB	Seg. HB

- The opcode forms the first byte of this instruction and the successive bytes are formed from the segment and the offset of the jump destination.
- While specifying the segment and offset, the lower byte (LB) is specified first and then the higher byte (HB) is specified.
- Finally, the opcode comes out to be EA 00 50 00 20. The procedure of coding the CALL instructions is similar.

3.3 Programming With An Assembler:

The procedure of hand – coding 8086 programs is somewhat tedious, hence in general a programmer may find it difficult to get a correct listing of the machine codes. Moreover, the procedure of handcoding is time consuming. This programming procedure is called as machine level programming.

Disadvantages of Machine Level Programming:

- The process is complicated and time consuming.
- The chances of error being committed are more at the machine level in hand – coding and entering the program byte – by – byte into the system.
- Debugging a program at the machine level is more difficult.
- The programs are not understood by anyone and the results are not stored ina user – friendly form.

Assembly Language Programming:

- A program called 'assembler' is used to convert the mnemonics of instructions along with the data into their equivalent object code modules.
- These object code modules may further be converted in executable code using the linker and loader programs.
- This type of programming is called as "assembly level programming".
- In Assembly Language Programming, the mnemonics are directly used in the user programs. The assembler performs the task of coding.

Advantages of Assembly Language:

- The programming in assembly language is not so complicated as in machine language because the function of coding is performed by an assembler.
- The chances of error being committed are less because the mnemonics are used instead of numerical opcodes. It is easier to enter an assembly language program.
- As the mnemonics are purpose – suggestive the debugging is easier.
- The constants and address locations can be labled with suggestive labels hence imparting a more friendly interface to user. Advanced assemblers provide facilities like macros, lists, etc., making the task of programming much easier.

- The memory control is in the hands of users as in machine language.
- The results may be stored in a more user – friendly form.
- The flexibility of programming is more in assembly language programming as compared to machine language because of advanced facilities available with the modern assemblers
- Basically, the assembler is a program that converts an assembly input file also called as source file to an object file that can further be converted into machine codes or an executable file using a linker.

3.4 Assembly Language Example Programs

Example 3.1.4

Write a program for addition of two numbers.

Solution:

The following program adds two 16-bit operands. There are various methods of specifying operands depending upon the addressing modes that the programmer wants to use. Accordingly, there may be different program listings to achieve a single programming goal. A skilled programmer uses a simple ligic and implements it by using a minimum number of instructions.

Let us now try to explain the following program.

```
ASSUME    CS,   DS : DATA
DATA      SEGMENT
OPR1      DW   1234H            : 1ST Operand
OPR2      DW    0002H           : 2nd Operand
RESULT    DW    01   DUP(?)     : A word of memory reserved for result

DATA      ENDS
CODE      SEGMENT
START:    MOV AX,  DATA         :  Initialize data segment
          MOV  DS, AX           :
          MOV AX, OPR1          : Take 1st operand in AX
          MOV  BX, OPR2         : Take 2nd operand in BX
           CLC                  : CLEAR PREVOIUS CARRY IF ANY
          ADD AX,   BX          : Add BX to AX
          MOV  DI, OFFSET RESULT   : Take offset of result in DI
          MOV [DI], AX          : Store the result at memory address in DI
          MOV AH, 4CH           :  Return to DOS prompt
          INT 21H
CODE      ENDS                  : CODE segment ends
          END START             : Program ends
```

3.5 Stack structure of 8086

3.5.1 Introduction to Stack:

- During interrupt and subroutine operations, the contents of specific internal registers of the 8086 may be overwritten. If these registers contain data that are needed after the return, they should be PUSHed to a section of memory known as the Stack.
- Here they may be maintained temporarily. At the completion of the service routine or subroutine, these values are POPped off the stack in the reverse order of the PUSH, and the register contents are restored with their original data.
- When an interrupt occurs, the 80x86 automatically PUSHes the current flags, the value in CS, and the value in IP onto the stack.
- As part of the service routine for the interrupt, the contents of other registers may be pushed onto the stack by executing PUSH instructions.
- An example is the instruction PUSH SI. It causes the contents of the Source Index register to be PUSHed onto the stack. At the end of the service routine, POP instructions can be included to restore the values from the stack back into their corresponding internal registers. For example, POP SI causes the value at the top of the stack to be popped back into the source index register.
- As indicated earlier, stack is implemented in the memory of the 8086. It is a maximum of 64K bytes long and is organized as 32K words. The lowest addressed byte in the current stack is pointed to by the contents of the stack segment (SS) register.
- Any number of stacks may exist in an 8086. A new stack can be brought in by simply changing the value in the SS register through software. For instance, executing the instruction MOV SS,DX loads a new value from DX into SS. Even though many stacks can exist, only one can be active at a time.
- Another register, the stack pointer (SP) contains an offset of the current top of the stack from the value in SS. The address obtained from the contents of SS and SP is the physical address of the last storage location in the stack to which data were PUSHed. This is known as the top of the stack.
- The value in the stack pointer starts at 0FFFFh upon initialization of the 8086. Combining this value with the current value in SS gives the highest addressed location in the stack: that is, the bottom of the stack.
- Since data transfers to and from stack are always 16- bit words, it is important to configure the system such that all stack locations are at even word boundaries. This minimizes the number of memory cycles required to PUSH or POP data for the stack and minimizes the amount of time required to perform a switch in program context.

- The 8086 PUSHes data and addresses to the stack one word at a time. Each time a register value is to be PUSHed onto the top of the stack, the value in the stack pointer is first decremented by 2 and then the contents of the register are written into memory.
- In this way, we see that the stack grows down in memory from the bottom of the stack, which corresponds to the physical address derived from SS and 0FFFFh toward the end of the stack, which corresponds to the physical address obtained from SS and offset 000016.
- When a value is popped from the top of the stack, the reverse of this sequence occurs. The physical address defined by SS and SP always points to the location of the last value pushed onto the stack.
- Its contents are first popped off the stack and put into the specified register within the 8086; then SP is incremented by 2. The top of the stack now corresponds to the previous value pushed onto the stack.
- A few things that we must remember are:
- We must make sure to create a Stack, and make it large enough for any program that has a CALL or any program that will use the PUSH and POP instructions.
    - o PUSH all registers that contain data or addresses that we need after the RETurn
    - o When w RETurn we must POP the registers in the reverse order of the PUSH
    - o We must POP each register that we PUSH
    - o If we PUSH before the CALL, we must after the RETurn (in the calling module or subroutine)
    - o If we PUSH after the CALL, we must POP before the RETurn (inside the subroutine) PUSH Examples

      PUSH AX – the contents of a 16-bit register PUSH EBX - the contents of a 32-bit register

      PUSHA (286 and higher) – Preserves all usable registers of 80286

      PUSHAD (386 and higher) – Preserves all usable registers of 80386

- There are corresponding POP instructions for each of the above examples. Execution of a PUSH instruction causes the data corresponding to the operand to be pushed onto the top of the stack.

  For instance, if the instruction is **PUSH AX** its execution results in the following:

SP <- SP - 1 ; SP is decremented
SS:SP <= AH ; AH is PUSHed on the Stack
SP <- SP - 1 ; SP is decremented
SS:SP <= AL ; AL is PUSHed on the Stack

- This shows that the two bytes of AX are saved in the stack part of memory and the stack pointer is decremented by 2 such that it points to the new top of the stack. On the other hand, if the instruction is POP AX, its execution results in the following:

AL <- SS:SP ; AL is POPped from the Stack
SP <- SP + 1 ; SP is incremented
AH <= SS:SP ; AH is POPped from the Stack
SP <- SP + 1 ; SP is incremented

Example 3.2.1

Let us assume we wish to write a program that CALLs a subroutine that will need all the general purpose registers, and those registers contain data that we will need after the subroutine has finished. We may preserve the contents of the registers using the PUSH instruction either by:

PUSHing before the CALL and POPping after the RETurn or
PUSHing after the CALL and POPping before the RETurn

Regardless of which method we use, we must POP in the reverse order of the PUSH. Consider the following.

```
PUSH AX
PUSH BX
PUSH CX
PUSH DX
CALL SUB1
POP DX
POP CX
POP BX
POP AX
```

The above code will preserve the registers on the Stack, and execute the subroutine, which PUSHes the IP onto the Stack. The last instruction of the subroutine is the RETurn which POPs the contents of the Stack into the IP, and then we restore the registers with the POP instruction. The above code will perform this correctly. Notice the register operand of the first POP corresponds to the register operand of the last PUSH, while the register operand of the last POP corresponds to the register operand of the first PUSH.

It is incorrect to perform the reverse. Consider the following.

```
PUSH AX
PUSH BX
PUSH CX
PUSH DX
CALL SUB1
POP AX
POP BX
POP CX
POP DX
```

- The previous code INCORRECTLY restores the registers. The contents of the AX register before the CALL is restored to DX. This code Swaps registers as follows
  - o The contents of the AX and DX registers are swapped and the contents of the BX and CX registers are swapped. It is difficult to say what effect this might have on your program, but if the data in any of these registers are used, errors could be significant.
- A disastrous scenario is if we PUSH outside the subroutine and POP outside the subroutine, or PUSH inside the subroutine and POP outside the subroutine. This causes a register content to be placed in the IP. We have no idea where our program will continue, but we should expect the program to crash.

Example 3.2.2

Write a program to change a sequence of sixteen 2-byte numbers from ascending to descending order. The numbers are stored in the data segment. Store the new series at addresses starting from 6000H. Use the LIFO property of the stack.

Solution:

```
ASSUME          CS : CODE, DS   : DATA, SS : DATA
DATA                SEGMENT
LIST            DW   10H
STACKDATA       DB    FFH   DUP (?)

0RG6000H
                  RESULT  DW  10H
DATA            ENDSCOUNT   EQU   10H
CODE            SEGMENT
START: MOV  AX, DATA        ; Initialize data segment and
       MOV  DS, AX          ; Stack segment
       MOV  SS, AX
```

```
        MOV  SP, OFFSET LIST    ; Initialize stack pointer
        MOV CL, COUNT           ; Initialize counter for word number
        MOV BX, OFFSET RESULT + COUNT    ;Initialize BX at last address
NEXT:   POP   AX                ; (stack) for destination series
        MOV DX, SP              ; Get the first word from the series
        MOV SP, BX              ; Save source stack pointer
        PUSH    AX              ; Save AX to stack
        MOV BX, SP              ; Save destination stack pointer
        MOV SP, DX              ; Get source stack pointer for the next number
        DCR CL                  ; Decrement Count
        JNZ NEXT                ; If count is not zero, go to the next num
        MOV  AH, 4CH            ; Else, return to DOS
        INT 21H                 ; prompt
CODE ENDS
  END START
```

3.6 Interrupts and Interrupt service routines:

- 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.

3.7 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.

Example 3.2.2

- 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.

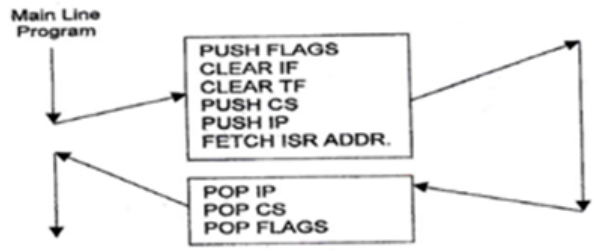
- ✓ If the interrupt flag is not set, the subsequent interrupt signals will not be acknowledged by the processor, till the current one is completed. The programmable interrupt controller is used for managing such multiple interrupts based on their priorities.
- ✓ At the end of ISR the last instruction 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.

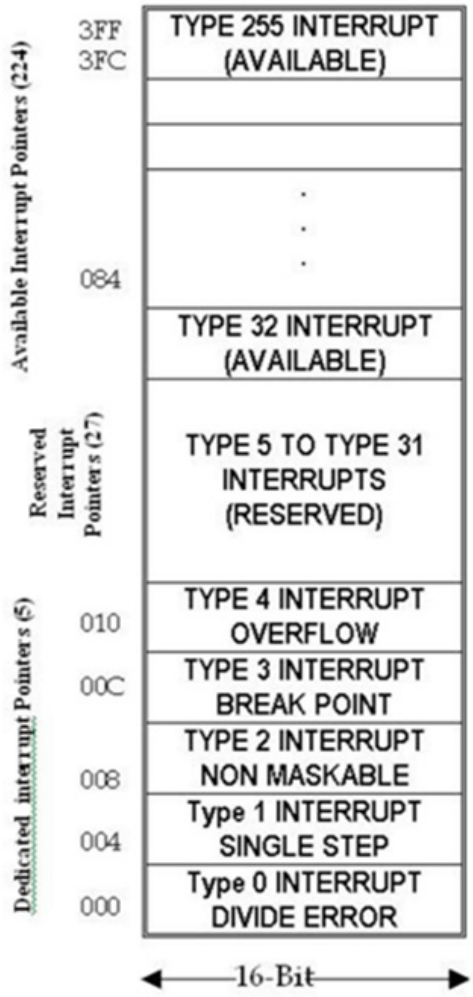
Following figure summarizes these steps in diagram form. As you can see, the 8086 pushes the flag register on the stack disables the INTR input and the single-step function, and does essentially an indirect far call to the interrupt service procedure. An IRET instruction at the end of the interrupt service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

Fig:8086/8088 Interrupt Response



• 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 of256 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 : 03FF H. The interrupt type N is multiplied by 4 and the hexadecimal multiplication obtained gives the offset address in the zeroeth code segment at which the IP and CS addresses of the interrupt service routine (lSR) are stored. The execution automatically starts from the new CS:IP.



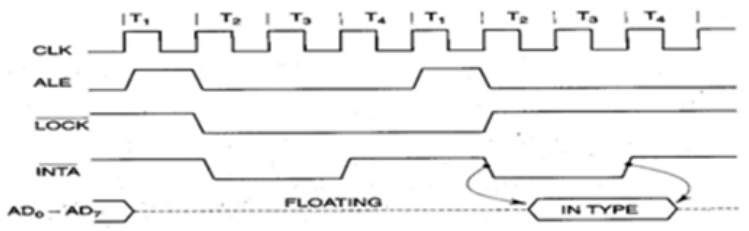
3.7.1 Non Maskable Interrupts:

- 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 intern al 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.

3.7.2 Maskable Interrupts (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.

3.8 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.

Example 3.2.3.1:

To read a string form the key board and convert the characters into upper case letters and display on the screen with enough messages displayed in between.

Solution:

```
data segment

msg1 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,09h
        int 21h
        mov ah,4ch
        int 21h
code ends

end start
```

3.9 Passing parameters to procedures

- Procedures or subroutines may require input data or constants for their execution. Their data or constants may be passed to the subroutine by the main program (host or calling program) or some subroutine may access readily available data of constants available in memory.

- Generally, the following techniques are used to pass input data / parameter to procedures in assembly language programs.
  - Using global declared variable
  - Using registers of CPU architecture
  - Using memory locations (reserved)
  - Using Stack
  - Using PUBLIC & EXTRN.

Besides these methods if a procedure is interactive it may directly accept inputs from input devices.

Example 3.1:

```
ASSUME CS : CODE1, DS : DATA
DATA SEGMENT
NUMBER  EQU 77H   GLOBAL
DATA ENDS
CODE1   SEGMENT
START :  MOV AX, DATA
            MOV  DS, AX
              .
              .
       MOV AX,NUMBER
              .
      CODE1   ENDS
ASSUME   CS : CODE2
CODE2    SEGMENT
   MOV  AX,DATA
    MOV  DS,AX
    MOV BX,NUMBER
CODE2    ENDS
       END START
```

- The CPU general purpose registers may be used to pass parameters to the procedures. The main program may store the parameters to be passed to the procedure in the available CPU registers and the procedure may use the same register contents for execution.
- The original contents of the used CPU register may change during execution of the procedure. This may be avoided by pushing all the register content to be used to the stack sequentially at the start of the procedure and by poping all the register contents at the end of the procedure in opposite sequence.

Example 3.2

```
ASSUME  CS : CODE
CODE   SEGMENT
START   :    MOV  AX, 5555H
             MOV  BX, 7272H
                  .
                  .
             CALL   PROCEDURE 1
                  .
                  .
PROCEDURE     PROCEDURE1   NEAR
                  ADD AX,BX
                  RET
PROCEDURE1    ENDP
CODE ENDS
                  END START
```

- Memory locations may also be used to pass parameters to a procedure in the same way as registers. A main program may store the parameter to be passed to a procedure at a known memory address location and the procedure may use the same location for accessing the parameter.

3.9.1 Handling Programs of Size More Than 64K:

- ❖ The maximum size of an 8086 segment is 64 KB. The same limitation is applicable to a code segment that contains executable program code. This obviously puts limitation on the maximum size of a program and thus how to write programs of size more than 64 k is going to be an interesting question, which is addressed in this question.
- ❖ The big programming task should be divided into independent modules, which may be developed and tested individual functions of the module to implement the complete tasks.
- ❖ As far as the programming methodology is concerned there are two approaches to solve this problem.
  - o Writing programs with more than one segment for Data, Code or Stack.
  - o Writing programs with FAR subroutines each of which can be of size up to 64K.

Example 3.1.1

A program with more than one segment

Solution:

```
ASSUME CS : CODE1,   DS: DATA1
CODE1   SEGMENT
          START  :  MOV AX, DATA1
                    MOV  DS, AX
                       .
                       .
 CODE1                ENDS
ASSUME   CS : CODE2,  DS : DATA2
CODE2     SEGMENT
                        MOV  AX, DATA2
                      MOV  DS, AX
                         .
                         .
CODE2                  ENDS
DATA1                  SEGMENT
DATA1                  ENDS
DATA2                  SEGMNET
DATA2                 ENDS

END START
```

Example 3.1.2 :

A program with more than one intersegment routine.

Solution :

```
ASSUME  CS : CODE1 , DS : DATA1
DATA  SEGMENT
 DATA  ENDS
CODE1   SEGMENT
START :             MOV  AX, DATA
                    MOV  DS, AX

          CALL FAR_PTR ROUTINE 1
          CALL FAR_PTR ROUTINE 2
CODE1                ENDS
PROCEDURE            ROUTINE1  FAR
ROUTINE1             ENDP
PROCEDURE            ROUTINE2  FAR
ROUTINE2             ENDP
END                  START
```

3.10 Macros

- The macro is also a similar concept. Suppose, a number of instructions are repeating through in the main program, the listings becomes lengthy. So a macro definition i.e., a label, is assigned with the repeatedly appearing string of instructions.
- The process of assigning a label or macro name to the string is called defining a macro.
- A macro within a macro is called a nested macro. The macro name or macro definition is then used throughout the main program to refer to that string of instructions.

Difference between a macro and a subroutine:

- In the macro the complete code of the instruction string is inserted at each place where the macro – name appears. Hence the EXE file becomes lengthy. Macro does not utilize the service of stack. There is no question of transfer of control as the program using the macro inserts the complete code of the macro at every reference of the micro name.
- Subroutine is called whenever necessary, i.e., the control of execution is transferred to the subroutine, every time it is called. The executable code in case of the subroutines becomes smaller as the subroutine appears only once in the complete code.
- The program using subroutine requires less memory space for execution than that using macro.
- Macro requires less time for execution, as it does not contain CALL and RET instructions as the subroutines do.

Defining a Macro:

- A MACRO can be defined anywhere in a program using the directives MACRO and ENDM. The label prior to MACRO is the macro name which should be used in the actual program. The ENDM directive marks the end of the instructions or statements sequence assigned with the macro name.

  The following macro DISPLAY displays the message MSG on the CRT. The syntax is as given:

```
DISPLAY  MACRO
        MOV AX, SEG MSG
        MOV DS, AX
        MOV DX,  OFFSET MSG
        MOV AH, 09 H
        INT  21 H
```

ENDM

- The above definition of a macro assigns the name DISPLAY to the instruction sequence between the directives MACRO and ENDM. While assembling, the above sequence of instructions will replace the label 'DISPLAY' , whenever it appears in the program.
- A macro may also be used in a data segment. In other words, a macro may also be used to represent statements and directives. The concept of macro remains the same independent of its contents.
  The following example shows a macro containing statements. The macro defines the strings to be displayed.
  STRINGS MACRO
  MSG1 DB 0AH, 0DH, "Program terminated normally", 0AH, 0DH, "$"
  MSG2 DB 0AH, 0DH, "Retry , Abort, Fail" , 0AH, 0DH, "$"
  ENDM
- A macro may be called by quoting its name, along with any values to be passed to the macro. Calling a macro means inserting the statements and instructions represented by the macro directly at the place of the macro name in the program.

Passing Parameters to a MACRO:

- Using parameters in a definition, the programmer specifies the parameters of the macro those are likely to be changed each time the macro is called.

Example:

The DISPLAY macro written in above can be made to display two different messages MSG1 and MSG, AS SHOWN

```
DISPLAY     MACRO  MSG
            MOV  AX,  SEG MSG
            MOV DS , AX
            MOV DX, OFFSET MSG
            MOV  AH, 09 H
            INT  21 H
    ENDM
```

This parameter MSG can be replaced by MSG1 and MSG2 while calling the macro.

3.11 Timings and Delays

- That every instruction requires a definite number of clock cycles for its execution. Thus every instruction requires a fixed amount of time, i.e. multiplication of the number of clock cycles required for the execution of the instruction period of the clock at which the microprocessor is running.

- ❖ The duration required for the execution of an instruction can be used to derive the required delays.
- ❖ A sequence of instructions, if executed by a microprocessor, will require a time duration that is the sum of all the individual time durations required for execution of each instruction.

The procedure of generating delays using a microprocessor based system can be stepwise described as follows:

1. Determine the exact required delay.
2. a) Select the instructions for delay loop. While selecting the instructions, care should be taken that the execution location or register used by the main program must note be modified by the delay routine.
   b) The instructions executed for the delay loop are dummy instructions in the sense that the result of those instructions is useless but the time required for their execution is an elemental part of the required delay.
3. Find out the number of clock states required for execution of each of the selected delay loop instructions. Further find out the number of clock states required (n) to execute the loop once by adding all the clock states required to execute the instructions individually.
4. Find out the period of the clock frequency at which microprocessor is running i.e. duration of a clock state (T).
5. Find out the time required for the execution of the loop once by multiplying the period T with thw number of clock states required (n) to execute the delay loop once.
6. Find out the count (N) by dividing the required time delay Td by the duration for execution of the loop once (n*T).

$$\text{Count } N = \frac{\text{Required Delay (Td)}}{n*T}$$

Assignment Questions

1. Write an ALP to convert a four digit hexadecimal number into decimal number.
2. Write an ALP find out transpose of a matrix.
3. Write an ALP to set and get the system time.
4. Describe the procedure for coding the intra segment and intersegment jump and call instructions.
5. Explain the stack structure of 8086 in detail.
6. What is interrupt vector table of 8086? Explain its structure.
7. How do you pass parameters to macro. Explain?
8. Write an ALP to calculate the hexadecimal factorial of a one digit hexadecimal number?

Short Questions (2, 3 marks)

1. What is an Assembler?
2. What is a Linker?
3. List advantages of ALP.
4. Write about DEBUG command?
5. What are the DOS function calls.
6. Find out the machine code for the following instructions.
   a. ADC AX,BX
   b. MUL [SI+5]
   c. LEA SI,[BX+500H]
   d. JMP 3000H:2000H
   e. CALL [5000H]
   7. Describe execution of a CALL instruction.
8. What is the difference between a NEAR and FAR procedure?
9. Explain the term “nested Interrupt”?
10. What you mean by Macro?
11. What you mean subroutine?
12. What is nested macro?
13. Differentiate software and hardware interrupt?
14. What is the vector address of NMI interrupt?
15. What is the vector address of INT 55H interrupt?

Review Questions

1. Write an ALP to find square root of a two digit number.
2. Write an ALP to sort list of numbers in ascending order.
3. Write an ALP to find average of a given string of data bytes.
4. Write an ALP to display message “Happy Birthday” on the screen after a key ‘A’ is pressed.

5. Define a macro SQUARE that calculates square of a number.
6. What is the role of stack in calling a subroutine and returning from the routine?
7. Draw and discuss interrupt structure of 8086?
8. How will you differentiate between subroutine and interrupt service routine procedures?
9. Write an ALP to generate following delays using 5MHz
   a) 100 ms b) 5 sec