Example

```
LEA DI, STR 1     ; DI = offset of STR 1
MOV AL, 05 H      ; AL = 05 H
CLD               ; DF = 0
MOV CX, 35 H      ; CX = 35 H i.e. counter = 35 H
REPE SCASB        ; Repeat, scan string byte operation till
                  ; CX ≠ 0 and ZF = 1.
```

B. REPNE/REPNZ (Repeat while not equal / Repeat while not zero)

Mnemonic REPNE/REPNZ

Algorithm

Check CX
if CX < > 0 then CX = CX – 1
If ZF = 0 then
repeat the instruction to which it is prefix
go back to check CX
else exit from REPNE cycle
else exit from REPNE cycle.

Operation

- These are two mnemonics for the same instruction.
- This will cause string instructions to be repeated as long as ZF = 1 and CX ≠ 0.
- If ZF = 1 or CX = 0, string instructions will not be repeated.

□□□

CHAPTER

13

Assembly Language Programming

13.1 Introduction to Assembly Language

Microprocessor chip designers create a basic set of instructions for every processor they design. These are really simplistic instructions that take baby steps as compared with high-level languages such as C++.

- Each instruction has a code so that the instruction decoder can decode them in order to energize the proper circuits to execute the function using the registers of the processor. These are referred to as ***operation code*** (OPCODE). *"Machine code is the instruction set that machine understands".*
- Machine language is the only language understood directly by the CPU in our computers.
- Humans, however, understand words, so each machine code is given an "English" equivalent. *These instructions in text form are called* ***Mnemonic***.
- Assembly language is a mnemonic representation of machine code.
- Assembly language is not just a simple mapping of numbers to words. It also contains many high-level-language type constructs to make data definition and program structuring easier.
- There is a one-to-one correlation between assembly language instructions and the machine code.

Reasons for learning the assembly language

The assembly language is learnt for the following reasons :

(1) It helps to learn about the internal architecture of the computer. The more you use the assembly language the more knowledge you gain about the architecture.

(2) So as to make use of the utilities of a computer e.g. to directly communicate with the microprocessor of the computer. This is useful under the circumstances where programming in high level language is difficult or even impossible.

(3) Most of the times a high level language program written for the above applications does not yield the required performance. So use of assembly level programming is always a better solution.

(4) The high level language programs may need a large memory space. Instead the memory requirement will reduce if we use the assembly language programming.

(5) There are less number of restrictions or rules as compared to the high level languages.

13.1.1 Programming Methodology

- Generally the programmers do not write large programs using the assembly language.
- Instead they write short, specific routines and subroutines in the assembly language.
- Write the main long program using the high level language and call the assembly subroutines written in assembly language as and when required.
- Assembly language subroutines can be written to handle operations which are not available in the higher level language.

13.2 Development of an Assembly Language Program

Tools used for development

The development of assembly language program needs following tools :

- Assembler
- Linker
- Loader
- Debugger and
- Emulator

Some of these tools are used for program development, some for the program execution and the remaining are useful for testing the assembly language program.

13.2.1 Steps for Developing an Assembly Language Program

- Developing an assembly language program is a four step process. The steps are as follows :
  - (i) To specify the source code as per the assembly language definition.
  - (ii) Assemble the program to create the object code.
  - (iii) Link the program to create an executable code.
  - (iv) Test and debug the program.

Fig. 13.2.1 shows the steps involved in developing and executing an assembly language program.

Description

- Refer Fig. 13.2.1 to understand the program development steps.

Step 1 :

- The first step is to analyse what the program is to do and how we want the program to do it.
- Then using an editor create the source file for program.

Step 2 :

- The second step is to assemble this source file.
- If the assembler indicates errors then use the editor for correcting them and again assemble this source file.

Step 3 :

- If the program consists of several modules, then use the linker to join them into one large object module.
- If the system needs to locate a program in order to specify its location in the memory then use the locator.
- At this stage the program is ready for loading into the memory and run.

Step 4 :

- If the developed program does not interact with any external hardware other than that connected directly to the system then use debugger for running and debugging your program.
- If the program is supposed to work with the external hardware system such a microprocessor based instrument then use emulator to run and debug the program.

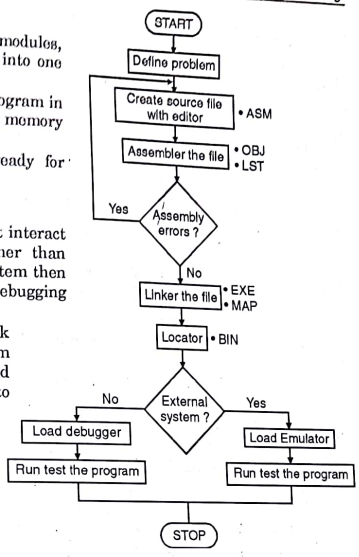


Fig. 13.2.1

13.3 Assembly Language Program Development Tools

- In this section we will go into the details of some of the assembly language program development tools.

13.3.1 Editor

- An editor is basically a software (i.e. a program).
- It helps the user to create a file that contains the assembly language statements.
- The examples of editors used for the assembly language programs are Wordstar, Edit, WordPad, Notepad etc.
- The job of the editor is to store the ASCII codes for the letters and numbers in the successive RAM locations.
- As the typing of program is over, this file is stored on a floppy or hard disk.
- This file is called as the "source file" and an ASM extension is given to it.
- The source file is then processed using an assembler.

13.3.2 Assembler

- Each assembly level instruction has a mnemonic. For example in the instruction MOV BX, DX. MOV represents the mnemonic.
- An assembler is a program which translates the assembly language mnemonics into corresponding binary codes.
- Two assemblers available for assembling the programs for IBM – PC are : 1. Microsoft micro assembler (MASM) and 2. Borland turbo assembler (TASM).

Assembler operation

- The assembler first reads the source file of program.
- Then it determines the displacement of data items, offsets of labels etc. and puts this information into a symbol table.
- Then it produces the binary codes for each assembly language instruction and inserts the offsets etc. calculated earlier.

File Generation in assembler

- An assembler generates two files namely the **object file** and the **assembler list file**.
- The object file is given extension .OBJ whereas the assembler list file is given extension . LST.
- **Object file :** It contains the binary codes of the program instructions and the information about the addresses of instructions.
- **List file :** It contains the assembly language statements, the binary codes for each instruction and the offset of each instruction.
- Any typing or syntax errors are indicated in the assembly listing if we take a print out of . LST file.

Error detection and correction

- The assembler is capable of only finding the syntax errors.
- To check if our program is working, we have to test and run the program.
- The errors indicated by the assembler should be edited using the editor.
- This edit-assemble loop should be executed till all the errors are corrected.

Assembling a program

- Once the program is written, it can be assembled to obtain the . OBJ (object) file by executing the MASM or TASM assemblers.
- It is necessary to mention the assembly language program file name along with the command as shown below :
  MASM <filename . ASM> or TASM <file name . ASM>.
- If the source file name is add, then to assemble it we can write.
  masm add . asm or
  tasm add . asm
- If there are errors then the assembler generates error messages. These errors are listed along with the line number.
- In case of no errors the OBJ file is created.

- In order to obtain the . EXE file the user will have to link the . OBJ file with the linker.
  For MASM LINK <file name>
  For TASM TLINK <file name>

13.3.3 Linker

- Linker is a program which is used for joining many object files into one large object file.
- When a large program is being written, it is always advisable to break it into small modules, so that each module can be separately tested and debugged. Then finally link their object modules together to form a large working program.
- e.g. the display routine can be kept in the library file and linked into the other programs when required.
- The linker produces a link file which contains the binary codes for all the combined modules.
- The linker produces a link map file. It contains the address of all the linked files.
- It is important to note that the linker does not assign absolute addresses. It only assigns relative addresses to the program starting from zero.
- It is relocatable. The linkers with MASM or TASM produce link files with . EXE extension.

13.3.4 Locator

- The locator is a program which is used for assigning the specific address of the locations where the segments of the object code are to be loaded into the memory.
- A locator program called EXE2BIN comes with the IBM. PC disk operating system (DOS).
- This program converts an . EXE file into a . BIN file which has physical addresses.

13.3.5 Debugger

- "A software tool that is used to detect the source of program or script errors, by performing step-by-step execution of application code and viewing the content of code variables."
- If a program is directly accessible from the microcomputer and does not need any external hardware, then we can use a debugger to run and test the program.
- Debugger is basically a program which permits the user to load object code program into the system memory, execute the program and debug it.
- The debugger also permits the change in register contents, memory locations and rerun the program.
- With the help of the debugger, we can stop the program execution after each instruction so that we can check or alter the memory and register contents.
- In other words we can put breakpoints in the program and execute the program from one breakpoint to the other.
- It is possible to examine the register and memory contents after partial execution of program between the breakpoints.

- We can use the debugger to check and correct the program till all the errors are corrected.
- For most IBM PC type computers the basic debugger comes by default.
- But more powerful debuggers such as Borland's Turbo Debugger (TD) or Microsoft code view debugger are available.
- They make the debugging easier and allow the user to see the contents of registers and memory locations as the program is executed.

13.3.6 Emulator

- The emulator is used to test and debug the hardware and software of an external system such as the microprocessor based system.
- Emulator is a combination of hardware and software.
- An emulator consists of a multi-wire cable that connects the host system to the external system.
- Through this cable the software of the emulator allows the user the download the object code program into RAM in the external system being developed.
  Like the debugger the emulator also allows the user to load the programs to be tested, run the, programs check and modify the contents of various registers and memory locations and also insert the breakpoints.
- As each instruction in the assembly language program is executed, the emulator as if takes "snapshot" of the register contents, activities on the address and data buses and the state of flag register.
- The emulator stores this data as "trace data".
- It is possible to take out the print out of the trace data so as to analyse the results produced in the program on the step by step basis.

13.4 Assembly Language Structure

- The assembly language instructions have a definite structure.
- In this section we are going to discuss the structure of assembly language statements and use of pseudo operations.
- Pseudo operations are not machine instructions but they are the instructions to the assembler.

13.4.1 Assembler Instruction Format

- The assembly language statement is divided into various fields.
- These fields are separated by spaces and tabs.
- The general format of an assembler instruction is as follows :

Label	Mnemonic	Operand	Comment
(Optional) Used to identify the instruction	Basically, this is the type of instruction.	The data which is being perated upon.	(Optional) Enclosed in brackets These are essential in Assembly Language programming !

Label

- The label field is an optional field containing a symbolic label for the current statement.
- Labels are used in assembly language as to mark the lines which can be jumped by the other instruction (as GOTO statement).
- In general, you should begin your labels in column one with the syntax :
  <label name> :
- A symbol is associated with some particular value. This value can be an offset within a segment, a constant, a string, etc.
- A symbolic name consists of a sequence of letters, digits, and special characters, with the following restrictions :
  1. First character must be alphabetic letter or special character like '$'. No numeric digit is allowed. Only '$' or '?' characters are not allowed (as these two characters have meanings).
  2. It cannot be reserved words.
  3. The maximum length for the characters is 31.
  4. It is not case-sensitive which is different from C.
  5. It treats the upper and lower case letters equivalently.

Mnemonic

- A mnemonic is an instruction name.
- All the instructions should have a mnemonic i.e. this field is compulsory.
- A mnemonic specifies the operation to be executed.

Operand

- The operand field contains the operands, or parameters, for the instruction specified in the mnemonic field.
- Operands never appear on lines by themselves.
- The type and number of the operands depend entirely on the specific instruction.
- Some instructions have no operands e.g. XLAT, AAM, DAA.
- Some instructions have two operands. In such a case, they are separated by a comma.

Comments

- The comment field allows you to annotate each line of source code in your program i.e. put a comment on each line of the program code so that it becomes simple to understand the program .
- When the assembler is processing a line of text with beginning of the notation " ; ", it completely ignores everything on the source line following a semi-colon.
- You can also have a comment on the line by itself.
- The following instruction illustrates the various fields clearly.

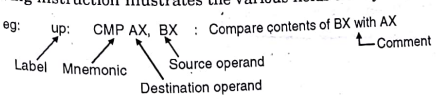


Fig. 13.4.1

13.4.2 Program Format and Assembler Directives

- First let us see the structure of general assembler program for 80813.
- The general assembler program structure for 8086 is shown in Figs. 13.4.2 (a) and 13.4.2 (b).

```
Line 1    . MODEL SMALL        ; Selects small model.
Line 2    . DATA               ; Indicates data segment.
            .....
            .....              ; Data declaration
Line 15   . Code               ; Indicates code segment.
            .....
            .....              ; Program body.
            .....
Line n      End                ; End of file.
```

Fig. 13.4.2(a) : Program structure

```
DATA_HERE SEGMENT
.....
.....                          ; Data declaration.
DATA_HERE ENDS
CODE_HERE SEGMENT
   ASSUME CS : CODE_HERE, DS : DATA_HERE
.....
.....                          ; Body of the program
.....
CODE_HERE ENDS
END
```

Fig. 13.4.2(b) : Program structure

13.5 Assembler Directives

Q. What are different types of assembler directives? Explain any four assembler directives.

Assembly language consists of two type of statements viz.

1) Executable statements
   These are the statements to be executed by the processor. It consists of the entire instruction set of 8086 (as seen in the chapter 3.)
2) Assembler directives
   These are the statements that direct the assembler to do something. As the name says, it direct the assembler to do a task.

- The speciality of these statements is that they are effective only during the assembly of a program but they do not generate any code that is machine executable.
- We can divide the assembler directives into two categories namely the general purpose directives and the special directives.

They are classified into the following categories based on the functions performed by them.

- Simplified segment directives
- Segment directives
- Code label directives
- Listing control directives
- Data allocation directives
- Macros related directives
- Scope directives
- Miscellaneous directives

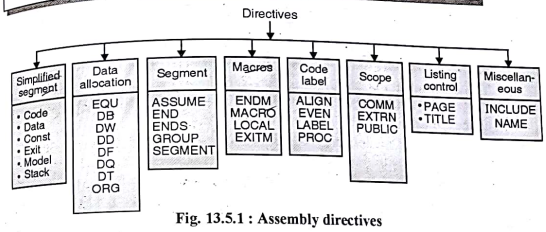


Fig. 13.5.1 : Assembly directives

1. **.CODE**
   - This assembler directive indicates the beginning of the code segment. Its format is as follows :
     . CODE [name]
   - The name in this format is optional.
   - For tiny, small and compact models the segment name is – TEXT always.
   - The medium and large memory models use more than one code segments which can be distinguished by name.
2. **.DATA**
   This directive indicates the beginning of the data segment.
3. **.MODEL**
   - This directive is used for selecting a standard memory model for the assembly language program.
   - Each memory model has various limitation depending on the maximum space available for code and data.
   - The general format for defining the Model directive is as follows :
     . MODEL [memory model] :
   - The memory model is chosen based on the user's requirement by referring to Table 13.5.1.

Table 13.5.1

Model	Number of Code Segments	Number of Data Segments
Small	One code segment and of size < = 64 Kbytes	One of size < = 64 Kbytes
Medium	Code segment may be of any number and any size.	One of size < = 64 Kbytes

Model	Number of Code Segments	Number of Data Segments
Compact	One of size < = 64 Kbytes	Data segment of any number and any size.
Large	Any number, any size	Any number, any size.
Huge	Any number, any size	Any number, any size.

- The size of a memory model can be anything from small to huge.
- The tiny model is meant for the . COM programs because they have their code, data and stack in only one 64 kB segment of memory.
- On the other hand the flat model is the biggest which defines one area upto 4 GB for code and data.
- The small model is useful for the student level programs because for this model the assembler assumes that the addresses are within a span of 64 kB and hence generates 16 kB offset addresses.
- In the compact model, the assembler can use 32 bit addresses. So the execution time for this model is longer.
- The huge model contains variables such as arrays which need a larger space than 64 kB.

4. STACK

- This directive is used for defining the stack. Its format is as follows :
  . STACK [size]
- The size of the stack is 1024 bytes by default but this size can be overridden.
- We can omit the . stack command if the stack segment is not to be used in a program.
- Example of the stack directive is
  . STACK 100

Which reserves 100 bytes for the stack segment.

5. EQU-Equate

- It is used to give a name to some value or symbol in the program.
- Each time when the assembler finds that name in the program, it replaces that name with the value assigned to that r variable .
- Its format is [name] EQU initial value.
  e.g. : FACTORIAL EQU 05H.
- This statement is written during the beginning of the program and whenever now FACTORIAL appears in an instruction or another directive, the assembler substitutes the value 5.
- The advantage of using EQU in this manner is that if FACTORIAL is used several times in a program and the value has to be changed, all that has to change the EQU statement and reassemble the program.
- The assembler with automatically put the new value each time it finds the name FACTORIAL.

6. Define Byte [DB]

- This directive defines the byte type variable.

- It is also useful to set one or more storage locations aside.
- The format of this directive is as follows :
  [name] DB initial value
- The initial value can be a numerical value (8 – bit long) or more than one 8 bit numeric values.
- It can be a constant expression, or a string constant or even a question mark.
- The initial value can be a signed or unsigned number. Its range is from – 128 to + 127 if unsigned and 0 to 255 if it is unsigned.

7. Define Word or Word [DW]

- The DW directive defines items that are one word (two bytes) in length.
- For unsigned numeric data the range of values is 0 to 65, 535
- For signed data the range of values is – 32, 768 to + 32, 767.
- Its format is
  [name] DW initial value.
  e.g. List DW 2534.

8. Define Double Word or DWORD [DD]

- It defines the data items that are a double word (four bytes) in length.
- It creates storage for 32 bit double words. The format is
  [name] DD initial value.
  e.g. BUFF DD ?

9. Define Quad Word or QWORD [DQ]

- This directive is used to tell the assembler to declare variable 4 words in length or to reserve 4 words of storage in memory.
- It may define one or more constants, each with a maximum of 8 bytes or 16 Hex digits.
- Its format is :
  [name] DQ initial value, [initial value].
  e.g. Num DQ . 1234567898765432H.

10. Define Ten Bytes or TBYTE [DT]

- It is used to define the data items that are 10 bytes long.
- Its format is
  [name] DT initial value, [initial value].
  e.g. unpack DT 1234567890.
- Unlike the other data directives with store hexadecimal numbers, DT will directly store the data in decimal form.

11. ORG-Originate

- The ORG directive allows us to set the location counter to any desired value at any point in the program.
- The location counter is automatically set to 0000H when the assembler reads a segment.
- Its format is ORG expression.
  e.g. ORG 500 H ; Set the location counter to 500 H.

- A $ is used to represent the current value of LC.
- The $ represents the next available byte location where assembler can put data or code byte.
- The $ is often used in ORG statements to inform the assembler to make change in location counter relative to its current value.

e.g. ORG $ + 50 ; Increments the location counter by 50 from its current value.

12. ASSUME

- The directive is used for telling the assembler the name of the logical segment which should be used.
- The format of the assume directive is as follows,

ASSUME segment register : segment-name :

- The segment register can be CS, DS, SS and ES.
- The example of Assume directive is as follows,

ASSUME CS Code, DS : Data, SS : Stack :

- ASSUME statement can assign upto 4 segment registers in any sequence.
- In this example, DS : Data means that the assembler is to associate the name of data segment with DS register.
- Similarly CS : Code tells the assembler to associate the name of code segment with CS register and so on.

13. END

- This is placed at the end of a source and it acts as the last statement of a program.
- This is because the END directive terminates the entire program.
- The assembler will neglect any statement after an END directive.
- The format of END directive is as follows :

END

14. SEGMENT and ENDS

- The SEGMENT directive is used to indicate the start of a logical statement.
- ENDS directive is used with the segment directive.
- ENDS directive indicates the end of the segment.
- Its format is

```
name SEGMENT        ; Begin Segment
name ENDS           ; End Segment.
e.g. : DATA SEGMENT
DATA ENDS.
```

15. GROUP

- This directive collects the segments of the same type under one name.
- It does it so that the segments that are grouped will reside within one segment usually data segment.
- Its format is

[name] GROUP Seg-name, [seg-name]

e.g. : NAME GROUP SEG 1, SEG 2.

16. MACRO AND ENDM

- The macros in the programs can be defined by MACRO directive.
- The ENDM directive is used along with the Macro directive.
- ENDM defines the end of macro.
- Its format is

```
DISP MACRO
        ; Statements inside the Macro
ENDM
```

17. ALIGN

- This directive will tell the assembler to align the next instruction on an address which corresponds to the given value.
- Such an alignment will allow the processor to access words and double words.

ALIGN number

→ This number should be 2, 4, 8, 16 .... i.e it should be a power of 2.

- The example of Align directive are ALIGN 2 and ALIGN 4.
- ALIGN 2 is used for starting the data segment on a word boundary whereas ALIGN 4 will start the data segment on a double boundary word.

18. EVEN (Align on Even Memory Address)

- It tells the assembler to increment its location counter if required, so that the next defined data item is aligned on an even storage boundary.
- The 8086 can read a word from memory in one bus cycle if the word is at an even address.
- If the word starts at an odd address, the microprocessor must do two read cycles to get 2 bytes of the word. In the first cycle it will read the LSB and in the second it will read MSB
- Therefore, a series of even words can be read more quickly if they are at an even address.

```
e.g. : EVEN   TABLE   DB   10 DUP (O)  ; It declares an array named
                                        ; TABLE of 10 bytes which are
                                        ; starting from an evenaddress.
```

19. LABEL

- This directive assigns name to the current value of the Location Counter.
- The LABEL directive must be followed by a term which specifies the type associated with that symbolic name.
- If label is going to be used as the destination for a jump or call, then the LABEL must be specified as type near or type Far.
- If the label is going to be used to reference a data item, then the label must be specified as type byte, word or double word.

```
e.g. : STACK SEGMENT
DW  50 DUP (O)                  ; Set aside 50 words for stack.
STACK_TOP   LABEL WORD          ; Give a symbolic name to next location
                                ; after the last word in stack STACK ENDS
```

20. PROC-Procedure

- This directive is used to indicate the start of a procedure.
- The procedures are of two types NEAR and FAR.
- If the procedure is within the same segment then the label NEAR should be given after procedure.
- If the procedure is in another module then the label FAR should be given after procedure.
- Its format is **[procedure-name]** PROC NEAR.

21. EXTRN

- It indicates that the names or labels that follow the EXTRN directive are in some other assembly module.
  e.g. : EXTRN DISP : FAR.
  The statement tells the assembler that DISP is a label of type far in another assembly module.
- To call a procedure that is in a program module assembled at a different time from that which contains the CALL instruction, the assembler has to be told that the procedure is external.
- The assembler will then put information in the object code file so that linker can connect the two modules together.
- The names or labels that are external in one module must be declared public with the PUBLIC directive in the module where they are defined.
- Its format is

```
e.g. : Procedure_HereSegment
       EXTRN FACT   :  FAR
       XTRN SUM     :  NEAR.
Procedure_HereEnds.
```

22. PUBLIC

- It informs the assembler and linker that the identified variables in a program are to be referenced by other modules linked with the current one.
- Its format is
- PUBLIC variable, [variable]
- The variable can be a number (up to two bytes) or a label or a symbol.
  e.g. PUBLIC MULTIPLIER, DIVISOR.

This makes the two variables Multiplier and Divisor available to other assembly modules.

23. PAGE

- This directive is used to specify the maximum number of lines on a page and the maximum number of characters on a line.
- The format of this directive is as follows :

PAGE [length], [width]

- [length] → Number of lines on a page
- [width] → Number of characters on a line.

- The example is PAGE 55, 102 which shows that the, there are 55 lines per page and 102 characters per line.
- The number of lines per page typically range from 10 to 255 and the number of characters per line will range from 60 to 132.

24. TITLE

- It helps the user for controlling the format of listing of an assembled program.
- It is used to give a title to program and print the title on the second line of each page of the program.
- The maximum number of characters allowed as a title is 60.
- The format of this assembler directive is as follows,
  TITLE Text

25. INCLUDE

- This directive is used to tell the assembler to insert a block of source code from the named file into the current source module. This shortens the source code.
- Its format is
  INCLUDE path : file name.

26. NAME

This directive assigns a specific name to each assembly module when programs consisting of several modules are written.

27. DUP Operator

- Whenever we want to allocate space for a table or an array the DUP directive can be used. The DUP operator it will be used after a storage allocation directive like (DB, DW, DQ, DT, DD).
- With DUP, we can repeat one or more values while assigning the storage values .
- Its format is
  [name] Data-Type Number DUP (value).

```
e.g. : List DB 20 DUP [0]          ; A list of 20 bytes, where each byte is zero.
       A DUP operator may be nested.
e.g. LIST 1 DB 4 DUP (4 DUP [0]. 3 DUP ['X']) ; The assembler assigns the values
                                                in memory here as follows,
       00, 00, 00, 00, x, x, x
       00, 00, 00, 00, x, x, x
       00, 00, 00, 00, x, x, x
       00, 00, 00, 00, x, x, x
```

28. GLOBAL – Declare Symbols as PUBLIC or EXTRN

- This directive can be used instead of PUBLIC directive or instead EXTRN directive.
- For a name or variable defined in the current module, the GLOBAL directive is used to make the variable available to all other modules.
- Its format is

```
e.g. : GLOBAL FACTOR   ; it makes the variable FACTOR public so that it can
                       ; be accessed from all other modules that are linked .
```

```
GLOBAL FACTOR : WORD  ; it tells assembler that variable FACTOR
                      ; type word which is in another assembly
                      ; or EXTRN.
```

29. LENGTH

- It is an operator.
- It informs the assembler to find the number of elements in a named data
  like a string or an array.
- The length of string is always stored in Hex by the 8086.
- Its format is :

  e.g. : MOV CX, LENGTH STRING ; Loads the Length of string in CX.

30. OFFSET

- It is an operator.
- It informs the assembler to determine the offset or displacement of a name
  data item.
- It may also determine the offset of a procedure from the start of the segment
  which contains it
- Its format is :

  e.g. : MOV AX, OFFSET NUM ; It will load the offset of NUM in the AX
  ; register.

31. ENDP-End Procedure

- This directive is used along with the name of the procedure to indicate the end
  of procedure
- ENDP defines the end of procedure.

```
e.g. :  FACT PROC FAR      ; Start procedure named FACT and informs the
                           ; assembler that the procedure is FAR.
        Body of Procedure
        FACT ENDP          ; End of Procedure FACT.
```

32. PTR-Pointer

- The pointer is an operator.
- It is used to assign a specific type to a variable or to a label.
- It is necessary to do this in any instruction where the type of operand is not
  clear.

```
e.g. (i) INC [AX]   ; The assembler does not know whether to increment the byte
                    ; pointed by BX or increment the word pointed to by BX.
```

This difficulty could be avoided by using PTR directive.

```
e.g.  INC BYTE PTR [BX]   :  it increments byte pointed to by [BX]
      INC WORD PTR [BX]   :  it increments word pointed to by [BX].
```

(ii) The PTR operator can be used to override the declared type of variable.

```
e.g.  WORD DB 23, 45, 10, 56.
```

The access to the array WORDS will be in terms of bytes.

```
MOV AL, BYTE PTR WORDS.
```

(iii) During indirect jump instruction, PTR could be used JMP [BX]. This instruction does not tell the assembler whether to code the instruction for

near jump or for far jump. For a near jump the instruction is written as

```
JMP WORD PTR [BX]
```

If far jump is required the instruction is written as

```
JMP DWORD PTR [BX].
```

33. SHORT

- Short is an operator.
- It is used to tell the assembler that only 1-byte displacement is needed to code
  a jump instruction.
- If the jump destination is after the jump instruction in the program, the
  assembler will automatically reserve 2-bytes for the displacement.

  e.g. JMP SHORT NEAR_LABEL.

34. TYPE

- The Type is an operator
- It informs the assembler to find the type of a specified variable.
- The assembler actually finds the number of bytes in the type of variable.
- For a byte type variable the assembler gives a value 1.
- For word type variable the assembler gives a value 2 and for double word 4.

13.6 Structured Programming

Structured Programming involves applying high level language programming concepts to assembly language programming. The term structured applies to certain high level program statements that can be translated to assembly code.

High level language programmers try to use a few basic program structures to solve all problems. The following are the standard program structures for high level languages.

IF.... THEN.... ELSE.
DO.....WHILE.
REPEAT.....UNTIL.
CASE.

Structured programming allows the user to think in pseudocode. Pseudocode is a descriptive sequence of operations that the user wants to carry out in the program. Pseudocode and flowcharts are two ways of thinking about the program. Pseudocode is a verbal thinking process and flowchart is a graphical thinking process. Both pseudocode and flowchart are useful to the programmer as it lets the programmer concentrate on the essential features of the program. Once the program is define d in pseudocode the programmer can code various high level structures into the assembly code.

13.6.1 IF...THEN...ELSE Program Structure

The structure has format

```
IF condition THEN
action
ELSE
action.
```

The compare and jump instructions form the basic structure for IF...THEN...ELSE program structures.

```
e.g. : L1 :   CMP AX, BX      ;   Compare data in AX and BX.
              JE NEXT         ;   If equal THEN go to NEXT
              JMP L1          ;   ELSE continue.
       NEXT : ADD AX, 30H.
```

The IF part of the structure is part of the compare and jump opcode sequence. A compare sets the appropriate flags and the jump is THEN_taken IF conditions are satisfied. ELSE is accommodated by the fact that the program continues at the instruction following the conditional jump.

13.6.2 DO...WHILE Program Structure

Its format is

WHILE some condition is present DO
action

An important point about this structure is that the condition is checked before any action is done. eg. in industrial control applications. It involves loops in the program. The DO...WHILE structure tests a condition, if conditions is satisfied loop is entered and program continues looping as long as the condition is satisfied.

```
e.g. :          MOV BX, Start       ;   Point to starting point in memory.
       NEXT :   INC BX              ;   increment to next byte.
                CMP BX, Finish      ;   Do loop while.
                JE L1               ;   If condition not satisfied exit.
                CMP [BX], NUM       ;   Compare data.
                JE L1               ;   If equal THEN exit
                JMP NEXT            ;   else continue
       L1 :     NOP
```

The DO...WHILE loop begins at memory location start and does byte search within a memory bound defined by finish. The test for BX pointing to the end of memory is CMP BX, Finish, is mode at beginning of DO...While structure.

13.6.3 REPEAT-UNTIL Program Structure

The format is

REPEAT
action
UNTIL some condition is present.

This structure performs a series of actions before the condition is check, whereas a DO...WHILE structure condition is satisfied and then actions are done. The REPEAT...UNTIL loop enters the loop and tests a condition at the end of the loop. Looping continues until the tested condition is satisfied. Both the DO...WHILE and REPEAT...UNTIL structures generally accomplish the same function.

```
e.g. :          MOV BX, Start       ;   Repeat Loop, point to starting address.
       NEXT :   CMP [BX], NUM       ;   Compare data with desired byte.
                JE L1               ;   if equal THEN exit
                INC BX              ;   ELSE continue, point to next byte.
```



```
                CMP BX, FINISH      ;   test at the end of loop.
                JNE NEXT            ;   UNTIL out of data area.
       L1 :     NOP
```

The test for end of memory. CMP BX, FINISH is done at the end of the Repeat...until loop.

Choosing between a DO...WHILE and REPEAT...UNTIL is a matter of personal preference.

13.6.4 CASE Program Structure

CASE statements are used to select one action from amongst the many choices. A simple CASE structure can be made up of repeated compare and jump instructions.

e.g. : To display a different message for any key pressed on the keyboard

```
CASE    inkey   OF
'a'     :       msg a
'b'     :       msg b
:
:
'z'     :       msg z.
```

Simulating CASE in assembly language.

```
MOV AH, 07 H        ;   get key
INT 21 H
CMP AL, 'a'         ;   Is choice = a ?
JE msg a            ;   if yes, go to label msg a.
CMP AL, 'b'         ;   is choice = b ?
JE msg b            ;   if yes, go to label msg b.
:
:
CMP AL, 'z'         ;   is choice = 'z' ?
JE msg z            ;   if yes, go to msg z.
```

13.7 Programs

Note : For outputs refer CD.

Program 1 : Add two 16 bit numbers.

Program statement :

- Assuming 4 digits are available in registers AX and BX, write a program in the assembly language of 8086 to add two 16 bit numbers.

Explanation :

- Consider that a word of data is present in the AX register and a 2nd word of data is present in the BX register.
- We have to add word in AX with the word in BX. Using ADD instruction, add the contents.
- Result will be stored in the AX register.

For example : AX = 1234 H

BX = 0100 H

```
  1234 H
+ 0100 H
-------
  1334 H
```

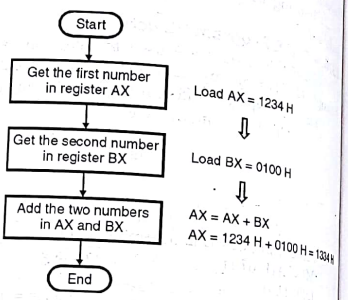
>> **Algorithm :**

Step I : Initialize the data segment.
Step II : Get the first number in AX register.
Step III : Get the second number in BX register.
Step IV : Add the two numbers.
Step V : Stop

>> **Flowchart :** Refer flowchart 1.

>> **Program :**

Instruction	Comment
.model small	
.data	
a dw 1234H	
b dw 0100H	
.code	
mov ax, @data	Initialize data section
mov ds, ax	
mov ax, a	Load number1 in ax
mov bx, b	Load number2 in bx
add ax, bx	add numbers Result in ax
end	



Flowchart 1

>> **Result : AX = 1334H**

Program 2 : Add series of N 16 bit numbers.

>> **Program statement :**

- Write an ALP to add a block of N numbers. Assume the result to be 16 bit.

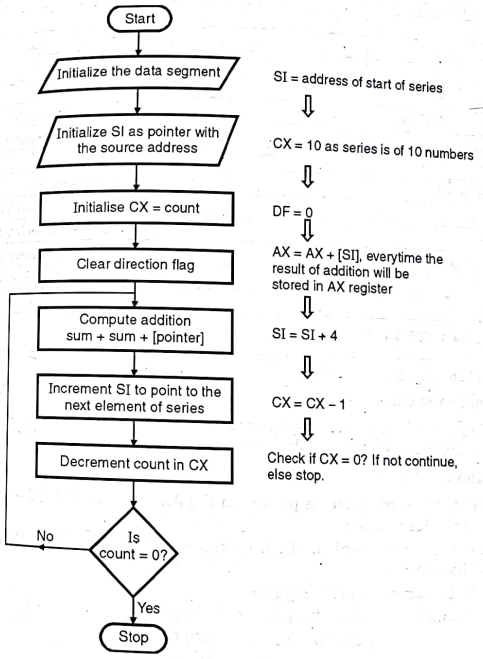
>> **Explanation :**

- Consider that a block of N bytes is present at source location.
- Let the number of bytes N = 10 for example.
- We have to add these N bytes.
- We will initialize this as count in the CX register.
- We know that source address is in the SI register. This SI register will act as pointer.
- Clear the direction flag.
- Using ADD instruction add the contents, byte by byte of the block.
- Increment SI to point to next element.
- Decrement the counter and add the contents till all the contents are added.
- Result is stored in AX.

>> **Algorithm :**

Step I : Initialise the data segment.
Step II : Initialise SI as pointer with source address.
Step III : Initialise CX register with count.

Step IV : Initialise direction flag to zero.
Step V : Add data, word by word.
Step VI : Increment pointer i.e. SI by 2 as 16 bit addition.
Step VII : Decrement counter CX.
Step VIII: Check for count in CX, if not zero goto step V else goto step IX.
Step IX : Store the result of addition.
Step X : Stop.



Flowchart 2

>> **Flowchart :** Refer flowchart 2.

Program :

Label	Instruction	Comment
	.model small	
	.data	
	series db 0111H, 0231H, 0341H, 0456H, 0578H, 06ABH, 0733H, 0845H, 0976, 0A12H	
	count dw 0AH	
	.code	
	mov ax, @data	initialise data segment
	mov ds, ax	
	mov ax, 0	
	mov si, offset blk1	initialise pointer
	mov cx, count	initialise counter
	cld	df=0
l1:	add ax, [si]	add numbers
	inc si	increment pointer
	inc si	increment pointer
	dec count	decrement counter
	jnz l1	check if all nos are added
	end	

Result : AX = 39FCH

Program 3 : To Subtract two 16 bit numbers.

Program statement :

- Assuming 4 digits are available in registers AX and BX, write a program in the assembly language of 8086 to subtract two 16 bit numbers.

Explanation :

- Consider that a word of data is present in the AX register and a 2nd word of data is present in the BX register.
- We have to subtract word in BX from the word in AX. Using SUB instruction subtract the contents.
- Result will be stored in the AX register.

For example : AX = 1234 H
BX = 0100 H

$$\begin{array}{r} 1234\ H \\ -\ 0100\ H \\ \hline 1134\ H \end{array}$$

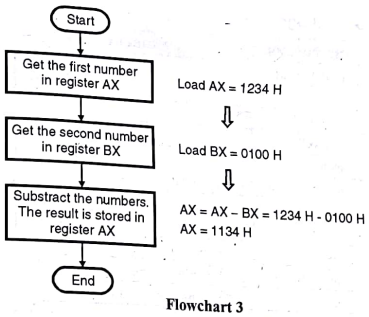
Algorithm :

Step I : Initialize the data segment.
Step II : Get the first number in AX register.
Step III : Get the second number in BX register.
Step IV : Subtract the two numbers.
Step V : Stop

Flowchart : Refer flowchart 3.

Program :

Instruction	Comment
.model small	
.data	
a dw 1234H	
b dw 0100H	
.code	
mov ax, @data	Initialize data section
mov ds, ax	
mov ax, a	Load number1 in ax
mov bx, b	Load number2 in bx
sub ax, bx	Subtract the numbers. Result in ax
end	



Flowchart 3

Result : AX = 1134H

Program 4 : To multiply two 8 bit unsigned numbers.

Program statement :

- Assuming that two digits are available in the AL and BL registers, write a program in assembly language of 8086 to multiply two 8 bit numbers.

Explanation :

- Consider that a byte of data is present in the AL register and second byte of data is present in the BL register.
- We have to multiply the byte in AL with the byte in BL.
- Using MUL instruction, multiply the contents of two registers.
- The multiplication of two 8 bit numbers may result into a 16 bit number. So result is stored in AX register.
- The MSB is stored in AH and LSB in AL.

For example : AL = 09 H
BL = 02 H

$$\begin{array}{r} 09\ H \\ \times\ 02\ H \\ \hline 0012\ H \end{array}$$

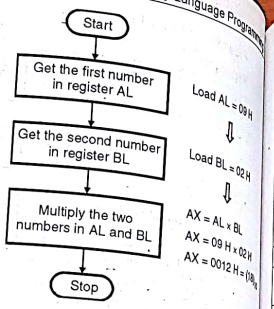
Algorithm :

Step I : Initialise the data segment.
Step II : Get the first number in AL register.
Step III : Get the second number in BL register.
Step IV : Multiply the two numbers.
Step V : Stop

Flowchart : Refer flowchart 4.

>> **Program :**

Instruction	Comment
.model small	
.data	
a db 09H	
b db 02H	
.code	
mov ax, @data	Initialize data section
mov ds, ax	
mov ah, 0	
mov al, a	Load number1 in al
mov bl, b	Load number2 in bl
mul bl	multiply numbers and result in ax
end	



Flowchart 4

>> **Result : 0012H**

Program 5 : Multiply two 16 bit unsigned numbers.

>> **Program statement :**

- Assuming that two words are available in registers AX and BX, write a program in the assembly language of 8086 to multiply two 16 bit unsigned numbers.

>> **Explanation :**

- Consider that a word of data is present in the AX register and 2nd word of data is present in the BX register.
- We have to multiply the word in AX with the word in BX. Using MUL instruction, multiply the contents of the 2 registers.
- The multiplication of the two 16 bit numbers may result into a 32 bit number. So result is stored in the DX and AX register.
- The MSB of result is stored in the DX register and LSB of result in AX register.

$$\begin{array}{rl} AX = 1234\ H & 1234\ H \\ BX = 0100\ H & \times \quad 0100\ H \\ \hline & 123400\ H \end{array}$$

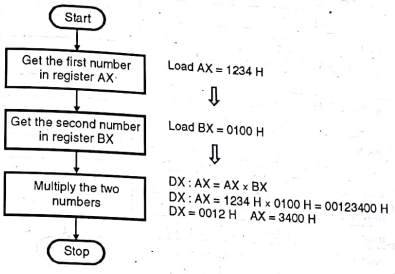
>> **Algorithm :**

Step I : Initialise the data segment.
Step II : Get the first number in AX register.
Step III : Get the second number in BX register.
Step IV : Multiply the two 16 bit numbers.
Step V : Stop

>> **Flowchart :** Refer flowchart 5.

>> **Program :**

Instruction	Comment
.model small	
.data	
a dw 1234H	
b dw 0100H	
.code	
mov ax, @data	Initialize data section
mov ds, ax	
mov ax, a	Load number1 in ax
mov bx, b	Load number2 in bx
mul bx	multiply numbers.
	Result in dx and ax
end	



Flowchart 5

>> **Result : 00123400H with DX = 0012H and AX = 3400H**

Program 6 : Multiply two 16 bit signed numbers.

>> **Program statement :**

- Assuming that two words are available in registers AX and BX, write a program in the assembly language of 8086 to multiply two 16 bit signed numbers.

>> **Explanation :**

- Consider that a word of data is present in the AX register and 2nd word of data is present in the BX register.
- We have to multiply the word in AX with the word in BX. Using IMUL instruction, multiply the contents of the 2 registers.
- The multiplication of the two 16 bit numbers may result into a 32 bit number. So result is stored in the DX and AX register.
- The MSB of result is stored in the DX register and LSB of result in AX register.

$$\begin{array}{rl} AX = 8000\ H & 8000\ H \\ BX = 2000\ H & \times \quad 2000\ H \\ \hline & F000\ 0000\ H \end{array}$$ Negation of the product generated by MUL instruction

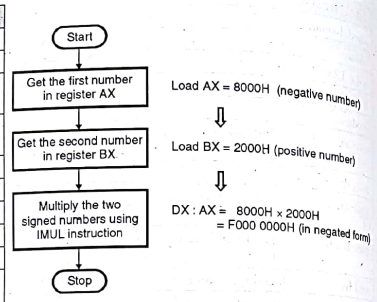
>> **Algorithm :**

Step I : Initialise the data segment.
Step II : Get the first number in AX register.
Step III : Get the second number in BX register.
Step IV : Multiply the two 16 bit signed numbers.
Step V : Stop

>> **Flowchart :** Refer flowchart 6.

>> **Program :**

Instruction	Comment
.model small	
.data	
a dw 8000H	Negative number as MSB = 1
b dw 2000H	Positive number as MSB = 0
.code	
mov ax, @data	Initialize data section
mov ds, ax	
mov ax, a	Load number1 in ax
mov bx, b	Load number2 in bx
imul bx	multiply numbers. Result in dx and ax
end	



Flowchart 6

>> **Result : F000 0000H with DX = F000H and AX = 0000H i.e. result in negated form as result is negative.**

Program 7 : Divide 16 bit unsigned numbers by an 8 bit unsigned numbers.

>> **Program statement :**

- Assuming that a word is available in the AX register and a byte is available in the BL register, write a program in assembly language of 8086 to divide 16 bit number in AX with 8 bit number in BL.

>> **Explanation :**

- Consider that a word of data is present in the AX register and byte of data is present in the BL register, we have to divide word in AX with the byte in BL.
- Using DIV instruction, divide the contents of two register. Result of division is stored in the AX register. AL contains the quotient and AH contains the remainder.

AX = 000F H (i.e. 15)
BL = 08 H

$$08 \overline{)\,000\,F\,(}\ 01$$
$$-\ 08 \quad \downarrow$$
Remainder 07 Quotient

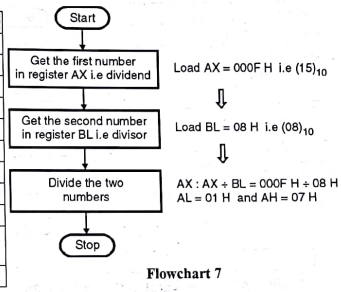
>> **Algorithm :**

Step I : Initialise the data segment.
Step II : Get the first number in AX register i.e. dividend.
Step III : Get the second number in BL i.e. divisor.
Step IV : Divide the two numbers.
Step V : Stop

>> **Flowchart :** Refer flowchart 7.

>> **Program :**

Instruction	Comment
.model small	
.data	
a dw 000FH	
b db 08H	
.code	
mov ax, @data	Initialize data section
mov ds, ax	
mov ax, a	Load number1 in ax
mov bl, b	Load number2 in bl
div bl	divide numbers. Quotient in AL and Remainder in AH
end	



Flowchart 7

>> **Result : AL = 01H (Quotient) and AH = 07H (Remainder)**

Program 8 : Divide 16 bit unsigned numbers by a 16 bit unsigned numbers.

>> **Program statement :**

- Assuming that a word is available in the AX register and a word is available in the BX register, write a program in assembly language of 8086 to divide 16 bit number in AX with 16 bit number in BX.

>> **Explanation :**

- Consider that a word of data is present in the AX register and word of data is present in the BX register, we have to divide word in AX with the word in BX.
- Using DIV instruction, divide the contents of two register. Result of division is stored in the AX register. AL contains the quotient and AH contains the remainder.

AX = 8000 H
BL = 2000 H

$$2000 \overline{)\,8000\,(}\ 04$$
$$-\ 8000 \quad \downarrow$$
Remainder 0000 Quotient

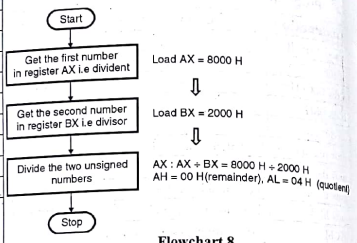
>> **Algorithm :**

Step I : Initialise the data segment.
Step II : Get the first number in AX register i.e. dividend.
Step III : Get the second number in BX i.e. divisor.
Step IV : Divide the two numbers.
Step V : Stop

>> **Flowchart :** Refer flowchart 8.

➢➢ **Program :**

Instruction	Comment
.model small	
.data	
a dw 8000H	
b dw 2000H	
.code	
mov ax, @data	Initialize data section
mov ds, ax	
mov ax, a	Load number1 in ax
mov bx, b	Load number2 in bx
div bx	divide numbers. Quotient in AL and Remainder in AH
End	



Flowchart 8

➢➢ **Result : AL = 04H (Quotient) and AH = 00H (Remainder)**

Program 9 : Divide 16 bit signed numbers by a 16 bit signed numbers.

➢➢ **Program statement :**

- Assuming that a word is available in the AX register and a word is available in the BX register, write a program in assembly language of 8086 to divide 16 bit number in AX with 16 bit number in BX.

➢➢ **Explanation :**

- Consider that a word of data is present in the AX register and word of data is present in the BX register, we have to divide word in AX with the word in BX.
- Using IDIV instruction, divide the contents of two register. Result of division is stored in the AX register. AL contains the quotient and AH contains the remainder.

AX = 8000 H 2000) 8000 (04

BL = 2000 H − 8000 ↓

Remainder 0000 Quotient

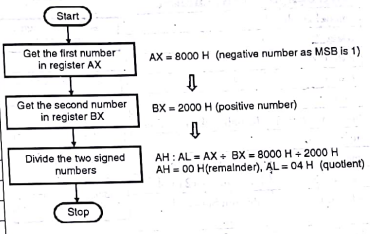
➢➢ **Algorithm :**

Step I : Initialise the data segment.
Step II : Get the first number in AX register i.e. dividend.
Step III : Get the second number in BX i.e. divisor.
Step IV : Divide the two numbers.
Step V : Stop

➢➢ **Flowchart :** Refer flowchart 9.

➢➢ **Program :**

Instruction	Comment
.model small	
.data	
a dw 8000H	
b dw 2000H	
.code	
mov ax, @data	Initialize data section
mov ds, ax	
mov ax, a	Load number1 in ax
mov bx, b	Load number2 in bx
idiv bx	divide numbers. Quotient in AL and Remainder in AH
End	



Flowchart 9

➢➢ **Result : AL = 04H (Quotient) and AH = 00H (Remainder)**

Program 10 : Add 8 bit BCD numbers.

➢➢ **Program statement :**

- Add two 2 digit BCD numbers present in the registers AL and BL.

➢➢ **Explanation :**

- Consider that a byte of data is present in the AL register and a second byte of data is present in the BL register. We have to add byte in AL with the byte in BL. Using add instruction, add the contents of 2 registers.
- Result will be stored in the AL register.
- Use DAA instruction that will check if BCD is valid, if it is not valid then 6 is added to give proper BCD result.

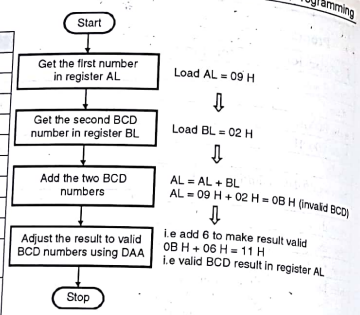
➢➢ **Algorithm :**

Step I : Initialize the data memory.
Step II : Get the first BCD number in AL.
Step III : Get the second BCD number in BL.
Step IV : Add the two BCD numbers.
Step V : Using DAA, adjust result to valid BCD number.
Step VI : Stop.

➢➢ **Flowchart :** Refer flowchart 10.

>> **Program :**

Instruction	Comment
.model small	
.data	
a db 09H	
b db 02H	
.code	
mov ax, @data	Initialize data section
mov ds, ax	
mov al, a	Load number1 in al
mov bl, b	Load number2 in bl
add al, bl	add numbers and result in al
daa	adjust result to valid BCD number
end	



Flowchart 10

>> **Result : AL = 11H**

Program 11 : Subtract 8 bit BCD numbers.

>> **Program statement :**

- Subtract two 2 digit BCD numbers which are present in the registers AL and BL.

>> **Explanation :**

- Consider that a byte of data is present in the AL register and a second byte of data is present in the BL register. We have to subtract byte in BL with byte in AL. Using SUB instruction, we will subtract the contents. The result is stored in the AL register. Using DAS instruction we will make sure that the result is valid BCD.

e.g. : AL = 32 BCD
BL = 17 BCD

```
    0 0 1 1   0 0 1 0
  -
    0 0 0 1   0 1 1 1
  ---------------------
    0 0 0 1   1 0 1 1   → invalid BCD, so subtract 6.
  -
DAS             0 1 1 0
  ---------------------
    0 0 0 1   0 1 0 1   = 15 BCD
```

- Invalid BCD, means if digit is greater than 9. If digit is invalid we subtract 6 to make it valid. DAS instruction (decimal adjust after subtraction), automatically adjusts to valid BCD result.

>> **Algorithm :**

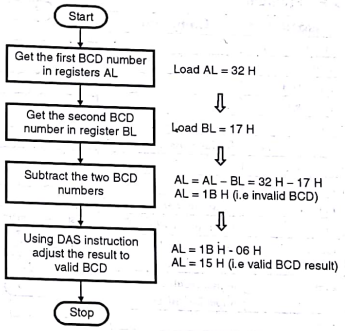
Step I : Initialize the data memory.
Step II : Get the first BCD number in AL.

Step III : Get the second BCD number in BL.
Step IV : Subtract the two BCD numbers.
Step V : Adjust result to valid BCD number.
Step VI : Stop.

>> **Flowchart :** Refer flowchart 11.

>> **Program :**

Instruction	Comment
.model small	
.data	
a db 32H	
b db 17H	
.code	
mov ax, @data	Initialize data section
mov ds, ax	
mov al, a	Load number1 in al
mov bl, b	Load number2 in bl
sub al, bl	subtract numbers and result in al
das	adjust result to valid BCD number
end	



Flowchart 11

>> **Result : AL = 15H**

Program 12 : Program to multiply two 8 bit BCD numbers.

>> **Program statement :**

- Assuming that two unpacked digits are available in the AL and BL registers. Write a program in the assembly language of 8086 to multiply two unpacked BCD numbers.

>> **Explanation :**

- Consider that two unpacked numbers are present in the AL and BL registers. We have to multiply the two numbers. The result is stored in the AX register. The AAM (BCD adjust after Multiply) is used to adjust the product to two unpacked BCD digits in AX.

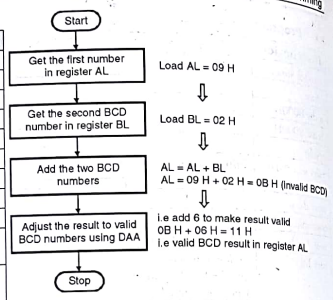
```
e.g.  AL = 0000    0100 = unpacked BCD 4
      BL = 0000    0110 = unpacked BCD 6
      MUL BL       AL x BL Result in AX.
      AX = 0000    0000 0001 1000 = 0018 H
AAM   AX = 0000    0010 0000 0100 = 0204 H i.e. Unpacked BCD for 24.
```

>> **Algorithm :**

Step I : Initialize the data segment.
Step II : Get the first unpacked BCD number.
Step III : Get the second unpacked BCD number.

➢➢ **Program :**

Instruction	Comment
.model small	
.data	
a db 09H	
b db 02H	
.code	
mov ax, @data	Initialize data section
mov ds, ax	
mov al, a	Load number1 in al
mov bl, b	Load number2 in bl
add al, bl	add numbers and result in al
daa	adjust result to valid BCD number
end	



Flowchart 10

➢➢ **Result : AL = 11H**

Program 11 : Subtract 8 bit BCD numbers.

➢➢ **Program statement :**

- Subtract two 2 digit BCD numbers which are present in the registers AL and BL.

➢➢ **Explanation :**

- Consider that a byte of data is present in the AL register and a second byte of data is present in the BL register. We have to subtract byte in BL with byte in AL. Using SUB instruction, we will subtract the contents. The result is stored in the AL register. Using DAS instruction we will make sure that the result is valid BCD.

e.g. : AL = 32 BCD — 0 0 1 1 0 0 1 0
BL = 17 BCD — 0 0 0 1 0 1 1 1
0 0 0 1 1 0 1 1 → invalid BCD, so subtract 6.
DAS — 0 1 1 0
0 0 0 1 0 1 0 1 = 15 BCD

- Invalid BCD, means if digit is greater than 9. If digit is invalid we subtract 6 to make it valid. DAS instruction (decimal adjust after subtraction), automatically adjusts to valid BCD result.

➢➢ **Algorithm :**

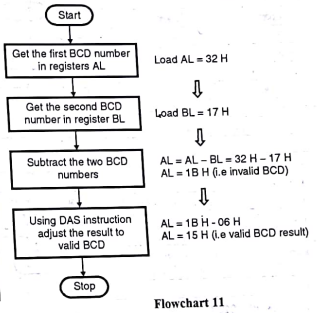
Step I : Initialize the data memory.
Step II : Get the first BCD number in AL.

Step III : Get the second BCD number in BL.
Step IV : Subtract the two BCD numbers.
Step V : Adjust result to valid BCD number.
Step VI : Stop.

➢➢ **Flowchart :** Refer flowchart 11.

➢➢ **Program :**

Instruction	Comment
.model small	
.data	
a db 32H	
b db 17H	
.code	
mov ax, @data	Initialize data section
mov ds, ax	
mov al, a	Load number1 in al
mov bl, b	Load number2 in bl
sub al, bl	subtract numbers and result in al
das	adjust result to valid BCD number
end	



Flowchart 11

➢➢ **Result : AL = 15H**

Program 12 : Program to multiply two 8 bit BCD numbers.

➢➢ **Program statement :**

- Assuming that two unpacked digits are available in the AL and BL registers. Write a program in the assembly language of 8086 to multiply two unpacked BCD numbers.

➢➢ **Explanation :**

- Consider that two unpacked numbers are present in the AL and BL registers. We have to multiply the two numbers. The result is stored in the AX register. The AAM (BCD adjust after Multiply) is used to adjust the product to two unpacked BCD digits in AX.

e.g. AL = 0000 0100 = unpacked BCD 4
BL = 0000 0110 = unpacked BCD 6
MUL BL AL × BL Result in AX.
AX = 0000 0000 0001 1000 = 0018 H
AAM AX = 0000 0010 0000 0100 = 0204 H i.e. Unpacked BCD for 24.

➢➢ **Algorithm :**

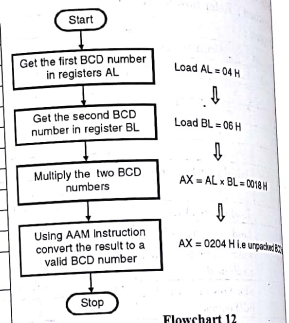
Step I : Initialize the data segment.
Step II : Get the first unpacked BCD number.
Step III : Get the second unpacked BCD number.

Step IV : Multiply the two numbers.
Step V : Adjust result to valid unpacked BCD number in AX.
Step VI : Stop.

>> **Flowchart :** Refer flowchart 12.

>> **Program :**

Instruction	Comment
.model small	
.data	
a db 04H	
b db 06H	
.code	
mov ax, @data	Initialize data section
mov ds, ax	
mov ah, 0	
mov al, a	Load number1 in al
mov bl, b	Load number2 in bl
mul bl	multiply numbers and result in ax
aam	adjust result to valid unpacked BCD
end	



Flowchart 12

>> **Result : 0204H**

Program 13 : Divide two BCD numbers .

>> **Program statement :**

- Assuming that a word is available in the AX register and a byte is available in BL register, write a program in assembly language of 8086 to divide 16 bit number in AX with 8 bit number in BL.

>> **Explanation :**

- Consider that a word of data is present in the AX register in unpacked BCD byte of data is present in the BL register, we have to divide word in AX by byte in BL.
- The AAD is before the division is perfomed to adjust the result after division unpacked BCD.
- Using DIV instruction, divide the contents of two register. Result of division stored in the AX register. AL contains the quotient and AH contains the remainder.

e.g AX = 0000 0110 0000 0111 = unpacked BCD 67
BL = 0000 1001 = unpacked BCD 9
AAD AX = 0000 0000 0100 0011 = 0043 H.
DIV BL AX / BL Result in AX.
AX = 0000 0100 0000 0111 = 0407 H

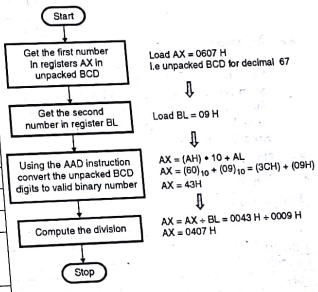
>> **Algorithm :**

Step I : Initialise the data segment.
Step II : Get the first number in AX register i.e. dividend.
Step III : Get the second number in BL i.e. divisor.
Step IV : Divide the two numbers.
Step V : Stop

>> **Flowchart :** Refer flowchart 13.

>> **Program :**

Instruction	Comment
.model small	
.data	
a dw 0607H	
b db 09H	
.code	
mov ax, @data	Initialize data section
mov ds, ax	
mov ax, a	Load number1 in ax
mov bl, b	Load number2 in bl
div bl	divide numbers. Quotient in AL and Remainder in AH
end	



Flowchart 13

>> **Result : AL = 07H (Quotient) and AH = 04H (Remainder)**

Program 14 : Add two 8 bit ASCII numbers.

>> **Program statement :**

- Add two 2 digit ASCII numbers present in the registers AL and BL.

>> **Explanation :**

- Consider that a byte of data is present in the AL register and a second byte of data is present in the BL register. We have to add byte in AL with the byte in BL. Using add instruction, add the contents of 2 registers.
- Result will be stored in the AX register.
- Use AAA instruction that will check if BCD is valid, if it is not valid then 6 is added to give proper BCD result.

Assume AL = 0 0 1 1 0 1 0 1 ASCII 5
BL = 0 0 1 1 1 0 0 1 ASCII 9

Example : ADD AL, BL

```
    0 0 1 1     0 1 0 1
+   0 0 1 1     1 0 0 1
    0 1 1 0     1 1 1 0   →  Result of addition
+               0 1 1 0      Invalid BCD
AAM             0 0 0 0     0 1 0 0
Carry →  1      Clear higher nibble
```

(04 H) → Result in AL valid BCD.
(01 H) → Result in AH. The carry is stored in AH register

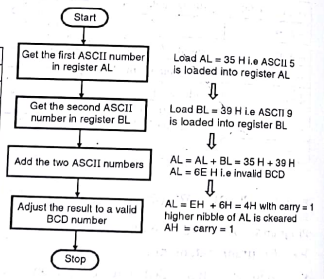
Algorithm :

Step I : Initialize the data memory.
Step II : Get the first ASCII number in AL.
Step III : Get the second ASCII number in BL.
Step IV : Add the two ASCII numbers.
Step V : Using AAA, adjust result to valid BCD number.
Step VI : Stop.

Flowchart : Refer flowchart 14.

Program :

Instruction	Comment
.model small	
.data	
a db 35H	
b db 39H	
.code	
mov ax, @data	Initialize data section
mov ds, ax	
mov al, a	Load number1 in al
mov bl, b	Load number2 in bl
add al, bl	add numbers and result in al
aaa	adjust result to valid BCD number
end	



Flowchart 14

Result : AX = 0104H

Program 15 : Subtract two 8 bit ASCII numbers.

Program statement :

- Subtract two 2 digit ASCII numbers present in the registers AL and BL.

Explanation :

- Consider that a byte of data is present in the AL register and a second byte of data is present in the BL register. We have to subtract byte in AL with the byte in BL. Using sub instruction, add the contents of 2 registers.

- Result will be stored in the AX register.
- Use AAS instruction that will check if BCD is valid, if it is not valid then 6 is added to give proper BCD result.

Assume AL = 0 0 1 1 1 0 0 1 ASCII 9
BL = 0 0 1 1 0 1 0 1 ASCII 5

Example : SUB AL, BL

```
    0 0 1 1     1 0 0 1
–   0 0 1 1     0 1 0 1
    0 1 1 0     0 1 0 0  →  Result of subtraction
                            valid BCD
AAM             0 0 0 0     0 1 0 0
Carry →  0      Clear higher nibble
```

(04 H) → Result in AL valid BCD.
(00 H) → Result in AH.

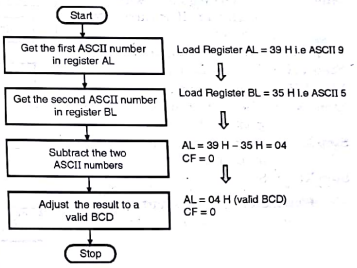
Algorithm :

Step I : Initialize the data memory.
Step II : Get the first ASCII number in AL.
Step III : Get the second ASCII number in BL.
Step IV : Subtract the two ASCII numbers.
Step V : Using AAS, adjust result to valid BCD number.
Step VI : Stop.

Flowchart : Refer flowchart 15.

Program :

Instruction	Comment
.model small	
.data	
a db 39H	
b db 35H	
.code	
mov ax, @data	Initialize data section
mov ds, ax	
mov al, a	Load number1 in al
mov bl, b	Load number2 in bl
sub al, bl	subtract the numbers and result in al
aas	adjust result to valid BCD number
end	



Flowchart 15

Result : AX = 0004H

Program 16 : To find the largest among the block of data.

Lab assignment

>> Program statement :

- Given an array(block) of 10 numbers. Write a program in ALP of 8086 to find the maximum number amongst the 10 numbers.

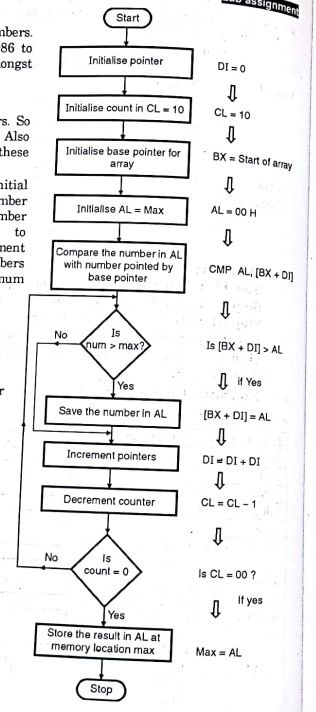
>> Explanation :

- We have an array of 10 numbers. So we initialize the counter with 10. Also we initialize a pointer to point these numbers.
- Compare first number with initial maximum number i.e. zero. If number > maximum number, save number otherwise increment pointer to compare next number. Decrement counter, compare till all the numbers are compared. Store the maximum number in MAX.

>> Algorithm :

Step I : Initialize the data segment.
Step II : Initialize pointer.
Step III : Initialize counter.
Step IV : Initialize base pointer for array.
Step V : Compare number with maximum. If no carry goto step VII
Step VI : Save the maximum number.
Step VII : Decrement counter.
Step VIII : Increment pointer.
Step IX : Check if count = 0. If not goto step V.
Step X : Store the maximum number.
Step XI : Stop.

>> Flowchart : Refer flowchart 16.



Flowchart 16

>> Program :

Label	Instruction	Comment
	.model small	
	.stack 100	
	.data	
	array db 61h, 05h, 42h, 05H, 12H, 15h, 09h, 14h, 56h, 38h	Array of 10nos
	max db 0	
	.code	
	mov ax, @data	Initialize DS
	mov ds, ax	
	xor di, di	Initialise pointer
	mov cl, 10	Initialise counter
	lea bx, array	Initialise base pointer for array
	mov al, max	Get maximum number
back:	cmp al, [bx+di]	Compare number with maximum
	jnc skip	
	mov dl, [bx+di]	If no > this no swap
	mov al, dl	
skip:	inc di	Increment pointer
	dec cl	Decrement counter
	jnz back	check whether all the nos have been scanned
	mov max, al	Store maximum number
	end	

>> Result : 61H

Program 17 : To find the smallest among the block of data.

Lab assignment

>> Program statement :

- Given an array (block) of 10 numbers. Write a program in ALP of 8086 to find the minimum number amongst the 10 numbers.

>> Explanation :

- We have an array of 10 numbers. So we initialize the counter with 10. Also we initialize a pointer to point these numbers.
- Compare first number with initial number inv array. If number < maximum number, save number otherwise increment pointer to compare next number. Decrement counter, compare till all the numbers are compared. Store the minimum number in MIN.

>> Algorithm :

Step I : Initialize the data segment.
Step II : Initialize pointer.
Step III : Initialize counter.

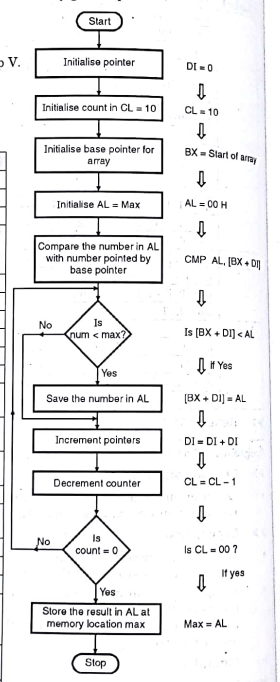
Step IV : Initialize base pointer for array.
Step V : Compare number with minimum. If no carry goto step VII
Step VI : Save the minimum number.
Step VII : Decrement counter.
Step VIII : Increment pointer.
Step IX : Check if count = 0. If not goto step V.
Step X : Store the minimum number.
Step XI : Stop.

➢➢ **Flowchart :** Refer flowchart 17.

➢➢ **Program :**

Label	Instruction	Comment
	.model small	
	.stack 100	
	.data	
	array db 61h, 05h, 42h, 25h, 12h, 15h, 09h, 14h, 56h, 38h	Array of 10nos
	min db 0	
	.code	
	mov ax, @data	Initialize DS
	mov ds, ax	
	xor di, di	Initialise pointer
	mov cl, 10	Initialise counter
	lea bx, array	Initialise base pointer for array
	mov al, max	Get maximum number
back:	cmp al, [bx+di]	Compare number with maximum
	jc skip	
	mov dl, [bx+di]	If no < this no swap
	mov al, dl	
skip:	inc di	Increment pointer
	dec cl	Decrement counter
	jnz back	check whether all the nos have been scanned
	mov min, al	Store minimum number
	end	

➢➢ **Result : 05H**



Flowchart 17

Program 18 : Program to sort the numbers in ascending order.

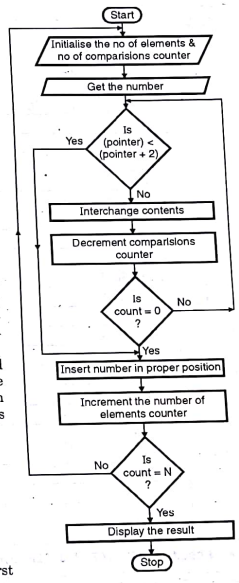
Lab assignment

➢➢ **Program statement :**

- Write a program in assembly language of 8086 to sort the given N words from a block in ascending order.

➢➢ **Explanation :**

- Consider that a block of N words is present. Now we have to arrange these N words in ascending order, Let N = 4 for example. We will use SI as pointer to point the block of N words.
- Initially in the first iteration we compare first number with the second number. If first number < second number, don't interchange the contents, otherwise if first number > second number swap the contents.
- In the next iteration we go on comparing the first number with third number. If first number < third number, don't interchange the contents. If first number > third number then swapping will be done.
- Since the first two numbers are in ascending order the third number will go to first place, first number in second place and second number will come in third place in the second iteration only if first number > third number.
- In the next iteration first number is compared with fourth number. So on comparisons are done till all N numbers are arranged in ascending order. This method requires approximately n comparisons.



Flowchart 18

➢➢ **Algorithm :**

Step I : Initialise the data segment.
Step II : Initialise the number of elements counter.
Step III : Initialise the number of comparisions counter.
Step IV : Compare the elements. If first element < second element goto step VIII else goto step V.
Step V : Swap the elements.
Step VI : Decrement the comparison counter.
Step VII : Is count = 0 ? if yes goto step VIII else goto step IV.
Step VIII : Insert the number in proper position

Step IX : Increment the number of elements counter.
Step X : Is count = N ? If yes, goto step XI else goto step II
Step XII : Stop.

>> **Flowchart :** Refer flowchart 18.

>> **Program :**

Label	Instruction	Comment
	.MODEL SMALL	
	.STACK 100	
	.DATA	
	NUM DW 0102H, 0154H, 0070H, 0005H	
	.CODE	
	mov ax, @data	Initialise data segment
	mov ds, ax	
	mov dx, 2	DX has location where number to be placed
loop2 :	mov cx, dx	
	dec cx	CX number of comparisons required
	mov si, cx	
	add si, si	si*2 as word ptr
	mov ax, num[si]	Get the number
loop1:	cmp num[si-2], ax	Compare it with previous number
	jbe next	if prev num <= this num goto next
	mov di, num[si-2]	else
	mov num[si] , di	insert in new posn
	dec si	decrement si
	dec si	
	dec cx	Decrement comparison counter
	jnz loop1	If not zero, repeat
next:	mov num[si], ax	Insert the number in proper position
	inc dx	Next number to be inserted
	cmp dx, 4	Check if all numbers are inserted
	jbe loop2	If not continue
	end	

>> **Result : 0005H, 0070H, 0102H, 0154H**

Program 19 : Program to sort the numbers in descending order. **Lab assignment**

>> **Program statement :**

- Write a program in assembly language of 8086 to sort the given N words from a block in descending order.

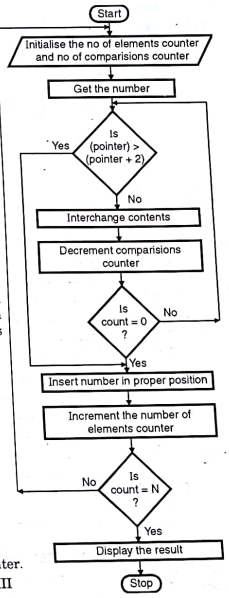
>> **Explanation :**

- Consider that a block of N words is present.
- Now we have to arrange these N words in descending order, Let N = 4 for example.
- We will use SI as pointer to point the block of N words.
- Initially in the first iteration we compare the first number with the second number. If first number > second number don't interchange the contents. If first number < second number swap their contents. Now at the end of this iteration first two elements are sorted in descending order.
- In the next iteration we will compare the first number alongwith third. If first > third don't interchange contents otherwise swap the contents. At the end of this iteration first three elements are sorted in descending order. Go on comparing till all the elements are arranged in descending order. This method requires approximately n comparisons.

>> **Algorithm :**

Step I : Initialise the data segment.
Step II : Initialise the number of elements counter.
Step III : Initialise the number of comparisons counter.
Step IV : Compare the elements.
If first element > second element goto step VIII else goto step V.
Step V : Swap the elements.
Step VI : Decrement the comparision counter.
Step VII : Is count = 0 ? If yes, goto step VIII else goto step IV.
Step VIII: Insert the number in proper position.
Step IX : Increment the elements counter.
Step X : Is count = N ? If yes, goto step XI else goto step II.
Step XI : Stop.

>> **Flowchart :** Refer flowchart 19.



Flowchart 19

➢➢ **Program :**

Label	Instruction	Comment
	.MODEL SMALL	
	.STACK 100	
	.DATA	
	NUM DW 0102H, 0154H, 0070H, 0005H	
	.CODE	
	mov ax, @data	Initialise data segment
	mov ds, ax	
	mov dx, 2	DX has location where num is to be inserted
loop2:	mov cx, dx	
	dec cx	CX number of comparisons required
	mov si, cx	
	add si, si	si*2 as word ptr
	mov ax, num[si]	Get the number
loop1:	cmp num[si-2], ax	Compare it with previous number
	jae next	if prev num >= this num goto next
	mov di, num[si-2]	else
	mov num[si], di	insert in new posn
	dec si	decrement si
	dec si	
	dec cx	Decrement comparison counter
	jnz loop1	If not zero, repeat
next:	mov num[si] , ax	Insert the number in proper position
	inc dx	Next number to be inserted
	cmp dx, 4	Check if all numbers are inserted
	jbe loop2	If not continue
	end	

➢➢ **Result : 0154H, 0102H, 0070H, 0005H**

Program 20 : Data block Transfer using string instructions. **Lab assignment**

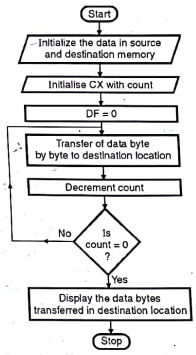
➢➢ **Program statement :**

- Write an ALP to move a block of N bytes of data from source to destination and display the result.

➢➢ **Explanation :**

- Consider that a block of data of N bytes is present at source location. Now this block of N bytes is to be moved from source location to a destination location.
- Let the number of bytes N = 10.
- We will have to initialize this as count in the CX register.

- We know that source address is in the SI register and destination address is in the DI register.
- Clear the direction flag.
- Using the string instruction move the data from source location to the destination location. It is assumed that data is moved within the same segment. Hence the DS and ES are initialized to the same segment value.



Flowchart 20

➢➢ **Algorithm :**

Step I : Initialise the data in the source memory and destination memory.
Step II : Initialise SI and DI with source and destination address.
Step III : Initialise CX register with the count.
Step IV : Initialise the direction flag to zero.
Step V : Transfer the data block byte by byte to destination.
Step VI : Decrement CX.
Step VII : Check for count in CX, if not zero goto step V else goto step VIII.
Step VIII : Stop.

➢➢ **Flowchart :** Refer flowchart 20.

➢➢ **Program :**

Label	Instruction	Comment
	.model small	
	.data	
	src_blk db 01, 02, 03, 04, 05, 06, 07, 08, 09, 0AH	
	dest_blk db 10 dup(?)	
	count dw 0AH	
	.code	
	mov ax, @data	initialize data & extra segment
	mov ds, ax	
	mov es, ax	
	mov si, offset src_blk	si to point to source block
	mov di, offset dest_blk	di to point to destination block
	mov cx, count	initialize counter
	cld	df=0
again :	rep movsb	transfer contents
	mov di, offset dest_blk	di to point to destination block
	mov bh, 0Ah	initialize counter
up:	mov bl, [di]	store result in bl
	end	

➢➢ **Result : 01H, 02H, 03H, 04H, 05H, 06H, 07H, 08H, 09H, 0AH**

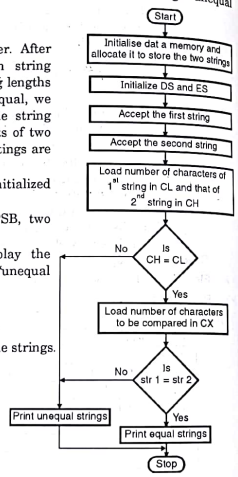
Program 21 : Program to compare given strings.

➢➢ **Program statement :**

- Write a program in the ALP of 8086 to check the data in two strings are equal, if equal display the message "equal strings", and if not display the message "unequal strings".

➢➢ **Explanation :**

- We will accept two strings from the user. After accepting the strings, the first step in string comparison is to check whether their string lengths are equal. If the string lengths are not equal, we print the message unequal strings. If the string lengths are equal, we check if the contents of two strings are equal. The lengths of the two stings are initialized in the CX register.
- The source and destination address are initialized in DS : SI and ES : DI registers.
- Using the string instruction REPE CMPSB, two data are compared character by character.
- If all the characters are matching display the message "equal strings" otherwise display "unequal strings".



Flowchart 21

➢➢ **Algorithm :**

Step I : Initialize the data memory.
Step II : Allocate data memory to save the strings.
Step III : Initialize DS and ES register.
Step IV : Accept the first string.
Step V : Accept the second string.
Step VI : Load the number of characters of first string in CL.
Step VII : Load the number of characters of second string in CH register.
Step VIII: Compare the lengths of the two strings. If not go to step XIII.
Step IX : Load number of characters to be compared in CX.
Step X : Compare the strings, character by character. If not same goto step XIII.
Step XI : Print "equal strings" using Macro.
Step XII : Jump to step XIV.
Step XIII: Print "unequal strings" using Macro.
Step XIV: Stop.

➢➢ **Flowchart :** Refer flowchart 21.

➢➢ **Program :**

Label	Instruction	Comment
	PRINT MACRO MES	Macro to display string
	MOV AH, 09H	
	LEA DX, MES	
	INT 21H	
	ENDM	
	.MODEL SMALL	
	.DATA	
	MS1 DB 10, 13, "ENTER FIRST STRING : $"	
	MS2 DB 10, 13, "ENTER SECOND STRING : $"	
	MS3 DB 10, 13, "EQUAL STRINGS $"	
	MS4 DB 10, 13, "UNEQUAL STRINGS $"	
	MS5 DB 10, 13, "$"	
	BUFF DB 25 , ? , 25 DUP('$')	
	BUFF1 DB 25 , ? , 25 DUP('$')	
	.CODE	
	mov ax, @data	Initialize DS and ES
	mov ds, ax	
	mov es, ax	
	print ms1	
	mov ah, 0ah	accept first string
	lea dx, buff	
	int 21h	
	lea si, buff	
	print ms2	
	mov ah, 0ah	accept other string
	lea dx, buff1	
	int 21h	
	mov cl, buff+1	Number of characters in str1
	mov ch, buff1+1	Number of characters in str2

Label	Instruction	Comment
	cmp ch, cl	check if length is same
	jnz para	
	mov ch, 00	
	mov cl, buff+1	Number of characters
	lea di, buff1	
	cld	
	repe cmpsb	Compare string char by char
	jnz para	if not same goto PARA
	print ms3	Strings are equal
	jmp quit	
para:	print ms4	Strings are Unequal
quit:	mov ah, 4ch	
	int 21h	
	end	

Program 22 : Program for ASCII-Binary conversion.

➢➢ **Program statement :**

- Write a program in the assembly language of 8086 to convert a given ASCII number into its binary code equivalent. Draw flowchart.

➢➢ **Explanation :**

- For ASCII-Gray conversion.
  (i) Divide the ASCII number by 2 until the quotient is 0.
- e.g. to convert 0EH i.e. decimal 14
- (1110 B) is the binary equivalent of 0E H.

2	14	
2	7	0
2	3	1
2	1	1
2	0	1

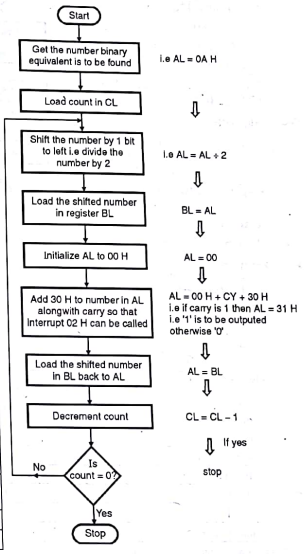
➢➢ **Algorithm :**

Step I : Get the number whose binary code equivalent is to be found.
Step II : Initialise count in CL = 08 H
Step III : Divide the number by 2 i.e. shift the number by 1 it to the left.
Step III : Display the bit shifted in carry.
Step IV : Decrement count
Step V : Check if count =0 ?
Step VI : if yes goto step VII else goto step III.
Step VII : Stop.

➢➢ **Flowchart :** Refer flowchart 22.

➢➢ **Program :**

Label	Instruction	Comment
	.model small	
	.data	
	a db 0AH	
	.code	
	mov ax, @data	Initialize data section
	mov ds, ax	
	mov al, a	Load number1 in al
	mov cl , 08H	
	mov ah, 00h	ah=00
up :	shl al, 01h	divide the number by 2 and SHL gives the same result
	mov bl, al	
	mov al, 00H	
	adc al, 30h	
	mov dl, al	
	mov ah, 02h	
	int 21h	
	mov al,bl	
	dec cl	
	jnz up	
	mov ah, 4cH	Terminate Program
	int 21H	
	end	



Flowchart 22

➢➢ **Result : 0000 1010**

Program 23 : Program for Binary-ASCII conversion.

➢➢ **Program statement :**

- Write a program in the assembly language of 8086 to convert a given binary number into its ASCII equivalent. Draw flowchart.

➢➢ **Explanation :**

- For Binary-ASCII conversion.
  (i) Multiply the binary number by increasing powers of 2 i.e. 2^0, 2^1, 2^2.... Starting from LSB

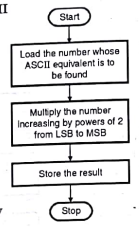
- e.g. to convert "0000 1010" to ASCII
  $= 0*2^0 + 1*2^1 + 0*2^2 + 1*2^3 + 0*2^4 + 0*2^5 + 0*2^6 + 0*2^7$
  $= 0 + 2 + 0 + 8 + 0 + 0 + 0 + 0$
  $= 10$
- 10 is the ASCII equivalent of "(0000 1010 B)".

➢➢ **Algorithm :**

Step I : Get the number whose ascii code equivalent is to be found.

Step II : Multiply the number by powers of 2.

Step III : Stop.



Load AL = " 0000 1010 B "

⇩

$AL = 0 \times 2^0 + 1 \times 2^1 + 0 \times 2^2 + 1 \times 2^3 + 0 \times 2^4 + 0 \times 2^5 + 0 \times 2^6 + 0 \times 2^7$

$AL = (10)_{10} = (0A)_{16}$

⇩

ANS = 0A H

Flowchart 23

➢➢ **Flowchart :** Refer flowchart 23.

➢➢ **Program :**

Label	Instruction	Comment
	.model small	
	disp macro arg	
	mov al, arg	
	and al, 0f0H	
	mov cl, 04H	
	ror al, cl	
	dispchar al, m1	
	mov al, arg	
	and al, 0fH	
	dispchar al, m2	
	endm	
	dispchar macro arg, l1	
	mov dl, arg	
	cmp dl, 09H	
	jle l1	
	add dl, 07H	
l1:	add dl, 30h	
	mov ah, 02H	
	int 21h	
	endm	
	.data	
	a db "00001010B"	
	res db ?	

Label	Instruction	Comment
	ans db ?	
	cnt db ?	
	.code	
	mov ax, @data	Initialize data section
	mov ds, ax	
	mov al, a	Load number1 in al
	mov cl, 07H	
	lea si, a	
up :	mov cnt, cl	
	call power	
	mov bl, res	
	mov al, [si]	
	sub al, 30H	
	mul bl	
	add ans, al	
	inc si	
	dec cl	
	jnz up	
	disp ans	
	mov ah, 4Ch	
	int 21h	
	power proc near	
	mov ch, cnt	
	mov bl, 01	
up1:	mov al, 02h	
	mul bl	
	mov bl, al	
	dec ch	
	cmp ch, 00	
	jnz up1	
	mov res, bl	
	ret	
	power endp	
	end	

➢➢ **Result : 0A**

Program 24 : Program for ASCII-BCD conversion.

➢➢ **Program statement :**

- Write a program in the assembly language of 8086 to convert a given ASCII number into its BCD code equivalent. Draw flowchart.

>> Explanation :

- For ASCII-BCD conversion.
  (i) Subtract the ASCII number by 30 H to get the valid BCD number.

```
  35 H
- 30 H
-------
  05 H
```

- e.g. to convert 35 H i.e. ASCII 5
- 05 is the BCD equivalent of 35 H.

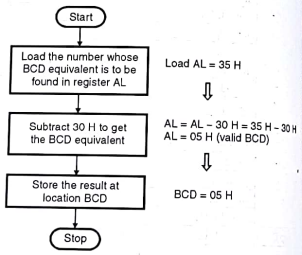
>> Algorithm :

Step I : Get the number whose BCD code equivalent is to be found.
Step II : Subtract the number by 30 H to get its BCD equivalent
Step III : Store the result at location BCD
Step IV : Stop.

>> Flowchart : Refer flowchart 24.

>> Program :

Instruction	Comment
.model small	
.data	
a db 35 H	
.code	
bcd db ?	
mov ax, @data	Initialize data section
mov ds, ax	
mov al, a	Load number1 in al
sub al, 30 H	
mov bcd, al	store the BCD result
mov ah, 4c H	Terminate Program
int 21 H	
end	



Flowchart 24

>> Result : 05

Program 25 : Program for BCD-ASCII conversion.

>> Program statement :

- Write a program in the assembly language of 8086 to convert a given BCD number into its ASCII code equivalent. Draw flowchart.

>> Explanation :

- For BCD-ASCII conversion.
  (i) Mask the lower digit of given BCD number.
  (ii) Add 30H to it to convert it to equivalent ASCII number
  (iii) Now mask the upper digit of given BCD number.
  (iv) Rotate the upper digit by 4 to make it lower nibble and then add 30H to get equivalent ASCII number.

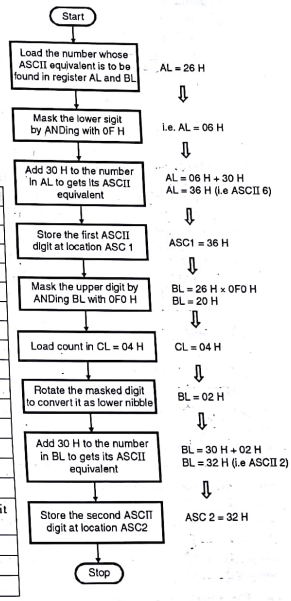
>> Algorithm :

Step I : Get the number whose ASCII code equivalent is to be found.
Step II : Mask the lower digit of given BCD number.
Step III : Add 30 H to masked digit get its ASCII equivalent
Step IV : Store the ASCII digit at location ASC1
Step V : Mask the upper digit of given BCD number.
Step VI : Rotate the upper digit by 4 to make it lower nibble
Step VII : Add 30 H to masked digit get its ASCII equivalent
Step VIII: Store the ASCII digit at location ASC2
Step IX : Stop.

>> Flowchart : Refer flowchart 25.

>> Program :

Instruction	Comment
.model small	
.data	
bcdno db 26H	
asc1 db ?	
asc2 db ?	
.code	
mov ax, @ data	Initialize data section
mov ds, ax	
mov al, bcdno	Load bcd number in al
mov bl, al	store the number in bl
and bl, 0f H	separate the lower BCD digit
add bl, 30 h	add 30 H to make it ASCII
mov asc1, bl	Store the ASCII digit in asc1
and al, 0f0 h	separate the upper bcd digit
mov cl, 04 h	load cl = 04 h
ror al, cl	rotate the upper digit to make it lower nibble
add al, 30 h	add 30 H to make it ASCII
mov asc2, al	Store the ASCII digit in asc2
end	



>> Result : ASC1 =32 and ASC2 = 36

Flowchart 25

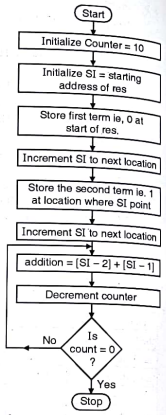
Program 26 : To find Fibonacci series of N given terms.

>> Program Statement

Write a program in the assembly language of 8086 to compute the Fibonacci series of N terms.

>> Explanation

- The Fibonacci series is
  0 1 1 2 3 5 8 13 21....
  The first two terms are 0 1, the third term is computed as 0 + 1 = 1, fourth term 1 + 1 = 2, fifth term = 1 + 2 = 3.
  i.e. n^{th} term = $(n-2)^{th}$ term + $(n-1)^{th}$ term.
  We will find the series say for 10 terms i.e. N = 10. We will initialize count = 10. An array res is initialized, so that the computed terms will be stored in it. SI is initialized to start of res. The first term i.e. 0 is stored at start. SI is incremented and next term is stored i.e. 1. Then addition of contents of two locations i.e. 0 and 1 is done. The next term is computed and result is stored at third location. Process is repeated till all 10 terms are computed. Display the result. The result in Hex will be
  0 1 1 2 3 5 8 0D 15 22.



Flowchart 26

>> Algorithm

Step I : Initialize the data section.
Step II : Initialize the counter = 10 i.e. 0AH.
Step III : Initialize SI to starting address of res.
Step IV : Store the first term at location where SI is pointing.
Step V : Increment SI to point next location.
Step VI : Store next term '1' at location where SI is pointing.
Step VII : Increment SI to next location.
Step VIII : next term = [SI – 2] + [SI – 1]
Step IX : Store the result to location that SI indicates.
Step X : Increment SI to point next location.
Step XI : Decrement counter.
Step XII : Check if count = 0, if not go to step VIII.
Step XIII : Stop.

>> Flowchart : Refer flowchart 26.

>> Program

Label	Instruction	Comment
	.model small	
	.data	
	count dw 0AH	count of no. of terms let n=10
	res db 10 dup(?)	
	.code	
	mov ax, @data	initialize data segment
	mov ds, ax	
	mov cx, count	initialize counter
	mov si, offset res	initialize si
	mov al, 00h	initialize al=0 i.e.first term of series
	mov bl, 01h	initialize bl=1 i.e.next term of series
	mov [si], al	store the first term of series
	inc si	increment si to next location
	mov [si], bl	store next term in series
	inc si	increment si to next location
up :	mov al, [si-2]	
	mov bl, [si-1]	
	add al, bl	compute the next term in series
	mov [si], al	store the computed term
	inc si	increment si to next location
	loop up	repeat till all terms are calculated
	mov ah, 4ch	terminate program
	int 21h	
	end	

Program 27 : Program to Display 'A' to 'Z' on CRT screen.

>> Program statement :

- Write a program in the assembly language of 8086 to display 'A' to 'Z' on the CRT screen.

Explanation :

- DOS interrupt function 02 displaces the contents of DL (ASCII code) on the screen. By loading the ASCII code of 'A' in the DL register, loading AH register with 02 and calling INT 21H it is possible to display the character 'A' on the screen.
- By increment the contents of DL the ASCII codes of B to Z can be obtained.
- The count of 26 is loaded into register to keep track of the display 'A' to 'Z' characters on the screen.

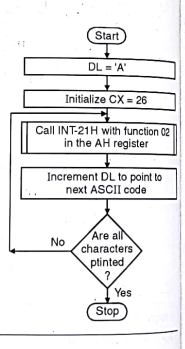
Algorithm :

Step I : Initialize DL-register with ASCII code of 'A'.
Step II : Initialize CX-register with number of alphabets (26).
Step III : CALL INT-21H with function 02 in the AH register.
Step IV : Increment 'DL' to have next ASCII code.
Step V : Check whether all the characters have been printed. If not go to step III.
Step VI : Stop.

Flowchart : Refer flowchart 27.

Program :

Label	Instruction	Comment
	.model small	
	.stack 100	
	.code	
	mov dl, 'A'	starting character to be displayed is A
	mov cl, 26	number of characters to be printed
print:	mov ah, 02h	function 02 under int21h
		(display character)
	int 21h	
	inc dl	
	loop print	go to next character
	mov ah, 4ch	
	int 21h	
	end	



Program 28 : To validate a user using Password.

Program statement :

- Write a program in ALP of 8086 to read a password from the keyboard, check whether the entered password matches with the stored password in memory, if matches display a message "Welcome to Programming" and if not display message "Invalid password". Use suitable DOS interrupts. Draw flowchart.

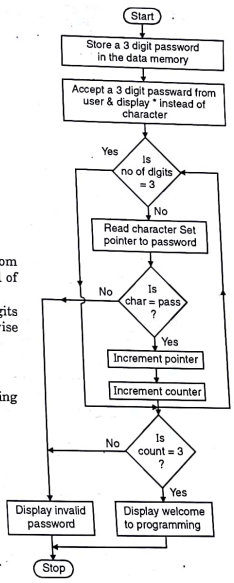
Explanation :

- Initialize the data memory and store the password on it.
- Let our password be a 3 digit password abc.
- Accept a password from the user.
- Check if the number of characters is less than 3. If yes, then character by character compare the entered password with the stored password. If character matches increment the pointer and compare next character, otherwise display invalid password. If all the characters match display "Welcome to Programming".

Algorithm :

Step I : Initialize the data memory.
Step II : Accept a 3 digit password from the user and display * instead of entered character.
Step III : Check if entered number of digits is 3, if yes go to step X otherwise go to step IV.
Step IV : Read character.
Step V : Set pointer to password.
Step VI : Compare characters. If matching continue otherwise go to step XIII.
Step VII : Increment pointer to next character.
Step VIII : Increment counter.
Step IX : Check if all characters are compared. If yes, go to step X otherwise go to step III.
Step X : Is count of number of characters compared equal to 3. If yes, go to step XI otherwise go to step XIII.
Step XI : Display "Welcome to Programming".
Step XII : Go to step XIV.
Step XIII : Display Invalid password.
Step XIV : Stop.

Flowchart : Refer flowchart 28.



Flowchart 28

>> **Program :**

Label	Instruction	Comment
	.MODEL SMALL	
	.DATA	
	PASS DB 'ABC'	
	MSG1 DB 10, 13, 'ENTER 3 CHARACTER PASSWORD $'	
	MSG2 DB 10, 13, 'WELCOME TO PROGRAMMING $'	
	MSG3 DB 10, 13, 'INVALID PASSWORD $'	
	.CODE	
	mov ax, @data	Initialize data segment
	mov ds, ax	
	mov ah, 09h	Display message
	lea dx, msg1	
	int 21h	
	mov cl, 00	Clear count
	mov dl, 00h	Clear number of match
	xor di, di	Initialize pointer
up:	cmp cl, 3	Check if count = 3 for
		max number of characters
	je down	
	mov ah, 07h	
	int 21h	Read character
	lea bx, pass	Set pointer to password
	mov ah, [bx+di	
	cmp al, ah	if character not match
		go to l4
	jne l4	
	add dl, 01	
	inc di	Increment pointer
l4:	inc cl	Increment counter
	mov ah, 02	display * instead of character
	mov dl, '*'	
	int 21h	
	jmp up	
down :	cmp dl, 3	count of number of characters matched
	jne l2	
	mov ah, 09h	
	lea dx, msg2	password is correct
	int 21h	

Label	Instruction	Comment
	jmp endd	
l2:	mov ah, 09h	
	lea dx, msg3	password invalid
	int 21h	
endd:	mov ah, 4ch	Terminate Program
	int 21h	
	end	

Program 29 : Program to accept Input from keyboard and display it

>> **Program statement :**

- Write a program using DOS interrupts to accept input from keyboard and display it on the screen. If '0' key as pressed program is terminated.

>> **Explanation :**

- Initialize the stack memory. Accept the input from keyboard using interrupt INT 21H function 01H. Check if the input is '0', (ASCII equivalent 30 H). If '0' is entered the program is terminated, otherwise the input is accepted.

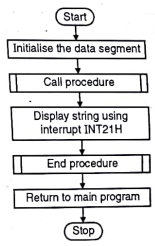
>> **Algorithm :**

Step I : Initialize the stack memory.
Step II : Accept input from keyboard.
Step III : Check if input = '0'. If yes go to step V otherwise go to step IV.
Step IV : Jump to step II.
Step V : Stop.

>> **Flowchart :** Refer flowchart 29.

>> **Program :**

Label	Instructions	Comment
	.model small	
	.stack 100h	
	.data	
	.code	
	mov ax, @data	Intialize data and extra segment
	mov ds, ax	
up:	mov ah, 01h	Get input from keyboard
	int 21h	
	cmp al, 30h	If 0 is entered End program
	je endd	
	jmp up	
endd:	mov ah, 4ch	Terminate Program
	int 21h	
	end	



Flowchart 29

Program 30 : Program to Display string using procedure.

>> **Program statement :**

- Write a procedure in assembly language of 8086 (with DOS interrupts) which will display "Study of Microprocessors is interesting" when called in the main program. Draw flowchart.

>> **Explanation :**

- Initialise the data memory section. Then call the procedure. Let the procedure be Near procedure. In the procedure using INT 21H, Function 09H display the string "Study of Microprocessors is interesting" and return back to the main program. Terminate the program.

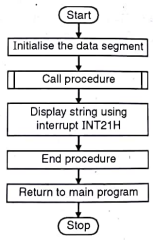
>> **Algorithm :**

Step I : Initialize the data segment.
Step II : Call procedure DISP.
Step III : Display string using INT 21H function 09H.
Step IV : Return to main program.
Step V : Stop.

>> **Flowchart :** Refer flowchart 30.

>> **Program :**

Instruction	Comment
.model small	
.data	
string db 'Study of Microprocessor is Interesting $'	
.code	
mov ax, @data	Intialize data and extra segment
mov ds, ax	
call disp	
mov ah, 4ch	Terminate Program
int 21h	
disp proc near	
mov ah, 09h	
lea dx, string	
int 21h	
endp	
ret	
end	



Flowchart 30

□□□

CHAPTER

14

Stacks and Subroutines

14.1 Introduction to Stack

- The stack is a reserved area of the memory where temporary information may be stored. A n-bit stack pointer is used to hold the address of the most recent stack entry. This location which has the most recent entry is called as the **top of the stack**.
- When the information is written on the stack, the operation is called **PUSH**. When the information is read from the stack, the operation is called **POP**. The stack works on the principle of **Last In First Out or First In Last Out.**
- The microprocessor stores the information/data like stacking plates. Fig. 14.1.1 shows stacked plates. If we want to remove the first stack plate, then we have to remove all the plates above the first i.e. we have to remove the fourth plate, third plate, second plate and then finally the first plate. This indicates that the first plate pushed onto the stack is the last one to be popped from the stack. This operation is called as **First In Last Out (FILO)**.

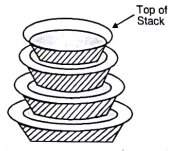


Fig. 14.1.1 : Stacked plates

- The stack is implemented with the help of special memory pointer register called as the **stack pointer**. The stack pointer's contents are automatically adjusted to point to the stack.
- The memory location that is currently pointed by the stack pointer is called as **top of stack**. As shown in Fig. 14.1.1 the 4th stacked plate represents top of stack.
- When data is to be stored on the stack, the SP increments before storing the data on stack and when the data is to be retrieved/popped from the stack the SP decrements to point next available byte of stored data.
- The stack array can reside anywhere in the on-chip memory.

Hence in short,
a) Stack is used to store information (data) temporarily.
b) Stack is a part of memory defined in program by THE USER.
c) STACK, the name it self, informs about the way data is stored i.e. stack of data.

14.2 Procedures

Q. Explain near and far procedures.

- Whenever we need to use a group of instructions several times throughout a program there are two ways we can avoid having to write the group of instructions each time we want to use them.
- One way is to write the group of instructions as a separate procedure. We can then CALL the procedure whenever we want to execute that group of instructions. For calling the procedure, the return address has to be stored back on to the stack.
- Another way is to write the group of instructions as a Macro.

The basic requirements which must be satisfied while calling a procedure are :

(1) A procedure call must save the address of the next instruction so that the return will be able to branch back to the proper place in the calling program.

(2) The registers used by the procedure need to be stored before their contents are changed and then restored just before the procedure is exited.

(3) A procedure must have a means of communicating or sharing data with the routine that calls it and other procedures.

The first requirement is met by the CALL and RET branch instructions. The second requirement is met by saving the register and flags by PUSH instruction and retrieving them back by POP instruction.

The procedure are delimited within the source code by the statement form

```
[procedure name]    PROC [attribute]    ;    at beginning of procedure
                    :
                    :
[procedure name]    ENDP                ;    at the end
```

The procedure name is the identifier used for calling the procedure and the attribute is NEAR or FAR. If the procedure is within the same code segment where the main program is stored then it is a NEAR procedure and the attribute NEAR is used for calling it. If the procedure is in some other segment than where the main program is stored then it is a FAR procedure. For a near procedure CALL instruction pushes only the IP contents on stack as contents of CS register remain unchanged for main program and procedure, whereas for a far procedure the CALL instruction pushes the contents of CS and IP both on the stack.

Sr. no.	NEAR PROCEDURE	FAR PROCEDURE
1.	It is declared in the same segment where the main program is stored.	It is declared in different segment than the segment where the main program is stored.
2.	When a NEAR procedure is called only the contents of IP are pushed as contents of CS remain unchanged.	When a FAR procedure is called the contents of CS and IP both are pushed onto the stack.
3.	Less stack memory is required.	More stack memory is required.
4.	NEAR CALL is referred to as an INTRA SEGMENT CALL.	FAR CALL is referred to as an INTER SEGMENT CALL.

Example 1 : Addition of Two Arrays using Procedure.

>> Program statement

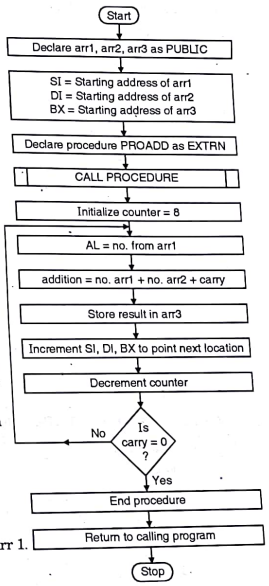
- Write a program in assembly language of 8086 to add two 8 bytes of data available in arr 1 and arr 2 and store the result in arr 3. Assume the LSB is available in first location array 1 and most significant byte in location arr 1 + 7 use procedure. The arrays are stored in code segment Module_1 and write FAR PROCEDURE in code segment MODULE_2 for addition of two arrays. Use PUBLIC and EXTRN directive. Create .OBJ files of both modules and link them to create an EXE file.

>> Explanation

- We will declare the two 8 byte arrays arr 1 and arr 2 as PUBLIC, so that they are available to other modules also. The result array arr 3 is also declared as PUBLIC.
- In the main program we will initialize SI to point to first element of arr 1, DI to point to first element of arr 2 and BX to point to first element of result array.
- Then call procedure which will perform addition.
- Display the result.

>> Algorithm

Step I : Initialize the data segment.
Step II : Declare arr 1, arr 2 arr 3 in public.
Step III : Let SI point to starting address of arr 1.
DI = Starting address of arr 2.
BX = Starting address of arr 3.
Step IV : Declare procedure proadd to be EXTRN.
Step V : Call procedure.
Step VI : Initialize counter = 8.
Step VII : Load AL with element of arr 1.
Step VIII : addition = number in arr 1 + number in arr 2 + carry.
Step IX : Store result in arr 3.
Step X : Increment SI, DI and BX to next location.
Step XI : Decrement counter.
Step XII : Check if counter = 0, if not goto step VII.
Step XIII : Return to calling program and end procedure.
Step XIV : Stop.



Flowchart 1

>> **Flowchart :** Refer Flowchart 1.

>> **Program**

Label	Instruction	Comment
	.model small	
	.data	initialize data segment
	public arr1, arr2, arr3	declaration of data
	arr1 db 01, 02, 03, 04, 05, 06, 07, 08	
	arr2 db 08, 07, 06, 05, 04, 03, 02, 01	
	arr3 db 8 dup(?)	declared space for result
	.code	initialize code segment
	mov ax, @data	initialize data section
	mov ds, ax	
	mov si, offset arr1	si to point to arr1
	mov di, offset arr2	di to point to arr2
	mov bx, offset arr3	bx to point to arr3
	extrn proadd:far	procedure in other file
	call proadd	call procedure
	mov ah, 4ch	terminate program
	int 21h	
	end	End

FAR PROCEDURE FOR ADDITION OF NUMBERS IN DIFFRENT ARRAYS

Label	Instruction	Comment
	.model small	
	.data	
	extrn arr1, arr2, arr3	declaring that variables are in other file
	.code	
	public proadd	making this procedure available to other file proadd proc
	mov cx, 03h	initialize counter=08
next:	mov al, [si]	get operand1 from arr1
	adc al, [di]	add=operand1+operand2+carry
	mov [bx], al	save result in location pointed by bx
	inc si	get next operand from arr1
	inc di	get next operand from arr2
	inc bx	point to next location from arr3
	loop next	repeat till all bytes are added
	ret	
	proadd endp	end procedure
	end	end program

Example 2 : Add N bytes of data in array using Procedure.

>> **Program statement**

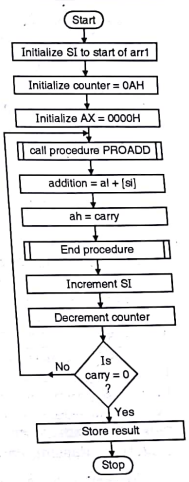
- Write a program in ALP of 8086 to add N bytes of data in an array using procedure. Use the attribute NEAR.

>> **Explanation**

- We are having an array of data N bytes. Let N = 10 for eg. now we have to add these 10 bytes of data. We will initialize pointer SI to starting address of the array and also CX = 10. The task of addition will be done in the procedure. Display the result of addition.

>> **Algorithm**

Step I : Initialize the data section and data segment.
Step II : SI = Start of array.
Step III : Initialize counter = 10, AX = 0000H.
Step IV : Call procedure.
Step V : addition = AL + [SI]
Step VI : Save carry if any in ah and Return.
Step VII : Increment SI to next location.
Step VIII : Decrement counter.
Step IX : Check if count = 0, if not goto step IV else go to step X.
Step X : Store the result in sum location.
Step XI : Stop.



Flowchart 2

>> **Flowchart :** Refer Flowchart 2.

>> **Program**

Label	Instruction	Comment
	.model small	
	.data	initialize data segment
	arr1 db 01, 02, 03, 04, 05, 06, 07, 08, 09, 0AH	
	sum db ?	
	count dw 0AH	
	.code	initialize code segment
	mov ax, @data	initialize data section
	mov ds, ax	
	mov ax, 00	ax=00
	mov si, offset arr1	initialize si to point to arr1
	mov cx, count	initalize counter
next:	call proadd	call near ptr proadd call procedure
	inc si	increment si to point to next element pop cx
	loop next	repeat till cx=0

Label	Instruction	Comment
	mov sum, al	store result at location sum
	mov ah, 4ch	normal termination to dos
	int 21h	
	proadd proc near	near procedure to add contents
	add al, [si]	al =al+[si]
	adc ah, 00	save carry in ah
	ret	
	proadd endp	end procedure
	end	end program

14.2.1 Passing Parameters to and from Procedures

Procedures are written to process data or address variables from the main program. To achieve this, it is necessary to pass the information about address, variables or data. This technique is called as **parameter passing**. The four major ways of passing parameters to and from a procedure are :

- Passing parameters using registers
- Passing parameters using memory
- Passing parameters using pointers
- Passing parameters using stack.

14.2.1(A) Passing Parameters Using Registers

The data to be passed is stored in the registers and these registers are accessed in the procedure to process the data.

e.g. :
```
.model small
.data.
MULTIPLICAND DW 1234 H
MULTIPLIER DW 4232 H
.Code
MOV AX, MULTIPLICAND
MOV BX, MULTIPLIER
CALL MULTI
:
:
MULTI PROC NEAR
MUL BX          ;   Procedure to access data from BX register.
RET
MULTI ENDP
:
:
END.
```


The **disadvantage** of using registers to pass parameters is that the number of registers limits the number of parameters you can pass.

eg : An array of 100 elements can't be passed to a procedure using registers.

14.2.1(B) Passing Parameters Using Memory

In the cases where we have to pass few parameters to and from a procedure registers are convenient. But, in cases when we need to pass a large number of parameters to procedure we use memory. This memory may be a dedicated section of general memory or a part of it.

e.g. :
```
.model small
.data.
MULTIPLICAND DW 1234 H      ; Storage for Multiplicand value
MULTIPLIER DW 4232 H        ; Storage for Multiplier value.
MULTIPLICATION DW ?         ; Storage for Multiplication result.
.Code
MOV AX, @ Data
MOV DS, AX
:
:
CALL MULTI
:
:
MULTI PROC NEAR.
MOV AX, MULTIPLICAND.
MOV BX, MULTIPLIER.
:
:
MOV MULTIPLICATION, AX      ; Store the Multiplication value in named memory location.
RET
MULTI ENDP
END.
```

14.2.1(C) Passing Parameters Using Pointers

A parameter passing method which overcomes the disadvantage of using data item names (i.e. variable names) directly in a procedure is to use registers to pass the procedure pointers to the desired data.

e.g.
```
. model small
. data
MULTIPLICAND    DB 12 H     ; Storage for Multiplicand value.
MULTIPLIER DB 42 H          ; Storage for Multiplier value.
MULTIPLICATION  DW ?        ; Storage for Multiplication result.
. Code
MOV  AX, @ Data
MOV  DS, AX
MOV  SI, OFFSET MULTIPLICAND
MOV  DI, OFFSET MULTIPLIER
```

```
MOV  BX, OFFSET MULTIPLICATON
CALL MULTI
:
:
MULTI PROCNEAR
:
:
MOV  AL, [SI]          ; Get Multiplicand value pointed by SI in accumulator.
MOV  BL, [DI]          ; Get Multiplier value pointed by DI in BL.
:
:
MOV  [BX], AX          ; Store result in location pointed out by BX.
RET
MULTI ENDP
END.
```

14.2.1(D) Passing Parameters Using Stack

In order to pass the parameters using stack we push them on the stack before the call for the procedure in the main program. The instructions used in the procedures read these parameters from the stack. Whenever stack is used to pass parameters it is important to keep a track of what is pushed on the stack and what is popped off the stack in the main program.

eg :

```
. model small
. data
MULTIPLICAND   DW 1234H
MULTIPLIER DW 4232H.
. Code
MOV AX, @ Data
MOV DS, AX.
:
:
PUSH MULTIPLICAND.
PUSH MULTIPLIER
CALL MULTI
:
:
MULTI PROCNEAR
        PUSH BP
        MOV BP, SP                ;
        MOV AX, [BP + 6]          ; copies offset of SP into BP.
                                  ; MULTIPLICAND value is available at
                                  ; [BP + 6] and is passed to AX.
        MUL WORD PTR [BP + 4]     ; MULTIPLIER value is passed.
        POP BP
        RET 4                     ; Increments SP by w to Return Address
        MULTI ENDP                ; End procedure.
        END.
```


14.2.2 Re-entrant and Recursive Procedures

There are two kinds of procedures :

- Re-entrant procedure
- Recursive procedure

(1) Re-entrant Procedure

While executing a program it may happen that a procedure 1 is called from main program, procedure 2 is called from procedure 1 and procedure 1 is again called from procedure 2. The program reenters the procedure 1. This type of procedure is called as re-entrant procedure.

Fig. 14.2.1 shows execution flow for a reentrant procedure.

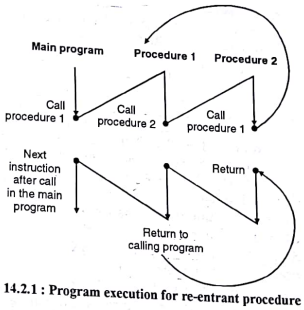


Fig. 14.2.1 : Program execution for re-entrant procedure

(2) Recursive Procedure

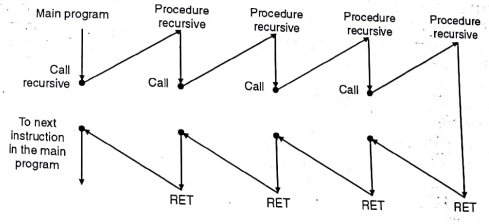


Fig. 14.2.2 : Flow diagram for recursive procedure with N = 4

A procedure which calls itself is called a recursive procedure. They are used when we work with complex data structures called trees. The recursive loop takes a long time to execute and occupies huge stack space.

For this type of procedures we normally define N i.e. the recursive depth. If N = 4, N is decremented by 1 after each procedure. Call the procedure until N = 0. Fig.14.2.2 shows flow diagram for N = 4

Example 3 : To find the Factorial of a Number using Recursive procedure.

➢➢ **Program Statement :**

Write a program in the ALP of 8086 to compute the factorial of a number. Use recursive procedure.

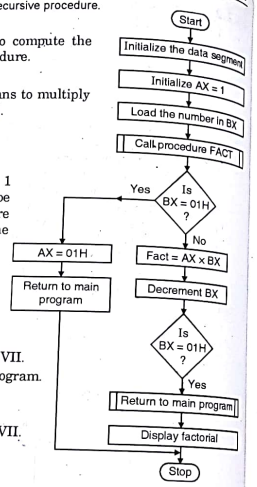
➢➢ **Explanation :**

- To compute the factorial of a number means to multiply the number n with $(n-1)(n-2)\ldots\times 2\times 1$.
  e.g. to compute 5 !
  $5! = 5\times 4\times 3\times 2\times 1$
  $= 120$.
- In our program, we will initialize AX = 1 and load the number whose factorial is to be computed in BX. Call recursive procedure fact, which will calculate the factorial of the number. Display the result.

➢➢ **Algorithm :**

Step I : Initialize the data segment.
Step II : Initialize AX = 1.
Step III : Load the number in BX.
Step IV : Call procedure fact.
Step V : Compare BX with 1, if not goto step VII.
Step VI : AX = 1 and return back to calling program.
Step VII : Fact = AX × BX.
Step VIII : Decrement BX.
Step IX : Compare BX with 1, if not goto step VII.
Step X : Return back to calling program.
Step XI : Display the result.
Step XII : Stop.

➢➢ **Flowchart :** Refer Flowchart 3.



Flowchart 3

➢➢ **Program :**

Label	Instruction	Comment
	.model small	
	.data	
	num dw 03h	
	.code	
	mov ax, @data	initalize data segment
	mov ds, ax	
	mov ax, 01	initalize ax=1
	mov bx, num	load the number in cx
	call fact	call procedure
	mov di, ax	store the lsb of result in di
	mov bp, 2	initalize count for no of times display is called
	mov bx, dx	store msb of result in reg bx
up1:	mov ch, 04h	count of digits to be displayed

Label	Instruction	Comment
	mov cl, 04h	count to roll by 4 bits
l2:	rol bx, cl	roll bl so that msb comes to lsb
	mov dl, bl	load dl with data to be displayed
	and dl, 0fH	get only lsb
	cmp dl, 09	check if digit is 0-9 or letter A-F
	jbe l4	
	add dl, 07	if letter add 37H else only add 30H
l4:	add dl, 30H	
	mov ah, 02	function 2 under INT 21H
	int 21H	
	dec ch	decrement Count
	jnz l2	
	mov bx, di	store lsb of result in bx
	dec bp	decrement bp
	cmp bp, 00	
	jnz up1	
	mov ah, 4ch	
	int 21h	
fact	proc near	function for finding the factorial
	cmp bx, 01	is bx=1?
	jz l11	if yes, ax=1
l12:	mul bx	find factorial
	dec bx	decrement bx
	cmp bx, 01	multiply till bx=1
	jne l12	
	ret	
l11:	mov ax, 01	initalize ax=1
	ret	return to called program
	fact endp	end procedure
	end	end program

14.3 Macros

When the repeated group of instructions is too short not appropriate to be written as a procedure, we use a macro. A macro is a group of instructions that perform a task. Each time we call the macro in our program the assembler will insert the group of defined instructions in place of "call".

The macros are useful for following purposes :

- To simplify and reduce the amount of repetitive coding.
- To reduce the errors caused by repetitive coding.
- To make the assembly language program more readable.

A macro executes faster because, there is no need of call and return.

The basic **format** of a Macro is

```
macro name  MACRO        ; Define macro
                         ; Body of macro
            ENDM         ; End macro.
```

The MACRO directive on the first line tells the assembler that the instructions that follow upto ENDM are a part of macro definition. The ENDM directive ends the macro definition. The instructions between MACRO and ENDM comprise the body of macro definition.

```
e.g. : INIT MACRO          ; define MACRO.
       MOV AX, @ Data      ;
       MOV DS, AX          ;   Body of MACRO
       MOV ES, AX          ;
       ENDM                ; END MACRO.
```

The assembler places the macro instructions in the program when it is invoked, this is called as Macro expansion.

14.3.1 Comparison of Procedure and Macro

Q. Compare procedure and macro.

Sr. No.	Procedure	Macro
1.	It resembles a call function of high level language. The processor branches to the procedure on call proc, instruction and returns back to the caller program after executing the procedure.	When the assembler comes across the instruction "CALL MACRO", it replaces this instruction with the group of instructions placed in the corresponding macro.
2.	Since the processor branches to another memory location and returns back, it consumes some time to store and fetch back the return address. Hence it has a latency period.	Macro does not required any latency period.
3.	Since the assembler stores the instructions of procedure only once in the memory, the program consumes less space in memory.	Since the assembler replaces all "Call macro" instruction by the group of instructions in the macro, the program consumes more space in memory.
4.	Procedures are to be used for repetitive task, if the task is very large (i.e. it has many instructions).	Macros are to be used for repeatative task, if the task is small (i.e. it has less number of instructions).

14.4 Timing and Delay Loops

Q. Write a short notes on :
(i) Software delay (ii) Hardware delay.

The speed at which the instructions in 8086 are executed is determined by a crystal-controlled clock with a frequency of few megahertz. Each instruction takes a fixed number of clock cycles to execute.

eg : DAA requires 4 clock cycles, JNZ requires 16 clock cycles.

If an 8086 is running with a 5 MHz clock, then each clock takes $\left(\frac{1}{5\text{ MHz}}\right) = 0.2\ \mu s$.

An instruction which takes 4 clock cycles, will take $4 \times 0.2 = 0.8\ \mu s$ to execute. A common programming problem is to introduce delay between execution of two instructions. eg traffic signal controller, real time clock, process control, serial communication.

There are two types of delay :

- Software delay
- Hardware delay

14.4.1 Software Delay

This delay is introduced using the instruction time. The microprocessor requires the number of T states while executing an instruction. By repeating the instructions for N number of times, the delay between two events can be easily obtained.

Time required for 1 T state = $\frac{1}{\text{Operating Frequency}}$.

eg 1 : To implement timer delay using an 8 bit counter.

First let us, write the program. Assume 8086 clock frequency to be 5 MHz.

```
. model small
. stack 100
. code
                                          T states required
        MOV  AL, count  ;  Load counter          4
Loop 1: DEC  AL         ;  Decrement counter     3
        JNZ  Loop1      ;  Repeat till counter = 0.   16 (when condition true) /
        end             ;  end program.               4 (when condition false)
```

Total delay time = $4 + \text{count} \times 3 + [\text{count} - 1] \times 16 + 4$

MOV DEC AL JNZ Loop 1

MOV AL, count is executed only once so T states = 4.

DEC AL is executed count times so T states = count.

JNZ Loop 1 is executed count – 1 times

It will not satisfy when count = 0, so T states = (count – 1) × 16 + 4

If count = 4

Timer delay = $4 + 4 \times 3 + 3 \times 16 + 4 = 4 + 12 + 48 + 4 = 68$ T states.

Operating frequency of 8086 = 5 MHz.

$\therefore$ Time required for 1 T state = $\frac{1}{5\text{ MHz}} = 0.2\ \mu s$.

$\therefore$ Total Time Delay = 68×0.2

Total Time Delay = 13.6 µs.

eg 2 : Implement a timer delay using 16 bit counter.

```
. model small
. stack 100
```

```
. code                                          T states required
        MOV  AX, count      ;  Load count            4
Loop 1: DEC  AX             ;  Decrement counter     2
        JNZ  loop1          ;  Repeat till count = 0 16/4
    end
```

$\therefore$ Time delay = 4 + count × 2 + (count − 1) × 16 + 4

If count 3

Time delay = 4 + 3 × 2 + 2 × 16 + 4 = 4 + 6 + 32 + 4
= 46 T states

8086 operates at 5 MHz.

$\therefore$ Total timer delay = 46 × 0.2

Timer delay = 9.2 μs.

14.4.1(A) Disadvantage of Software Delay

In software delay, microprocessor wastes its valuable time doing nothing. If microprocessor time is to be used at its best, hardware delay is preferred.

14.4.2 Hardware Delay

In this, one can use 555 timer or 8253 programmable interval timer which will interrupt 8086 microprocessor, after a specified time duration. Let's implement RTC (Real Time Clock) using 555 timer.

In clock second, minutes and hour are important. After each second, increment second location. After 60 seconds reset second location and Increment minute. After 60 minutes, Increment hour and reset minute as well as second. This application asks for 1 Hz (or 1 sec) time base signal. 555 timer IC is used in Astable mode. Interface the output of 555 to NMI input. Refer Fig. 14.4.1(a).

Software is having basic two routine, one is main routine and other is subroutine (ISR).

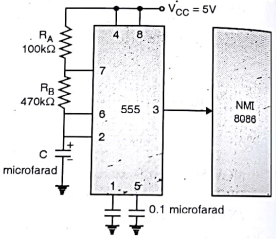


Fig. 14.4.1(a) : 1 Hz pulse source used to interrupt 8086

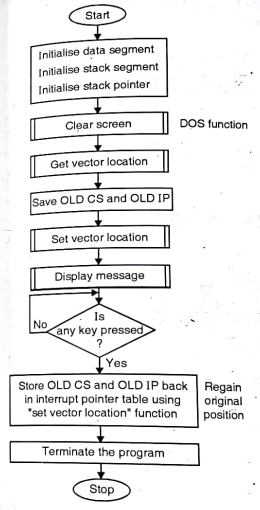
Refer Fig. 14.4.1(b) for flowchart of main routine.

Application - Real Time Clock using NMI

Label	Instruction	Comment
	TITLE RTC	
	.MODEL SMALL	
	.STACK 100	
	.DATA	
	MES1 DB 0AH, 0DH, `REAL TIME CLOCK', `$'	
	OLDCS DW 0H	
	OLDIP DW 0H	

Label	Instruction	Comment
	SEC DB 0H	
	MIN DB 0H	
	HOUR DB 0H	
	.CODE	
	MOV AX, @DATA	initialise data
	MOV DS, AX	
	MOV AX, STACK	initialise stack
	MOV SS, AX	
	MOV SP, OFFSET STACK + 200	init stack pointer
	MOV AL, 0H	clear screen
	MOV AL, 03H	text mode `3'
	INT 10H	BIOS Interrupt
	MOV AH, 35H	get vector locations
	MOV AL, 02H	Type 2
	INT 21H	returns ES : BX = Segment : offset
	MOV OLDCS, ES	save oldcs
	MOV OLDIP, BX	save oldip
	PUSH DS	save ds
	PUSH CS	save cs
	POP DS	load cs in ds
	LEA DX, NMI_ISR	load offset of ISR
	MOV AL, 02H	set vector location
	MOV AH, 25H	function 25h
	INT 21H	DOS interrupt
	POP DS	recall ds
	MOV AH, 09	display Message
	MOV DX, OFFSET MES1	load Offset
	INT 21H	DOS Interrupt
	MOV AH, 01H	wait for key
	INT 21H	DOS interrupt
	MOV DS, OLDCS	load oldcs
	MOV DX, OLDIP	load oldip
	MOV AL, 02H	regain original condition by
	MOV AH, 25H	set vector location

Label	Instruction	Comment
	INT 21H	DOS interrupt
	MOV AX, 4C00H	Terminate
	INT 21H	DOS interrupt
NMI_ISR	PROC FAR	
	PUSH AX	push registers
	PUSH DS	
	MOV AX, @DATA	load data register
	MOV DS, AX	
	MOV AL, SEC	load seconds
	ADD AL, 01H	seconds = seconds + 1
	DAA	BCD value
	CMP AL, 60H	check for 60 sec over
	JNZ NO_Q	
	MOV SEC, 0H	clear second
	MOV AL, MIN	load minute
	ADD AL, 01H	minute = minute + 1
	DAA	BCD value
	CMP AL, 60H	check for 60 minutes
	JNZ NO_Q	no quit
	MOV MIN, 0H	minutes = 0
	MOV AL, HOUR	load hour count
	ADD AL, 01H	increment hour
	DAA	BCD value
	CMP AL, 24H	check for 24 hours
	JNZ NO_Q	no, continue
	MOV HOUR, 0H	hour = 0
NO_Q:	POP DS	POP registers
	POP AX	
	IRET	
NMI_ISR	ENDP	
	END	



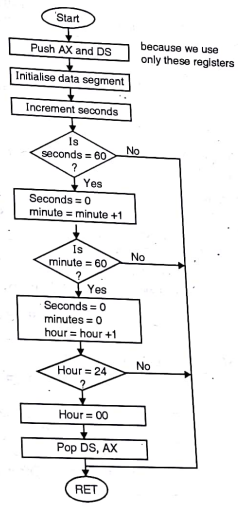


Fig. 14.4.1

Ex. 14.4.1 : Write an Assembly Language Program to generate a delay of 1 sec using a microprocessor running at 5 MHz. Also show the delay calculations.

Soln. :

Program

	Instruction	Clock cycles required
	MOV BX, Multiplier count	
REPE :	MOV CX, Count	; 4
	DEC CX	; 2
	JNZ BACK	; 16 / 4
	DEC BX	
	JNZ REPE	

Step I : To calculate the delay generated by inner loop with maximum count (FFFF H).

Delay generated by = $[4 + (65535 - 1) \times (2 + 16) + (2 + 4)] \times 0.2\ \mu s$

Inner loop for count = 235.9244 msec

FFFF H (i.e. 65535)

Step II : Calculate the multiplier count to get delay of 1 second

$$\text{Multiplier count} = \frac{\text{Required delay}}{\text{delay provided by inner loop}} = \frac{1 \times 1s}{235.9244\ \text{msec}}$$

$$= 4.239$$

$$= 04\ H.$$

Ex. 14.4.2 : Write an 8086 ALP to generate a delay of 100 ms, if 8086 system frequency is 10 MHz.

Soln. :

Program

	Instruction	Clock cycles required
	MOV CX, Count	; 4
BACK :	DEC CX	; 2
	JNZ BACK	; 16 / 4

Step I : To calculate the numbr of required clock cycles.

$$\text{Clock cycles required} = \frac{\text{Required delay time}}{\text{Time for 1 clock cycle}} = \frac{100\ ms}{0.1\ \mu s}$$

$$= 1000000.$$

Step II : To find the required count.

$$\text{Count} = \frac{\text{Number of clock cycles required} - 4 - (2 + 4)}{\text{execution time for one loop}} + 1$$

$$\text{Count} = \frac{1000000 - 4 - 6}{(16 + 2)} + 1$$

$$\text{Count} = (55556)_{10}$$

$$\textbf{Count} = \textbf{D904 H}$$

□□□

CHAPTER

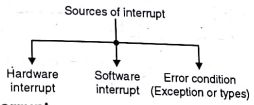
15

8086 Interrupt Structure

15.1 8086 Interrupt Structure

Q.1 Explain 8086 interrupt structure.

In 8086, we have three sources of interrupt.



15.1.1 Hardware Interrupt

In this type of interrupt, physical pins are provided in the chip. In 8086 we have two pins :

(i) NMI (Non Maskable Interrupt). (ii) INTR.

To interrupt the processor we have to apply signal to these pins. As name suggests, NMI is non maskable i.e. microprocessor has to service this interrupt, it can not avoid it. Whereas INTR is maskable, if IF flag in flag register is **'0'**, microprocessor will not recognise interrupt available on the pin.

15.1.2 Software Interrupt

Software interrupt, in 8086 we have INT instruction. When INT instruction is executed interrupt will occur.

15.1.3 Error Conditions (Exception Or Types)

We know that 8086 supports division, multiplication, addition etc. Suppose by mistake if user asks microprocessor to divide any number by ZERO, then you know that dividing any number by ZERO produces answer **'∞ *(infinity)*'**. So in this case microprocessor will generate an interrupt ***"Automatically"*** and interrupt current execution. In ISR, user can display message ***"Divide by zero error"***. (Possibly you may have come across this error). Instead of showing the answer as **'∞ *(infinity)*"**.

Thus, internally generated errors produces an interrupt for microprocessor, normally referred as ***"TYPE"*** by Intel engineer and referred as ***"Exception"*** by motorola engineer.

Thus we conclude that 8086 has a simple and versatile interrupt system. Every interrupt is assigned a ***"type code"*** that identifies it to the CPU. The 8086 can handle upto 256 different interrupt types. Interrupts may be initiated by devices external to the CPU; in addition, they also may be triggered by software interrupt introductions and under certain condition, by the CPU itself. Fig. 15.1.1 shows interrupt sources for 8086.

As shown in Fig. 15.1.1 8086 have two lines that external device may use to signal interrupts. The INTR line is usually driven by an Intel 8259A (PIC), which in turn connected to the devices that need interrupt services.

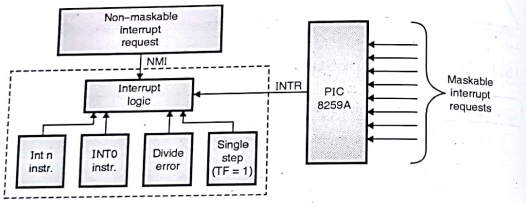


Fig. 15.1.1 : Interrupt sources (INTO-Interrupt on Overflow)

15.2 Software Interrupts

- The INT instruction of the 8086 can be used to do one of the 256 interrupts (Type 0-255).
- The interrupt type is specified by the number as a part of the instruction. e.g. INT 0 instruction can be used to send execution to a divide by zero interrupt service routine.
- With the help of these software interrupts we can call the routines from different programs in the system e.g. BIOS. The BIOS routines are called with INT instructions.

15.3 Interrupt Service Routines

Q. What is an ISR ? How does 8086 acknowledge an interrupt ? Draw flowchart for interrupt processing sequence.

At the end of each instruction cycle, the 8086 checks to see if any interrupts have been requested. Therefore whenever interrupt occurs, it won't be immediately checked by microprocessor. Microprocessor first completes, execution of current instruction and then checks for an interrupt.

Now the main important point is, how 8086 acknowledges it. The same has been graphically presented in flowchart (Refer Fig. 15.3.1).

(1) First, microprocessor will complete execution of current instruction.
(2) It checks for any internal interrupt, suppose the same is not present.
(3) Then it checks for NMI i.e. hardware interrupt.

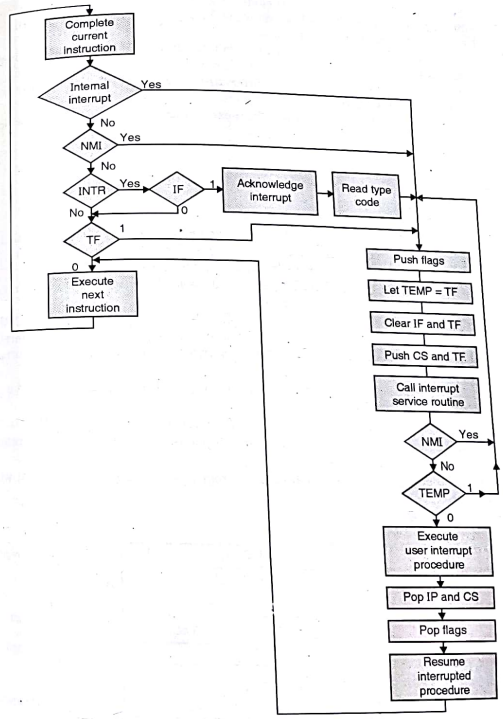


Fig. 15.3.1 : Interrupt processing sequence/ priority sequence

(4) If NMI is not present, it will check for INTR.
(5) In absence of INTR, it will continue checking for TF (Trap, single step) flag.
(6) If TF is not equal to 1, it will switch over to next instruction.

(7) If you observe flowchart, you conclude that except INTR, all interrupt comes to point of pushing flags. Therefore first we will take path of INTR and after that next discussion is common to all interrupts.

(8) Suppose microprocessor finds that INTR is present, then it checks for IF (interrupt enable flag), If it is 0, it will not service INTR and return back to check TF flag.

(9) Suppose IF = 1, then it will execute ***"interrupt acknowledge"*** cycle. In this cycle it captures ***"TYPE CODE"***.

(10) After capturing the ***"type code"***, the next sequence will be executed which is common to all interrupt.

(a) Push flag register.

(b) Temp = TF (Save present status of TF).

(c) Clear IF and TF. This disables INTR input and single step function.

(d) Push CS and IP (OLD CS : OLD IP).

(e) CALL INTERRUPT service routine.
This CALL is equivalent of an intersegment indirect called instruction. By calling this routine, microprocessor receives NEW CS and NEW IP value from vector table (also referred as Interrupt Pointer Table).

(f) These NEW CS and NEW IP values, will be loaded into code segment register and instruction pointer, respectively.

(g) Before transferring control to NEW CS : NEW IP again checks for NMI. If yes, it will jump to point (a), else check for TEMP. In TEMP we had stored TF status. So if TF = 1, i.e. single step is activated, then microprocessor jumps to point (a).

(h) But if TEMP = TF = 0, it will execute USER Interrupt procedure, i.e. ISR written by user.

(i) At the end of ISR, user will write IRET i.e. return from an interrupt.

(j) In response to IRET, microprocessor will pop up CS and IP (normally referred as OLD CS and OLD IP).

It will also pop flag status and will resume program execution from the point where it was interrupted.

To resume interrupted procedure microprocessor simply loads OLD CS and OLD IP value to CS register and IP register respectively. The full process is diagrammatically presented in Fig. 15.3.2.

Going through the full discussion of interrupt response, you came across ***"Vector table"*** OR ***"Interrupt vector table"***. So let's have a look at the table to see how it looks like.

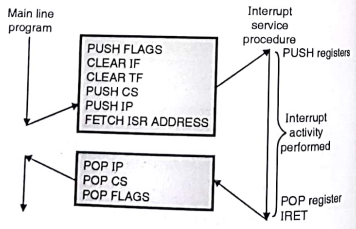


Fig. 15.3.2 : 8086 interrupt response

15.4 Interrupt Vector Table (IVT)

Q. Explain type 0, 1, 2 interrupts found in the interrupt vector table of 8086 microprocessor ?

The interrupt vector (or interrupt pointer) table is the link between an interrupt type code and the procedure that has been designated to service interrupts associated with that code. 8086 supports total 256 types i.e. 00H to FFH. For each type it has to reserve four bytes i.e. double word. This double word pointer contains the address of the procedure that is to service interrupts of that type. The higher addressed word of the pointer contains the base address of the segment containing the procedure. This base address of the segment is normally referred as NEW CS. The lower addressed word contains the procedure's offset from the beginning of the segment. This offset we normally refer as NEW IP. Thus NEW CS : NEW IP provides NEW physical address from where user ISR routine will start.

As for each type, four bytes (2 for NEW CS and 2 for NEW IP) are required, therefore interrupt pointer table occupies upto the first 1k bytes (i.e. 256 × 4 = 1024 bytes), of low memory.

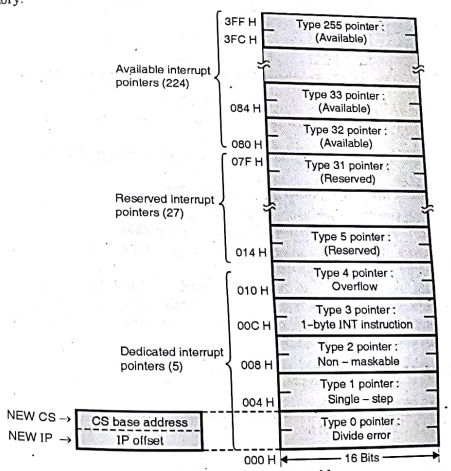


Fig. 15.4.1 : Interrupt vector table

Thus 00000H to 003FFH, these locations of 8086 microprocessor are reserved for interrupt vector table.

In section 15.3, point 10 (e), we specified that microprocessor receives NEW CS and NEW IP value from vector table, after calling internal service routine. This point is referred back here because, we want to know that, what activity is performed by this routine to get NEW CS and NEW IP. We know that for each ***"type"*** we have 4 locations reserved. So call routine, will simply sense the type number, multiply the same with 4 and will get double word pointer from that location. Suppose for example type code is '3'. Then $3 \times 4 = (12)_{10} = 0C$ H.

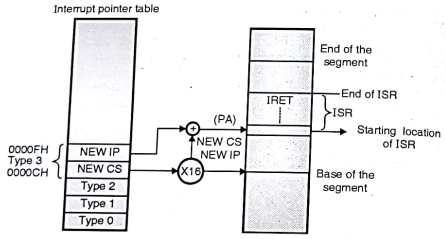


Fig. 15.4.2 : Pointing to ISR routine via interrupt vector table

Then 0CH/0DH location will provide OFFSET IP (NEW IP) and 0EH/0FH will provide base address of the segment (NEW CS). The same is graphically presented in Fig. 15.4.2.

If you observe Fig. 15.4.1, you will find that, total interrupt vector table is divided into three groups :

(1) Dedicated interrupt pointers (Type 0/1/2/3/4)
(2) Reserved interrupt pointers (Type 5 to Type 31)
(3) Available interrupt pointers (Type 32 to Type 225).

Type 0 to 4 are dedicated interrupt pointers by Intel for divide by zero error, single step, NMI, 1-byte INT instruction and overflow.

Interrupt types from 5 to 31 are reserved by Intel for use in more complex microprocessors, such as 80286, 80386, 80486. Finally the types from 32 to 255 i.e. total 224, are available to user to use for either hardware or software interrupt. Now let's start our study with dedicated interrupt pointers.

15.5 Dedicated Interrupt Pointer

Under this, we have type 0 to 4.

- Type 0 : Divide by zero interrupt
- Type 1 : Single step interrupt (INT 1)
- Type 2 : NMI (INT 2)
- Type 3 : One byte interrupt /breakpoint interrupt (INT 3)
- Type 4 : Interrupt on overflow (INT O)

15.5.1 Type 0 - Divide by Zero Interrupt

8086 supports division (unsigned / signed), instruction. 8086 will automatically do a type 0 interrupt if the result of DIV or IDIV operation is too large to fit in destination register. When type 0 interrupt is internally generated, microprocessor will :

(a) Push flag register.
(b) Reset TF and IF.
(c) Push CS and IP (i.e. return address).
(d) Get NEW CS and NEW IP. For this, microprocessor takes type no, i.e. **'0'**, multiply by 4. Therefore we get $0 \times 4 = 000$ H. So microprocessor gets, NEW IP from 00000H / 00001H location and NEW CS from 00002H/00003H location.
(e) NEW CS and NEW IP will be loaded into the CS and IP register. Thus we get branching to ISR routine.
(f) After returning from ISR, microprocessor will pop CS and IP (OLD CS/OLD IP). Microprocessor will also pop flag register.

Here important point regarding Type 0, is, it is automatic and cannot be disabled anyway i.e. non-maskable. User have to a account it in the program where he/she uses DIV/IDIV instruction. Normally user will write an interrupt service procedure which takes desired action when an invalid division occurs. To avoid this interrupt, user can check, before division, that divisor is not zero.

15.5.2 Type 1 - Single Step Interrupt (INT 1)

We use debug utility to debug the program. In that we can ***"Single step"*** the program. You must have found this utility very useful for the beginner. You must have observed that when you tell a system to single step, it will execute one instruction and stop. One can examine the contents of memory / register if required, else execute next instruction. The conclusion is, in this mode, a system will stop after it executes each instruction and wait for further direction from user. The 8086 trap flag and type 1 interrupt makes it easy to implement a single step feature.

How to set trap flag ?

8086 has no instructions to directly set/reset trap flag. Therefore to initiate single stepping follow next steps :

(1) Copy the flags onto the stack. (2) Setting the TF bit on the stack. (3) Popping the flag.

The same is achieved by following instructions :

```
PUSHF                          ; Push flag register
MOV BP, SP                     ; Use BP as a pointer as it is stack segment
OR WORD PTR [BP + 0], 0100 H   ; Set TF bit
POPF                           ; Restore flag register
```

One more way we can have is,

```
PUSHF                          ; Push flag register
MOV SI, SP                     ; Read SI pointer
OR Word PTR [SI], 0100 H       ; Set TF = 1
POPF                           ; Pop flag register
```

After, TF = 1, microprocessor will automatically do a Type 1 interrupt after each instruction executes. The interrupt sequence saves the flags and program counter, then

resets TF flag to allow the single step routine to execute normally. To return to the routine under test, an interrupt return restores the IP, CS and flags with TF set. This allows the execution of next instruction in the program under test before trapping back to the single step routine.

Note that single step is not masked by the IF bit in the flag register.

Locations used by type 1 interrupt are 0004H / 5H / 6H / 7H.

15.5.3 Type 2 - NMI (INT 2)

This is the highest priority hardware interrupt and is non maskable. The input is edge triggered, but is synchronized with the CPU clock and must be active for two clock cycles to generate recognition. The interrupt signal may be removed prior to entry to the service routine. Since the input must make a LOW to HIGH transition to generate an interrupt, ***"spurious"*** transition on the input should be suppressed. If the input is normally HIGH, the NMI low time to guarantee triggering is two CPU clock times. When type 2 (NMI) is generated, in response to the same microprocessor executes steps which are same as that listed in section 15.3 point 10 (a) to (j).

The only point to be added is, the location in interrupt pointer table.

Type no. $\times 4$ = Location in interrupt pointer table.

2×4 = 00008H

Thus from 00008H/9H microprocessor gets NEW IP and from 0000AH/BH microprocessor gets NEW CS.

Basically NMI interrupt input is used for catastrophic failures, for example power failure, time out of system watchdog timer.

15.5.4 Type 3 - One Byte Interrupt / Breakpoint Interrupt (INT 3)

This type is invoked by a special form of the software interrupt instruction which requires a single byte of code space i.e. CCH (INT 3). This interrupt is primarily used as a breakpoint interrupt for software debug. When you insert a breakpoint in your program, the system executes instructions upto the breakpoint and then goes to the breakpoint procedure.

When user informs debugger program to insert breakpoint at some point in your program, they actually do it by temporarily replacing the instruction byte at that address with CC H, i.e. code for INT 3 instruction. Thus this single byte instruction can be mapped into the smallest instruction for absolute resolution in setting breakpoints.

The steps taken by microprocessor are same as that taken in section 15.3 point 10 (a) to (j). Regarding location in interrupt pointer table, microprocessor will take type number and multiply the same with 4, therefore $3 \times 4 = 12 = 0C$ H. Then from 0000CH/DH we get NEW IP and from 0000EH/FH we get NEW CS.

A breakpoint ISR routine usually saves all the register contents on the stack. If user wants, the same can be displayed on CRT screen. At this point system is now waiting for next command from the user. Thus this feature of 8086 allows fast debugging, i.e. wait or stop at a point suspicious by the programmer, else continue.

15.5.5 Type 4 - Interrupt on Overflow (INT O)

This interrupt occurs if the overflow flag (OF) is set in the flag register. The OF flag is set if the signed result of an arithmetic operation on two signed number is too large to

be represented in destination register or memory location. Thus this interrupt is used to capture overflow errors.

One more way of branching to this interrupt is INT 0 (Interrupt on Overflow). The action taken by microprocessor in response to type 4 and INTO is same, which is already given in section, 6.3, point 10 (a) to (j). For type 4/INTO, we get NEW IP from location 10H/11H and NEW CS from 12H/13H. This interrupt is non maskable.

Interrupt types 0 and 2 can occur without specific action by the programmer (except for performing a divide for Type 0). While types 1, 3 and 4 require a conscious act by the programmer or generate these interrupt types. All, but type 2 are invoked through software activity and are directly associated with a specific instruction.

15.6 Hardware Interrupts

Q. How does 8086 respond to an INTR signal ?

8086 provides this hardware pin so that some external signal can interrupt program sequence executed by 8086. Let's start INTR discussion in question - answer form.

Can we mask INTR ? If yes, how ?

INTR is maskable interrupt, so that INTR cannot cause an interrupt. Masking is achieved by forcing IF = 0 in flag register. IF can be cleared at anytime using CLI (Clear Interrupt Instruction).

What we have to do to unmask INTR ?

To unmask INTR, set IF = 1: The same is achieved by STI (Set Interrupt Instruction).

What will be the status of IF flag, when 8086 is resetted ? Why ?

When 8086 is resetted IF flag is automatically cleared. To allow 8086 to service INTR you have to use STI instruction. IF is cleared at power on reset or reset by user, just because, before starting actual operation 8086 has to perform following jobs i.e. :

(a) Initialise peripherals 8255, 8155, 8279 etc.
(b) Initialise pointers
(c) Initialise counters
(d) Initialise variables etc.

So it is expected that unless and until ***"initialisation"*** is over 8086 SHOULD NOT service INTR interrupt. In short, by not enabling INTR we are allowing 8086 to ***"get ready"*** to service INTR, 8086 will take some time to get ready therefore during that part, IF = 0.

How INTR signal is sensed by 8086 ? Is the instruction of 8086 affects the sensing of INTR signal ?

During the last clock cycle of each instruction, the state of the INTR pin is sampled. The 8086 deviates from this rule when the instruction is MOV or POP to a segment register. For this case, the interrupts are not sampled until completion of the following instruction. This allows a 32 bit pointer to be loaded to the stack pointer registers SS and SP without the danger of an interrupt occurring between the two loads. Another exception is WAIT instruction which waits for a low active input on the $\overline{\text{TEST}}$ pin. This instruction also continuously samples the interrupt request during its execution and allows servicing

interrupts during the wait. When an interrupt is detected, the WAIT instruction is again fetched prior to servicing the interrupt to guarantee the interrupt routine will return to the WAIT instruction.

Also, since prefixes are considered part of the instruction they precede, the 8086 will not sample the interrupt line until completion of the instruction, the prefix(es) precede(s). An exception to this (other than HLT or WAIT) is the string primitives preceded by the Repeat (REP) prefix. The repeated string operations will sample the interrupt line at the completion of each repetition. This includes repeat string operations and also includes the lock prefix. If multiple prefix precedes a repeated string operation and the instruction is interrupted, only the prefix immediately preceding the string primitive is restored.

When 8086 responds to interrupt, it clears IF flag. Why ?

IF is cleared because of two reasons :

(a) It prevents a signal on the INTR input from interrupting a higher priority interrupt service procedure in progress.

(b) IF is made '0', so that signal on INTR input does not cause 8086 to interrupt itself continuously INTR is active HIGH. So when INTR = 1, 8086 is interrupted. If INTR is not disabled during the first response, the 8086 would be continuously interrupted and would never get to the actual interrupt service routine.

Suppose user wants, another INTR input to be able to interrupt as interrupt procedure in progress, what should be done ?

Simple, use STI command to make IF = 1, to allow sensing of another interrupt on INTR line.

How 8086 responds, to INTR signal ? Do we need any extra hardware for the same ?

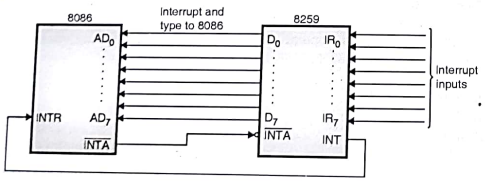


Fig. 15.6.1 : Interfacing 8259 to 8086

8086 response to INTR is somewhat different than the response to other interrupts. The main difference is, for INTR, interrupt type number is sent to 8086 from an external hardware. Thus we, do need external hardware. The most widely used device is 8259 (PIC). Refer Fig. 15.6.1 for interfacing.

Now we will study the interrupt acknowledge machine cycles.

(1) When 8259 receives an interrupt signal on one of its IR (Interrupt request) inputs, it sends an interrupt request signal to INTR input of 8086.

(2) If IF flag is set, then 8086 responds to 8259, which is diagrammatically presented in Fig. 15.6.2.

In response to INTR 8086 executes an interrupt acknowledge sequenc guarantee the interrupt will be acknowledged, the INTR input must be held activ until the interrupt acknowledge is issued by the CPU. If BIU is running a bus cycle when the interrupt condition is detected (as would occur if the BIU is fetching an instruction when the current instruction completes), the interrupt must be valid at 8086, 2 clock cycles prior to T_4 of the bus cycle if the next cycle is to be an interrupt acknowledge cycle. If the 2 clock setup is not satisfied another pending bus cycle will be executed before the interrupt acknowledge is issued. If a held request is also pending, the interrupt is serviced after the hold request is serviced.

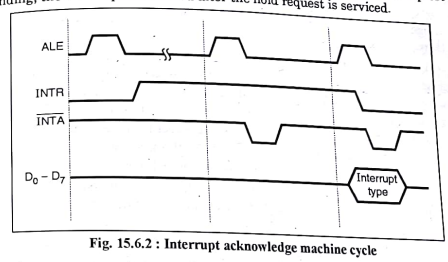


Fig. 15.6.2 : Interrupt acknowledge machine cycle

(3) The interrupt acknowledge cycle consists of two $\overline{INTA}$ bus cycles separated by two idle clock cycles. During the first bus cycle the $\overline{INTA}$ command is issued rather than read. No address is provided by 8086 during either bus cycle ($\overline{BHE}$ and status are valid), however, ALE is still generated and will load the address latches with indeterminate information.
This condition requires that devices in the system do not drive their outputs without being qualified by the Read Command.

(4) During second $\overline{INTA}$ bus cycle DT/$\overline{R}$ and $\overline{DEN}$ are conditioned to allow the 8086 to receive a one byte interrupt type number from the interrupt system.

(5) The first $\overline{INTA}$ bus cycle signals an interrupt acknowledge cycle is in progress and allows the system to prepare to present the interrupt type number on the next $\overline{INTA}$ bus cycle. The CPU does not capture information on the bus during the first cycle, instead it will float its bus, AD_0 to AD_{15}.

(6) During the second interrupt acknowledge cycle the 8086 sends out another pulse on its $\overline{INTA}$ output pin. In response to second pulse on, $\overline{INTA}$ the 8259 puts the interrupt type (number) on the lower eight lines of the data bus, where it is read by the 8086. This implies that devices which present interrupt type numbers to the 8086 must be located on the lower half of the 16 bit data bus.

(7) In the minimum mode system, the $M/\overline{IO}$ signal will be low indicating I/O during the $\overline{INTA}$ bus cycles. The 8086 internal $\overline{LOCK}$ signal will be active from T_2 of the first bus cycle until T_2 of the second to prevent the BIU from honouring a hold request between the two $\overline{INTA}$ cycles.

(8) Once the 8086 has the interrupt type number (from the bus for hardware interrupts, from the instruction stream for software interrupts or from the predefined condition), the type number is multiplied by four to form the displacement to the corresponding interrupt vector in the interrupt vector table. The four bytes of the interrupt vector are, for CS and IP register.

(9) During the transfer of control, the CPU pushes the flags, current code segment register and instruction pointer onto the stack. The new code segment and the instruction pointer values are loaded and the single step and interrupt flags are reset. Resetting the interrupt flag disables response to further hardware interrupts in the service routine unless the flags are specifically reenabled by the service routine. The CS and IP values are read from the interrupt vector table with data read cycles. No segment registers are used when referencing the vector table during the interrupt context switch. The vector displacement is added to ZERO to form the 20 bit physical address. $S_1S_0 = (10)_2$ indicating code segment or none.

(10) The number of clock cycles from the end of the instruction during which the interrupt occurred to the start of interrupt routine execution is 61 clock cycles. For software generated interrupts, the sequence of the bus cycles is the same except no interrupt acknowledge bus cycles are executed. This reduces the delay to service routine execution to 51 clocks for ***"INT"*** instruction and single step, 52 clocks for INT3 and 53 clocks for INTO. If wait states are inserted by either the memories or the device supplying the interrupt type number, the given clock times will increase accordingly.

15.7 Interrupt Procedure

When an interrupt service procedure is entered, the flags, CS and IP are pushed onto the stack and TF and IF are cleared. The procedure may enable external interrupts with the STI (set interrupt-enable flag) instruction, thus allowing itself to be interrupted by a request on INTR. (Note, however, that interrupts are not actually enabled until the instruction *following* STI has executed). An interrupt procedure always may be interrupted by a request arriving on NMI. Software or processor-initiated interrupts occurring within the procedure also will interrupt the procedure. Care must be taken in interrupt procedures that the type of interrupt being serviced by the procedure does not itself inadvertently occur within the procedure. For example, an attempt to divide by 0 in the divide error (type 0) interrupt procedure may result in the procedure being reentered endlessly. Enough stack space must be available to accommodate the maximum depth of interrupt nesting that can occur in the system.

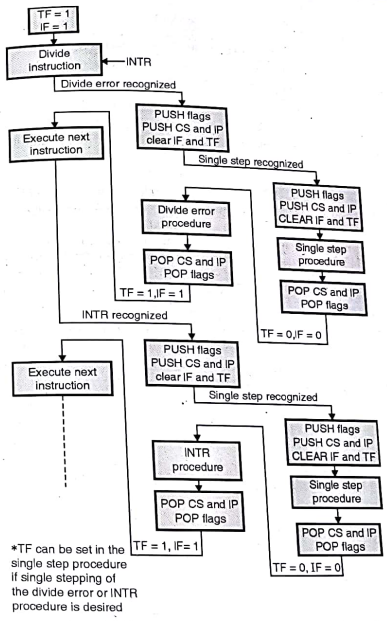


Fig. 15.7.1 : Processing simultaneous interrupts

15.8 Priority of 8086 Interrupts

Q. 5 How does 8086 decide the priority of interrupts ?

Here we have favourite question that, what will happen if two or more interrupt occur at the same time ? The answer is higher priority interrupt will be serviced and then next highest priority interrupt will be serviced. So let's know about priority of 8086 interrupt. Following Table 15.8.1 depicts the same.

Table 15.8.1

Interrupt	Priority
NMI	Highest
Divide Error, INT n, INTO	
INTR	
Single Step	Lowest

When considering the precedence of interrupts for multiple simultaneous interrupts, the following guideline apply :

1. INTR is the only maskable interrupt and if detected simultaneously with other interrupts, resetting of IF by the other interrupts will mask INTR. This causes the INTR to be the lowest priority interrupt serviced after all other interrupts unless the other interrupt service routine enable interrupts.
2. Of the non-maskable interrupts (NMI, single step and software generated), in general, NMI has highest priority (will be serviced first) followed by software, followed by single step software interrupt. This implies following three cases :

CASE 1 : Simultaneous NMI and single step will cause NMI routine to be followed by single step.

CASE 2 : Simultaneous software trap and single step trap will cause the software interrupt service to be followed by single step.

CASE 3 : Simultaneous NMI and software trap will cause NMI routine to be executed followed by the software interrupt service routine.

❑❑❑

CHAPTER

16

8255 PPI

16.1 8255

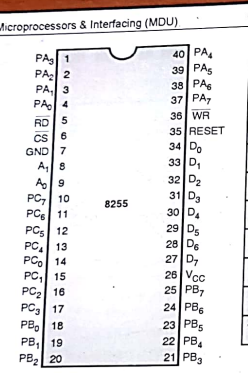
- The 8255 is a programmable peripheral interface i.e. PPI 8255.
- It is a general purpose programmable parallel I/O device.
- It contains 3 I/O ports which can be programmed in different modes.
- To program the function to all three I/O ports it contains a register called as ***control register.*** The control register defines the function of each I/O port and in which mode they should operate.
- 8255 is a general purpose in nature and provides many facilities for connecting different devices. So it is used frequently in different applications.

16.1.1 Features of 8255

- It is a programmable parallel I/O device.
- It contains 24 programmable I/O pins arranged as 2-8 bit ports and 2-4 bit ports.
- It has 3, 8 bit ports : Port A, Port B and Port C, which are arranged in two groups of 12 pins.
- Fully compatible with Intel microprocessor families.
- TTL compatible.
- Direct bit set/reset capability is available for port C.
- Improved DC driving capability.
- It can operate in 3 modes :
  (a) Mode 0 - Simple I/O
  (b) Mode 1 - Strobed I/O
  (c) Mode 2 - Strobed bi-directional I/O

16.2 Pin Configuration of 8255

The pin configuration of 8255 programmable peripheral interface is as shown in Fig. 16.2.1.



PA_0–PA_7	I/O	Port A pins
PB_0–PB_7	I/O	Port B pins
PC_0–PC_7	I/O	Port C pins
D_0–D_7	I/O	Data pins
RESET	I	Reset input
$\overline{RD}$	I	Read input
$\overline{WR}$	I	Write input
A_0–A_1	I	Address pins
$\overline{CS}$	I	Chip select
V_{CC}–GND	I	+5 V supply ground

Fig. 16.2.1 : Pin configuration

Symbol	Name and function
(1) $D_0 - D_7$	**Data bus :** These are 8 bit bi-directional data bus lines, connected to system data bus for data transfer between CPU and 8255.
(2) $\overline{CS}$	**Chip select :** This is an active LOW input signal used to select 8255 IC. If $\overline{CS}$ = 0 then 8255 will get selected and take part in data transfer from/to CPU, otherwise 8255 will be in inactive state.
(3) $\overline{RD}$	**Read :** This is an active LOW input signal used in co-ordination with other signals to send data to CPU through data lines.
(4) $\overline{WR}$	**Write :** This is an active LOW input signal used in co-ordination with other signals to send data to 8255.
(5) $A_0 - A_1$	**Address lines :** These are input, active HIGH address lines used to distinguish different ports of 8255 such as port A, port B, port C, control register. These lines are I internally decoded by 8255 to select ports as follows :

A_1	A_0	Selected port
0	0	Port A
0	1	Port B
1	0	Port C
1	1	Control register

Symbol	Name and function
(6) RESET	**Reset :** This is an active HIGH input signal used to reset 8255. When 8255 is reset it clears control word register and all ports are set to input mode.
(7) $PA_0 - PA_7$	**Port A pins 0 to 7 :** These are 8 bit bi-directional I/O pins used to send data to peripheral or to read data from peripheral. The contents are transferred to/from Port A.
(8) $PB_0 - PB_7$	**Port B pins 0 to 7 :** These are 8 bit bi-directional I/O pins used same as $PA_0 - PA_7$.
(9) $PC_0 - PC_7$	**Port C pins 0 to 7 :** These are 8 bit bi-directional I/O pins. These lines are divided in 2 sections i.e. PC_0 to PC_3 and PC_4 to PC_7. These two sections can be individually used to transfer 4 bits of data from two separate port C sections i.e. upper port C and lower Port C.

Table 16.2.1 : Port and register select signals summary

A_1	A_0	$\overline{RD}$	$\overline{WR}$	$\overline{CS}$	Operations
					Input (Read) Operation
0	0	0	1	0	Port A to data bus
0	1	0	1	0	Port B to data bus
1	0	0	1	0	Port C to data bus
					Output (Write) Operation
0	0	1	0	0	Data bus to port A
0	1	1	0	0	Data bus to port B
1	0	1	0	0	Data bus to port C
1	1	1	0	0	Data bus to control register
					Disable Function
×	×	×	×	1	Data bus tri-stated
1	1	0	1	0	Illegal condition
×	×	1	1	0	Data bus tri-stated

16.3 8255 Functional Block Diagram

Q. Draw the block diagram of 8255A.

The block diagram of 8255 is as shown in Fig. 16.3.1.
It contains following blocks

- Data bus buffer
- Group A and group B control
- Port C
- Read/Write control logic
- Port A and port B

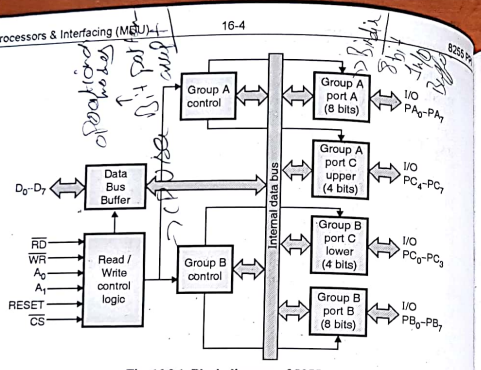


Fig. 16.3.1 Block diagram of 8255

1. **Data Bus Buffer**
   - The 8 bit bi-directional, tristate data bus buffer is used to interface 8255 internal data bus with system data bus.
   - The direction of data buffer is decided by read and write control signals.
   - When read is activated, it transmits data to the system data bus.
   - When write is activated, it receives data from system data bus.
2. **Read / Write Control Logic**
   - This block accepts inputs from system control bus and address bus and performs operations as shown in Table 16.2.1.
   - The control signals are $\overline{RD}$ and $\overline{WR}$ and address signals used are A_0 and A_1 and $\overline{CS}$.
   - The signals $\overline{RD}$ and $\overline{WR}$ are connected to $\overline{IOR}$, $\overline{IOW}$ or $\overline{MEMR}$, $\overline{MEMW}$.
   - A_0 and A_1 of 8086 are directly connected to address lines A_0 and A_1 of 8255.
   - $\overline{CS}$ is connected to address chip select decoder.
   - The 8255 operation / selection is enabled/disabled by $\overline{CS}$ signal.
3. **Group A and Group B Control**
   - The 8255 I/O ports are divided into 2 sections Group A (GA) and Group B (GB).
   - Group A consists of port A and port C upper.
   - Group B consists of port B and port C lower.
   - Each group is programmed through software.

   - The GA and GB control block receives commands from the R/W control logic to accept bit pattern from CPU.
   - GA control will contro! GA ports and GB control will control GB ports.
   - The bit pattern given by CPU consists of information (i) To control the operation of GA and GB (ii) The mode in which they should be operated.
4. **Port A and Port B**
   - The port A and port B consists of 8 bit bi-directional data output latch/ buffer and 8 bit data input buffer.
   - The function of port A and B is decided by control bit pattern available in GA and GB control.
   - The function of ports A and B are also dependent on mode of operation.
5. **Port C**
   - The port C consists of 8 bit bi-directional data output latch/buffer and 8 bit data input buffer.
   - It is divided into 2 sections, port C upper PC_U and port C lower PC_L These two sections can be programmed and used separately as a 4 bit I/O ports.
   - The port C function is dependent on mode of operation.
   - It can be used as : (i) simple I/O (ii) handshake signals (iii) status signal inputs.
   - For handshake signals and status signals it is used in co-ordination with port A and port B.
   - The direct bit set/reset capability is provided by port C only.

16.4 8255 Operating Modes

Q. 1 Explain different modes of 8255A.

Q. 2 Explain Bit set/Reset mode of 8255A.

- The 8255 IC provides one control word register.
- It is selected when $A_0 = 1$, $A_1 = 1$, $\overline{CS} = 0$ and $\overline{WR} = 0$.
- The read operation is not allowed for control register.
- The bit pattern loaded in control word register specify an I/O function for each port and the mode of operation in which the ports are to be used.
- There are 2 different control word formats which specify 2 basic modes :
  - BSR - Bit set reset mode
  - I/O mode.
- The two basic modes are selected by D_7 bit of control register. When $D_7 = 1$ it is a I/O mode and when $D_7 = 0$; it is a BSR mode.

16.4.1 BSR Mode

- The BSR mode is a port C bit set/reset mode.
- The individual bit of port C can be set or reset by writing control word in the control register.
- The control word format of BSR mode is as shown in Fig. 16.4.1.

- The pin of port C is selected using bit select bits [b b b] and set or reset is decided by bit S/R.
- The BSR mode affects only one bit of port C at a time.
- The bit set using BSR mode remains set unless and until you change the bit. So to set any bit of port C, bit pattern is loaded in control register.
- If a BSR mode is selected it will not affect I/O mode.

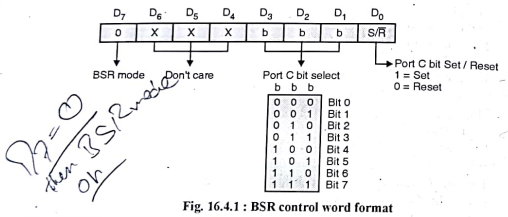
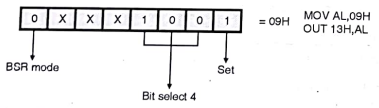


Fig. 16.4.1 : BSR control word format

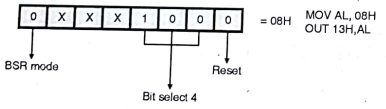
Ex. 16.4.1 : Write a set of instructions to perform the following :
(1) Set bit 4 of port C. (2) Reset bit 4 of port C.
Assume the address of PA = 10 H, PB = 11 H, PC = 12 H and Control reg. = 13 H.

Soln. :

(1) To set bit 4 of port C the bit pattern required in BSR mode will be,



(2) To reset bit 4 of port C the bit pattern required in BSR mode will be,



16.4.2 I/O Modes

There are three I/O modes of operation :

(1) Mode 0 - Basic I/O (2) Mode 1 - Strobed I/O

(3) Mode 2 - Bi-directional I/O

The I/O modes are programmed using control register. The control word format of I/O modes is as shown in Fig. 16.4.2.

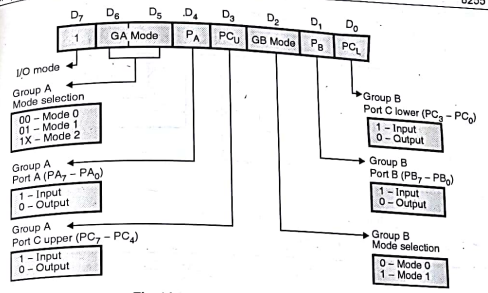


Fig. 16.4.2 : I/O modes control word format

Function of each bit is as follows :

(1) **D_7** : When the bit $D_7 = 1$ then I/O mode is selected, if $D_7 = 0$ then BSR mode is selected. The function of bits D_0 to D_6 is dependent on mode (I/O mode or BSR mode).

(2) **D_6 and D_5** : In I/O mode the bits D_6 and D_5 specifies the different I/O modes for group A i.e. Mode 0, Mode 1 and Mode 2 for port A and port C upper.

(3) **D_4 and D_3** : In I/O mode the bits D_4 and D_3 selects the port function for group A. If these bits = 1 the respective port specified is used as input port. But if bit = 0, the port is used as output port.

(4) **D_2** : In I/O mode the bit D_2 specifies the different I/O modes for group B i.e. Mode 0 and Mode 1 for port B and port C lower.

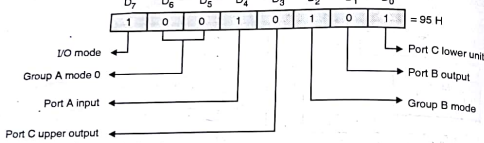
(5) **D_1 and D_0** : In I/O mode the bits D_1 and D_0 selects the port function for group B. If these bits = 1 the respective port specified is used as input port. But if bit = 0, the port is used as output port.

From the above explanation you can observe that all the 3 modes i.e. Mode 0, Mode 1 and Mode 2 are only for group A ports, but for group B only 2 modes i.e. Mode 0 and Mode 1 are provided.

When 8255 is reset, it will clear control word register contents and all the ports are set to input mode. The ports of 8255 can be programmed for other modes by sending appropriate bit pattern to control register.

Ex. 16.4.2 : Write a set of instructions to perform the following :
(1) Initialise port A as input, port B as output, port C upper as output and port C lower as input.
(2) Use Mode 0 for group A and Mode 1 for group B.

Soln. : The control word format required will be,



The initialization instructions are :

```
MOV AL, 95H.
OUT CWR, AL.
```

16.5 I/O Operating Modes

Q. Write an assembly language program to use 8255 as per the following requirements :
(i) Group A for hand-shake input mode.
(ii) Group B for hand-shake output mode. Assume IC is just reseted.

- The 8255 can operate in basic 3 I/O modes; mode 0, mode 1 and mode 2. Now we will see details of 8255 modes.

- Mode 0 – Simple Input / Output mode
- Mode 1 – Strobed I/O
- Mode 2 – Strobed Bi-directional I/O

16.5.1 Mode 0 (Simple Input / Output Mode)

- In Mode 0, all ports i.e. port A, port B, port C provide simple input or output operation separately.
- The data is simply read from a port or it is simply written to a port. In Mode 0 there is no restriction between function of ports.
- The port C, 2 sections port C upper and port C lower can be individually programmed as 4 bit ports.

Features of Mode 0

- Two 8 bits ports P_A and P_B and two 4 bit ports PC_L and PC_U.
- All the ports can be separately programmed as input or output.
- If the port is programmed as output, the outputs are latched.
- If the port is programmed as input, the inputs are buffered.
- As 4 ports are to be used, 16 different I/O configurations are possible.
- No facility for interrupt driven I/O.

Mode 0 - Input mode

- The 8255 is initialized in input mode by using control register.
- When CPU wants to read data from an input port, the CPU will first send address of port on address lines so $\overline{CS}$, A_0 and A_1 will select the appropriate port. After selecting a port CPU will send a control signal $\overline{RD}$ to read data from external peripheral through port and data bus.

- The timing waveform for Mode 0 input mode is as shown in Fig. 16.5.1.

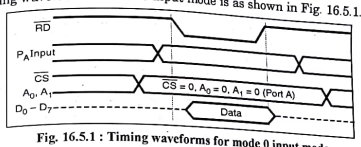


Fig. 16.5.1 : Timing waveforms for mode 0 input mode

Mode 0 - Output mode

- The 8255 is initialized in output mode by using control register.
- When CPU wants to send data to an output port, the CPU will first send address of port on address lines so $\overline{CS}$, A_0 and A_1 will select the appropriate port. After selecting a port CPU will send data and control signal $\overline{WR}$ to write data to port through data bus.
- As the port is in output mode the contents will get latched in the port.
- The timing waveform for Mode 0 output mode is as shown in Fig. 16.5.2.

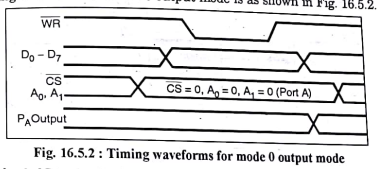


Fig. 16.5.2 : Timing waveforms for mode 0 output mode

16.5.2 Mode 1 (Strobed I/O)

- In mode 1, group A and/or group B can be used.
- Each group consists of 8 bit latched input or output port and 3 bits of port C.
- When port A is used in Mode 1, port C upper bits are used to control the port A.
- When port B is used in Mode 1, port C lower bits are used to control the port B.
- When P_A is used as input in Mode 1 the bits PC_3 PC_4 and PC_5 are used for control.
- When P_A is used as output in Mode 1, the bits PC_3, PC_6 and PC_7 are used for control.
- When P_B is used in Mode 1 the bits PC_0, PC_1 and PC_2 are used for control in both input and output mode.
- In all, if both P_A and P_B are programmed in Mode 1, it uses 6 port C bits to control and remaining 2 bits are not used for Mode 1.
- The 2 unused bits can be used as simple I/O. The function of 2 unused bits is decided by bit D_3 of the control word. If $D_3 = 1$ the bits are used as input pins and if $D_3 = 0$, the bits are used as output pins.

- The Mode 1 provides facility to transfer data to/from an external peripheral to 8255 using handshake signals.
- When an external device wants to send data to 8086 through 8255, it will send data to port and inform 8255 about it.
- The data is available in 8255 and next data should not be sent by external device so 8255 will generate a signal to indicate data present. In addition to this 8255 will inform 8086 about availability of data.
- When 8086 gets this signal, 8086 will read data from 8255 port. Now the 8255 is ready to accept next data byte so it will remove the signal of data present. If next data byte is available in peripheral device, then it will repeat the above process.
- The above function of handshake signals can also be performed using Mode 0 but in that case 8086 has to read the status of peripheral, check the status and then take decision on that. This process will waste valuable time of processor.

Features of Mode 1

- Two 8 bit ports A and B function as I/O ports.
- Each port uses 3 lines from port C as handshake signals.
- The remaining lines of port C can be used as simple I/O.
- The input and output data are latched.
- Port C handshake signals can be used to interrupt the CPU i.e. interrupt logic is supported by 8255 to transfer data from 8255 to CPU.

Mode 1 - Input Mode

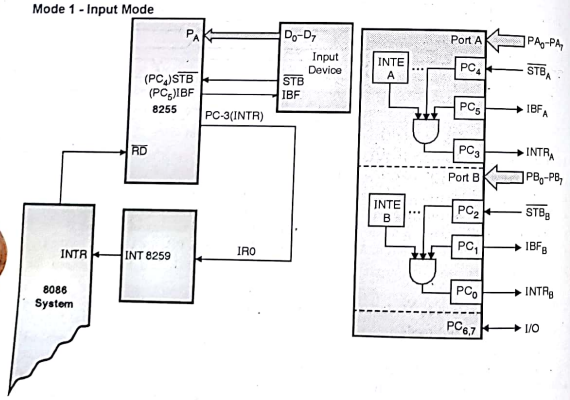


Fig. 16.5.3 : Mode 1 : Input mode interfacing *Fig. 16.5.4 : P_A, P_B and P_C in mode 1 input mode*

- In Mode 1 input mode port A uses PC_3, PC_4, PC_5 and port B uses PC_0, PC_1, PC_2 signals as handshake signals.
- The port A, port B and handshake signals are interfaced with peripheral as shown in Fig. 16.5.3. The aim of interfacing is to transfer data from peripheral to CPU through 8255. The Fig. 16.5.3 shows interfacing I/P port to PA, similarly it can be done with PB.
- The different handshake signals used are $\overline{STB}$, IBF and INTR. The internal organisation of these signals is as shown in Fig. 16.5.4.

(1) $\overline{STB}$ (Strobe input)

- This is active low input signal for 8255.
- If IBF signal is not present this signal is generated by peripheral to indicate that it has transmitted data or written data to input port.
- When 8255 gets data byte with STB signal the input buffer contains a data byte so 8255 will generate IBF signal.

(2) IBF (Input buffer full)

- This is an active high output signal generated by 8255.
- This signal is generated by 8255 in response to $\overline{STB}$ signal to give acknowledge to peripheral device.
- The input buffer full signal indicates that a data byte is present in input port latch.
- This signal is checked by peripheral device to take decision, for sending next byte of data.
- If IBF signal is active the peripheral will not send next data. IBF is reset when CPU reads input port.

(3) INTR (Interrupt request)

- This is an active high output signal given by 8255.
- If interrupt driven I/O is used the INTR signal is used to interrupt CPU.
- The INTR signal is conditioned by $\overline{STB}$, IBF and INTE signals. If $\overline{STB}$ = 1, IBF = 1 and INTE = 1 then INTR = 1, for all other combinations INTR = 0. The INTR signal is used to indicate CPU to read input port. The falling edge of $\overline{RD}$ will reset INTR.

(4) INTE (Interrupt enable)

- This is an internal flip-flop used to enable or disable interrupt signal.
- If INTE flip-flop is set the interrupt will be generated depending on $\overline{STB}$ and IBF signals.
- The two flip-flops $INTE_A$ and $INTE_B$ are used for group A and group B separately.
- These INTE flip-flops are set/reset by using BSR mode only. The $INTE_A$ is set/reset through PC_4 bit and $INTE_B$ is set/reset through PC_2 bit.

- If interrupt driven I/O is used for data transfer then INTE bit must be set, which will be used to generate INTR signal.
- The INTR logical equations will be,

$$INTR_A = INTE_A \cdot \overline{STB_A} \cdot IBF_A$$
$$INTR_B = INTE_B \cdot STB_B \cdot IBF_B$$

- The timing diagram of control signals in Mode 1 input mode is as shown in Fig. 16.5.5.

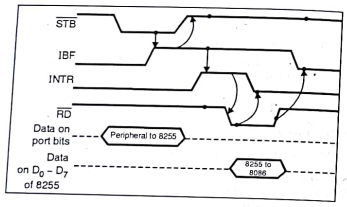


Fig. 16.5.5 : Timing diagram of mode 1 input mode

(1) **Interrupt driven I/P** In this case the INTR signal is connected as interrupt input to 8086. The sequence of events will be as follows :

- The data is transferred by peripheral to 8255 and STB signal is made low.
- When $\overline{STB}$ signal is received by 8255. 8255 generates a signal IBF.
- When IBF is generated the condition of $\overline{STB}$, IBF and INTE is satisfied to generate INTR signal.
- In response to INTR signal, 8086 reads data from 8255 input port.
- The read operation will reset IBF signal and INTR signal.
- When the peripheral wants to send next data byte it will check IBF signal. If it is low it will repeat the above steps to transfer next data byte.

(2) **Status driven I/P**

- In Mode 1 of 8255 the port C is used as status word. In this method the INTR signal of 8255 is not used to interrupt 8086. INTE bit will be reset. The service to 8255 is given by polling the status register. The sequence of events will be as follows :
  - CPU will read port C of 8255.
  - It will check IBF signal if it is high then a data is read from port.
  - If IBF is low the CPU will go on reading and checking the signal IBF.

The port C is used as status word and its definitions will be as shown in Fig. 16.5.6.

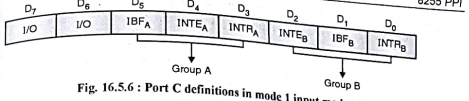


Fig. 16.5.6 : Port C definitions in mode 1 input mode

Mode 1 - Output Mode

- In Mode 1 output mode port A uses PC_3, PC_6 and PC_7 and port B uses PC_0, PC_1, PC_2 signals as handshake signals.
- The port A, port B and handshake signals are interfaced with peripheral as shown in Fig. 16.5.7. The aim of interfacing is to transfer data from CPU to peripheral through 8255.
- The different handshake signals used are $\overline{OBF}$, $\overline{ACK}$ and INTR the internal organisation of these signals is as shown in Fig. 16.5.8.

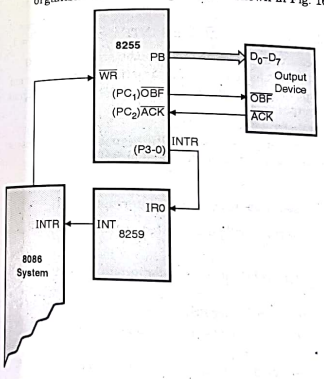


Fig. 16.5.7 : Mode 1 : Output mode interfacing

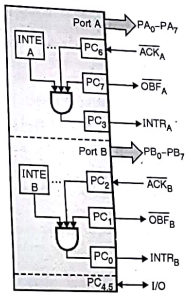


Fig. 16.5.8 : P_A, P_B and P_C in mode 1 output mode

The function of these handshake signals is as follows :

(1) **$\overline{OBF}$ (Output buffer full)**

- This is an active low output signal generated by 8255.

- When CPU writes data to output port, the data is available in output port. It is indicated by giving $\overline{OBF}$ signal. This signal is used to indicate, that new data is ready to be read. In response to $\overline{OBF}$ signal peripheral reads data byte from output port and acknowledges it by $\overline{ACK}$ signal. So after receiving $\overline{ACK}$ signal, the 8255 removes OBF signal.

(2) $\overline{ACK}$ (Acknowledge)

- This is an active low input signal for 8255.
- The peripheral reads the data byte and gives acknowledge to 8255. The acknowledge signal indicates that new data byte can be loaded in output buffer.

(3) INTR (Interrupt request)

- This is an active high output signal given by 8255.
- The INTR signal is used to interrupt CPU, if interrupt driven I/O system is to be used.
- The INTR signal is conditioned by $\overline{OBF}$, $\overline{ACK}$ and INTE signals. If $\overline{OBF}$ = 1, $\overline{ACK}$ = 1 and INTE = 1 then INTR = 1, for all other combinations INTR = 0.
- The INTR = 1 condition specifies that output buffer is empty, peripheral is not reading data and interrupt system is enabled. In response to INTR signal CPU writes data to 8255 output port.
- The above steps for data transfer will be repeated.
- When a data is written to output port it makes OBF signal LOW and resets INTR signal. Again when peripheral reads data, INTR signal is set.

(4) INTE (Interrupt enable)

- This is an internal flip-flop used to enable or disable interrupt signal.
- If INTE flip-flop is set, the interrupt will be generated depending on $\overline{OBF}$ and $\overline{ACK}$ signals.
- The two flip-flops $INTE_A$ and $INTE_B$ are used for group A and group B separately.
- These INTE flip-flops are set/reset by using BSR mode only.
- The $INTE_A$ is set/reset through PC_6 bit and $INTE_B$ is set/reset through PC_2 bit.
- If interrupt driven I/O is used, for data transfer, then INTE bit must be set, which will be used to generate INTR signal.
- The INTR logical equations will be,
- $INTR_A = INTE_A \cdot \overline{ACK_A} \cdot \overline{OBF_A}$; $INTR_B = INTE_B \cdot \overline{ACK_B} \cdot \overline{OBF_B}$
- The timing diagram of control signals in Mode 1 output mode is as shown in Fig. 16.5.9.

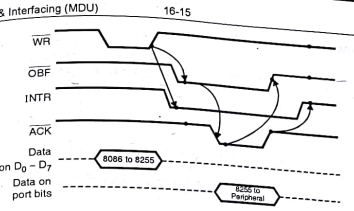


Fig. 16.5.9 : Timing diagram of mode 1 output mode

(1) Interrupt driven output

In this case, the INTR signal is connected as interrupt input to 8086. The sequence of events will be as follows :

- The data is transferred by CPU to 8255 output port, by using $\overline{WR}$ signal.
- The $\overline{OBF}$ will go LOW to indicate data is available in output port.
- When peripheral detects $\overline{OBF}$ signal, it reads data from output port and acknowledge it by giving $\overline{ACK}$ signal.
- The acknowledge from peripheral will make $\overline{OBF}$ signal HIGH and condition of IBF, ACK and INTE is satisfied to generate INTR signal.
- In response to INTR, CPU writes next data to output port and above steps for data transfer will get repeated.

(2) Status driven output

In Mode 1 of 8255, the port C is used as status word. In this method, the INTR signal of 8255 is not used to interrupt 8086. INTE bit will be reset. The service to 8255 is given by polling the status register. The sequence of events will be as follows :

- CPU will read port C of 8255.
- It will check OBF signal if it is high then data is transferred to output port.
- If OBF signal is low the CPU will not write data to output port and will go on reading and checking the OBF signal.

The port C is used as status word and its definitions will be as shown in Fig. 16.5.10.

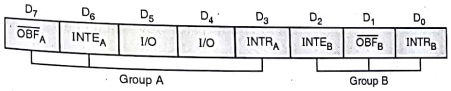


Fig. 16.5.10 : Port C definitions in mode 1 output mode

16.5.3 Mode 2 - Strobed Bi-directional I/O

- In this mode group A is used as input and output i.e. for transmitting and receiving data from peripheral through 8255 as shown in Fig. 16.5.11.
- The transfer of data is achieved by port C handshake signals. The group B can be in Mode 0 or Mode 1.
- The bi-directional data is transferred through port A so it consists of input and output latch.
- The Mode 2 is combination of Mode 1 input and output both at a time to port A.
- The interrupt signals of input and output mode are combined to generate common interrupt signal to CPU. The internal organization of these signals is as shown in Fig. 16.5.12.
- The different handshake signals used are $\overline{OBF_A}$, $\overline{ACK_A}$, $\overline{STB_A}$, IBF_A and $INTR_A$. 2 handshake signals are used for output operation, 2 are used for input operation and one is common to both.

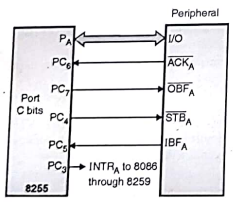


Fig. 16.5.11 : Mode 2 interfacing

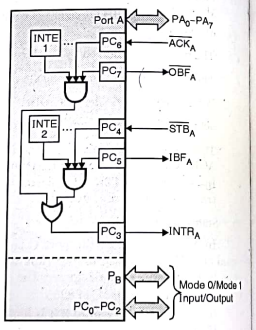


Fig. 16.5.12 : P_A, P_B and P_C in mode 2

Output operation

$\overline{OBF}$ (Output buffer full) : This is an active low output signal generated by 8255. When CPU writes data to output port 8255 will enable OBF signal to indicate peripheral that data is available in output buffer.

$\overline{ACK}$ (Acknowledge) : This is an active low input signal for 8255. When the peripheral detects $\overline{OBF}$ signal, it reads data from 8255 port and makes $\overline{ACK}$ = 0 and the

ACK signal is used to acknowledge 8255 that data is read from port so 8255 will remove $\overline{OBF}$ signal to indicate output buffer is empty.

Input operation

$\overline{STB}$ (strobe) : This is an active low input signal. When the peripheral writes data to input buffer, it generates a signal STB to indicate 8255 that it has written data.

IBF (Input buffer full) : When data is available in input buffer 8255 will enable IBF signal to indicate that data is available in input buffer.

INTR (Interrupt request)

- This is an output signal given by 8255 to request CPU service.
- The INTR is generated in two different conditions input and output.
- The interrupt is generated for input mode when IBF = 1, $\overline{STB}$ = 1 and $INTE_1$ = 1 and for output mode when $\overline{OBF}$ = 1, $\overline{ACK}$ = 1 and $INTE_2$ = 1. The $INTE_1$ and $INTE_2$ are set/reset using BSR mode, port C bits used are PC_6 and PC_4 respectively.

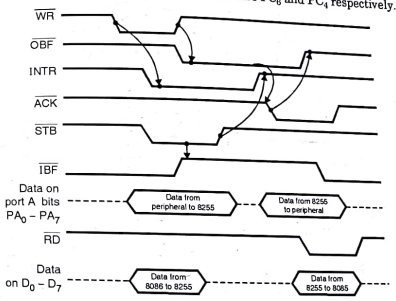


Fig. 16.5.13 : Timing diagram of mode 2

- The logical equation will be,

$$INTR_A = INTE_1 \cdot \overline{ACK_A} \cdot \overline{OBF_A} + INTE_2 \cdot \overline{STB_A} \cdot IBF_A$$

The timing diagram of Mode 2 bi-directional data transfer for data transfer from peripheral to CPU and CPU to peripheral are as shown in Fig. 16.5.13.

The Mode 2 also supports both modes of data transfer i.e. Interrupt drive I/O and status driven I/O. The port C is used as status word and its definitions are as follows :

$\overline{OBF_A}$	$INTE_1$	IBF_A	$INTE_2$	$INTR_A$	x	x	x

Q. 1 If PC_7 is set for input mode and P_A is in mode 1 input. Does BSR operation affect the PC_7 bit?

Ans. : When port C pin is used in input mode it does not get affected because of BSR mode.

Q. 2 If P_A and P_B are in mode 0 and P_C lower is being used as output. Does BSR operation affect the PC_1 bit ?

Ans. : When port C pin is being used in output mode and is not being used in handshaking mode the port C bit can change because of BSR operation.

Q. 3 Consider port A and port B has been configured as Mode 1 input. Which bit of port C can be changed by the instruction OUT P_C.

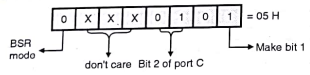
Ans. : When port A and port B are used in handshaking mode they use port C bits as handshake signals. So only bits PC_6 and PC_7 remains unused. If these bits are programmed in output mode they will get changed.

Now let's .summarize, three modes of 8255.

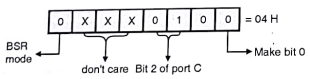
Ex. 16.5.1 : Blink port C bit 2 of 8255.

Program statement : Write a program to blink port C bit 2 of 8255. Assume address of control word register of 8255 as 83 H. Use Bit Set / Reset mode.

Soln. : Let us write the control word to make bit 2 high.



Control word to make bit 2 low



Program

Label	Instruction	Comments
L1 :	MOV AL, 05H	Load control word to make PC_2 high
	OUT 83H,AL	
	CALL DELAY	
	MOV AL, 04H	Load control word to make PC_2 low
	OUT 83H,AL	
	CALL DELAY	
	JMP L1	

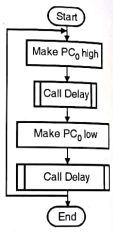


Fig. P. 16.5.1

16.6 Applications using 8255 Chip

8255 is most widely used chip for many applications. For example,

(1) LED / Relay Interface (2) Key Board Interface
(3) Display Interface (4) ADC / DAC Interface
(5) Stepper motor Interface (6) Traffic Signal Controller
(7) Lift Controller etc.

16.7 Interfacing 8255 with 8086

Fig. 16.7.1 shows interfacing of 8255 with 8255 in I/O mapped I/O technique. Only lower data bus $D_0 - D_7$ is used as 8255 is an 8 bit device. The Reset out signal is connected to the Reset signal of 8255. In case of interrupt driven I/O INTR signal (PC_3 or PC_0) from 8255 is connected to INTR input of 8086.

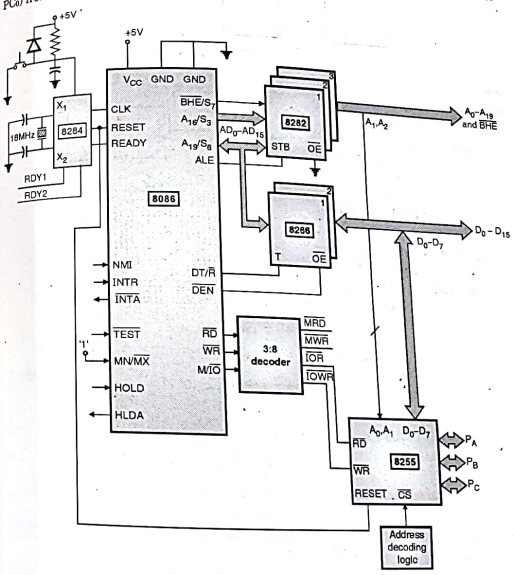


Fig. 16.7.1

16.8 Interfacing Examples with 8086

Ex. 16.8.1 : Design a system using 8255 to generate a square wave of 1KHz. Write corresponding program.

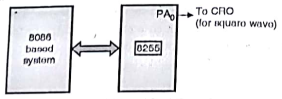


Fig. P. 16.8.1

Soln. :

Program

```
        MOV AL, 00H
again : OUT 80H, AL
        CALL delay_500 μsec
        NOT AL
        JMP again
```

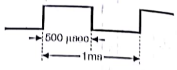


Fig. P. 16.8.1(a)

(Assume PA address = 80H)

delay 500 μsec subroutine assumed :

Generation of delay of 500 μsec :

Crystal frequency : 15 MHz

Operating frequency : 5 MHz

$$1 \text{ machine cycle} = 4 * \text{T-state} = 4 * \frac{1}{5} \mu\text{sec} = 0.8 \mu\text{sec}$$

$$\text{Delay required} = 500 \mu\text{sec}$$

$$\text{Machine required} = \frac{500}{0.8} = (625)_{10} = (0271)_{16}$$

```
        Delay_500μsec
        MOV CX, 026F H      3B    2m/c
next :  Loop next           2B    1m/c
        RET
```

$\therefore$ Total machine required = 2 + 1 * count

$625 = 2 + 1 * \text{count}$

$\therefore$ count = $(623)_{10} = (26F)_{16}$

$\therefore$ Program assume PA address = 80H

CW address = 86H

```
MOV AL, 80H
OUT 86 H, AL
MOV AL, 00H
```



```
again : OUT 80H, AL
        CALL delay_500 μs
        NOT AL
        JMP again
```

Ex. 16.8.2 : Interfacing key board and display to 8255.

Interface a key to 8255 and indicate its position using a LED i.e. :

If switch is closed – LED should be ON

If switch is open – LED should be OFF.

Soln. :

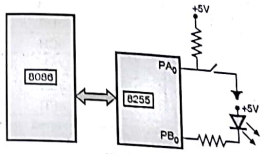


Fig. P. 16.8.2

Control Word

I/O	GA mode		PA	PCU	GB mode		PB	PCL
1	0	0	1	0	0		0	0
		9				0		

Assume address

PA = 80H PB = 82H

PC = 84H CW = 86H

Program

```
        MOV AL, 90H   } initialize CW
        OUT 86H, AL   }
again : IN AL, 80H    }
        OUT 82, AL    } input to PA = output of PB
        JMP again     }
```

[Logic : if switch is closed – Logic 0 If switch is open – Logic 1]

Ex. 16.8.3 : Write a program to interface 8 keys using 8255 and display the key pressed on 7 segment display.

No	a	b	c	d	e	f	g	dp	Common anode Hex no.	Hex
0	1	1	1	1	1	1	0	0	FCH	03H
1	0	1	1	0	0	0	0	0	60H	9FH
2	1	1	0	1	1	0	1	0	DAH	25H
3	1	1	1	1	0	0	1	0	F2H	0DH

No	a	b	c	d	e	f	g	dp	Common anode Hex no.	Hex
4	0	1	1	0	0	1	1	0	66H	99H
5	1	0	1	1	0	1	1	0	B6H	49H
6	1	0	1	1	1	1	1	0	BEH	41H
7	1	1	1	0	0	0	0	0	F0H	1FH
8	1	1	1	1	1	1	1	0	FEH	01H
9	1	1	1	1	0	1	1	0	F6H	09H
A	1	1	1	0	1	1	1	0	EEH	11H
B	0	0	1	1	1	1	1	0	3EH	C1H
C	1	0	0	1	1	1	0	0	9CH	63H
D	0	1	1	1	1	0	1	0	7AH	85H
E	1	0	0	1	1	1	1	0	9EH	61H
F	1	0	0	0	1	1	1	0	8EH	71H

Assume this look up table for common anode is stored on memory location from 3200 H onword.

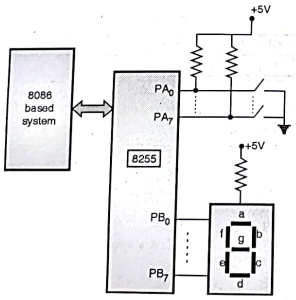


Fig. P. 16.8.3

Control word :	I/O	GA mode		PA	PCU	GB mode	PB	PCL
(90 H)	1	0	0	1	0	0	0	0

Assume : PA = 80H PB = 82H
PC = 84H CW = 86H

Program

Label	Instruction	Comments
	MOV AX, 3000H	
	MOV DS, AX	
	MOV BX, 2000H	
	MOV AL, 90H	C.W.
	OUT 86H, AL	C.W.
	MOV CH, 00H	
	MOV AL, FFH	if no switch
	OUT 82H, AL	pressed
again :	IN AL, 80H	
	MOV CL, 08H	initialize counter
	SHL AL, 1	to check which bit
	JC OVER	zero that no is
	MOV AL, CL	∴ that no is given to
	DEC AL	∴ that no is given to
		XLAT
nothing :		
	OUT 82H, AL	
	JMP again	
over :	LOOP next	
	MOV AL, FFH	if no key pressed
	JMP nothing	if no key pressed

Interface Hex keypad with 7 segment display using 8255 to display switch press.

Ex. 16.8.4 : Switch debouncing.

When human being presses a switch there are some bounces as shown in Fig. P. 16.8.4(a).

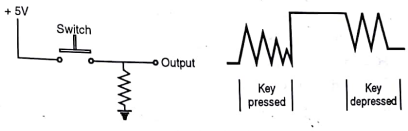


Fig. P. 16.8.4(a)

A bounce laps for not more than 20 msec.

Hence switch debouncing implements checking of the key pressed 2 times at the interval of 20 msec to confirm a valid switch pressed.

Control word

I/O	GA mode		PA	PCU	GB mode		PB	PCL
1	0	0	0	0	0		0	0
8				0				

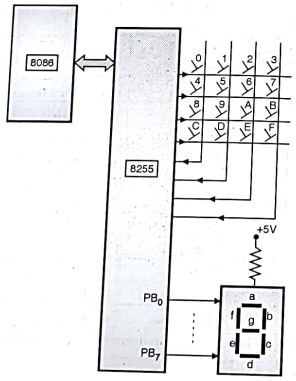


Fig. P. 16.8.4(b)

Switch pressed = row number * 4 + column number

Assume PA ⇒ 80H (address)
PB ⇒ 82H (address)
PC ⇒ 84H (address)
CW ⇒ 86H (address)

Program

Label	Instruction	Comments
	MOV AX, 3000H	
	MOV DS, AX	
	MOV BX, 2000H	
	MOV AL, 81H	
	OUT 86H, AL	
next :	MOV AL, 00H	
	OUT 80H, AL	
again :	IN AL, 84H	
	AND AL, 0FH	
	CMP AL, 0FH	
	JZ again	
	CALL delay_20 ms	debouncing
	IN AL, 84H	debouncing
	AND AL, 0FH	debouncing
	CMP AL, 0FH	debouncing
	JZ again	debouncing
	MOV CL, 00H	
rep1 :	ROR AL, 1	to get column no
	JNC over	
	INC CL	
	CMP rep1	
over :	MOV CH, 00	to get column no
	MOV AL, 0E H	
rep2 :	MOV DL, AL	
	OUT 80H, AL	
	IN AL, 0FH	
	CMP AL, 0FH	
	JNZ over1	
	INC CH	
	MOV AL, DL	
	SHL AL, 01	
	JMP rep2	

Label	Instruction	Comments
over1:	MOV AL, 40H	
	MOV CH	
	ADD AL, CL	
	XLAT	
	OUT 82H, AL	
	JMP next	

16.9 Interfacing 8255 to 8085 in I/O Mapped I/O

Step 1 : In I/O mapped I/O scheme the address of I/O ports is of 8 bits. The $A_0 - A_7$ contents are copied on A_8 to A_{15}. So we can use any part or combination of the 2 for decoder logic.

Step 2 : The 8255 requirements are A_0 and A_1 should be directly connected. So the remaining lines A_2 to A_7 or A_{10} to A_{15} can be used for chip select decoder logic.

Step 3 : If IO/$\overline{M}$, $\overline{RD}$ and $\overline{WR}$ are used to generate control signals $\overline{IOR}$, $\overline{IOW}$, $\overline{MEMR}$ and $\overline{MEMW}$ from these four, $\overline{IOR}$ and $\overline{IOW}$ are connected to $\overline{RD}$ and $\overline{WR}$ of 8255.

Step 4 : Now we will decide the lines for decoder logic.

$\frac{A_{15}}{A_7}$	$\frac{A_{14}}{A_6}$	$\frac{A_{13}}{A_5}$	$\frac{A_{12}}{A_4}$	$\frac{A_{11}}{A_3}$	$\frac{A_{10}}{A_2}$	$\frac{A_9}{A_1}$	$\frac{A_8}{A_0}$	
						0	0	Port A
						0	1	Port B
						1	0	Port C
						1	1	CWR

Used to enable decoder Used as input to decoder

We are using 3 : 8 decoder and so decoder logic will be as shown in Fig. 16.9.1.

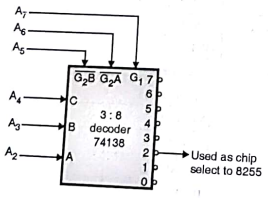


Fig. 16.9.1 : Chip select decoder logic

Step 5 : Depending on chip select decoder logic connections address of 8255 will be as follows :

A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0		
1	0	0	0	1	0	0	0	= 88 H	Port A
1	0	0	0	1	0	0	1	= 89 H	Port B
1	0	0	0	1	0	1	0	= 8A H	Port C
1	0	0	0	1	0	1	1	= 8B H	Control word register

Step 6 : The detailed interfacing diagram will be as shown in Fig. 16.9.2.

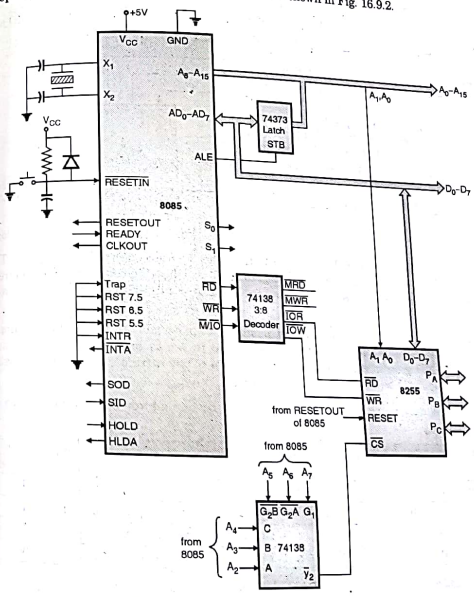


Fig. 16.9.2 : 8255 interfacing in I/O mapped I/O

16.10 Interfacing 8255 to 8085 in Memory Mapped I/O

Step 1 : In memory mapped I/O the addresses of I/O ports are of 16 bits. The 8255 requires A_0 and A_1 lines to be directly connected. The remaining lines A_2 to A_{15} can be used to generate chip select signal.

Step 2 : In memory mapped I/O the control signals used for I/O devices are $\overline{MEMR}$ and $\overline{MEMW}$. These control signals can be generated by using IO / $\overline{M}$, $\overline{RD}$ and $\overline{WR}$ signals and 3.8 decoder.

If we use IO/$\overline{M}$ signal in chip select decoder logic to enable decoder then $\overline{RD}$ and $\overline{WR}$ can be directly connected minimising the hardware.

Step 3 : In this case we are using IO/$\overline{M}$ in chip select decoder logic. The A_2 to A_{15} address lines are to be used for generation of chip select. It will not be possible to use all 14 address lines to 3 : 8 decoder. So some lines will remain unconnected i.e. don't cares. Suppose the address lines and IO/$\overline{M}$ are connected as shown in Fig. 16.10.1.

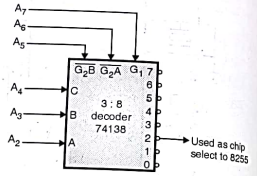


Fig. 16.10.1 : Chip select decoder logic

When IO/$\overline{M}$ = 0, A_{15} = 1 and A_{14} = 0, then the 3 : 8 decoder will be enabled. The zeroth output of decoder is used as chip select to 8255. So when the combination on address lines A_{13} = 0, A_{12} = 0 and A_{11} = 0 is present the zeroth output line gets activated and selects 8255 IC.

The decoding can also be done by using logic gates and A_2 to A_{15} lines. Simple example is use 14 input NAND gate, connect all A_2 to A_{15} as input to NAND gate and output of NAND gate is connected as chip select to 8255. In this case no line remains unconnected so there will be no don't care condition .

Step 4 : Depending on chip select decoder logic connection addresses for 8255 will be as follows

A_{15}	A_{14}	A_{13}	A_{12}	A_{11}	A_{10}	A_9	A_8	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0	Port selected
1	0	0	0	0	x	x	x	x	x	x	x	x	x	0	0	P_A
1	0	0	0	0	x	x	x	x	x	x	x	x	x	0	1	P_B
1	0	0	0	0	x	x	x	x	x	x	x	x	x	1	0	P_C
1	0	0	0	0	x	x	x	x	x	x	x	x	x	1	1	CWR

The address in hex. if we consider all don't cares to be zero's will be,

P_A = 8000 H, P_B = 8001 H, P_C = 8002 H, CWR = 8003 H.

Step 5 : The detailed interfacing diagram will be as shown in Fig. 16.10.2.

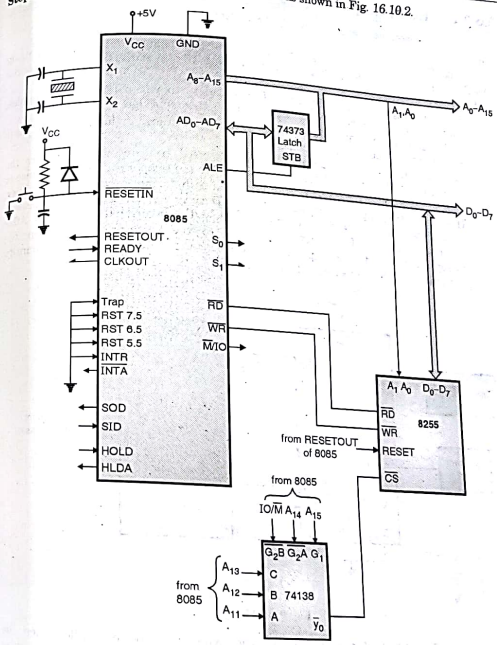


Fig. 16.10.2 : 8255 interfacing in memory mapped I/O

16.11 Interfacing Examples with 8085

Ex. 16.11.1 : Temperature Measurement and Control System :

Soln. :

Fig. P. 16.11.1 shows the temperature sensing and heater control circuitry using ON/OFF control. It includes sensing circuitry analog to digital converter, circuit required to drive the controller.

- The sensing circuitry consists of resistance thermometer, thermocouples, thermistors, pyrometers place in the arm of wheatstone bridge.
- A change in temperature causes a change in the resistance giving a voltage that is proportional to the change in temperature.
- Generally RTD or thermocouples are widely used transducers to measure the temperature. The output of RTD is proportional to the temperature of the furnace or oven in microvolts. This voltage is to be amplified by using a multistage amplifier. The amplified voltage is then applied to an A/D converter. The 8085 microprocessor sends a start of conversion (SOC) signal to the A/D Converter through 8255. After the conversion an EOC signal is given to the microprocessor. Then the microprocessor reads the output of the A/D converter which is a digital quantity proportional to the temperature to be measured.

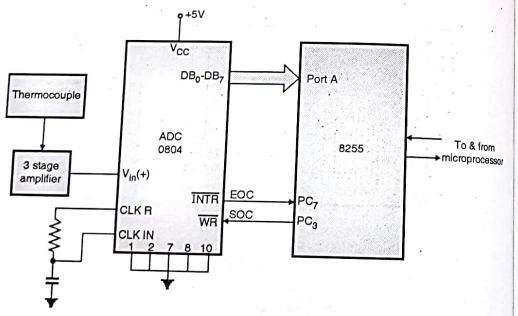
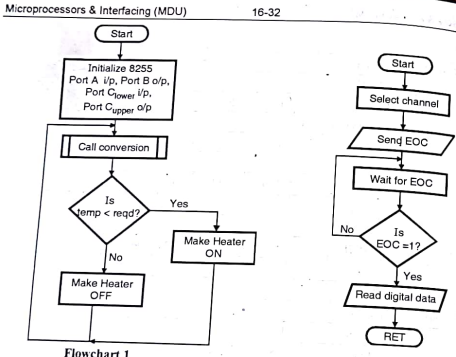


Fig. P. 16.11.1

- If the temperature of the furnace or the oven is to be controlled then the microprocessor first measures the temperature compares it with a reference temperature at which the temperature is to maintained.
- If the temperature is higher than the reference temperature, the microprocessor sends signals to reduce the temperature. If temperature is less than the reference temperature then control signal is sent to increase the temperature.

Program :

Label	Instructions	Comments
	MVI A,91H	
	OUT CWR	Initialize 8255
L2 :	CALL CONVERSION	
	CPI 80H	compare with set point
	JC L1	
	MVI A,0EH	Reset PC_7 bit to switch off heater with BSR mode
	OUT CWR	
	JMP L2	
L1 :	MVI A,0FH	Set PC_7 bit to switch ON heater with BSR mode
	OUT CWR	
	JMP L2	
Subroutine Conversion		
	MVI A,00H	
	OUT PB	Send address to select input 0
	MVI A,08H	Latch address by making ALE high.
	OUT PB	
	MVI C,0AH	
L3 :	DCR C	
	JNZ L3	Give delay
	MVI A,18H	
	OUT PB	Make SOC high
	MVI A,08H	Make SOC Low.
	OUT PB	
	MVI A,00H	
	OUT PB	Make ALE Low
L4 :	IN PC	
	ANI 01H.	
	JZ L4.	Wait for EOC.
	IN PA	
	RET.	



Flowchart 1

Flowchart 2

Ex. 16.11.2 : With a neat labelled diagram show how six – 7 segment displays can be interfaced to 8085 through 8255. Write a program to display " HELLO" on that display.

Soln. : Fig. P. 16.11.2 shows the multiplexed six-7 segment display connected in the 8085 system using 8255. Port A and Port B are used as simple latched output ports. Port A provides the segment data inputs to the display and Port B provides a means of selecting a display position at a time for multiplexing the displays.

- $A_0 - A_7$ lines are used to decode the addresses for 8255.
  PA = 00 H PB = 01 H
  PC = 02 H CWR = 03 H
- The values of the registers are chosen such that the segment current is 80 mA. This current is required to produce an average of 10 mA per segment as the displays are multiplexed.
- Only one digit is displayed at a time.

The control word for 8255 is

BSR	Mode A		P_A	P_{CU}	Mode B	P_B	P_{CL}
1	0	0	0	×	0	0	×

= 80 H

Program to initialise 8255 :

Instruction	Comment
MVI A, 80 H	Load control word in register
OUT CR	Load control word in control word register

Subroutine to display message "HELLO" on the six 7 segment display. The subroutine is stored in the memory from address C600 H. It must be called continuously to display the 7-segment coded message.

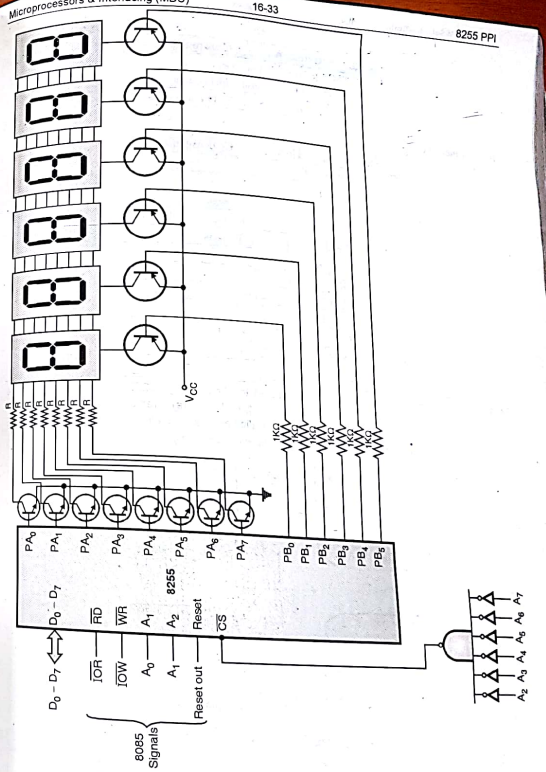


Fig. P. 16.11.2 : Interfacing of six 7 segment display using 8255

Setting up registers for display

Label	Instruction	Comments
	MVI B, 06H	Load count
	MVI C,7FH	Load select pattern
	LXI H, C600H	Starting address of message

Display message

Label	Instruction	Comments
L1 :	MOV A, C	
	OUT PB	
	MOV A,M	Get message
	OUT PA	Display message "HELLO"
	CALL DELAY	Wait for sometime
	MOV A, C	
	RRC	
	MOV C, A	adjust selection pattern
	INX H	
	DCR B	Decrement count.
	JNZ L1	Repeat
	RET	

Ex. 16.11.3 : A traffic controller is to be interfaced to 8085 through 8255. The traffic signals are located on cross roads as shown in Fig. P. 16.11.3(a). Traffic is allowed from North to South or East to West only. No turns allowed. Traffic is allowed in one direction for 55 seconds and yellow to red transition time is 5 seconds.

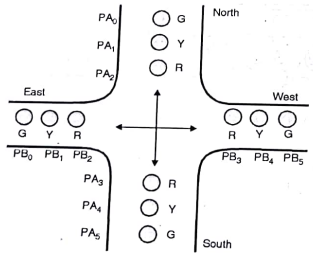


Fig. P. 16.11.3(a)

Soln. :

Step 1 : Traffic signals use normal bulbs which require 230 V AC as supply. The microcomputer works on 5 V DC so we require a special interfacing arrangement as shown in Fig. P. 16.11.3(b). The interfacing uses a transistorised circuit to drive relay. The relay will make bulb ON or OFF which depends on input to transistorised circuit. When base of Q_1 is logic high, the transistor will be ON and pass current through relay coil, which will make bulb ON. When base of Q_1 is logic low, the transistor will be OFF and will not pass current through relay coil, which will make bulb OFF.

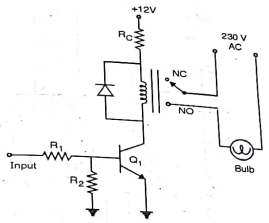


Fig. P. 16.11.3(b) : Bulb interfacing

Step 2 : The 8255 is interfaced to 8085 in I/O mapped I/O. In it, lines A_0, A_1 are directly connected and remaining lines A_2 to A_7 or A_{10} to A_{16} are used for decoder logic to generate chip select signal.

Step 3 : The transistorised circuit is interfaced to 8255 port bits and total interfacing schematic is as shown in Fig. P. 16.11.3(c).

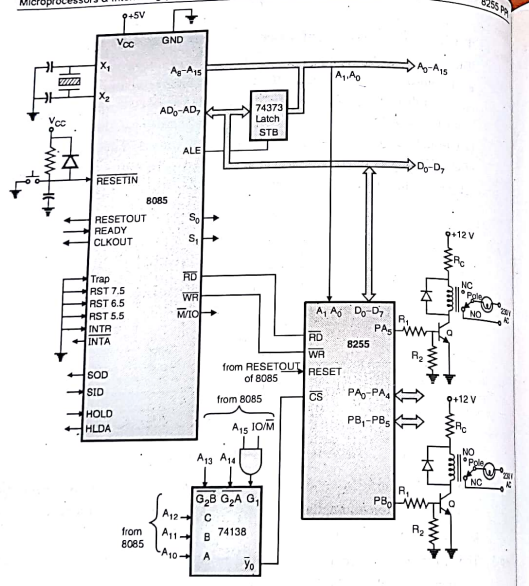


Fig. P. 16.11.3(c) : Traffic controller interface

Step 4 : The addresses of 8255 ports will be as follows :

A_{15}	A_{14}	A_{13}	A_{12}	A_{11}	A_{10}	A_1	A_0		
1	0	0	0	0	0	0	0	= 80 H	Port A
1	0	0	0	0	0	0	1	= 81 H	Port B
1	0	0	0	0	0	1	0	= 82 H	Port C
1	0	0	0	0	0	1	1	= 83 H	CWR

Step 5 : The flowchart required to implement traffic controller steps will be as shown in Fig. P. 16.11.3(d).

Step 6 : The port A connected to NS traffic, port B connected to EW traffic. There are 4 possible combinations of time slots as follows :

Time slot	Time	NS	EW
T_1	55 sec	Green	Red
T_2	5 sec	Yellow	Red
T_3	55 sec	Red	Green
T_4	5 sec	Red	Yellow

The data required to achieve the above combination of time slots will be as follows :

(1) **T_1 slot :**

ON NS $PA_0 = 1$, $PA_1 = 0$, $PA_2 = 0$, $PA_3 = 0$, $PA_4 = 0$ $PA_5 = 1$,
OFF EW $PB_0 = 0$, $PB_1 = 0$, $PB_2 = 1$, $PB_3 = 1$, $PB_4 = 0$, $PB_5 = 0$

NS	X	X	1	0	0	0	0	1	= 21 H
EW	X	X	0	0	1	1	0	0	= 0C H

(2) **T_2 slot :**

NS	X	X	0	1	0	0	1	0	= 12 H
EW OFF	X	X	0	0	1	1	0	0	= 0C H

(3) **T_3 slot :**

NS OFF	X	X	0	0	1	1	0	0	= 0C H
EW ON	X	X	1	0	0	0	0	1	= 21 H

(4) **T_4 slot :**

NS OFF	X	X	0	0	1	1	0	0	= 0C H
EW	X	X	0	1	0	0	1	0	= 12 H

Step 7 : Program :

Label	Instructions	Operation
	LXI SP, FFFFH	FFFF → SP
	MVI A, 80H	80 → A
	OUT 63H	A → CWR
UP :	MVI A, 21H	21 → A
	OUT 60H	A → Port A
	MVI, 0CH	0C → A
	OUT 61H	A → Port B
	CALL DELAY1	Wait for 55 Sec
	MVI A, 12H	12 → A
	OUT 60H	A → Port A
	MVI A, 0CH	0C → A
	OUT 61H	A → Port B
	CALL DELAY2	Wait for 5 sec.
	MVI A, 0CH	0C → A
	OUT 60H	A → Port B
	MVI A, 21H	21 → A
	OUT 61H	A → Port B
	CALL DELAY1	Wait for 55 sec

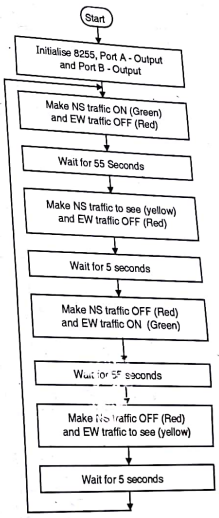


Fig. P. 16.11.3(d)

Label	Instructions	Operation
	MVI A, 0CH	0C → A
	OUT 60H	A → Port A
	MVI A, 12H	12 → A
	OUT 61H	A → Port B
	CALL DELAY2	Wait for 5 sec.
	JMP UP	

Ex. 16.11.4 : Stepper Motor Interfacing :

Lab assignment

Interfacing a stepper motor :

Interface a stepper motor to 8085 using 8255. Interface 8255 in I/O mapped I/O.

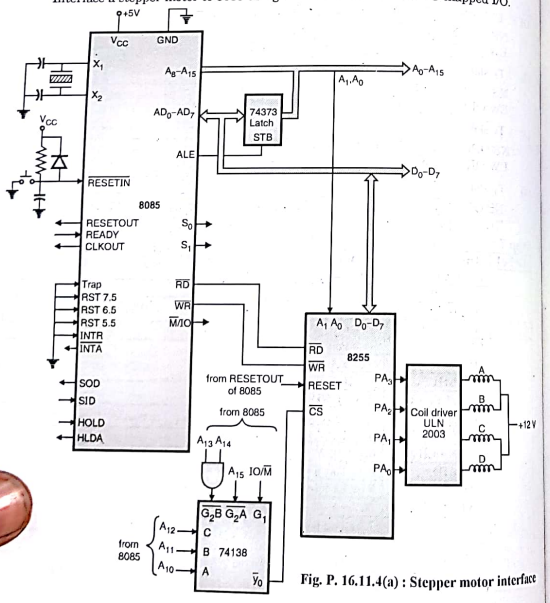


Fig. P. 16.11.4(a) : Stepper motor interface

Step 1 : The 8255 is interfaced to 8085 in I/O mapped I/O. The port A is used to give steps to stepper motor. So function of port A will be output. No other signals except 4 step signals are required.

Step 2 : The 8255 provides very less current which will not be able to drive stepper motor coils so a coil driver ULN 2003 is generally used or a transistorised driver can be used instead of ULN 2003.

Step 3 : The interfacing diagram of stepper motor using 8255 will be as shown in Fig. P. 16.11.4(a).

Step 4 : There are two possible step modes :

Mode 1 full stepping L :

In this case the motor moves by, to do this 2 bits are changed at a time, the bit patterns will be as follows :

A	B	C	D		
1	0	1	0	=	0A
1	0	0	1	=	09
0	1	0	1	=	05
0	1	1	0	=	06

Reverse

Forward

Mode 2 half stepping :

In this case the motor moves by, to do this 1 bit is changed at a time, the bit patterns will be as follows :

A	B	C	D		
1	0	1	0	=	0A
1	0	0	0	=	08
1	0	0	1	=	09
0	0	0	1	=	01
0	1	0	1	=	05
0	1	0	0	=	04
0	1	1	0	=	06
0	0	1	0	=	02
1	0	1	0	=	0A

Reverse

Forward

Step 5 : The data is stored in memory and accessed using memory pointer. Depending on sequence in which data is stored, the motor will move in forward or reverse direction.

Step 6 : To prepare a program for forward direction in full stepping mode, the flowchart required will be as shown in Fig. P. 16.11.4(b).

Step 7 : Program for full stepping mode,

Label	Instructions	Operation
	LXI SP, FFFFH	FFFF → SP
	MVI A, 80H	80 → A
	OUT CWR	A → CWR
BACK :	LXI H, C200H	C200 → HL
	MVI C, 04H	04 → C
UP :	MOV A, M	(HL) → A
	OUT P_A	A → P_A
	CALL DELAY	Wait for delay
	INX H	HL + 1 → HL
	DCR C	C – 1 → C
	JNZ UP	Is C = 0,
		if no go to UP
	JMP BACK	Go BACK

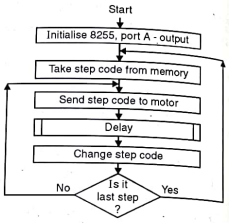


Fig. P. 16.11.4(b)

Data stored in memory for forward direction in full stepping mode will be as follows :

Address	Data
C200	0A
C201	09
C202	05
C203	06

The forward direction in full stepping mode can be easily changed to reverse direction by changing the sequence of data stored in memory as follows :

Address	Data
C200	06
C201	05
C202	09
C203	0A

If you are not interested in changing data stored in memory just make few changes in program. The changes to be done are :

(i) LXI H, C203 instead of LXI H, C200

(ii) DCX H instead of INX H.

Step 8 : To prepare a program for forward direction in half stepping mode the flowchart required will be same as shown in Fig. P. 16.11.4(b).

Step 9 : Program for half stepping mode,

Label	Instructions	Operation
	LXI SP, FFFFH	FFFF → SP
	MVI A, 80H	80 → A
	OUT CWR	A → CWR
BACK :	LXI H, C200H	C200 → HL
	MVI C, 08H	08 → C
UP :	MOV A, M	(HL) → A
	OUT PA	A → PA
	CALL DELAY	Wait for delay
	INX H	HL + 1 → HL
	DCR C	C –1 → C
	JNZ UP	Is C = 0, is no go to UP
	JMP BACK	Go BACK

Data stored in memory for forward direction in half stepping mode will be as follows :

Address	Data
C200	0A
C201	08
C202	09
C203	01
C204	05
C205	04
C206	06

□□□

Note

CHAPTER 17

8237 DMA Controller

17.1 Introduction

Q. Explain about any DMA Controller.

In I/O data transfer, data is transferred, by using microprocessor. The microprocessor will read data from I/O device and then will write data to memory. In this case, there are two operations for single data transfer. If the ***data*** is less, then microprocessor will not waste its time; transferring data from I/O to memory or back. But suppose, ***data*** is huge, then the transfer rate from I/O to memory or back will slow down because of microprocessor intervention. In such case, to speed up the process of transferring the data, we can think, ***Can I/O have direct access to memory ?***; and the answer is, yes. It can have ***Direct memory access (DMA)***, but under ***Supervision***. The device which supervises, data transfer is named as ***DMA controller***. Now let's have diagrammatic representation of the scheme, which depicts microprocessor, DMA controller, memory and I/O device.

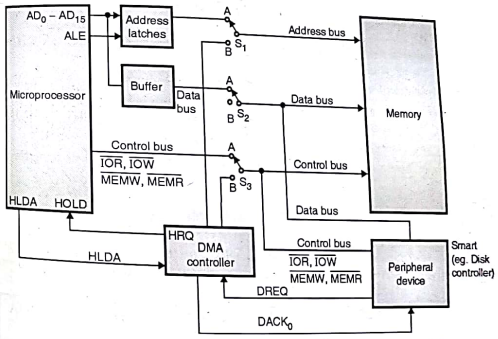


Fig. 17.1.1 : DMA controller scheme

Let's understand the concept clearly.

1. Initially, switches S_1, S_2 and S_3 are at position A.
2. No direct access to memory by I/O.
3. Microprocessor is MASTER of all three buses; address, data and control.
4. Microprocessor treats **DMA controller**, as I/O device ONLY.
5. Using IN/OUT instruction, you can program **DMA controller** chip, for various modes.
6. Whenever, peripheral device is ready to transfer data, DIRECTLY to memory, it will generate REQUEST (DRQ → DMA REQUEST), to DMA controller, asking for direct access.
7. In response to DRQ, DMA controller will activate, HRQ (HOLD request); connected to HOLD pin of microprocessor. By activating HOLD line, DMA controller request microprocessor, to HOLD for sometime and allow him to become a master of all three buses.
8. Moment HOLD pin is HIGH, microprocessor will complete the present job and also activate HLDA (HOLD acknowledge) signal; informing, DMA controller to become master.
9. Microprocessor tristates, all its buses, so total cutoff from memory and I/O device. Thus microprocessor relinquishes the buses and provides control to DMA controller.
10. Now DMA controller is master. It will position all three switches to position B.
11. DMA controller will also generate $\overline{DACK}$ (DMA acknowledge) signal to peripheral device; informing that, direct access is allowed.
12. Now it will generate address and control signal. Data will flow from memory to I/O or vice versa.
13. After completing data transfer, DMA controller will deactivate HOLD line. It also positions, switches back to position A.
14. Now microprocessor will regain the control over the three buses.
15. Microprocessor will start executing instructions from main program. Till DMA is inactive or not master of the bus, is referred as **DMA IDLE Cycle**. When DMA controller gains the control, it is referred as **DMA Active Cycle**.

This is about basics of DMA. Now let's study the different methods of transferring data.

17.2 DMA Controller : Data Transfer Modes

The DMA controller functions as a bus master and bus slave. It performs data transfer operations. DMA controlled input/output is further divided into the following categories :

- Burst or Block transfer DMA.
- Cycle steal or Single byte transfer DMA.
- Transparent or Hidden DMA.

17.2.1 Burst or Block Transfer DMA

- It is the fastest DMA mode.
- In this mode, two or more data bytes are transferred continuously.
- The microprocessor is disconnected from the system bus during DMA transfer i.e. the microprocessor cannot executes its own program during this transfer.
- N number of DMA cycles are added into the machine cycles of the microprocessor where N is number of bytes to be transferred.
- In this mode, the **DMA** controller sends ***'HOLD'*** signal to the microprocessor and waits for HLDA signal.
- After receiving HLDA signal, the DMA controller gains control of the system bus and executes a DMA cycle to transfer one byte.
- After transferring one byte, it increments memory address, decrements counter and transfers next byte.
- In this way, it transfers all data bytes between memory and I/O devices. After transferring all data bytes, the DMA controller disables ***'HOLD'*** signal and enters into slave mode.

17.2.2 Cycle Steal or Single Byte Transfer DMA

- In cycle steal transfer only one byte of data is transferred at a time.
- This type of DMA is slower than burst DMA.
- In this mode, only one DMA cycle is added between two machine cycles of the microprocessor, hence the instruction execution speed of the microprocessor is reduced slightly.
- In this mode the DMA controller sends ***'HOLD'*** signal to the microprocessor and waits for HLDA signal.
- After receiving HLDA signal, the DMA controller gains control of the system bus and executes only one DMA cycle.
- After transferring one byte, it disables ***'HOLD'*** signal and enters into slave mode.
- The microprocessor then gains control of the system bus and executes next machine cycle. If the count is not zero and next data is available then the DMA controller sends ***'HOLD'*** signal to the microprocessor and transfers next byte of data block.

17.2.3 Transparent or Hidden DMA Transfer

- The microprocessor executes some states during which it floats the address and data buses.
- During these states, the microprocessor is isolated from the system bus.
- The DMA controller transfers data between memory and I/O devices during these states. This operation is transparent to microprocessor.

- This is the slowest DMA transfer, In this mode, the instruction execution speed of microprocessor is not reduced. But, the transparent DMA requires logic to detect the states when the microprocessor is floating the buses.

Now, we will study DMA controller chip 8237.

17.3 8237 High Performance Programmable DMA Controller

17.3.1 Features

- It provides various modes of DMA.
- It provides on chip four independent DMA channels. The number of channels can be increased by cascading DMA controller chips.
- Each channel can be used in auto initialization mode.
- It can transfer data between two memory blocks in DMA mode i.e. memory to memory transfer.
- In memory to memory transfer a single word can be written into all location of memory block.
- The address of memory is either incremented or decremented after each DMA cycle depending upon the mode.
- The clock frequency is 3 MHz (8237) or 5 MHz (8237-2).
- The data transfer rate is very high e.g. 1.6 M bytes/second for 8237-2 at 5 MHz.
- Directly expandable to any number of channels. It doesn't require any additional chip for cascading. There is no limitations on cascading.
- It provides EOP line that is used to terminate DMA operation. This signal can be generated by external hardware.
- The DMA can be requested by setting an appropriate bit of request register.
- Independent control for DREQ and DACK signal. DREQ and DACK signals can be initialized either for active high or active low.
- It provides compressed timings to improve throughput of the system. It can compress the transfer time to two cycles (2S).

17.4 Pin Configuration of 8237

Q. 1 Draw and explain pin configuration of 8237.

Q. 2 Explain the significance of AEN, ADSTB, $\overline{EOP}$, READY pins of 8237.

The Fig. 17.4.1 shows pin configuration of 8237.

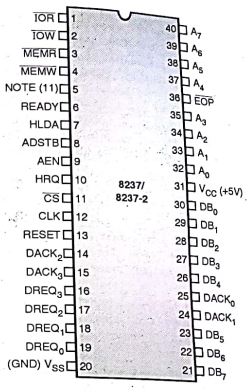


Fig. 17.4.1 : Pin configuration of 8237

Symbol	Description
CLK	This is a clock input line ignored in slave mode. In master mode, this signal controls all internal and external DMA operations. The data transfer rate depends upon the frequency of this signal.
$\overline{CS}$	In slave mode, this signal is generated by address decoder to select 8237 chip for communication between CPU and 8237. In master mode, this signal is ignored.
Reset	It is an asynchronous input line. This signal clears the command, status, request and temporary register and forces 8237 into slave mode.
READY	In master mode, this signal is used to add wait states into a DMA cycle.
HRQ	It is a hold request output line. It is connected to hold input of the CPU. It is used to request control of the system bus.
HLDA	It is a hold acknowledge input line. This signal is generated by CPU. In response to this signal, the 8237 gains control of the system bus and enters into master mode.
$\overline{IOR}$	It is an active low bi-directional tristate line. In slave mode, it acts as an input line and used to read contents of 8237 registers. In

Symbol	Description
	master mode, it acts as an output line. This signal is generate during DMA cycle to read data from I/O device.
$\overline{IOW}$	It is an active low bi-directional line. In slave mode, it acts as input line and used by CPU to write contents to 8237 registers. master mode, it acts as an output line. This signal is generate during DMA read cycle to write data into I/O device.
A_0 - A_3	These are bi-directional, address lines. In slave mode, these lin act as input lines, used to select one of the registers of 8237. I master mode, the 8237 provides lower bits of memory address these lines.
A_4 - A_7	These are tristate address output lines. These lines are tristated slave mode. In master mode, the 8237 transfers bits of memor address on these lines.
$\overline{MEMR}$	It is an active low tristate control output line. It is tristated in slav mode. In master mode, this signal is generated during DMA rea cycle or during memory to memory transfer cycle to read contents o source memory.
$\overline{MEMW}$	It is an active low tristate output line. It is tristated in slave mode In master mode, this signal is activated during DMA write c during memory to memory transfer cycle to write data into destination memory.
DB_0 - DB_7	These are bi-directional tristate buffered data lines. In slave mode these lines are used to transfer data between CPU and 8237 registers. In master mode, these lines act as address output lines The 8237 places higher byte of address on these lines during DMA cycles.
AEN	This active high output enables the 8 bit latch that drives the upper 8 bit address bus. The AEN pin is used to disable other bus drivers during DMA transfers.
ADSTB (Address Strobe)	This output line is used to strobe the upper address byte generated by 8237 in master mode into an external latch.
$DREQ_0$ -$DREQ_3$	These are asynchronous DMA channel request lines used by peripheral. The polarity of each signal is programmable i.e. these lines can be used as either active high or active low input. DREQ must be maintained until the corresponding DACK is activated.
$DACK_0$ - $DACK_3$	These are DMA acknowledge output lines. The polarity of each line is programmable. This signal indicates that the requesting peripheral has been granted for DMA cycle.
$\overline{EOP}$	End of Process : It is an active low bi-directional signal. This line is also used to terminate DMA cycle. The DMA cycle can be terminated by pulling $\overline{EOP}$ input low. The 8237 also generates EOP pulse, when the terminal count for any channel is reached.

17.5 Block Diagram of 8237

Q. Draw and explain the block diagram of 8237 DMA.

The Fig. 17.5.1 shows an internal block diagram of 8237. It consists of logic blocks and internal register.

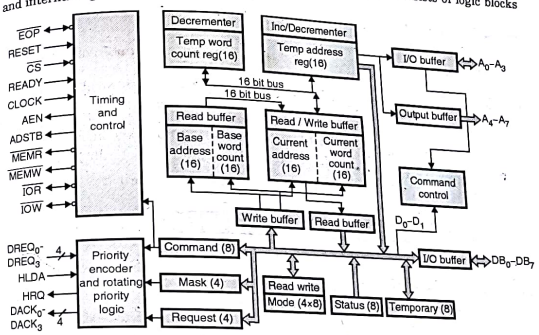


Fig. 17.5.1 : Internal block diagram of 8237

8237 Registers

It contains 344 bits of internal memory in the form of registers. It contains 12 types of registers.

- Current address register
- Base address register
- Command register
- Request register
- Status register
- Current word register
- Base count register
- Mode register
- Mask register
- Temporary register

(1) Current address register

- Each channel has a 16 bit current address register.
- This register holds the address of memory location to be accessed during current DMA cycle. The address stored in this register is automatically incremented or decremented after each transfer.
- It is a read and write register.
- It is divided into two parts : lower byte and higher byte.

- In autoinitialization mode, it is initialized automatically with original address after $\overline{EOP}$ signal. The current address register format is as shown in Fig. 17.5.2.

A_{15}	A_{14}	A_{13}	A_{12}	A_{11}	A_{10}	A_9	A_8	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0

Fig. 17.5.2 : Current address register

(2) Current word register

- Each channel has a 16 bit current word count register.
- The original value stored in this register indicates the number of bytes to be transferred.
- The word count is decremented after each transfer. The current count indicates the numbers of pending transfers.
- When the count value in the register goes to zero, a TC will be generated. It is a read and write register. It is also divided into two parts : lower byte and higher byte.
- In autoinitialization mode, this register is initialized automatically with original count value after $\overline{EOP}$ signal. The current word register format is as shown in Fig. 17.5.3.

W_{15}	W_{14}	W_{13}	W_{12}	W_{11}	W_{10}	W_9	W_8	W_7	W_6	W_5	W_4	W_3	W_2	W_1	W_0

Fig. 17.5.3 : Current word register

(3) Base address register

- Each channel has a 16 bit base address register.
- It is a write only register.
- It holds original value of address during all DMA transfers i.e. the contents of this register is not updated during DMA transfers.
- When $\overline{EOP}$ is activated, the 8237 transfers contents of base address registers into current address register in autoinitialization mode.
- This register is written along with current address register during initialization. Format is same as current address register.

(4) Base count register

- Each channel has a 16 bit base word count register.
- It is write only register.
- It holds original count value during all DMA cycle i.e. the contents of this register are not updated during DMA transfers.
- When $\overline{EOP}$ is activated, the 8237 transfers contents of this register into current word register in autoinitialization mode.
- This register is written along with current address register during initialization. Format is same as current word register.

(5) Command register

- It is an 8 bit control register. It is a write only register.
- This register is cleared by reset signal. It is used to initialize operation modes of 8237.
- The format of command register is as shown in Fig. 17.5.4.

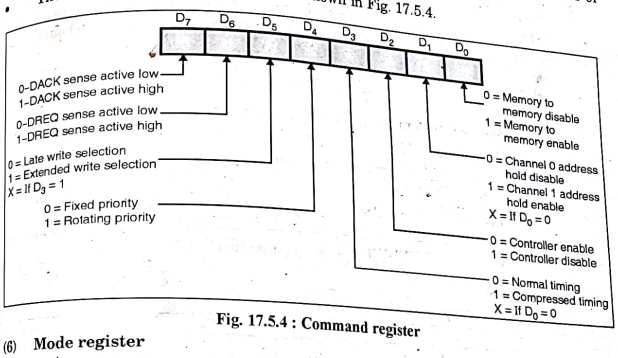


Fig. 17.5.4 : Command register

(6) Mode register

- It is an 8 bit write only register.
- It is used to set operating modes of 8237.
- Each channel has a 6 bit mode register.
- All registers are cleared by reset signal. The format of mode register is as shown in Fig. 17.5.5.

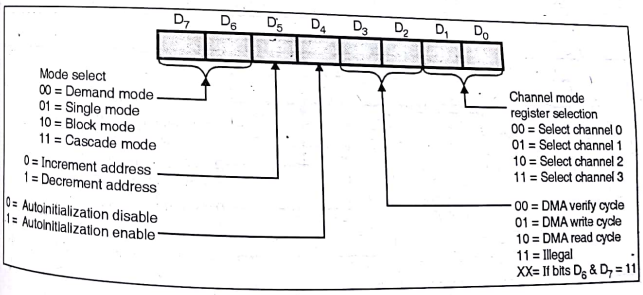


Fig. 17.5.5 : Mode register

(7) Request register

- It is an 8 bit write only register.

- It is used to request DMA through software. Each channel has a request b associated with it in the 4 bit request register.
- Each register bit is set or reset separately under software control. The request automatically cleared after TC. It is also cleared by external $\overline{EOC}$ signal.
- This register is cleared by reset signal.
- Software request will be serviced only if the channel is in block mode.
- In memory to memory transfer, the software request for channel 0 should be se The format of request register is as shown in Fig. 17.5.6.

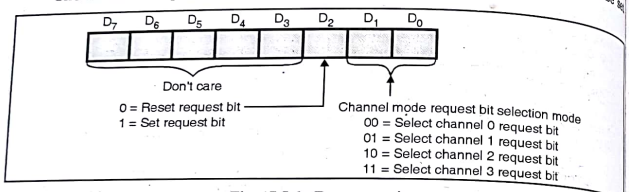


Fig. 17.5.6 : Request register

(8) Mask register

- It is an 8 bit write only register.

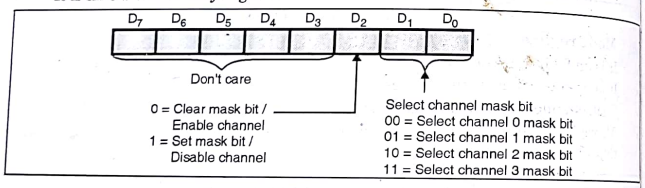


Fig. 17.5.7 : Byte written mask register

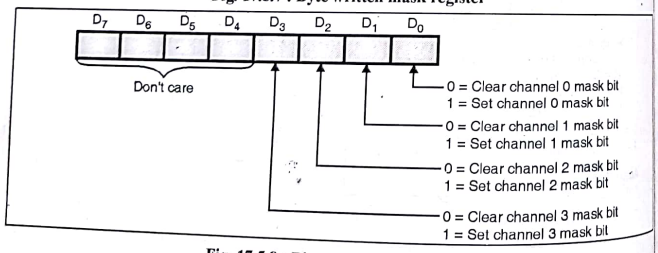


Fig. 17.5.8 : Bit written mask register

- It is used to maskout the channel DMA request.

- In normal mode, mask bit is set automatically after TC. It is not affected in autoinitialization mode.
- The reset signal sets mask bit of all channels i.e. it disables all channels.
- The channel mask bit is selected by D_1 and D_0 bits of the mask format as shown in Fig. 17.5.7. Each bit of the 4 bit mask register can be set or reset separately under software control.
- All 4 bits of the mask register may also be written with a single command as shown in Fig. 17.5.8.

(9) Status register

- It is an 8 bit read only register.
- It indicates which channels have reached a terminal count and which channels have pending DMA requests.
- Bits 0-3 are set every time a TC is reached by that channel or external EOP signal is applied.
- These bits are cleared automatically on reading the status register and upon reset signal.
- Bits 4-7 are set whenever their corresponding channel is requesting service. The format of status register is as shown in Fig. 17.5.9.

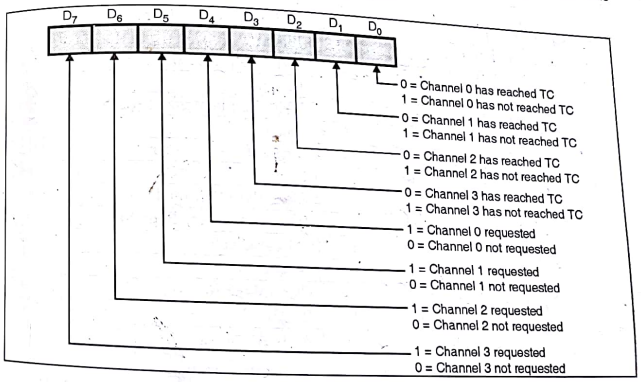


Fig. 17.5.9 : Status register

(10) Temporary register

- This register is used to hold data during memory to memory transfer.
- It is an 8 bit read only register.
- The microprocessor can read last byte of memory to memory transfer.
- It is cleared by reset signal.

(11) Software commands

- These are additional special write only software commands.
- These commands do not require a specific data pattern. i.e. writing only data into corresponding address enables such commands.

(12) Clear first / last flip-flop

- This command is used to reset internal first / last flip-flop.
- After issuing this command the microprocessor can access lower byte of any 16 bit register. The F/L flip-flop is toggled after each read or write 16 bit register.

(13) Master clear

- It is a software reset command.
- It clear command, status, request, temporary registers and F/L flip-flop.
- It sets mask register and forces 8237 in slave mode.
- The Tables 17.5.1 and 17.5.2 shows addresses of all registers.

Table 17.5.1 : Software command codes

Signals						Operation
A_3	A_2	A_1	A_0	$\overline{IOR}$	$\overline{IOW}$	
1	0	0	0	0	1	Read Status Register
1	0	0	0	1	0	Write Command Register
1	0	0	1	0	1	Illegal
1	0	0	1	1	0	Write Request Register
1	0	1	0	0	1	Illegal
1	0	1	0	1	0	Write Single Mask Register Bit
1	0	1	1	0	1	Illegal
1	0	1	1	1	0	Write Mode Register
1	1	0	0	0	1	Illegal
1	1	0	0	1	0	Clear Byte Pointer Flip-Flop
1	1	0	1	0	1	Read Temporary Register
1	1	0	1	1	0	Master Clear
1	1	1	0	0	1	Illegal
1	1	1	0	1	0	Clear Mask Register
1	1	1	1	0	1	Illegal
1	1	1	1	1	0	Write All Mask Register Bits

Table 17.5.2 : Word count and address register command codes

Channel	Register	Operation	Signals							Internal Flip-Flop	Data Bus $DB_0 - DB_7$
			$\overline{CS}$	$\overline{IOR}$	$\overline{IOW}$	A_3	A_2	A_1	A_0		
0	Base and Current Address	Write	0	1	0	0	0	0	0	0	$A_0 - A_7$
			0	1	0	0	0	0	0	1	$A_8 - A_{15}$
	Current Address	Read	0	0	1	0	0	0	0	0	$A_0 - A_7$
			0	0	1	0	0	0	0	1	$A_8 - A_{15}$

Channel	Register	Operation	Signals							Internal Flip-Flop	Data Bus $DB_0 - DB_7$
			$\overline{CS}$	$\overline{IOR}$	$\overline{IOW}$	A_3	A_2	A_1	A_0		
	Base and Current Word Count	Write	0	1	0	0	0	0	1	0	$W_0 - W_7$
			0	1	0	0	0	0	1	1	$W_8 - W_{15}$
	Current Word Count	Read	0	0	1	0	0	0	1	0	$W_0 - W_7$
			0	0	1	0	0	0	1	1	$W_8 - W_{15}$
1	Base and Current Address	Write	0	1	0	0	0	1	0	0	$A_0 - A_7$
			0	1	0	0	0	1	0	1	$A_8 - A_{15}$
	Current Address	Read	0	0	1	0	0	1	0	0	$A_0 - A_7$
			0	0	1	0	0	1	0	1	$A_8 - A_{15}$
	Base and Current Word Count	Write	0	1	0	0	0	1	1	0	$W_0 - W_7$
			0	1	0	0	0	1	1	1	$W_8 - W_{15}$
	Current Word Count	Read	0	0	1	0	0	1	1	0	$W_0 - W_7$
			0	0	1	0	0	1	1	1	$W_8 - W_{15}$
2	Base and Current Address	Write	0	1	0	0	0	1	0	0	$A_0 - A_7$
			0	1	0	0	0	1	0	1	$A_8 - A_{15}$
	Current Address	Read	0	0	1	0	0	1	0	0	$A_0 - A_7$
			0	0	1	0	0	1	0	1	$A_8 - A_{15}$
	Base and Current Word Count	Write	0	1	0	0	0	1	1	0	$W_0 - W_7$
			0	1	0	0	0	1	1	1	$W_8 - W_{15}$
	Current Word Count	Read	0	0	1	0	0	1	1	0	$W_0 - W_7$
			0	0	1	0	0	1	1	1	$W_8 - W_{15}$
3	Base and Current Address	Write	0	1	0	0	0	1	1	0	$A_0 - A_7$
			0	1	0	0	0	1	1	1	$A_8 - A_{15}$
	Current Address	Read	0	0	1	0	0	1	1	0	$A_0 - A_7$
			0	0	1	0	0	1	1	1	$A_8 - A_{15}$
	Base and Current Word Count	Write	0	1	0	0	0	1	1	0	$W_0 - W_7$
			0	1	0	0	0	1	1	1	$W_8 - W_{15}$
	Current Word Count	Read	0	0	1	0	0	1	1	0	$W_0 - W_7$
			0	0	1	0	0	1	1	1	$W_8 - W_{15}$

17.6 DMA Data Transfer Modes

Q. Explain different data transfer modes of 8237.

The 8237 can be operated in four DMA modes viz. : single transfer mode, block transfer mode, demand transfer mode and cascading mode.

- Single transfer mode
- Burst or block transfer mode
- Demand transfer mode
- Cascade mode

17.6.1 Single Transfer Mode

- In this mode, the 8237 transfers only one byte.
- It decrements word count register and increments or decrements address register after each transfer.
- After transferring one byte (even though count is not zero) the 8237 disables HRQ and enters into idle state or slave mode. It is also called as cycle steal mode.
- When TC is reached, it disables corresponding channel.
- Microprocessor machine bytes are added between two successive DMA cycles of 8237.
- In 8237, the single byte transfer is achieved by clearing count register. But count register should be cleared after each transfer.
- In 8237, this mode is set by appropriate bits of mode register.
- In autoinitialization, the channel is reinitialized after TC.

17.6.2 Burst or Block Transfer Mode

- In this mode, all bytes are transferred continuously.
- After each transfer, it decrements count register and increments or decrements address register.
- It maintains HRQ high during all DMA cycles.
- It transfers data bytes until a terminal count is reached or $\overline{EOP}$ is activated.
- In autoinitialization mode, the channel is reinitialized after TC. DREQ needs to be held active until DACK becomes active.

17.6.3 Demand Transfer Mode

- In this mode, the number of bytes to be transferred is controlled by I/O device.
- The I/O device can discontinue the DMA operation by activating $\overline{EOP}$ signal or disabled DREQ signal.
- When DREQ signal is disabled, the 8237 stores intermediate values of count and address in current count and address registers respectively.

- In autoinitialization mode, the $\overline{EOP}$ signal reinitializes the current channel. Hence it is called demand transfer mode.

17.6.4 Cascade Mode

- This mode is used to cascade more than one 8237 together to increase the number of channels.
- The HRQ and HLDA lines of level 2 8237 are connected to DREQ and DACK lines of level 1 8237 respectively. The Fig. 17.6.1 shows cascading of 8237.
- There is no limitation of number of cascading levels. In the case, channel 0 and channel 1 of level 1 8237 should be operated in cascaded mode.
- In cascaded mode the 8237 activates only HRQ and DACK.

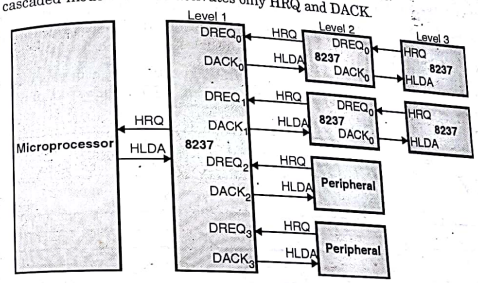


Fig. 17.6.1 : Cascaded 8237's

17.7 Types of Transfer

Q. Explain different transfer modes of 8237.

The 8237 provides two basic types of transfers peripheral transfers and memory to memory transfer.

- Peripheral transfer
- Memory to memory transfer

17.7.1 Peripheral Transfers

- There are three types of peripheral transfer DMA read, DMA write and DMA verify.

17.7.2 Memory to Memory Transfer

Q. Explain memory to memory transfer technique of 8237 with timing diagram.

- In this transfer, the 8237 transfers data byte from source memory location to destination memory location by using channel 0 and channel 1.

- The address of source memory location is specified by channel 0 address register while the address of destination memory location is specified by channel 1 address register.
- In memory to memory transfer, the 8237 operates in block transfer mode.
- The signals AEN, DACK, $\overline{IOR}$ and $\overline{IOW}$ are not activated.
- ADSTB signal is used to latch higher byte of address.
- The memory to memory transfer is initialized by setting the software DREQ for channel 0.
- To transfer one byte, the 8237 executes two (4S) cycles.
- During 1st cycle it reads a byte from memory and store this byte into temporary register. Then it increments or decrements channel 0 address register.
- During 2nd cycle, it writes stored byte into destination memory location. It increments or decrements channel 1 address register. It also decrements channel 1 count register. When this count is zero, it disables memory to memory transfer. It also activates $\overline{EOP}$ signal. When external $\overline{EOP}$ is activated in memory to memory transfer the 8237 terminates this transfer. The Fig. 17.7.1 shows a timing diagram of memory to memory transfer.

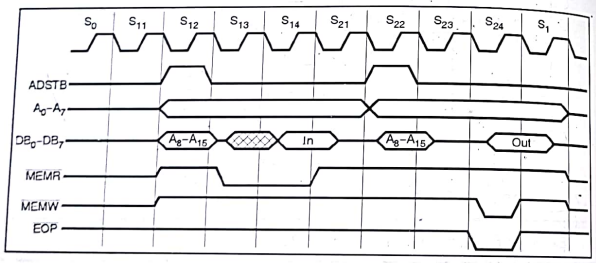


Fig. 17.7.1 : Memory to memory transfer timing diagram

17.8 8237 Operating Modes

Q. 1 What are the different operating modes of 8237 ?
Q. 2 Give the timing diagram of DMA operation.
Q. 3 Explain different DMA operation (cycles).

- Autoinitialization mode
- Normal mode
- Compressed timing
- Priority modes
- Extended write mode

(1) **Autoinitialization mode**

- In this mode, the original values of the current address and current word count registers are automatically restored from the base address and base word count registers of that channel following $\overline{EOP}$.
- The mask bit is not set when the channel is in autoinitialization.

(2) **Priority modes**

- The 8237 provides two priority modes viz. fixed priority mode and rotating priority mode.
- These modes are similar to 8257 priority modes.

(3) **Normal mode**

- In this mode, $\overline{Read}$ ($\overline{IOR}$ and $\overline{MEMR}$) pulse is activated during S_3 and S_4 and write pulse is activated during S_4 as shown in Fig. 17.8.1.

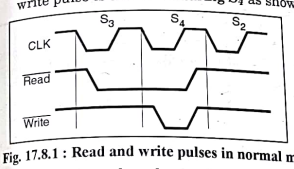


Fig. 17.8.1 : Read and write pulses in normal mode

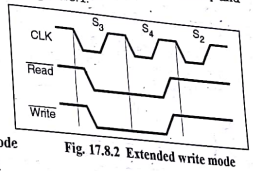


Fig. 17.8.2 Extended write mode

- The minimum length of DMA cycle is 3S.

(4) **Extended write mode**

- In this mode, write ($\overline{IOW}$ and $\overline{MEMW}$) and read ($\overline{IOR}$ and $\overline{MEMR}$) pulses are activated during S_3 and S_4 as shown in Fig. 17.8.2.
- The minimum length of DMA cycle is 3S. But one clock cycle (S_3) is wasted.

(5) **Compressed timing**

- In order to achieve even greater throughput the 8237 can compress the transfer time to two clock cycles.
- In extended write mode, the state S_3 is wasted.
- In compressed time, the 8237 eliminates state S_3. Hence 8237 execute only S_2 and S_4 as shown in Fig. 17.8.3.
- The minimum length of DMA cycle is 2S.

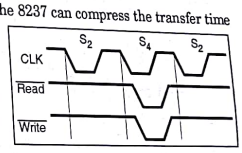


Fig. 17.8.3 : Compressed timing

17.9 8237 Address Generation

Address generation

- The 8237 places lower byte of memory address on A_0 - A_7 lines and higher byte on D_0 - D_7 lines. It places higher address byte during S_1 of a DMA cycle.
- In block and demand transfer mode, the address generated will be sequential.

- When the borrow or carry is generated from lower byte of address, the 8237 places new value of higher byte on data bus. i.e. the higher byte will remain same for many transfers. Once the higher byte is latched then the state S_1 is not required. When the higher byte is constant the 8237 executes only $S_2 S_3$ and S_4 states.
- When borrow/carry is generated from lower address byte during decrement/increment address, the 8237 executes S_1 to latch new higher byte. In most cases, the length of DMA cycle will be 3S.

17.10 Interfacing 8237 with 8086 (Minimum Mode)

Q. 1 Explain with neat diagram, how 8237 can be interfaced with 8086 in minimum mode?

Q. 2 Draw interfacing diagram to connect PDMAC 8237 with 8086 in minimum mode.

- Fig. 17.10.1 shows the interfacing of 8237 with 8086.
- Address generated by 8237 is only 16-bit; A_0 - A_3 is directly connected, A_4 - A_7 is also directly connected while A_8 - A_{16} is demultiplexed from DB_0 to DB_7 as shown in the Fig. 17.10.1.

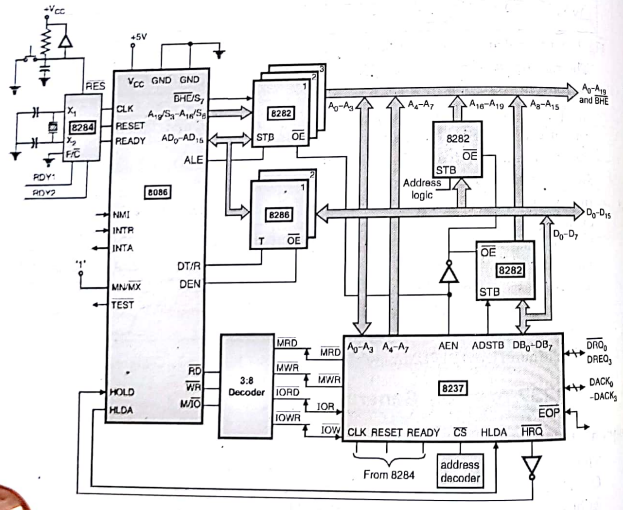


Fig. 17.10.1 : Interfacing of 8086 with 8237

- The upper 4-bit address i.e. A_{16} to A_{19} is provided through a latch. This latch is to be written on by the programmers. This latch can be written with 8086 program, treating the latch as an IO device.
- Another important thing to be noted is $\overline{OE}$ pin of the latches. When 8237 issues the address, corresponding latches are enabled and similarly for 8086 issuing the address. This is controlled by the AEN pin of 8237.

For interfacing 8237 with 8086 in maximum mode, the following circuit is required. This circuit is to be implemented as there is no HOLD and HLDA pin in maximum mode. Instead $\overline{RQ}$ / $\overline{GT}$ is to be used.

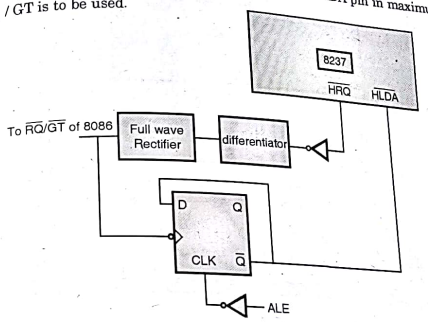


Fig. 17.10.2 : Interfacing 8237 with 8086 in maximum mode

The address of 8237, is A0H. If the device is to be programmed for burst mode (block transfer mode), then mode byte will be

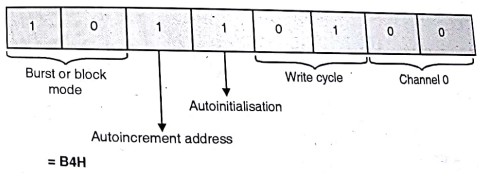


Fig. 17.10.3

Program

```
MOV AL,0B4 H    ; load mode word
MOV DX,0A5 H    ;
OUT DX,AL       ;
```

17.11 Interfacing 8237 with 8085

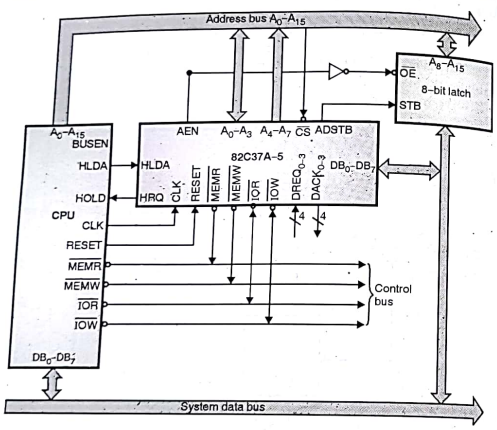


Fig. 17.11.1

The address of 8237, is A0H. If the device is to be programmed for burst mode (block transfer mode), then mode byte will be,

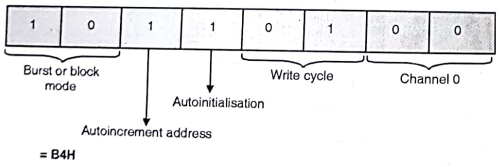


Fig. 17.11.2

Program

```
MVI A, B4H
OUT A0H
```


17.12 Programmable DMA Controller 8257

General features :

(1) It is a 4 channel DMA controller i.e. 4 peripheral input output devices can be interfaced to 8257. Each channel is individually programmable.
(2) It is compatible with Intel microprocessors.
(3) Each channel provides a 16 bit address register and a 14 bit counter, hence each channel can transfer 16 k bytes without microprocessor intervention.
(4) It provides on chip priority resolver that resolves priority of channels in fixed or rotating priority mode.
(5) It provides on chip channel inhibit logic.
(6) It generates a TC signal to indicate the peripheral that the programmed number of data bytes have been transferred.
(7) It generates a MARK signal to indicate the peripheral to which that 128 data bytes have been transferred.
(8) It requires single phase TTL clock. (ϕ_1)
(9) The maximum frequency is 3 MHz and minimum frequency is 250 kHz.
(10) It can be used in **block and cycle steal transfers**.
(11) In response to peripheral DMA request, the 8257 activates HOLD input of the CPU, which then relinquishes hold on the buses. It gains control of the system bus and resolves the priority of channel and executes appropriate DMA cycle for highest priority requesting channel.
(12) It executes three DMA cycles : (i) DMA read (ii) DMA write (iii) DMA verify.
(13) It can be operated in auto load mode.
(14) It provides AEN signal that can be used to isolate CPU and other devices from the system bus.
(15) The external device can terminate DMA operation.

Normally the DMA controller operates in two modes viz : (i) Slave mode (ii) Master mode.

17.12.1 Comparison between Slave and Master Mode

Q. Compare slave and master mode of 8257.

Sr. No.	Slave mode (Normal mode of the system)	Master mode (DMA mode of the system)
(1)	In this mode, microprocessor functions as a bus master and DMA controller functions as a bus slave.	In this mode, the DMA controller functions as a bus master and the microprocessor is disconnected from the system bus.
(2)	The data is transferred through microprocessor, hence two operations are required to transfer one byte between I/O device and a memory. As a result transfer rate is reduced.	The data is not transferred through microprocessor as well as DMA controller. The data is transferred between I/O device and memory directly. Hence only one operation is required to transfer one byte between I/O device and a memory. As a result the transfer rate is increased.

Sr. No.	Slave mode (Normal mode of the system)	Master mode (DMA mode of the system)
(3)	The microprocessor performs arithmetic, logical data transfer and decision making operations, hence data can be processed.	The DMA controller performs only data transfer operation, hence data cannot be processed.
(4)	All address and control signals are generated by microprocessor.	All address and control signals are generated by DMA controller.
(5)	The initialization of DMA controller, DMA and non DMA I/O devices is done by microprocessor.	The DMA controller cannot initialize devices.

17.13 8257 Block Diagram

Q. 1 Draw the Diagram of 8257 DMA controller. Explain the function of each part and control signals used.

Q. 2 Explain command words of 8257.

Q. 3 Explain the architecture, organization and various modes of operation of a programmable DMA controller IC.

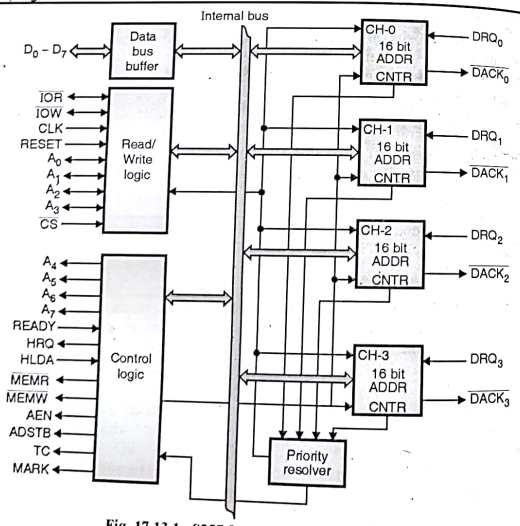


Fig. 17.13.1 : 8257 functional block diagram

The block diagram of 8257 is as shown in Fig. 17.13.1. It contains following blocks :

- Data bus buffer
- Read/write control logic
- Control logic block
- Priority resolver
- DMA channels.

(1) **Data bus buffer :**

- It is a tristate, bi-directional buffer. In slave mode, it transfers data between microprocessor and internal bus.
- The direction of data bus buffer is set by read/write control logic.
- In master mode, it outputs memory address.

(2) **Read/write control logic :**

- In slave mode, it accepts address bits and controls signals from the microprocessor.
- In master mode, it generates address bits and control signals.
- It controls all internal read/write operations.
- It contains F/L flip-flop.

(3) **Control logic block :**

- It contains control logic, mode set register and status register.
- Control logic controls the sequence of DMA operations during all DMA cycles in master mode.
- It generates address and control signals.
- It increments 16 bit address and decrements 14 bit count register.
- It activates a HRQ signal on channel DMA request.
- It is disabled in slave mode.

(a) **Mode set register :**

- It is used to set operating modes of 8257.
- This register must be programmed after initialization of DMA address registers and terminal count registers.
- The format of mode set register is shown in Fig. 17.13.2.

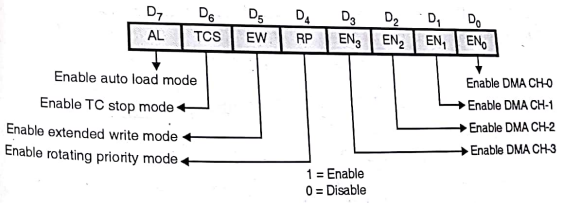


Fig. 17.13.2 : Mode set register

(b) **Status register :**

- It provides the status of DMA channels. The format of status register is as shown in Fig. 17.13.3

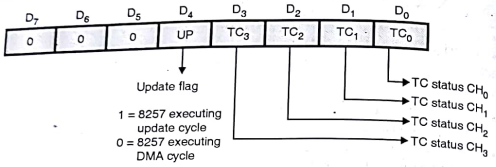


Fig. 17.13.3 : Status register

- The TC status bits are set when the TC signal is activated for that channel.
- TC status bits remain set until the status register is read or the 8257 is reset.
- The update flag is not affected by a status read operation.
- The UP bit is set during update cycle.
- It is cleared automatically after completion of update cycle. The update cycle is used in autoload mode only.

(c) F/L flip-flop (First/last flip-flop) :

- The word length of 8257 is 8 bit, but the address and count registers are 16 bit. $A_0 - A_3$ address lines are used to distinguish between 16 bit registers, but they are not used to distinguish between lower and higher bytes of the 16 bit registers.
- The 8257 provides on chip F/L flip-flop to distinguish between lower and higher bytes of a 16 bit register.
- It is reset by external RESET signal and whenever the mode set register is loaded.
- To access lower byte of a 16 bit register, the F/L flip-flop should be reset and for higher byte it should be set.
- This flip-flop is toggled after each channel register access.

The selection of registers is as shown in Table 17.13.1.

Table 17.13.1

A_3	A_2	A_1	A_0	F/L	Register		
0	0	0	0	0	LSB	CH-0	Address register
0	0	0	0	1	MSB	CH-0	Address register
0	0	0	1	0	LSB	CH-0	Terminal count register
0	0	0	1	1	MSB	CH-0	Terminal count register
0	0	1	0	0	LSB	CH-1	Address register
0	0	1	0	1	MSB	CH-1	Address register
0	0	1	1	0	LSB	CH-1	Terminal count register

A_3	A_2	A_1	A_0	F/L	Register		
0	0	1	1	1	MSB	CH-1	Terminal count register
0	1	0	0	0	LSB	CH-2	Address register
0	1	0	0	1	MSB	CH-2	Address register
0	1	0	1	0	LSB	CH-2	Terminal count register
0	1	0	1	1	MSB	CH-2	Terminal count register
0	1	1	0	0	LSB	CH-3	Address register
0	1	1	0	0	MSB	CH-3	Address register
0	1	1	1	0	LSB	CH-3	Terminal count register
0	1	1	1	1	MSB	CH-3	Terminal count register
1	0	0	0	0	Mode set register (write only)		
1	0	0	0	0	Status register (read only)		

(4) Priority resolver :

- It contains priority resolving logic circuit that resolves the priority of each channel. It can be initialized either in rotating or fixed priority mode.

(5) DMA channels :

- 8257 provides four separate channels. Each channel contains two 16 bit registers.
  - DMA address register
  - Terminal count register.

(1) DMA address register :

Q. What is the function of DMA register and terminal count register in 8257 ? How are their addresses determined ? Explain its different modes of operation.

- It is a 16 bit register. It is used to hold the starting address of memory.
- This register is incremented after each DMA cycle. If the address register is read in the middle of a DMA operation, it provides the address of next memory location.
- The format of address register is as shown in Fig. 17.13.4. A memory address must be programmed before the channel is enabled.

A_{15}	A_{14}	A_{13}	A_{12}	A_{11}	A_{10}	A_9	A_8	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0

Fig. 17.13.4 : DMA address register

(2) Terminal count register :

- It is a 16 bit register divided into two fields, 14 bit count and cycle control bits as shown in Fig. 17.13.5.

- The $TC_0 - TC_{13}$ bits indicate number of (DMA cycles bytes to be transferred – 1. So to transfer N bytes this value should be N – 1. The 14 bit count value is decremented after each DMA cycle. RD and WR bits indicate type of DMA cycle or direction of data transfer. The count of DMA cycles and type of DMA cycle must be programmed before channel is enabled.

RD	WR	TC_{13}	TC_{12}	TC_{11}	TC_{10}	TC_9	TC_8	TC_7	TC_6	TC_5	TC_4	TC_3	TC_2	TC_1	TC_0
↓	↓	← 14 bit binary count N–1 where N is number of bytes to be transferred →													

RD	WR	DMA cycle
0	0	DMA verify cycle
0	1	DMA write cycle
1	0	DMA read cycle
1	1	Illegal

Fig. 17.13.5 : Terminal count register

17.14 Operating Modes of 8257

Q. 1 *Explain different operating modes of 8257.*

Q. 2 *What do you understand by the following terms ?*

(a) *Rotating priority mode.*

(b) *TC STOP mode.*

The 8257 operates in the following modes :

- Rotating Priority mode
- Fixed Priority mode
- Extended write mode
- TC stop mode
- Autoload mode

17.14.1 Rotating Priority Mode

- If the RP bit of mode set register is set then the 8257 operates in rotating priority mode.
- After each DMA cycle, the priority of each channel changes.
- Hence all channels will get equal opportunity, if they are enabled and their DMA requests exist. Initially CH-0 gains highest priority while CH-3 gains lowest priority. The channel which has just been serviced will get the lowest priority after the DMA cycle and other channels move up to the next higher priority levels.
- The rotating pattern of channels is as shown in Fig. 17.14.1.

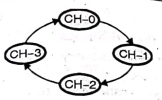


Fig. 17.14.1

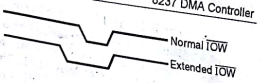


Fig. 17.14.2

17.14.2 Fixed Priority Mode

- If the RP bit of mode set register is reset then 8257 operates in fixed priority mode.
- In fixed priority mode, channel 0 has highest priority and channel 3 has lowest priority.
- The priority is resolved during state 4 of each DMA cycle.

17.14.3 Extended Write Mode

- If the EW bit of mode set register is set, then 8257 generates advanced or extended write control signals ($\overline{IOW}$ and $\overline{MEMW}$). i.e. the write control signals will go LOW, one clock cycle earlier, as shown in Fig. 17.14.2.
- This mode is used to interface slower devices to the system.
- If the memory device or I/O device connected is slower, then for synchronization READY signal is used.
- In this method the write signal is delayed by adding wait states into a DMA cycle. This reduces the speed of transfer.
- But in extended write mode, the write signal is extended earlier without adding states. i.e. the set up time of write input signal of an I/O device or memory is increased in extended write mode without reducing the speed of transfer.
- This signal allows more time to external logic for deciding if additional wait states are needed.

17.14.4 TC Stop Mode

- If the TC stop bit in mode set register is set, then 8257 disables the channel whose TC is reached. Thus it stops further DMA operations on that channel.
- If the TC stop bit is reset, then the TC have no effect on channel, corresponding channel must be disabled by the microcomputer system through software.
- The TC stop bit option should be common for all channels.

17.14.5 Autoload Mode

- If AL bit of mode set register is set, the 8257 operates in autoload mode.
- In this mode the data is transferred by channel 2 only i.e. other channels are not used for data transfer.
- It can be used for repeat block or block chaining operations.

(i) **Repeat block operation :**

- If the AL bit is set, the parameters (memory address and terminal count) of CH-2 are duplicated into CH-3 register.

- The EN_3 and TC bits are irrelevant.
- The new parameters are not written into the CH-3 registers.
- The CH-2 transfers first DMA block between memory and I/O device. After transferring first DMA block, the 8257 executes an ***'update cycle'***. During this cycle the contents of CH-3 register are transferred to CH-2 register and update flag is set in status register.
- Thus, the repeat block operation continues with the programming of only CH-2. It can be used in CRT or LED refreshing.

(ii) Block chaining operation :

- In this operation CH-2 transfers two or more different data blocks. CH-3 must be loaded with different parameters after initialization of CH-2 registers.
- In this mode both the channels have to be enabled.
- The TC stop bit is irrelevant.
- The autoload timing diagram is shown in Fig. 17.14.3.

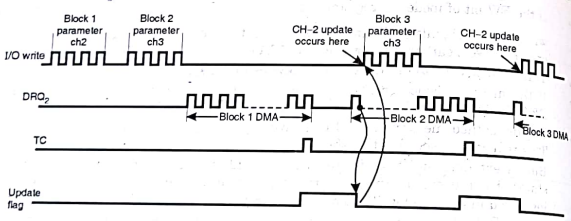


Fig. 17.14.3 : Autoload timing

- Initially CH-2 and CH-3 register are initialized with block 1 and block 2 parameters respectively. Then, CH-2 transfers data block.
- During last DMA cycle it activates TC signal and sets the update flag.
- The 8257 executes an update cycle and transfers contents of CH-3 registers into CH-2 registers.
- The update flag is cleared at the end of first DMA cycle of next data block.
- The microprocessor writes new parameters (Block parameter) into CH-3 registers.
- The CH-2 then transfers second byte of next data block. In this way, CH-2 transfers two or more data blocks.
- The TC signal can be used to interrupt the microprocessor to load block parameters.

17.15 8257 DMA Operation State Diagram

17.15.1 State Diagram

Fig. 17.15.1 shows a state diagram of 8257.

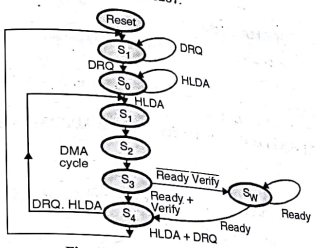


Fig. 17.15.1 : State diagram of 8257

The operation of 8257 is as follows :

State S_1 : After reset, the 8257 enters into idle state (S_1). In S_1 state, the 8257 samples all DRQ input signals. When it finds one or more DRQ inputs HIGH, the 8257 activates HRQ signal and enters into S_0 state, otherwise the 8257 remains in S_1 state.

State S_0 : In S_0 state, the 8257 samples its HLDA input. When it finds HLDA HIGH, it resolves the priority of channels and executes appropriate DMA cycle for highest priority channel, otherwise remains in S_0 state.

DMA cycles : The 8257 executes three types of DMA cycles programmed by D_{15} and D_{14} of TC register :

- DMA read
- DMA write
- DMA verify.

(i) **DMA read cycle :**

During this cycle, the data is transferred from memory to I/O device i.e. $\overline{MEMR}$ and $\overline{IOW}$ signals are activated.

(ii) **DMA write cycle :**

During this cycle, the data is transferred from I/O device to memory i.e. $\overline{MEMW}$ and $\overline{IOR}$ signals are activated.

(iii) **DMA verify cycle :**

- During this cycle the data is not transferred between memory and I/O device.
- This cycle may be used by the peripheral device to verify the data that has been recently transferred.

- The peripheral device can enable parity generator/checker circuits or a check sum or CRC (cyclic redundancy character) byte to detect error in data.
- During this cycle, $\overline{IOR}$, $\overline{IOW}$, $\overline{MEMR}$ and $\overline{MEMW}$ signals are not activated, but $\overline{DACK}$ is activated.

17.15.2 Operation of DMA Cycle :

The timing diagram of DMA cycle is shown in Fig. 17.15.2.

State S_1 :

- In S_1, the 8257 places lower byte of memory address on $A_0 - A_7$ lines and higher byte of memory address on $D_0 - D_7$ lines.
- It activates AEN signal at the falling edge of S_1 and ADSTB signal at the rising edge of S_1.

State S_2 :

- It activates $\overline{MEMR}$ (DMA read) or $\overline{IOW}$ (DMA write) at the rising edge of S_2.
- It activates $\overline{DACK}$ at the falling edge of S_2.
- In extended write mode, it activates $\overline{MEMW}$ (DMA write) or $\overline{IOR}$ (DMA read) at the rising edge of S_2.

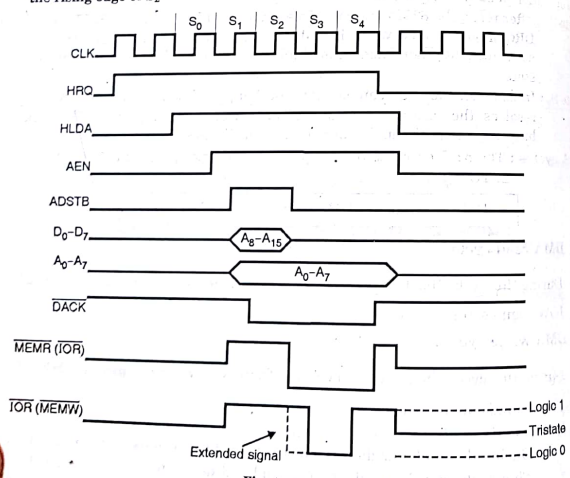


Fig. 17.15.2

State S_3 :

- It activates $\overline{MEMW}$ (DMA write) or $\overline{IOR}$ (DMA read) at the rising edge of S_3 in normal write mode.
- It activates TC and MARK signals in appropriate DMA cycles.
- In S_3, it samples the ready input.
- When it finds READY low, it adds wait states between S_3 and S_4, otherwise enters into S_4. It continues wait states until a high level at READY input is detected.
- The READY input is not sampled during S_3 of DMA verify cycle. Hence, the wait states are not added into DMA verify cycle.

State S_4 :

- It disables $\overline{MEMR}$, $\overline{MEMW}$, $\overline{IOR}$, $\overline{IOR}$ and $\overline{DACK}$ signals.
- It disables TC and MARK signals in appropriate DMA cycle.
- If TC stop bit is set, then it disables channel during S_4 of last DMA cycle. It samples HLDA and DRQ inputs.
- When it finds both HLDA and DRQ high, it resolves the priority of the channels and executes next DMA cycle for highest priority channel, otherwise it enters into S_1 state.

17.16 8257 Interfacing

Q. Interface the DMA controller 8257 with 8085

17.16.1 I/O Mapped I/O

Fig. 17.16.1 shows the interfacing of 8257 with 8085 microprocessor. In slave mode, the 8257 functions as an I/O device and the system bus is controlled by microprocessor. In master mode, the microprocessor must be isolated from the system bus and this isolation is done by an AEN signal.

1. **8212 latch 1 :** This latch is used to demultiplex $AD_0 - AD_7$ lines of microprocessor. The $\overline{DS_1}$ input line is connected to an AEN output of 8257. Hence latch is enabled in slave mode and disabled in master mode. This latch is strobed by ALE of the microprocessor. Functionally, 8212 is similar to 74373 latch.
2. **Decoder :** The 8085 microprocessor does not provide separate control signals for memory and I/O devices. Hence, a decoder is used to generate, $\overline{MEMR}$, $\overline{MEMW}$, $\overline{IOR}$ and $\overline{IOW}$ control signals in slave mode. It is disabled in master mode ($\overline{OE}$ line is connected to the AEN output line of 8257). In decoder all system control signals are generated by 8257 itself.
3. **8212 latch 2 :** It is used to hold higher byte of memory address in master mode. It is disabled in slave mode (S_2 is connected to the AEN). The input lines of this latch are connected to the data lines of the 8257 and the output lines are connected to the high order system address lines ($A_8 - A_{16}$). This latch is strobed by ADSTB signal of the 8257. The I/O address decoder is disabled in master mode, hence DMA I/O devices are selected by $\overline{DACK}$ signals only.

The HRQ output of 8257 is connected to the HOLD input of 8085 microprocessor and HLDA input of the 8257 is connected to HLDA output of 8085 microprocessor. The RESET OUT of 8085 microprocessor is connected to the the RESET IN of 8257. After reset, the 8257 operates in slave mode. The clock signal is generated by external clock generator. The 8257 requires single phase (ϕ_1), 50% duty cycle TTL clock. This can be obtained by inverting CLOCK OUT of 8085 microprocessor.

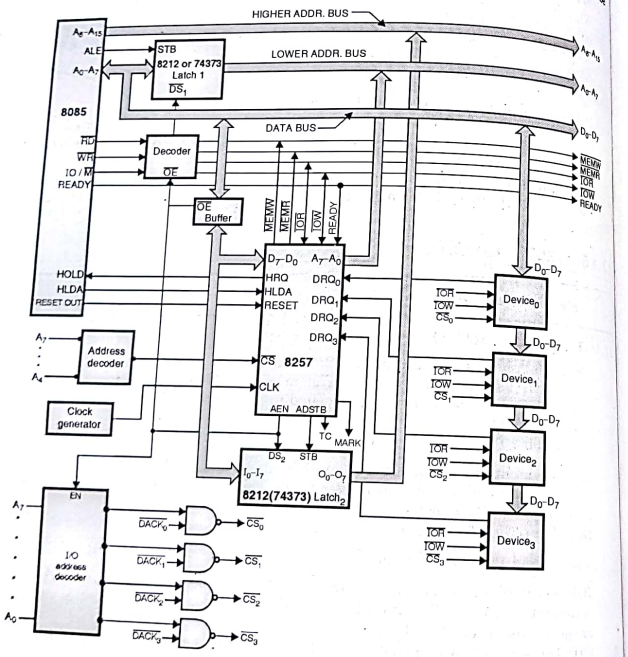


Fig. 17.16.1 : Interfacing of 8257 with 8085 (I/O mapped I/O)

17.16.2 8257 Interfacing (Memory Mapped I/O)

It is similar to I/O mapped I/O except the following connections :

(1) $\overline{MEMR}$ of 8257 is connected to the $\overline{IOR}$ of system bus

(2) $\overline{IOR}$ of 8257 is connected to the $\overline{MEMR}$ of system bus

(3) $\overline{MEMW}$ of 8257 is connected to the $\overline{IOW}$ of system bus

(4) $\overline{IOW}$ of 8257 is connected to the $\overline{MEMW}$ of system bus

(5) The address decoder uses $A_4 - A_{16}$ lines to generate $\overline{CS}$ signal.

In memory mapped, the data is transferred from memory to I/O during DMA write cycle and I/O to memory during DMA read cycle respectively.

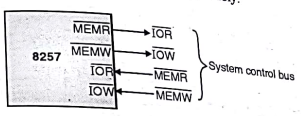


Fig. 17.16.2

- In memory mapped I/O the system memory control lines are connected to the I/O control lines of 8257 and system I/O control lines are connected to the memory control lines of 8257.

Ex. 17.16.1 : Write an initialization program to transfer data from a peripheral to memory for the following specifications.

(i) Bytes to be transferred = 256
(ii) Memory starting address = 2050 H.
(iii) Channel used =1
(iv) Priority = rotating.
(v) Extended write disabled.
(vi) DREQ senses active high.
(vii) DACK senses active low
(viii) Mode of transfer : block transfer.
(ix) Addresses

Registers	Addresses
Command	88 H
Mode	8B H
Current word count for channel	83 H
Current address for channel	82 H

Soln. :

Step I : The number of bytes to be transferred = 256

$\therefore$ terminal count = N – 1 = 255 = FFH

This value must be loaded into current word register.

Step II : The memory starting address is 2050 H. This value must be loaded in the current address register. The data is to be transferred from a peripheral device to the system memory. This indicates that the operation is a DMA write operation.

Step III : The command register and mode register words are

Command register

0	1	0	1	0	0	0	0

= 50 H

Mode register

1	0	1	1	0	0	0	1

= B1H

Program :

Instruction	Comments
LXI H, 2050 H	Initialize memory pointer to 2050 H
MVI A, 50 H	Send command word
OUT 88 H	
MVI A, 50 H	Send lower byte of address
OUT 82 H	
MVI A, 20 H	Send higher byte of address
OUT 82 H	
MVI A, FFH	Send lower byte of count
OUT 83 H	
MVI A, 00H	send higher byte of count
OUT 83 H	
MVI A, B1H	Send mode word
OUT 8B H	

❑❑❑

CHAPTER

18

8259
(Programmable Interrupt Controller)

18.1 Polling and Interrupts

- Whenever more than one I/O devices are connected to a microprocessor based system, any one of the I/O devices may ask service at any time. There are two methods in which the microprocessor can service these I/O devices. One method is to use the **polling routine**, while the other method employs **interrupt**.
- In the polling routine the microprocessor checks whether any of the I/O devices is requesting for service.
- The polling routine is a simple program that keeps a check for the occurrences of interrupt. For e.g. : Let us assume that our polling routine is servicing I/O ports 1, 2, 3,.....8. The polling routine will check the status of the I/O ports in a proper sequence.
- The polling routine will first transfer the status of the I/O port 1 to the accumulator. It then checks the contents of accumulator to determine if the service request bit is set. If the bit is set then I/O port 1 service routine is called, otherwise the polling routine will move forward to check if port 2 is requesting service. On completion of the service to port1, the polling routine will test port2. The process is repeated till all the 8 ports are tested and all the I/O ports those are demanding service are processed. On completion of the polling routine, the microprocessor will resume with the execution of the program. Fig. 18.1.1 shows the sequence for polling routine.
- The polling routine has priorities assigned to the different I/O devices. Once the routine begins port1 will always be checked first, then port2 and so on.
- Another way that allows the microprocessor stop with the execution of the program and give service to the I/O devices is **interrupt**. It is an external asynchronous input that informs the microprocessor to complete the instruction that it is currently executing and fetch a new routine in order to offer service to the I/O device. Once the I/O device is serviced, the microprocessor will continue with the execution of its normal program.

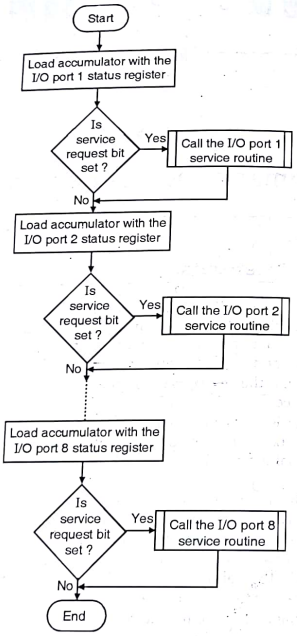


Fig 18.1.1 : Polling sequence

18.2 8259A Programmable Interrupt Controller

- For applications where we require multiple interrupt sources, we need to use an external device called as a priority interrupt controller (PIC).
- By connecting a PIC to the microprocessor we can increase the interrupt handling capacity of the microprocessor.
- 8259A is the commonly used priority interrupt controller.

18.2.1 Features of 8259A

- It is a LSI chip which manages 8 levels of interrupts i.e. it is used to implement 8 level interrupt system.

- It can be cascaded in a master slave configuration to ma interrupts.
- It can identify the interrupting device.
- It can resolve the priority of interrupt requests i.e. it, does not require any external priority resolver.
- It can be operated in various priority modes such as fixed priority and rotating priority.
- The interrupt requests are individually maskable.
- The operating modes and masks may be dynamically changed by the software at any time during execution of programs.
- It accepts requests from the peripherals, determines priority of incoming request, checks whether the incoming request has a higher priority value than the level currently being serviced and issues an interrupt signal to the microprocessor.
- It provides 8 bit vector number as an interrupt information.
- It does not require clock signal.
- It can be used in polled as well as interrupt modes.
- The starting address of vector number is programmable.
- It can be used in buffered mode (Buffered mode is applicable for multiprocessor system).

18.3 8259 Block Diagram

Q. Explain 8259 with functional block diagram.

The block diagram of 8259 is as shown in Fig. 18.3.1. It contains following blocks :

- Data bus buffer
- Cascade buffer and comparator
- IRR (Interrupt Request Register)
- Priority resolver
- Read/write logic
- Control logic
- InSR (In-Service Register)
- IMR (Interrupt Mask Register)

(1) ***Data bus buffer*** : It is used to transfer data between microprocessor and internal bus.

(2) ***Read/Write control logic*** : It sets the direction of data bus buffer. It controls all internal read/ write operations. It contains initialization and operation command registers.

(3) ***Cascaded buffer and comparator*** : In master mode, it functions as a cascaded buffer. The cascaded buffers outputs slave identification number on cascade lines. In slave mode, it functions as a comparator. The comparator reads slave identification number from cascade lines and compares this number with its internal identification number. In buffered mode it generates an $\overline{EN}$ signal.

(4) ***Control logic*** : It generates an INT signal. In response to an $\overline{INTA}$ signal, it releases three byte CALL address or one byte Vector number. It controls read/write control logic, cascade buffer/comparator, in service register, priority resolver and IRR.

(5) *Interrupt request register (IRR) :*

Q. Explain how will you read the IRR.

It is used to store all pending interrupt requests. Each bit of this register is set at the rising edge or at the high level of the corresponding interrupt request line. The microprocessor can read contents of this register by issuing appropriate, command word.

(6) *In service register (InSR) :*

It is used to store all interrupt levels currently being serviced. Each bit of this register is set by priority resolver and reset by End of interrupt command word. The microprocessor can read contents of this register by issuing appropriate command word.

(7) *Priority resolver* : It determines the priorities of the bit set in the IRR. To make decision, the priority resolver looks at the ISR. If the higher priority bit in the InSR is set then it ignores the new request. If the priority resolvers finds that the new interrupt has a higher priority than the highest priority interrupt currently being serviced and the new interrupt is not in service, then it will set appropriate bit in the InSR and send the INT signal to the microprocessor for new interrupt request.

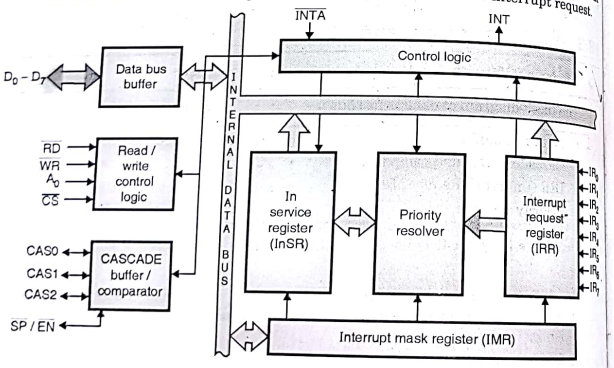


Fig. 18.3.1 : Functional block diagram of 8259

(8) *Interrupt mask register (IMR) :*

Q. Explain how will you read the IMR.

It is a programmable register.It is used to Mask unwanted interrupt request, by writing appropriate command word. The microprocessor can read contents of this register without issuing any command word.

18.4 Pin Configuration of 8259

Q. Explain in detail 8259 with pin configuration.

The pin configuration of 8259 programmable interrupt controller is as shown in Fig. 18.4.1.

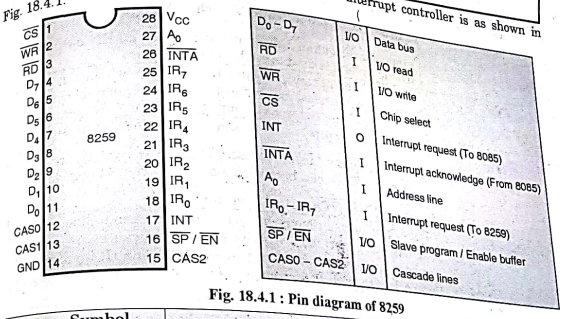


Fig. 18.4.1 : Pin diagram of 8259

Symbol	Description
(1) $D_0 - D_7$	These are bi-directional, tristate, buffered, non multiplexed data lines. These lines are connected to the system data lines directly (in non-buffered mode) or through data buffers (in buffered mode).
(2) $\overline{RD}$ *(Read)*	It is an active low control input line. It is used to read contents of internal registers. It is connected to $\overline{IOR}$ or $\overline{MEMR}$ of the system bus.
(3) $\overline{WR}$ *(Write)*	It is an active low control input line. It is used to write data into registers. It is connected to $\overline{IOWR}$ or $\overline{MEMWR}$ of the system bus.
(4) A_0	It is an address input line. It is used to select appropriate control register. It is connected to one of the address lines of the system address bus.
(5) $\overline{CS}$	It is an active low chip select input line. It is used to select 8259 A chip. This signal is generated by address decoder.
(6) *CAS0 – CAS2*	These are bi-directional 3 bit cascade lines. These lines are used in cascade mode only. In master mode these lines function as output lines. In this mode, the PIC places 3 bit slave identification number on cascade lines. In slave mode, these lines function as input lines. In this mode, the PIC reads 3 bit slave identification number from master PIC via cascade lines.

Symbol	Description
(7) $\overline{SP}$ / $\overline{EN}$ (Slave Program / Enable)	It is an active low bi-directional control line. In non buffered mode, it functions as $\overline{SP}$ input line. In this mode, $\overline{SP}$ is used to distinguish between master and slave PIC's. i.e. $\overline{SP}$ pin of master PIC is connected to V_{CC} while $\overline{SP}$ pin of slave PIC's is grounded. In buffered mode it functions as an $\overline{EN}$ output line. In this mode it is used to enable data buffers.
(8) INT	It is an interrupt output line. It goes high whenever a valid interrupt (unmasked and highest priority) request is activated. It must be connected to INTR input of the microprocessor.
(9) $\overline{INTA}$	It is an interrupt acknowledge input line. This signal is generated by the microprocessor. The 8259A accepts two $\overline{INTA}$ pulses to release one byte vector number. The 8259 A does not work without $\overline{INTA}$ pulses in vectored mode.
(10) $IR_0 - IR_7$	These are asynchronous interrupt request input lines. These signals are generated by peripherals. They can be used either in edge triggered or level triggered mode. The IR input should make low to high transition in level as well as edge triggered mode.

18.5 Priority Modes

Q. Explain various priority modes of 8259 A.

The various priority modes of 8259A are :

- Fully nested mode.
- Rotating priority mode.
- Special fully nested mode.
- Special masked mode.

18.5.1 Fully nested mode (FNM)

- After initialization, the 8259A operates in fully nested mode i.e. it is default priority mode. It continues to operate in this mode until the mode is changed through OCWs. It is also called as default mode.
- In this mode, IR_0 has highest priority and IR_7 has lowest priority. When the interrupt is acknowledged, it sets the corresponding bit of ISR.
- This bit will prevent all interrupts of the same or lower level, however it will accept higher priority interrupt requests.
- The bit in the InSR will remain set until an EOI (End of Interrupt) command is issued by the microprocessor at the end of ISR (Interrupt Service Routine).
- If the AEOI (Automatic End of Interrupt) bit is set, the bit in the InSR resets at the trailing edge of last $\overline{INTA}$

End of Interrupt (EOI)

Q. Explain the importance of EOI command of 8259.

(1) The InSR bit can be reset by on EOI command that is issued by the microprocessor before existing from the interrupt routine.

(2) In the FNM, the highest level in the InSR would correspond to the last interrupt that is acknowledged and serviced in such a case, a non-specific EOI command can be issued

(3) If the fully nested mode is not used, the 8259 PIC may not be able to determine the last interrupt that is acknowledged. In such a case a specific EOI command needs to be issued.

(4) In the cascade mode, the EOI command should be issued twice once for master and once for salve.

(A) ***Nonspecific EOI command*** :

This command informs the 8259A that the current interrupt service routine has been completed. It resets current bit (highest priority bit) of the InSR. This command is independent of the interrupt level and is thus called a nonspecific EOI. It must be used in fully nested mode.

(B) ***Specific EOI command*** :

If the fully nested mode is not used, the 8259A may not be able to tell which interrupt was just acknowledged. In such a case a specific EOI command must be issued. This command resets a bit of InSR, which is specified by $L_2L_1L_0$.

Automatic end of interrupt (AEOI)

In this mode the 8259 will perform a non-specific EOI on its own and on the trailing edge third $\overline{INTA}$ pulse. It can be used only for master and not for salve.

18.5.2 Special Fully Nested Mode (SFNM)

- In the fully nested mode, on the acknowledgement of an interrupt, the further interrupts of the same level are disabled.
- Consider a large system that uses cascaded 8259s and where the interrupt levels within each salve have to be considered. An interrupt request input to the salve causes the salve to place an interrupt request to the master and on one of the master's inputs.
- Interrupts to the same salve will cause the salve to place the request to the master on same input to the master. However, these interrupts will not be recognized.
- This is because further interrupts to the same level are disabled by the master as its InSR bit is set.
- The special fully nested mode is used to prevent the problem.

18.5.2(A) Difference between Special Fully Nested Mode And Fully Nested Mode

(1) Whenever an interrupt request form a salve is being serviced, the slave is allowed to place further requests and if these requests are of a higher priority than the request that is currently being serviced. These interrupts are recognized by the master. It initializes interrupt requests to the microprocessor unit.

(2) Before termination form the ISR, a non-specific EOI must be sent to the slave. Its InSR must be read to determine if it was the only interrupt to the slave. If the InSR is empty, a non-specific EOI command can be sent to the master. If the InSR is not empty, then the same IR level input to the master can be reserviced. This is because there are multiple interrupts on the slave EOI must not be sent to the master.

18.5.3 Rotating Priority Mode

Q. Explain rotating priority mode in case of 8259 PIC.

The rotating priority mode can be set as :

- Automatic rotation
- Specific rotation

18.5.3(A) Automatic Rotation

- In this mode, a device after being serviced becomes the lowest priority, and consecutive next interrupt becomes highest priority.
- The device that has been just serviced will receive the seventh priority. Here IR_i has just been serviced, hence it becomes lowest priority and IR_{i+1} becomes highest priority.

IR_0	IR_1	IR_2	IR_3	IR_4	IR_5	IR_6	IR_7
3	4	5	6	7	0	1	2
				↑	↑		
				Lowest priority	Highest priority		

18.5.3(B) Specific Rotation

- In the automatic rotation mode, the interrupt request that is serviced last is assigned the lowest priority.
- In the specific rotation mode, lowest priority can be assigned to any interrupt input (IR_0 to IR_7) by a specific rotation command.
- e.g. if lowest priority is assigned to IR_3, all other interrupt priorities are shown.

IR_0	IR_1	IR_2	IR_3	IR_4	IR_5	IR_6	IR_7
4	5	6	7	0	1	2	3

18.5.4 Special Masked Mode

- If an interrupt is in service, then the corresponding bit in the InSR is set and the lower priority interrupts are inhibited.
- Sometimes it is desired for the interrupt service routine to dynamically after the systems interrupt priority structure. In such cases we have to used special masked mode.
- Special masked mode inhibits further interrupts at that level and enables interrupts form all other levels that are not masked. Thus, as interrupt can be enabled by loading the mask register.

18.5.5 Operating Modes

8259 has two operating modes viz. interrupt driven and polling mode. In interrupt driven mode, 8259 interrupts the processor with the INT pin whenever it gets an interrupt.

Polled mode

Q. Explain POLL mode of 8259.

- In this mode the INT output is not used. The microprocessor checks the status of the interrupt request by issuing poll command.
- The microprocessor reads contents of 8259 A after issuing the poll command.
- During the read operation the 8259A provides polled word and sets the InSR bit of highest active interrupt request in following format.

I	X	X	X	X	X	W_2	W_1	W_0

I = 1 One or more interrupt requests activated
I = 0 No interrupt request activated

Binary code of highest priority active interrupt request

18.6 Programming the 8259A

Q. Explain initialisation command words of 8259 IC.

The 8259 can be programmed through a sequence of simple I/O operations. It accepts two types of command words. They are :

- Initialization command words (ICWs)
- Operation command words (OCWs)

- The S259A can be initialised with four ICWs, the first two are compulsory and the other two are optional based on the modes being used.
- The words must be issued in a given sequence.
- Fig. 18.6.1 shows the initialisation sequence.
- After the initialisation the 8259A can be set up to operate in various modes by using three different OCWs. However, they are not necessary to be issued in a specific sequence. They may be loaded any time after initialisation of 8259 to dynamically alter the priority modes.

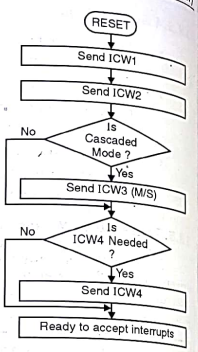


Fig. 18.6.1 : 8259A initialization sequence

18.6.1 Initialization Command Word 1 (ICW1)

It is used to program the basic operation of 8259A. To write this command word into ICW1 register A_0 pin should be at logic 1. After accepting this command the 8259A performs the following internal operations :

(1) It resets edge sense circuit, hence an interrupt request must make a low to high transition to trigger IR input after initialization.

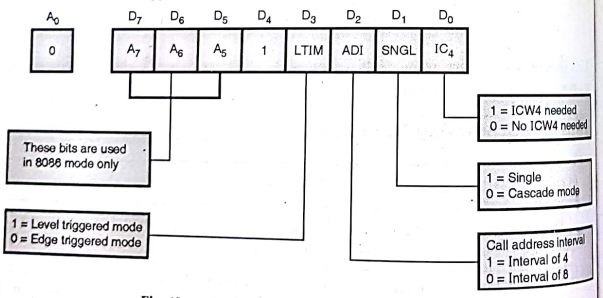


Fig. 18.6.2 : Initialization command word 1 (ICW1)

(2) It clears interrupt mask register hence all interrupt are unmasked.

(3) It assigns lowest priority to IR_7 and highest priority to IR_0.

(4) It sets slave identification number of slave 7 (1,1,1), if D_1 bit of ICW1 is '0'.

(5) It clears special mode and sets the status read to IRR, i.e. microprocessor can read IRR without issuing a special command word.

(6) If D_0 bit of ICW1 is reset, then it clears all functions associated with ICW4.

Refer Fig. 18.6.2.

(1) **IC_4 :** This bit indicates whether ICW4 is required or not. If this bit is cleared, no ICW4 will be issued and the default parameters will be set. If this bit is set, then ICW4 must be issued.

(2) **SNGL :** This bit indicates whether the single 8259 is used or multiple cascaded 8259's are used. This bit also decides whether ICW3 is required or not. If single 8259 then ICW3 is not required.

(3) **ADI :** This bit is used in 8085 mode only. ICW1 and ICW2 are given address A_{15} to A_5 for interrupt vector. If ADI = 1 the address interval is of 4 i.e. A_4 to A_0 will be 00000b for IR0, 00100b for IR1 and so on. If ADI = 0, the address interval is of 8 i.e. A_5 to A_0 will be 000000b for IR0, 001000b to IR1 and so on.

(4) **LTIM (Level triggered interrupt mode) :** This bit determines if the interrupt request are to be recognized in the level triggered mode or in the edge triggered mode.

(5) **$A_5 - A_7$:** These bits are used in 8086 mode only.

18.6.2 Initialization Command Word 2 (ICW2)

The ICW2 is used to program 8 bit vector number of interrupt type.

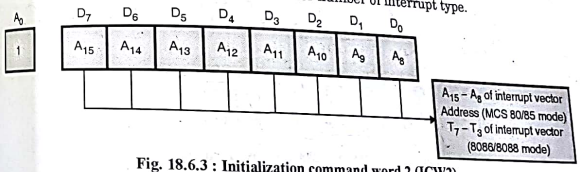


Fig. 18.6.3 : Initialization command word 2 (ICW2)

A write command issued after ICW1 with $A_0 = 1$ is considered as ICW2. The ICW2 format is as shown in Fig. 18.6.3.

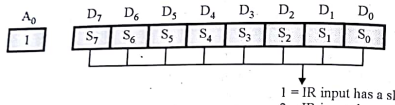
In 8085 mode, ICW2 bits are used to program A_8 to A_{15} address bits of ISR address. In 8086 mode, D_3 to D_7 bits are used to program T_3 to T_7 bits of 8 bit vector number. The lower bits T_0 to T_2 are provided by 8259 depending on which interrupt input is activated.

18.6.3 Initialization Command Word 3 (ICW3)

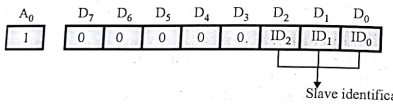
It is used in cascaded mode only. There are two types of ICW3's viz master ICW3 and slave ICW3. The master ICW3 is used to specify whether it has a slave 8259 connected to its interrupt request input. The slave ICW3 is used to assign a slave identification number. Identification number is used to tell slave 8259 on which IR input it is connected to master. A write command is issued after ICW1 and ICW2 and multiple

8259 system with A_0 = 1 is considered as ICW3. The ICW3 format is as shown in Fig. 18.6.4.

(ICW3) Master format



(ICW3) Slave format



ID_2	ID_1	ID_0	Slave PIC
0	0	0	Slave on IR0
0	0	1	Slave on IR1
0	1	0	Slave on IR2
0	1	1	Slave on IR3
1	0	0	Slave on IR4
1	0	1	Slave on IR5
1	1	0	Slave on IR6
1	1	1	Slave on IR7

Fig. 18.6.4 : (ICW3) Master/slave format

18.6.4 Initialization Command Word 4 (ICW4)

The ICW4 is used to initialize the 8259A in the following modes :

- Special fully nested mode.
- Auto EOI mode.
- Buffered mode.
- 8086/8088 mode.

A write command issued after ICW3 and ICW4 bit needed bit set in ICW1 with A_0 = 1 is considered as ICW4. The ICW4 format is as shown in Fig. 18.6.5.

(1) ***μPM*** : This bit indicates whether 8259 A is operated in 8085 or 8086 mode.

(2) ***AEOI (Auto End of Interrupt)*** : This bit indicates whether 8259 A is operated in normal EOI or auto EOI mode.

(3) ***M/S (Master / slave)*** : This bit is irrelevant in non buffered mode. In buffered mode it is used to distinguish between master and slave PICs.

(4) ***SFNM (Special fully nested mode)*** : This bit is used in cascade mode only. It is used to operate master PIC in special fully nested mode.

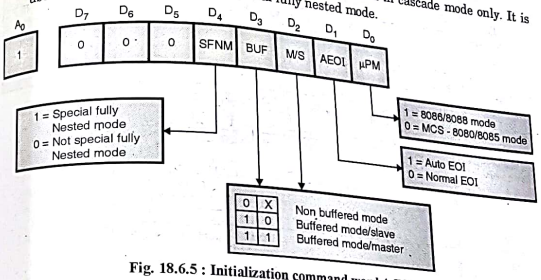


Fig. 18.6.5 : Initialization command word 4 (ICW4)

18.6.5 Operation Command Word 1 (OCW 1)

The OCW1 is used to mask unwanted interrupt request inputs (IR inputs). A write command issued after initialization with A_0 = 1 is considered as OCW1. The OCW1 format is as shown in Fig. 18.6.6.

OCW1 Format

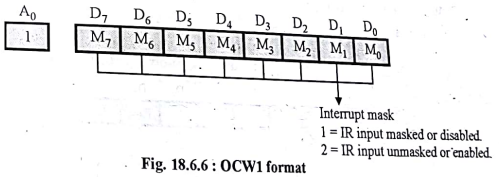


Fig. 18.6.6 : OCW1 format

18.6.6 Operation Command Word 2 (OCW2)

The OCW2 is used to program EOI (End Of Interrupt), rotate priorities and combination of both. A write command issued with A_0 = 0 and D_4D_3 = 00 is considered as OCW2. The OCW2 format is as shown in Fig. 18.6.7.

A_0	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	
0	0	0	1						Non specific EOI command
	0	1	1						Specific EOI command. The bit of ISR to be cleared is specified by $L_2 L_1 L_0$
	1	0	1						Rotate priorities on non specific EOI command
	1	0	0						Rotate priorities in auto EOI mode set command
	0	0	0						Rotate priorities in auto EOI mode clear command
	1	1	1						Rotate priorities on specific EOI command. The IR input, to be assigned at lowest priority is specified by $L_2 L_1 L_0$
	1	1	0						Set priority command. The IR input to be assigned lowest priority is specified by $L_2 L_1 L_0$
	0	1	0						No operation.

(R, SL, EOI: D_7 D_6 D_5; L_2 L_1 L_0: IR level to be acted upon)

End of interrupt

Non Specific rotation

Automatic rotation of enable / disable

Specific rotation of priorities

L_2	L_1	L_0	Interrupt
0	0	0	IR_0
0	0	1	IR_1
0	1	0	IR_2
0	1	1	IR_3
1	0	0	IR_4
1	0	1	IR_5
1	1	0	IR_6
1	1	1	IR_7

Fig. 18.6.7 : OCW2 format

18.6.7 Operational Command Word 3 (OCW3)

The OCW3 is used to program

- Special mask mode.
- Polled mode and
- Read IRR and InSR. A write command issued with $A_0 = 0$ and $D_1 D_3 = 01$ is considered as OCW3. The OCW3 format is as shown in Fig. 18.6.8.

OCW3 Format

A_0	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	
0	0	ESMM	SMM	0	1	P	RIR	RIS	
	0	X							No action
	1	0							Reset special mask mode
	1	1							Set special mask mode
	0	X							No action
	1	0							Read IRR
	1	1							Read InSR

(P: 1 = Polled mode)

Fig. 18.6.8 : OCW3 format

Status read operations

The OCW3 is used to read interrupt request register or in service register.

(i) **IRR status read :** The contents of IRR can be read by issuing read IRR command word (OCW3). After issuing this command word the microprocessor can read contents of IRR. To read contents of IRR, A_0 pin should be at logic 0. (Note that this command must be written into the 8259A only once for successive IRR status read operations).

(ii) **InSR status read :** The contents of InSR can be read by issuing read InSR command word (OCW3). After issuing this command word the microprocessor can read contents of InSR. To read contents of InSR, A_0 pin should be at logic 0. This command word needs to be written into the 8259A only once.

(iii) **IMR status read :** The microprocessor can read contents of IMR without issuing OCW3. To read contents of IMR, A_0 pin should be at logic 1.

- As discussed in the above subsections, the initialisation sequence shown in Fig. 18.6.1 should be followed.
- According to the flowchart in ICW1 and ICW2 should be sent to any 8259A in the system.
- If the system has any slave 8259As, then an ICW3 must be sent to the master and a different ICW3 must be sent to the slave. If certain special conditions are to be specified, then an ICW 4 needs to be sent to each master and slave.

Ex. 18.6.1 : Write the initialisation instructions for 8259A interrupt controller to meet the following specifications.
(i) Edge triggered, single.
(ii) Mask interrupts IR_1 and IR_3.
(iii) Interrupt vector type for IR_0 is 50 H.

Soln. :

ICW 1

A_7	A_6	A_5	1	LTIM	ADI	SNGL	IC_4	
0	0	0	1	0	0	1	1	= 13 H

ICW 2 : It is type of Interrupt.

B_7	B_6	B_5	B_4	B_3	B_2	B_1	B_0	
0	1	0	1	0	0	0	0	= 50 H

We do not require ICW3 as it is single 8259 connection.

ICW 4

D_7	D_6	D_5	SFNM	BUF	M / S	AEIO	µPM	
0	0	0	0	0	0	0	1	= 01 H

OCW 1

An OCW 1 must be sent to an 8259A to unmask any IR inputs. Here we want to mask IR_1 and IR_3, so we will put 1's in these two bits and 0's in the rest of bits.

M_7	M_6	M_5	M_4	M_3	M_2	M_1	M_0	
0	0	0	0	1	0	1	0	= 0AH

Program

Instruction	Comments
MOV AL,13H	Edge triggered, single and ICW4 required
OUT 40H,AL	Send ICW1
MOV AL,50H	Interrupt type for IR0
OUT 41H,AL	Send ICW2
MOV AL,01H	Microprocessor 8086 selection
OUT 41H,AL	Send ICW4
MOV AL,0AH	Mask interrupts IR1 and IR3
OUT 41H,AL	Send OCW1

Ex. 18.6.2 : Write the initialisation instructions required for the master slave configuration. Assume the INTR of slave is routed through IR_2 of the master 8259 to the microprocessor. The addresses of the master and slave being D0 H, D1 H and F0 H and F1H. Write the initialisation for automatic rotation and auto end of interrupt.

The master and slave interrupt vector type are 50 H and 60 H.

Soln. :

ICW 1 (Master)

A_7	A_6	A_5	1	LT1M	ADI	SNGL	IC_4
0	0	0	1	1	0	0	1

= 19H

ICW 2 (Master)

B_7	B_6	B_5	B_4	B_3	B_2	B_1	B_0
0	1	0	1	0	0	0	0

= 50 H

ICW 3 (Master)

S_7	S_6	S_5	S_4	S_3	S_2	S_1	S_0
0	0	0	0	0	1	0	0

= 04 H

ICW 4 (Master)

D_7	D_6	D_5	SFNM	BUF	M / S	AEIO	μPM
0	0	0	0	0	0	1	1

= 03 H

Operation Command word 2 (OCW 2) (Master)

R	SL	EOI	D_4	D_3	L_2	L_1	L_0
1	0	0	0	0	0	0	0

= 80 H

ICW 1 (Slave)

A_7	A_6	A_5	1	LT1M	ADI	SNGL	IC_4
0	0	0	1	1	0	0	1

= 19H

ICW 2 (Slave)

B_7	B_6	B_5	B_4	B_3	B_2	B_1	B_0
0	1	1	0	0	0	0	0

= 60 H

ICW 3 (Slave)

–	–	–	–	–	ID_2	ID_1	ID_0
0	0	0	0	0	0	1	0

= 02 H

ICW 4 (Slave)

D_7	D_6	D_5	SFNM	BUF	M / S	AEIO	μPM
0	0	0	0	0	0	1	1

= 03 H

Operation Command word 2 (OCW 2) (Slave)

R	SL	EOI	D_4	D_3	L_2	L_1	L_0
1	0	0	0	0	0	0	0

= 80 H

Program

Instruction	Comments
MOV AL,19H	Level triggered, cascaded and ICW4 required
OUT D0H,AL	Send ICW1 (Master)
MOV AL,50H	Interrupt type for IR0 (for Master)
OUT D1H,AL	Send ICW2 (for Master)
MOV AL,04H	Slave on IR2 of Master
OUT D1H,AL	Send ICW3 (for Master)
MOV AL,03H	Microprocessor 8086 selection and Automatic EOI
OUT D1H,AL	Send ICW4 (for Master)
MOV AL,80H	Enable Automatic rotation
OUT D0H,AL	Send OCW2 (for Master)
MOV AL,19H	Level triggered, cascaded and ICW4 required
OUT F0H,AL	Send ICW1 (for Slave)
MOV AL,60H	Interrupt type for IR0
OUT F1H,AL	Send ICW2 (for Slave)
MOV AL,02H	Identity (ID) of the slave since connected on IR2
OUT F1H,AL	Send ICW3 (for Slave)
MOV AL,03H	Microprocessor 8086 selection and Automatic EOI
OUT F1H,AL	Send OCW1 (for Slave)
MOV AL,80H	Enable Automatic rotation
OUT F0H,AL	Send OCW2 (for Slave)

18.7 Interfacing 8259 with 8086

Q. Explain interfacing of single 8259 with 8286 in minimum mode.

8259 can the interfaced with 8086 in following different possibilities.

1) 8259 in single mode and 8086 in minimum mode
2) 8259 in single mode and 8086 in maximum mode
3) 8259 in cascaded mode and 8086 in minimum mode
4) 8259 in cascaded mode and 8086 in maximum mode

In this system, only one PIC is connected to the microprocessor in vectored mode (interrupt driven I/O). CAS0, CAS1, CAS2 and $\overline{SP}/\overline{EN}$ lines of PIC are not used. In this system 8259A requires two or three initialization command words. Fig. 18.7.1 shows the connection of a PIC to the 8086 microprocessor.

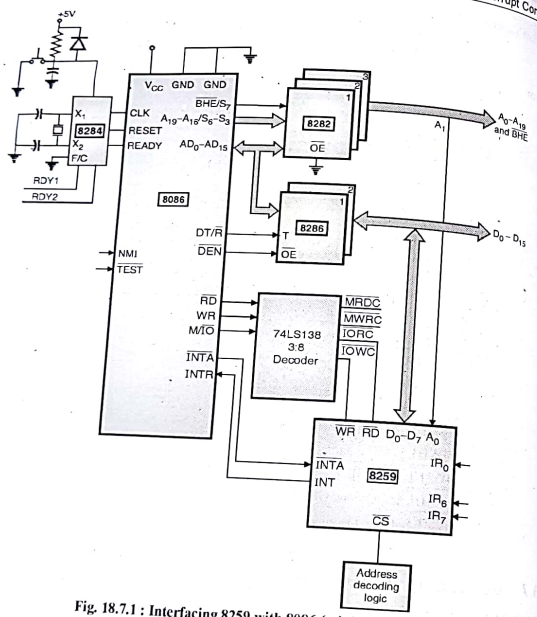


Fig. 18.7.1 : Interfacing 8259 with 8086 (minimum mode)

Interrupt operation :

Step 1 : The peripherals activate one or more interrupt requests.

Step 2 : These signals set corresponding bits in the IRR.

Step 3 : The priority resolver, resolves priorities of these interrupt requests and send an INT signal to the microprocessor if appropriate (If the new request has not been masked and has highest priority than the interrupt currently being serviced).

Step 4 : The microprocessor completes current instruction cycle and executes interrupt acknowledge cycle (6T). During this cycle it activates $\overline{INTA}$ pulse.

Step 5 : First $\overline{INTA}$ pulse will inform 8259 to be ready to send *Type number* for 8086 in next $\overline{INTA}$ cycle.

Step 6 : In 2nd $\overline{INTA}$ cycle 8259 provide 1 byte Type number to 8086. The same will be captured by the microprocessor. IN AEOI, it resets the corresponding bit of ISR.

at the falling edge of third $\overline{INTA}$ pulse. In normal EOI mode, the ISR bit remains set and can only be reset by issuing EOI command. The P/C disables an INT signal.

Step 7 : After receiving vector or type number, 8086 performs standard operation.

18.7.1 Interfacing Single 8259 with 8086 in Maximum Mode

Q. Explain interfacing single 8259 with 8086 in maximum mode.

The control signals are provided by 8086 in case of minimum mode while in maximum mode, bus controller 8288 provides control signals. Fig. 18.7.2 shows 8259 in single mode and 8086 in maximum mode.

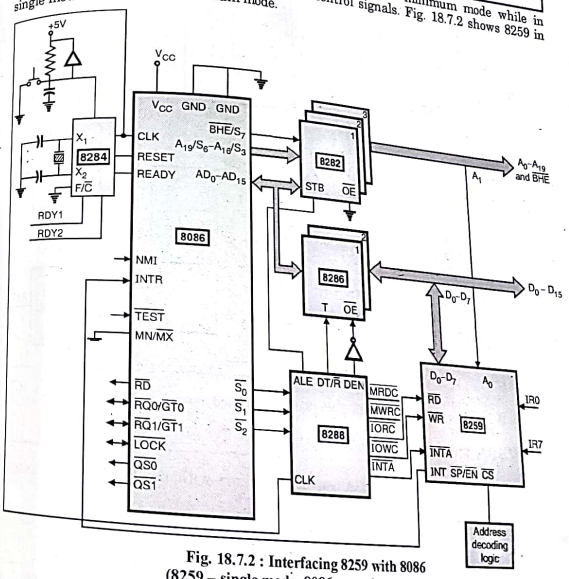


Fig. 18.7.2 : Interfacing 8259 with 8086 (8259 – single mode, 8086 – maximum mode)

18.7.2 Interfacing 8259 in Cascaded Mode

Q. Explain with neat interface diagram cascading of two 8259's.

For cascaded 8259, the INT pin of the slaves are connected to interrupt request pins (IR0 – IR7) and INTA to the INTA of master 8259. The CAS2 – CAS0 lines work as output for the master and input for the slave. Fig. 18.7.3 shows cascaded 8259 interfaced with 8086 in maximum mode.

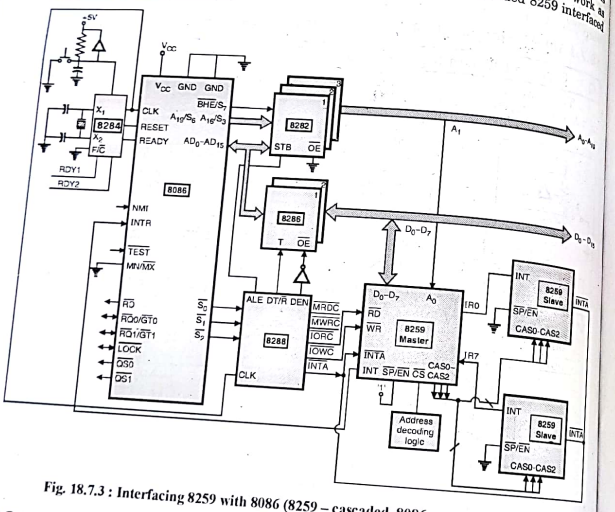


Fig. 18.7.3 : Interfacing 8259 with 8086 (8259 – cascaded, 8086 – maximum mode)

18.8 8259A Interfacing with 8085

Fig. 18.8.1 shows how an 8259A can be interfaced with the 8085 microprocessor system.

Address of 8259A :

A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0	
1	1	1	1	0	0	0	X	= F0 H or F1 H

- The 74LS138 address decoder will assert the $\overline{CS}$ input of 8259A when an I/O base address in F0H or F1H on the address bus.

- A_0 input of 8259A is used to select one of the two internal addressed in the device A_0 of 8259A is connected to the system line A_0. Hence, the system addresses for the two internal addressed are F0H and F1H.

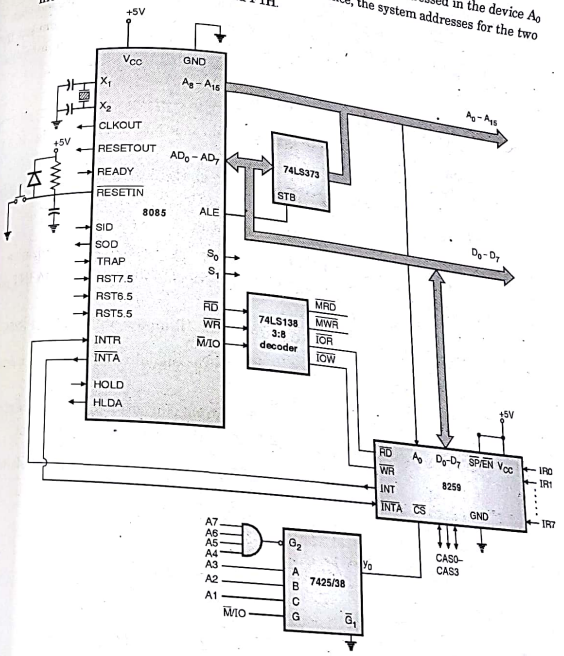


Fig. 18.8.1 : Interfacing 8259 to 8085

- The data lines of an 8259A are connected to the $AD_0 - AD_7$ of system data bus. $\overline{RD}$ and $\overline{WR}$ signals are connected to the system $\overline{RD}$ and $\overline{WR}$.
- The interrupt request signal INT from the 8259A is connected to the INTR input pin of the 8085.

- The $\overline{INTA}$ of 8085 is connected to $\overline{INTA}$ of 8259A.
- Single 8259A is used. Hence $\overline{SP}$ / $\overline{EN}$ is tied high $CAS_0 - CAS_2$ lines are left open.
- The eight IR inputs $IR_0 - IR_7$ are available for interrupt signals. The unused IR inputs will be tied to ground so that a noise pulse cannot cause an interrupt suddenly.

18.9 Cascade PICs System with 8085

- The 8259A can be easily interconnected to obtain multiple interrupts Fig. 18.9.1 shows 8259A connected in cascade mode.
- When 8259A IC's are connected in cascade, one 8259A is configured as master while all other 8259A's are configured as salve.
- In Fig. 18.9.1 8259-1 is in master mode and others are in slave mode.
- Each slave is recognized by a number that is assigned as a part of its initialization.
- The 8085 microprocessor has only one INTR input. Hence only one of 8259A INT pin is connected to the 8085 INTR pins.
- The INT pins from other 8259s are connected to the IR inputs of the master 8259A.

These cascaded 8259s are called as **slave**.The $\overline{INTA}$ signal is connected to master as well as slave 8259A.

- The cascade pins CAS_0-CAS_2 are connected from master to corresponding slave pins. For master they function as outputs while for slave these pins function as input.
- The $\overline{SP}$/ $\overline{EN}$ signal is tied high for the master. However it is grounded for all slaves.
- Addresses for 8259 As.

For 8259A – 1

A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0
1	1	1	1	0	0	0	×

= F0 or F1H

For 8259A – 2

A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0
1	1	1	1	0	0	1	×

= F2 H or F3H

For 8259A – 3

A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0
1	1	1	1	0	1	0	×

= F4 H or F5H

- Upto 8 PICs can be cascade together to act as a slave unit to master PIC. Thus, a total of 64 peripherals can be connected to the microprocessor.

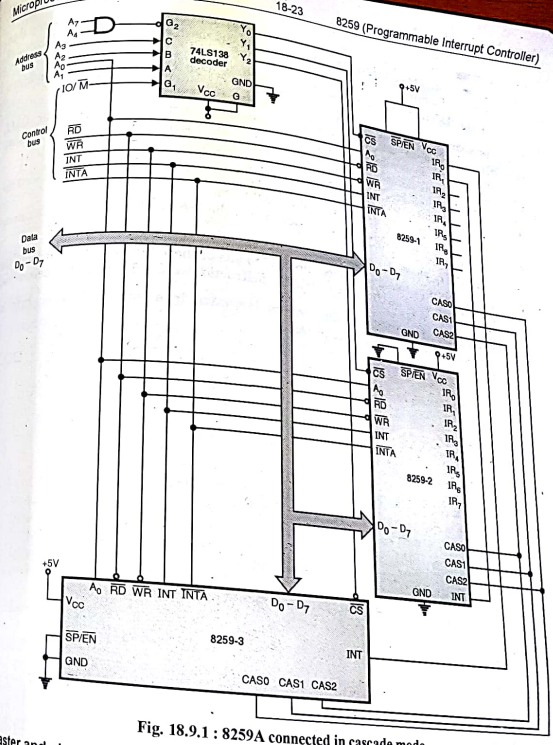


Fig. 18.9.1 : 8259A connected in cascade mode

Master and slave operation

- When the slave receives an interrupt signal on one of its IR inputs, it checks the mask condition and priority of the interrupt request.
- If the interrupt is unmasked and its priority is higher than any other interrupt level being serviced in the slave, then the slave will send an INT signal to the IR input of the master.

- If that IR input of the master is unmasked and if that input has higher priority than other IR inputs currently being serviced, then the master will send an INT signal to 8085 INTR input.
- If the INTR interrupt is enabled, the 8085 will go through its INTR interrupt procedure and sends three $\overline{INTA}$ pulses to both the master and the slave.
- In response to the first interrupt acknowledge signal the opcode for CALL instruction is available on the data bus. The master outputs a 3 bit slave identification number on $CAS_0 - CAS_2$ lines. This enables the slave.
- When the slave receives second $\overline{INTA}$ pulse from the 8085, the slave will send the low-order address byte of the ISR on the data bus.
- Finally, the slave sends the high order byte of the ISR on the data bus on receiving third $\overline{INTA}$ pulse.
- If the interrupt signal is applied directly to one of the IR inputs of the master, the master will send the opcode for CALL instruction to 8085 when it receives first $\overline{INTA}$ pulse from 8085. It then sends low-order byte and high-order byte in successive interrupt acknowledge cycles.

□□□

CHAPTER 19

8253

19.1 Introduction

Q. Write short note on 8254, Programmable interval timer.

In Microprocessor Based System, we come across two important modes i.e. timer → to provide delay and counter → to count incoming pulses. Presently, the way we implement timer/counter in 8086 based system as studied in earlier has following drawbacks :

(i) **Delay :** The 8086 can provide delays of any value, but it uses software to implement the delay. The instructions are arranged to waste time. The main disadvantage of this scheme is 8086 is executing some instructions, it means that it is busy in doing work.

(ii) **Counter :** The 8086 can count number of pulses arriving at port. To implement this 8086 goes on checking port, if it is active it increments counter by 1 and again goes on checking port. **The same disadvantage, 8086 will have to execute instructions.**

In large systems, where 8086 wastage time is critical, a separate timer IC 8254 can be used. The 8254 consists of 3 identical 16 bit counters. These counters can work as counter or can provide accurate time delays. To operate as a counter, a 16 bit count is loaded and the desired mode of operation is selected. The counters will work independently and generate the desired output. Now the 8086 job is to initialise and load counters.

In this chapter, we will discuss 8254 IC. Unless and until it is specified for 8253, the discussion of all points remains same in all respects to both ICs.

The two IC's 8253 and 8254 are timer ICs. Both are similar except following differences :

Q. How does 8254 differ from 8253 ?

8253	8254
1. Operating frequency 0 - 2.6 MHz.	Operating frequency 0 - 10 MHz.
2. Uses N-MOS technology.	Uses H-MOS technology.
3. Read-Back command is not available.	Read-Back command is available.
4. Reads and writes of the same counter cannot be interleaved.	Reads and writes of the same counter can be interleaved.

19.2 Features of Programmable Interval Timer

- Three independent 16 bit down counters.
- Counters can be programmed in 6 different programmable counter modes.
- Counting facility in both binary or BCD number system.
- Compatible with Intel and other microprocessors.
- Single + 5V supply.
- 24 pin dual in-line package.
- It is completely TTL compatible.
- It has a powerful command called READ BACK COMMAND which allows the user to check the count value, programmed mode and current mode and the current status of the counter (Only for 8254).
- Operating frequency range; For 8253 - DC to 2.6 MHz; For 8254 - DC to 10 MHz.

19.3 Pin Configuration of 8254

Q. Draw and explain the pin configuration of 8254.

The pin configuration of 8254 programmable interval timer is as shown in Fig. 19.3.1

$D_7 - D_0$	I/O	DATA BUS
CLK_N	I	Counter inputs
$GATE_N$	I	Counter gate inputs
OUT_N	O	Counter outputs
$\overline{RD}$	I	Read
$\overline{WR}$	I	Write
$\overline{CS}$	I	Chip select
$A_0 - A_1$	I	Counter select
V_{CC} / GND	I	+ 5 V supply/Ground

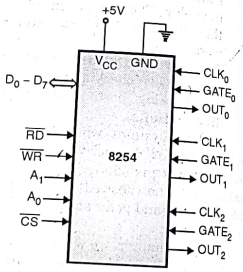


Fig. 19.3.1 : Pin diagram of 8254

Pin Description

Sr. No	Symbol	Name and function
1	D_0-D_7	**Data bus :** These are 8 bit bidirectional data bus lines, connected to the system data bus for data transfer between 8086 and 8254.
2	$\overline{CS}$	**Chip select :** This is an active low input signal, used to select the 8254 IC. If $\overline{CS}$ = 0 then 8254 will be active and take part in data transfer from/to 8086, otherwise 8254 will be in deactive state.

Sr. No	Symbol	Name and function
3	$\overline{RD}$	**Read :** This is an active low input signal, used in coordination with A_0 , A_1 to send data from appropriate counter to data lines D_0 - D_7.
4	$\overline{WR}$	**Write :** This is an active low input signal, used in coordination with A_0 , A_1 to load counters or to initialize counters.
5	A_0 - A_1	**Address lines :** These are input address lines used to distinguish different parts of 8254 such as Counter 0, Counter 1, Counter 2, Control word register.
6	CLK_{0-2}	**Clock input :** These are clock inputs to 3 independent counters. The pulses applied at these pins will be counted by respective counters.
7	$GATE_{0-2}$	**Gate control :** These are active high, input signals used to allow external hardware to control the respective counter. The function of gate input is dependent on operating mode.
8	OUT_{0-2}	**Output :** These lines are active high, output lines. The output is dependent on operating mode.

A_1	A_0	Selected part
0	0	Counter 0
0	1	Counter 1
1	0	Counter 2
1	1	Control word register

19.4 8254 Functional Block Diagram

Q. Explain 8254 with its functional block diagram.

The block diagram of 8254 is as shown in Fig. 19.4.1.

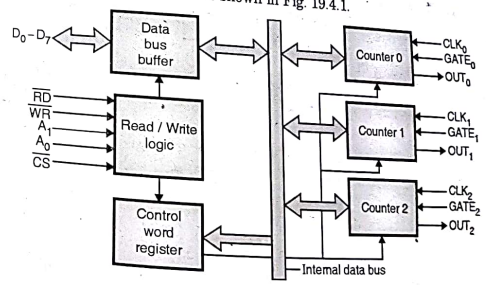


Fig. 19.4.1 : Functional block diagram of 8254

It includes data bus buffer, read/write logic, control word register and counters 0, 1, 2.

1) Data bus buffer

- It is tristate, bi-directional 8 bit data bus buffer.
- It is used to interface 8254 data bus with system data bus.
- It is internally connected to internal data bus and its outer pins D_0 - D_7 are connected to system data bus. The direction of data buffer is decided by read and write control signals.

2) Read/Write logic

- This block accepts inputs from system control bus and address bus.
- In I/O mapped I/O, the signals $\overline{RD}$ and $\overline{WR}$ are connected to $\overline{IOR}$ and $\overline{IOW}$.
- In memory mapped I/O, $\overline{RD}$ and $\overline{WR}$, are connected to $\overline{MEMR}$ and $\overline{MEMW}$.
- A_0 and A_1 are directly connected to address lines A_0 and A_1.
- $\overline{CS}$ is connected to address decoder.
- The 8254 operation/selection is enabled/disabled by $\overline{CS}$ signal. A_0, A_1 selects a specific part $\overline{WR}$, $\overline{RD}$ decides writing data to 8254 or reading data from 8254.
- The control word registers and the counters are selected according to the signals on lines A_0 and A_1.

A_1	A_0	Selection
0	0	Counter 0
0	1	Counter 1
1	0	Counter 2
1	1	Control word register

3) Control word register

- This register of 8254 gets selected when $A_0 = 1$ and $A_1 = 1$.
- It is used to specify the BCD or binary counter to be used, its mode of operation and the data transfer to be used i.e. read or write the data bytes (LSB, MSB or both).
- If the CPU performs a write operation, the data is stored in the control word register and is referred to as Control Word. It is used to define counter operation.
- The data can only be written into control word register, no read operation is allowed. Status information is available with the help of Read Back Command.

4) Counters

- There are three independent, 16 bit down counters.
- They can be programmed separately through control word register to decide mode of counter.

- Each counter is having 2 inputs viz. CLK and GATE.
- CLK is used as an input to counter and GATE is used to control the counter.
- The counters give output on OUT pin.
- The loaded count value in counter will be decremented by counter at each clock input pulse. The programmer can read counter without disturbing counter operation.

19.5 Control Word Register Format

Q. Explain the control word register format of 8254.

- The control word register format is as shown in Fig. 19.5.1.

Note : $SC_0 = 1$ and $SC_1 = 1$, this combination is used to give read back command (Only for 8254).

i) The control word register bits, SC_1 and SC_0, select the control word register for counter. When $SC_0 = 0$ and $SC_1 = 0$, the control word for counter 0 is selected. Similarly, other counter control words, are selected by SC_0 and SC_1 and used to initialize the counters. A_0 and A_1 selects counter(s), but they are used to read/load counters by microprocessor.

ii) The bits RL_0 and RL_1 are used to read/load, data bytes i.e. LSB byte, MSB byte or both LSB and MSB bytes.

iii) The bits M_2, M_1 and M_0 decides the mode of operation for selected counter, Mode0 - Mode5.

iv) The bit, BCD, decides the mode of counting i.e. BCD counter or binary counter.

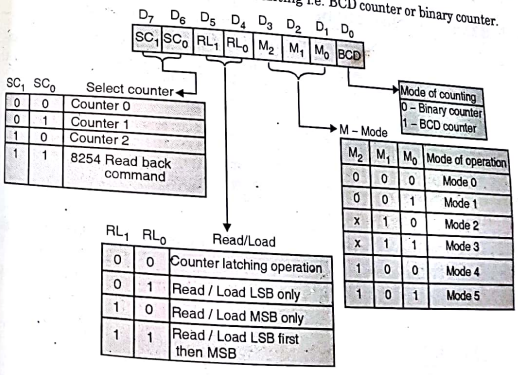


Fig. 19.5.1 : Control word register format

19.6 8254 Write Operation

The 8254 initialisation and count value loading operations are important operations performed by system software. Once 8254 is programmed it is ready to perform operations. The order of initialisation and count value loading for counter 0, counter 1 and counter 2 are sequence independent. But the number of data bytes programmed by RL_1 and RL_1 must be completed. The example format for this is as follows :

Sr. No.	A_1	A_0	
1	1	1	Control word register for counter 0
2	0	0	LSB count for counter 0
3	0	0	MSB count for counter 0
4	1	1	Control word register for counter 1
5	0	1	LSB count for counter 1
6	0	1	MSB count for counter 1
7	1	1	Control word register for counter 2
8	1	0	LSB count for counter 2
9	1	0	MSB count for counter 2

In above format, all the 3 counters are used with 16 bit count values.

Interleaved read and write

The special feature of 8254 is that the read and write operation of any counter may be interleaved. This feature allows us to read LSB byte then write LSB byte and read MSB byte then write MSB byte.

19.7 8254 Read Operations

The counter application, requires reading the value of the counter in progress. The 8254 contains logic that allows the programmer to read the contents of counters without disturbing the actual count in progress. There are 3 methods for 8254 as follows :

- Method 1 : Simple Read
- Method 2 : Counter Latch Operation
- Method 3 : Read Back Command (Only for 8254)

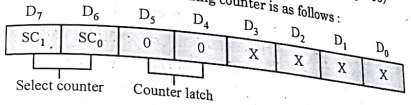
19.7.1 Method 1 : Simple Read

- The method 1 involves, the simple I/O read operations for the selected counter by using A_0, A_1 address inputs.
- The precaution should be taken that the RL_0- RL_1 bits programmed for reading the number of data bytes is followed.
- To perform the operation microprocessor issues $\overline{RD}$ control signal and takes count value from selected counter.

- If two bytes are programmed to be read, then two bytes must be read, before any write operation to the same counter.
- The requirement of this method for stable count reading is the counter should be inhibited. Because if a counter is changing its state and in the mean time when you read the contents, the count value will not be correct. To avoid such conditions the counter can be inhibited either by controlling the gate input or by external logic that inhibits the clock input. For example if the current count is 0100H. You read the lower byte and get it as 00H. Before you read the higher byte a pulse is applied on CLKn pin and the counter decrements i.e. it becomes 00FFH. Now when you read the upper byte, you get it as 00H. Hence the count you read is 0000H, which is incorrect, 0100H or 00FFH was correct.

19.7.2 Method 2 : Counter Latch Operation

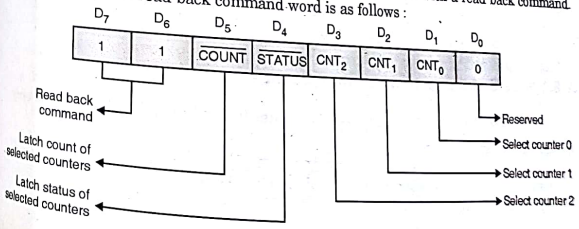
- The problem with the method 1 is, there are chances of reading incorrect count. To avoid this the 8254 provides counter latch operation. (RL_1 RL_0 = 00)
- The control word format for latching counter is as follows :



- When the above format is loaded in control register, it will latch the count value at that instant into special register.
- The contents of special register will be an accurate and stable quantity at that instant.
- The programmer then gives normal read commands to counter same as method 1 and contents of the latched register will be available. This type of reading count value contents is also called as ***reading on fly***.

19.7.3 Method 3 : Read Back Command (Only for 8254)

- In this method to get a stable count a counter is latched with a read back command. The format for read back command word is as follows :



- In above control word format, when D_7 and D_6 bits of control word registers are 1, it is a read back command. The definition of other bits also changes as follows :
  D_5 - latch count of selected counters.
  D_4 - latch status of selected counters.
  D_3 D_2 D_1 - select counter.
- The advantage of this method is, you can latch one, two or all the three counters by putting 1 in the appropriate select counter bits. Once the counters are latched, the reading is performed by simple I/O read operation same as method 1.
- The status word available, when a status is latched will be as follows :

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
Output	Null count	RL_1	RL_0	M_2	M_1	M_0	BCD

D_7 - Output : This gives the logic level of OUT pin of selected counter when bit = 0, OUT pin is LOW bit = 1, OUT pin is HIGH.

D_6 - Null count : This bit gives the status of counter. If the counter is zero, this bit will be set. If counter is not zero, this bit will be reset.

D_5 and D_4 : RL_1, RL_0 gives status of read load count value

D_3, D_2, and D_1 : M_2, M_1 and M_0 - gives status of mode

D_0 : BCD gives status of counter mode

19.8 Operating Modes of 8254

Q. Write down the names of different modes of operation of 8253 timer (PIT).

The programmable timer IC provides following modes of operations.

- Mode 0 : Interrupt on Terminal Count
- Mode 1 : Programmable One Shot / Hardware Triggerable One Shot
- Mode 2 : Rate Generator / Pulse Generator
- Mode 3 : Square Wave Generator
- Mode 4 : Software Triggered Strobe
- Mode 5 : Hardware Triggered Strobe

19.8.1 Mode 0 : Interrupt on Terminal Count

- Control word format required for mode 0, counter 0, Read / Load LSB data byte only and BCD counter will be,

0	0	0	1	0	0	0	1

= 11 H

The three cases are as shown in Fig. 19.8.1 and their details are as follows :

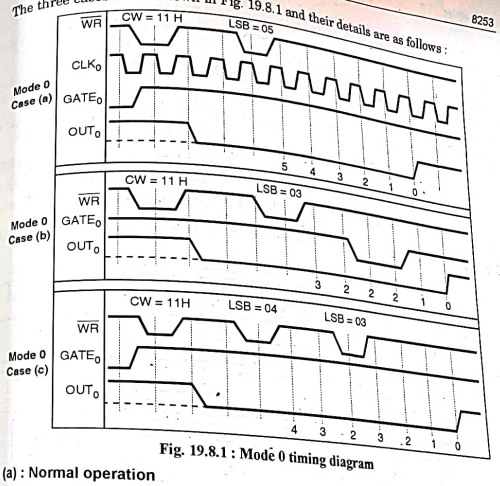


Fig. 19.8.1 : Mode 0 timing diagram

Case (a) : Normal operation

(i) The counter initialisation and loading operation is performed using two write operations. First write to load control word register for counter 0 (CWR = 11 H) and second to load count value (Data = 05 H). When counter is initialised in mode 0, the output goes LOW, at next negative going edge.

(ii) When the data 05 H is loaded, it is transferred to counter on next negative going edge of CLK_0 after $\overline{WR}$. Loading count value will start counter and counter will decrement count by 1 for each CLK_0 pulse.

(iii) When counter reaches zero the OUT_0 pin will go HIGH and remain HIGH. In case (a) we assume the status of GATE input is HIGH.

Case (b) : Gate disable

In this case counter operation is same as case (a). Only change is GATE input. If the GATE input becomes LOW the counter suspends counting. Again, when GATE becomes HIGH it will resume counting upto zero.

Case (c) : New count

In this case, counter operation is same as case (a). The only change is, loading new count value previous reaches to ZERO.

(i) If a new count value is loaded before counter reaches zero, the counter will take new count value and restart decrementing count value upto zero.

(ii) If a 16 bit counter (i.e. LSB and MSB) is used, then loading of LSBs, stops the current count and loading of MSB, restarts counter with new value.

The output in all 3 cases is made HIGH when count reaches zero. It remains HIGH until a new count value is loaded or mode of operation is changed. This output can be used as an interrupt, therefore the name given is ***Interrupt on terminal count***

19.8.2 Mode 1 : Programmable One Shot / Hardware Triggerable One Shot

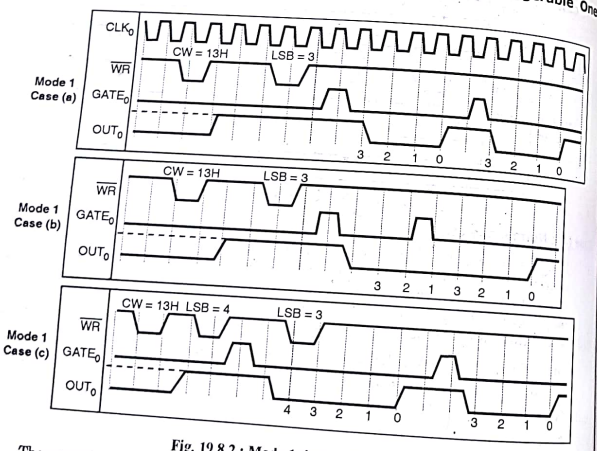


Fig. 19.8.2 : Mode 1 timing diagram

The control word format required for Mode 1, counter 0, Read / Load LSB data byte and BCD counter will be

0	0	0	1	0	0	1	1

= 13 H

The three cases are as shown in Fig. 19.8.2 and their details are as follows :

Case (a) : Normal operation

(i) The counter initialisation and loading operation is performed using two write operations. First write to load control word register for counter 0 (CWR = 13 H) and second to load count value (Data = 03 H). When the counter is initialised in mode 1 the output (OUT_0) goes HIGH.

(ii) The mode is, i.e. monostable, having HIGH as a stable state and LOW as quasistable state. The counter remains LOW for count value pulses.

(iii) The positive going edge on gate input will start counter, therefore it is called as ***trigger input***.
When the positive going edge occurs at gate input, the count value is transferred to counter. The counter output changes to LOW and counter goes on decrementing the count value by 1 on every negative going edge of CLK input. When the counter becomes zero, the output is changed back to HIGH i.e. a stable state.

(iv) When a trigger input is applied the counter will restart counting by making output low.

Case (b) :

In this case counter operation is same as case (a); the only change is two trigger pulses. If another trigger pulse is applied before the counter reaches zero, the positive edge on trigger will restart counter.

Case (c) :

In this case counter operation is same as case (a); the only change is count value loading. If new count value is loaded, before counter reaches zero. The new count value will not be considered for counter and counter will continue counting. But if a trigger pulse is applied after loading, the new count value is transferred to counter and counter will restart counting from new count value.

19.8.3 Mode 2 : Rate Generator / Pulse Generator

Q. Explain mode 2 in detail with suitable waveforms.

The control word format required for Mode 2, counter 0, Read / Load LSB data byte and BCD counter will be

0	0	0	1	×	1	0	1

= 15 H

The three cases are as shown in Fig. 19.8.3 and their details are as follows :

Case (a) : Normal operation

(i) The counter initialisation and loading operation is performed using two write operations; first write to load control word register for counter 0 (CWR = 15 H) and second write to load count value (Data = 04 H). When counter is initialised in Mode 2, the output is made HIGH.

(ii) When data is loaded, it is transferred to counter and counter starts counting. Each time, it decrements counter by 1.

(iii) When the counter reaches 1, it makes output LOW. The output remains LOW for one clock pulse.

(iv) When the counter becomes zero the output is made high, the count value is reloaded in counter and counter continues counting. In case (a) we assume the status of GATE input is HIGH.

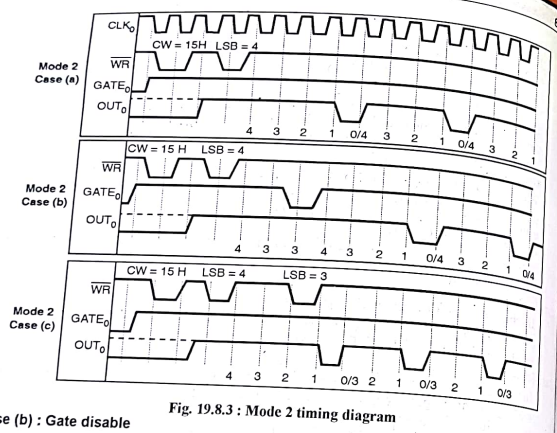


Fig. 19.8.3 : Mode 2 timing diagram

Case (b) : Gate disable

In this case counter operation is same as case (a) only change is gate input. If GATE goes LOW, the counter stops counting. If gate is again made HIGH, it reinitialises counter and count value is reloaded in counter. If GATE goes LOW when the output is low, the output is set high immediately. The trigger sequence is reloaded and normal sequence is repeated.

Case (c) : New count

In this case counter operation is same as case (a), the only change is loading the count value. If new count value is loaded, before counter reaches to zero, this new value is not considered. When counter becomes zero, the new count value is transferred to counter and counter continues with new count value. In this mode count of 1 is not allowed.

19.8.4 Mode 3 : Square Wave Generator

Q. Explain mode 3 as square wave generator with timing diagram.

The control word format required for Mode 3, counter 0, Read / Load LSB data byte and BCD counter will be,

0	0	0	1	×	1	1	1

= 17 H

The four cases are as shown in Fig. 19.8.4 and their details are as follows :

Case (a) : Normal operation with even count

(i) The counter initialisation and loading operation is performed using two write operations; first write to load control word register for counter 0 (CWR = 17 H) and second to load count value (Data = 04 H). When the counter is initialised in Mode 3, the output is made HIGH.

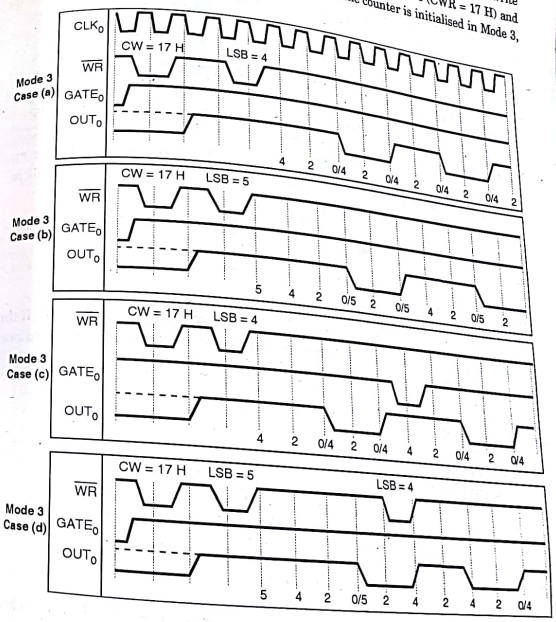


Fig. 19.8.4 : Mode 3 timing diagram

(ii) When data is loaded in counter, the counter starts counting. The Mode 3 is a square wave generator, output remains HIGH for half count CLK pulses and remains LOW for half count CLK pulses.

(iii) To implement this, the counter is decremented by 2 each time a negative going edge of clock input occurs. When the counter reaches 0, the count value is reloaded in counter. When a count value is reloaded, it changes its output level i.e. HIGH to LOW or LOW to HIGH.

(iv) If a count value is even, you get a perfect square wave output. The output frequency is decided by count value and input CLK frequency. In case (a) we assume the status of GATE input is HIGH.

Case (b) : Normal operation with odd count

(i) In this case counter operation is same as case (a) but the count value loaded is odd. When the count value is odd the counter is decremented by 1 and then decremented by 2 each time, upto zero. When the counter reaches zero, the counter changes its output to LOW and the count value is reloaded in counter.

(ii) Now the counter is decremented by 3, and then decremented by 2 each time upto zero. When the counter reaches zero, the counter changes its output to HIGH and the count value is reloaded in counter. The counter will continue counting.

(iii) The output will remain high for $\frac{N+1}{2}$ clock pulses and remain low for $\frac{N-1}{2}$ clock pulses.

Case (c) : Gate disable

In this case counter operation is same as case (a) only change is gate input. If the gate becomes LOW, the counter stops counting and makes output HIGH. When gate input becomes HIGH, the rising edge will initiate counting and the counting will continue.

Case (d) : New count

In this case counter operation is same as case (a) any change is loading the count value. If a trigger is received after writing a new count, before the end of current half cycle of square wave. The counter will be loaded with the new count on the next CLK pulse and counting will continue from the new count. Otherwise new count will loaded at the end of current half cycle.

19.8.5 Mode 4 : Software Triggered Strobe

The control word format required for mode 4, counter 0, Read/Load LSB data and BCD counter will be,

0	0	0	1	1	0	0	1

= 19 H

The three cases are as shown in Fig. 19.8.5 and their details are as follows :

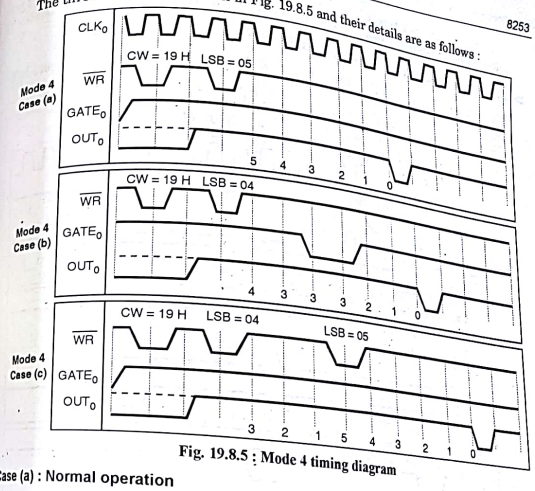


Fig. 19.8.5 : Mode 4 timing diagram

Case (a) : Normal operation

(i) The counter initialisation and loading operation is performed using two write operations; first write, to load control word register for counter 0 (CWR = 19 H) and second to load count value (Data = 05 H). When the counter is initialised in Mode 4, the output is made HIGH.

(ii) When data is loaded, it is transferred to counter and counter start counting. Each time, it decrements counter by 1. When the counter reaches zero, it changes output to LOW.

(iii) The output remains LOW for 1 clock pulse and again the output is made HIGH, In case (a) we are assuming the status of gate input is HIGH.

Case (b) : Gate disable

In this case counter operation is same as case (a) only change is gate input. If the GATE is made LOW, it will suspend the counting. Again when gate is made HIGH the counting will resume.

Case (c) : New count

In this case counter operation is same as case (a) only change is count value loading. If new count value is loaded, the new count value is reloaded in counter on next negative going edge of clock input. The counter will count from new count value

upto zero. The resetting of counter is done by loading new count value. There is no resetting available by using gate input so name given is software triggered strobe.

19.8.6 Mode 5 : Hardware Triggered Strobe

The control word format required for mode 5, counter 0, Read / Load LSB and BCD counter will be,

0	0	0	1	1	0	1	1

= 1BH

The three cases are as shown in Fig. 19.8.6 and their details are as follows :

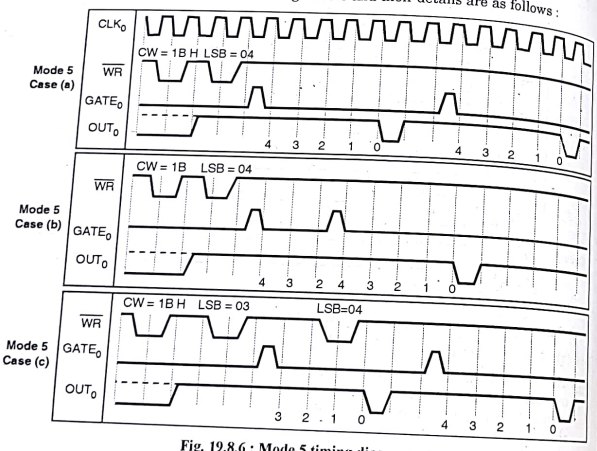


Fig. 19.8.6 : Mode 5 timing diagram

Case (a) : Normal operation

(i) The counter initialisation and loading operation is performed using two write operations; first write to load control word register for counter 0 (CWR = 1B H) and second to load count value (Data = 04 H). When counter is initialised in mode 5, the output is made HIGH.

(ii) To start the counter, a positive going edge on GATE input is required. When it is applied, count value is loaded in counter on next negative going edge of clock input and counter starts counting.

(iii) When the counter reaches zero, the output is made LOW for 1 clock pulse and again it is made HIGH. If another pulse at gate input is applied, the same process will be repeated.

Case (b) : Retriggering

- In this case counter operation is same as case (a) only change is gate input. If another trigger pulse is applied before the counter reaches zero the count value is reloaded and counter will restart from count value. When counter reaches zero, the output is made LOW for 1 clock pulse and again made HIGH.

Case (c) : New count

- In this case counter operation is same as case (a) only change is count value loading. If a new count value is loaded that value is not considered for counting. But if trigger input is applied after that, the new count value is loaded in counter and counter will start counting.

In all 3 cases the control is given to GATE input only so the name given is hardware triggered strobe.

Gate / Signal status / modes	Low or going low	Rising	High
0	Disables counting	–	Enables counting
1	–	i) Initiates counting ii) Resets output after next clock	–
2	i) Disables counting ii) Sets output immediately high	Initiates counting	Enables counting
3	i) Disables counting ii) Sets output immediately high	Initiates counting	Enables counting
4	Disables counting	–	Enables counting
5	–	Initiates counting	–

19.9 Interfacing 8254/8253 with 8086

To interface 8254 or 8253 with 8086, we need to do the following connections

1) Data lines D_0 - D_7 of 8253 are to be connected to data lines D_0 to D_7 (or D_8 to D_{15}) of 8086
2) A_0 and A_1 of 8253 / 8254 are taken from A_1 and A_2 of 8086
3) Control signals like $\overline{RD}$, $\overline{WR}$ and $\overline{CS}$ are to be connected.

Fig. 19.9.1 shows interfacing of 8253 / 8254 with 8086 in maximum mode. Similar connection can be done in minimum mode.

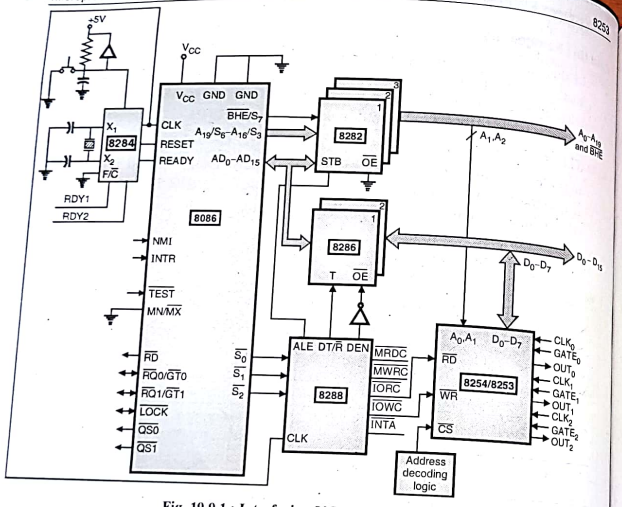


Fig. 19.9.1 : Interfacing 8253 / 8254 with 8086

19.10 Interfacing Examples with 8086

Ex. 19.10.1 : For given Fig. P. 19.10.1 determine the input frequency for counter 2 and time period after which output changes its state if counter 2 is initialised with 40,000 count. Write a small routine, which will initialise 8254 for following cases.

I : IO mapped IO, Indirect address 5000 H of 8254 (2)

II : IO mapped IO, direct address 50 H of 8254 (2)

III : Memory mapped IO, address 80000 H of 8254 (2).

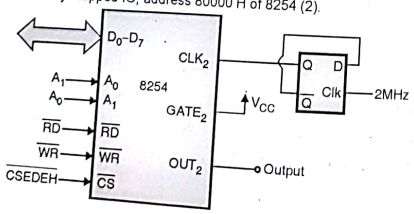


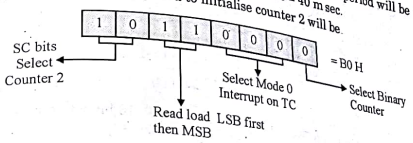
Fig. P. 19.10.1

Soln. :

Step 1 : The DFF will divide the input frequency 2 MHz by 2. So the input frequency to counter 2 will be 1 MHz.

Step 2 : If we load (40,000)10 count in counter 2. The total time period will be Count x CLK period $40,000 \times 1\ \mu\text{sec} = 40,000\ \mu\text{sec} = 40$ m sec.

Step 3 : The control word required to initialise counter 2 will be.



Step 4 : $(40,000)_{10}$ corresponds to 9C40 H

Step 5 : Initialisation

Label	Instruction	Comments
CASE I :	Indirect mode of addressing in IO mapped IO.	
	MOV AL, B0 H	
	MOV DX, 5006 H	CWR to 8254 (2)
	OUT DX, AL	
	MOV DX, 5002 H	
	MOV AL, 40 H	Load counter 2 address
	OUT DX, AL	Load Lower byte
	MOV AL, 9C H	
	OUT DX, AL	Load MSB
CASE II :	Direct addressing in IO mapped IO.	
	MOV AL, B0 H	
	OUT 56 H, AL	CWR
	MOV AL, 40 H	Load LSB
	OUT 52 H, AL	
	MOV AL, 9C H	Load MSB
	OUT 52 H, AL	
CASE III :		
	MOV AX, 8000 H	Initialize DS with 8000H
	MOV DS, AX	
	MOV BYTE PTR [0006 H], 0B0 H	Load CWR
	MOV BYTE PTR [0002 H], 40 H	Load LSB
	MOV BYTE PTR [0002 H], 9C H	Load MSB

Ex. 19.10.2 : *Explain the 8254 control word format and set up the 8254 as a square wave generator with 1 ms period if the input frequency to the 8254 is 1 MHz.*

Soln. :

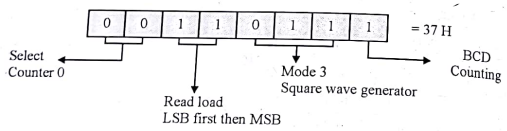
Step 1 : Assume the Counter 0 is used to generate square wave and addresses of Counter 0 = 5000H and Control register = 5006 H

Step 2 : The output period required is 1 ms and the input frequency is 1 MHz so input period = 1 ms

$$\text{Count value} = \frac{\text{Required period}}{\text{Input period}}$$

$$= \frac{1 \text{ ms}}{1 \mu \text{s}} = (1000)_{10}$$

Step 3 : The control word format to initialise Counter 0, 16 bit counter, BCD counting and square wave generator mode will be as follows :



Step 4 : The 8254 initialisation program will be as follows. In this case as the count value is (1000), it is less than 16 bits so we can use it as count value in BCD without converting,

Label	Instruction	Comments
CASE I :	MOV AL, 37H	Counter 0, mode 3 CWR
	MOV DX, 5006	CWR address
	OUT DX, AL	
	MOV DX, 5000H	Counter 0 address
	MOV AL, 00H	
	OUT DX, AL	LSB first
	MOV AL, 10H	
	OUT DX, AL	MSB Next
CASE II :	MOV AL, 37H	CWR
	OUT 56H, AL	
	MOV AL, 0H	LSB
	OUT 50H, AL	
	MOV AL, 10H	MSB
	OUT 50H, AL	Counter 0

Label	Instruction	Comments
CASE III :	MOV AX, 8000H	
	MOV DS, AX	Init D.S
	MOV BYTE PTR [0006 H], 37H	
	MOV BYTE PTR [0H],0H	CWR
	MOV BYTE PTR [0H],10H	LSB
		MSB

Ex. 19.10.3 : *Design a pulse train generator for a pulse train of frequency 1 kHz and duty cycle of the pulse 25 % using 8254. Assume suitable clock frequency.*

Soln. :

Step 1 : Duty cycle of pulse required is 25 % i.e. if 4 parts of wave forms are there it should remain high for 1 part and low for remaining 3 parts i.e. remain high for 1 clock pulse and low for 3 clock pulse, this can be implemented by using mode 2 rate generator and if you wish an inverter at OUT pin will give you the 25 % duty cycle output.

Step 2 : We want 1 kHz output frequency and 4 count pulses to get 25% duty cycle so input frequency will be selected 4 kHz. The counter is used as shown in Fig. P. 19.10.3

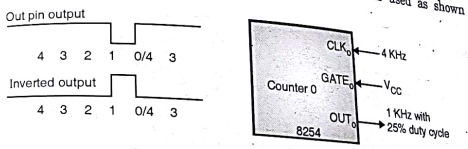
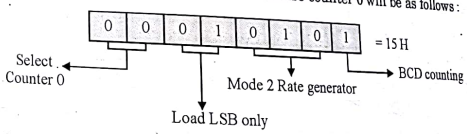


Fig. P. 19.10.3

Step 3 : The control register required to initialise counter 0 will be as follows :



Initialisation program will be as follows :

Instruction	Operation
MOV AL,15H	CWR
OUT 56H, AL	
MOV AL, 04H	Load LSB count value to Counter 0
OUT 50H, AL	

Ex. 19.10.4 : *Design a programmable timer using 8253 and 8086. Interface 8253 at an address 0040H for counter 0. Draw interfacing diagram. Operating frequency for 8086 and 8253 is 6 MHz and 1.5 MHz respectively. Write ALP to generate square wave of 1 ms. Show control word formation and delay calculations clearly.*

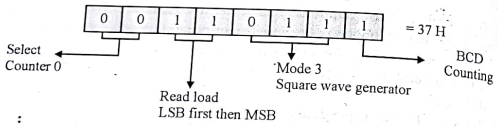
Soln. :

Step 1 : Assume the counter 0 is used to generate the square wave.

Step 2 : The output period required is 1 ms. The input frequency of 8253 is 1.5 MHz

$$\therefore \text{ Count value } = \frac{\text{Required period}}{\text{Input period}} = \frac{1 \text{ ms}}{0.667 \ \mu s} = 1500$$

Step 3 : The control word format



Step 4 :

Program :

Instruction	Comment
MOV AL, 37H	
MOV DX, 0046H	CWR address
OUT DX, AL	
MOV DX, 0040H	Counter 0 address
MOV AL, 00H	
OUT DX, AL	
MOV AL, 10H	
OUT DX, AL	

For interfacing diagram refer section 19.9.

Ex. 19.10.5 : *Design a system using 8253 to generate a square wave of 10 KHz and write corresponding program.*

Soln. :

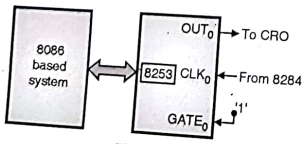


Fig. P. 19.10.5

Crystal frequency = 15 MHz

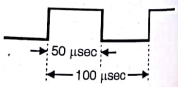


Fig. P. 19.10.5(a)

Operating frequency = 5 MHz

1 clock pulse $= \frac{1}{5}$ μsec = 0.2 μsec

Delay required = 100 μsec

$\therefore$ Total clock pulse to be counted $= \frac{100 \ \mu sec}{0.2 \ \mu sec} = (500)_{10} = (01F3)_{16}$

Control word

SC1	SC0	RW1	RW0	M_1	M_2	M_0	BCD
0	0	1	1	0	1	1	0

⇒ 36 H

3 6

Assume count 0 address = 80H

CW address = 80H

Program :

Instruction	Comments
MOV AL, 36H	Initializing CW
OUT 86H, AL	Initializing CW
MOV AL, F3H	Moving LSB
OUT 80, AL	Moving LSB
MOV AL, 01H	Moving MSB
OUT 80H, AL	Moving MSB

19.11 Interfacing of 8253/8254 with 8085

Fig. 19.11.1 shows 8253/8254 interfacing with 8085.

- 8253/54 have separate address and data lines. Hence, it is essential to demultiplex the signals $AD_0 - AD_7$.
- Demultiplexing is done with the help of an external decoder and ALE signal.
- 8253/54 decodes the A_0 and A_1 lines internally, inorder to select one of its ports or the control register.
- In I/O mapped I/O mode, the higher order address bus $A_{15} - A_8$ duplicates the lower address bus i.e. $A_7 - A_0$.

  Hence, the lines A_8 and A_9 are indirectly involved in selection of control register or ports. The remaining address lines $A_7 - A_2$ or $A_{15} - A_{10}$ can be used to generate the chip select signal.
- Decoder 74LS138 is used to generate the chip select signal. Another decoder can be used to generate $\overline{IOR}$, $\overline{IOW}$, $\overline{MEMW}$ signals from $\overline{RD}$, $\overline{WR}$ and IO / $\overline{M}$ signals.

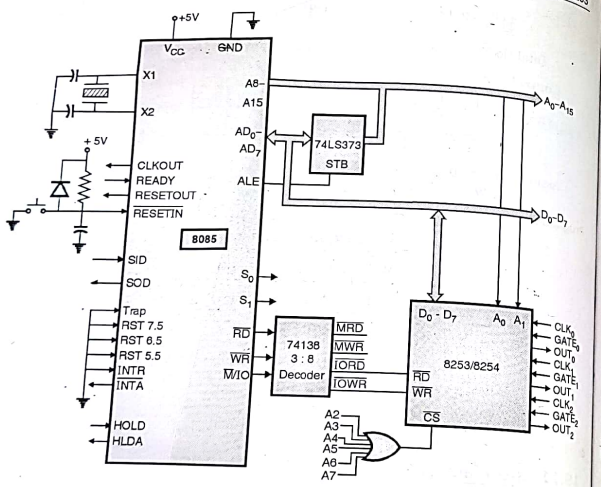


Fig. 19.11.1 : Interfacing 8253/8254

Address map :

Ports / Control Register	Address line								Address
	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0	
Counter 0	0	0	0	0	0	0	0	0	00 H
Counter 1	0	0	0	0	0	0	0	1	01 H
Counter 2	0	0	0	0	0	0	1	0	02 H
Counter 2	0	0	0	0	0	0	1	1	03 H

19.12 Interfacing Examples with 8085

Ex. 19.12.1 : Explain the 8253 control word format and set up the 8253 as a square wave generator with 1 ms period if the input frequency to the 8253 is 1 MHz.

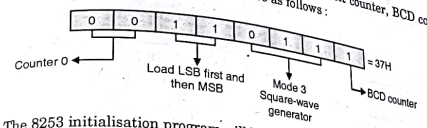
Soln. :

Step 1 : Assume the Counter 0 is used to generate square wave and addresses of Counter 0 = 10 H and Control register = 13 H.

Step 2 : The output period required is 1 ms. The input frequency is 1 MHz, so input period = 1 s

$$\text{Count value} = \frac{\text{Required period}}{\text{Input period}} = \frac{1\text{ ms}}{1\,\mu\text{s}} = (1000)_{10}$$

Step 3 : The control word format to initialise Counter 0, 16 bit counter, BCD counting and square wave generator mode will be as follows :



Step 4 : The 8253 initialisation program will be as follows :

Instruction	Comment
MVI A,37H	Initialise counter 0, mode 3,
OUT 13H	16 bit count and BCD counter
MVI A,00H	Load LSB count value to counter 0
OUT 10H	
MVI A,10H	Load MSB count value to counter 0
OUT 10H	

Ex. 19.12.2 : Interface 8253 to 8085 so as to generate interrupt which will be received at RST 6.5. Configure 8253 to generate interrupt at some known interval and use it to drive a clock which will be displayed at address and data field in the HH : MM : SS format. Show all calculations clearly. Assume the display subroutines for address and data fields are directly available such that the HL pair contents are displayed in address field and A register contents are displayed in data field. Assume the additional data if required.

Soln. :

Step 1 : The 8253 is used as a timer. The output of 8253 is connected as interrupt to 8085.

Step 2 : The 8253 should generate interrupt at fixed interval for example after every 1 second. So the mode which provides reloading facility should be used. The output of the mode should remain high for very less time and should remain low for maximum time. So the 8253, mode 2 can be used for this function but the output should be inverted and used so as to meet the requirements of system.

Step 3 : To decide the count value for counter,

$$\text{Count} = \frac{\text{Required period}}{\text{Input period}}$$

Assume input frequency = 1 MHz, $\text{Count} = \frac{1\text{sec}}{1\,\mu\text{s}} = 1000000$

Note : If 2 MHz frequency then the count will be 2000000.

Step 4 : The count value is much large than 16 bits even though we convert to hex. So two methods can be used.

Method 1 : Using One Counter :

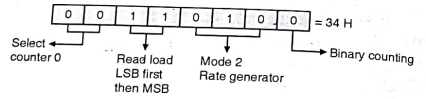
Step 5 : Assume counter is loaded with count value 50000. It will interrupt 8085 after every 50 ms. To use this to count seconds we must use 8085 counter, when 20 such interrupts are given we have to increment seconds by one.

Note : For 2 MHz, 40 interrupts will be given.

Step 6 : The interfacing diagram is as shown in Fig. P. 19.12.2(a)

The addresses of counter will be counter 0 = 60 H, counter 1 = 61 H, counter 2 = 62 H and control register = 63 H.

Step 7 : To initialise control register for counter 0 and with count value 50000, first convert 50000 to hex. $(50000)_{10} = C350$ H.



Control register = 34 H; Count value = C350 H.

Step 8 : The flowchart for clock program is shown in Fig. P. 19.12.2(b).

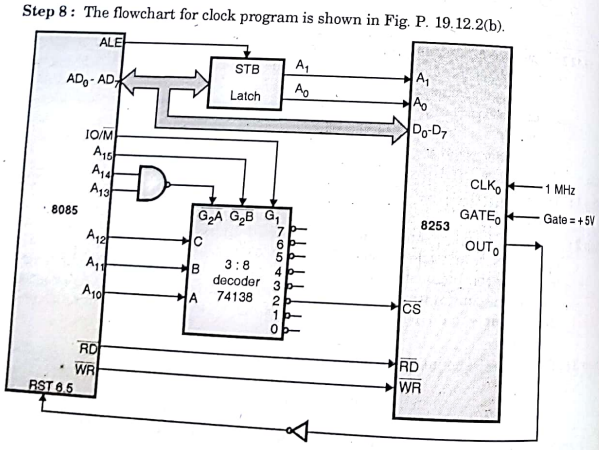


Fig. P. 19.12.2(a)

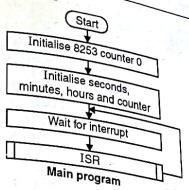


Fig. P. 19.12.2(b)

Step 9 : The program for above method 1 will be as follows :

Main Program :

Label	Instructions	Comments
	LXI SP,FFFFH	Initialise stack pointer
	MVI A,34H	Initialise counter 0 in
	OUT 63H	generator mode.
	MVI A,50H	Load LSB count value
	OUT 60H	in counter 0
	MVI A,C3H	Load MSB count value
	OUT 60H	in counter 0.
	LXI H,0000H	Initialise minutes and hours
	MVI B,00H	Initialise seconds.
	MVI C,00H	Initialise counter to count interrupts.
	MVI A,1DH	Enable RST 6.5
	SIM	Mask RST 5.5 and RST 7.5
UP :	EI	Enable interrupts
here :	JMP here	Wait for interrupt
	JMP UP	Go back to enable interrupts.

Interrupt Service Routine :

	INR C	Increment counter = counter + 1
	MOV A,C	
	CPI 14H	Is (20)10 interrupts occurred
	RNZ	If no, go back to main program
	MVI C,00H	Clear counter.

8253

Label	Instructions	Comments
	MOV A,B	Seconds = seconds + 1
	ADI 01H	
	DAA	Convert to BCD
	MOV B, A	
	CPI 60H	Are 60 seconds completed
	JNZ DISPLAY	If no, go to display
	MVI B,00H	Clear seconds
	MOV A,L	Minutes = minutes + 1
	ADI 01H	
	DAA	Convert to BCD
	MOV L,A	
	CPI 60H	Are 60 minutes completed
	JNZ DISPLAY	If not, go to display
	MVI L,00H	Clears minutes
	MOV A,H	Hours = hours + 1
	ADI 01H	
	DAA	Convert to BCD
	MOV H, A	
	CPI 24	Are 24 hours completed
	JNZ DISPLAY	If not, go to display
	MVI H,00H	Clear hours
DISPLAY:	MOV A, B	Take seconds to A reg.
	PUSH H	Store contents on to stack
	PUSH PSW	
	CALL DISPLAY1	Display H and L in address field
	POP PSW	Take and store back the contents on to stack
	PUSH PSW	
	CALL DISPLAY2	Display A reg in data field
	POP PSW	Take back the contents from stack.
	POP H	
	RET.	Go back to main program

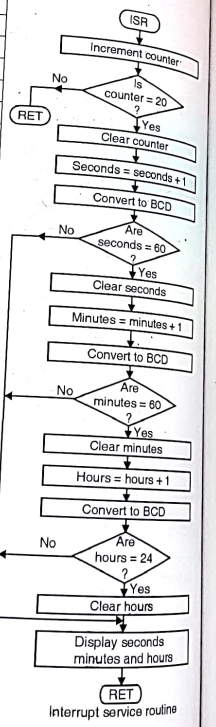


Fig. P. 19.12.2(c)

8253

Method 2 : Using Two Counters :

Step 5 : Instead of using single counter, two counters can be used in cascade as shown in Fig. P. 19.12.2(d). The input 1 MHz is connected as clock to one counter and output of first is used as input to another. The output of. second is used as interrupt signal.

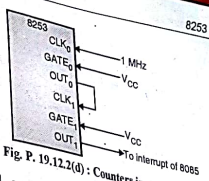


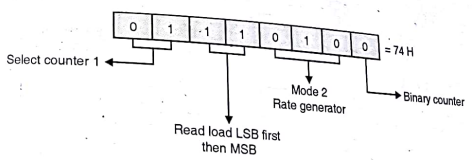
Fig. P. 19.12.2(d) : Counters in cascade mode

Step 6 : The Counter 1 will be initialised same as method 1 i.e. mode 2 and count = 50000. The Counter 0 will be initialised in any one mode, mode 2 or mode 3 i.e. rate generator or square wave generator. The count value loaded will be $(0020)_{10}$. Now 8085 will get interrupt after every second.

Step 7 : Flowchart for clock program will be as shown in Fig. P. 19.12.2(e).

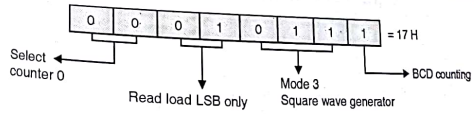
Step 8 : The interfacing diagram remains same as shown in Fig. P. 19.12.3(b) with some changes which are shown in Fig. P. 19.12.2(d). The addresses of 8253 remain same 60 H to 63 H.

Step 9 : To initialise control register for Counter 1 and with count value 50000. The bits selected will be as follows :



Control register = 74 H; Count value = C350 H.

To initialise Counter 0 with count value 20 H. The bits selected will be as follows :



Control register = 17 H and count value = 20 H

Step 10 : The program for method 2 will be as follows :

Main Program :

Label	Instructions	Comments
	LXI SP,FFFFH	Initialise stack pointer.
	MVI A,34H	Initialise counter 1, as rate generator
	OUT 63H	
	MVI A,50H	Load counter 1 with LSB count value.
	OUT 61H	
	MVI A,C3H	Load counter 1 with MSB count value.
	OUT 61H	
	MVI A,17H	Initialise counter 0 as square wave generator
	OUT 63H	
	MVI A,20H	Load LSB count value in counter 0.
	OUT 60H	
	LXI H,0000H	Initialise minutes and hours
	MVI B,00H	Initialise seconds.
	MVI A,1DH	
	SIM	Enable RST 6.5, Mask 7.5 and 5.5.
UP:	EI	Enable interrupts
here:	JMP here	Wait for interrupt
	JMP UP	Go back to enable interrupts

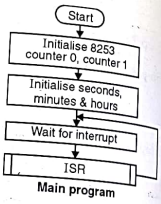


Fig. P. 19.12.2 (e)

Interrupt Service Routine :

Instructions	Comments
MOV A,B	
PUSH H	
PUSH PSW	
CALL DISPLAY1	Display H and L in address field
POP PSW	
PUSH PSW	
CALL DISPLAY2	Display A in data field
POP PSW	
POP H	
ADI 01H	Secs = secs + 1

Instructions	Comments
DAA	Adjust to BCD
MOV B, A	
CPI 60H	Check if 60 seconds are completed
RNZ	If not go back to main program
MVI B,00H	Secs. = 00
MOV A, L	
ADI 01H	Minutes = minutes + 1
DAA	Adjust to BCD
MOV L, A	
CPI 60H	Check if 60 minutes are completed
RNZ	If not, go back to main program
MVI L, 00	Minutes = 00
MOV A, H	
ADI 01H	Hours = hours + 1
DAA	Adjust to BCD
MOV H, A	
CPI 24	Check if 24 hours are completed
RNZ	If not, go back to main program
MVI H,00H	Hours = 00
RET	Go back to main program

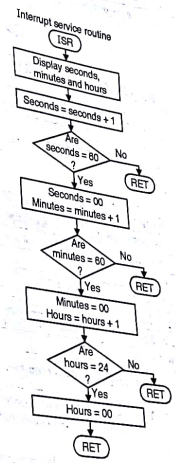


Fig. P. 19.12.2(f) : Interrupt service routine

Ex. 19.12.3 : Write a program to generate square wave of 1 KHz assuming that a 2 MHz clock is available and the 8253 registers are mapped on the I/O address space of 8085 at locations B0, B1, B2 and B3. Use BCD counting.

Soln. :

Step 1 : Assume that counter 0 is used.

Here, address B0 = Counter 0
B1 = Counter 1
B2 = Counter 2
and B3 = Control register

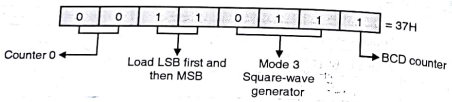
Step 2 : To generate output frequency of 1 KHz with 2 MHz microprocessor frequency.

So the count value will be

$$\frac{\text{Clock frequency}}{\text{Frequency}} \text{ or } \frac{\text{output period}}{\text{clock period}} = \frac{2\text{MHz}}{1\text{KHz}} = (2000)_{10}$$

$$\text{or} \quad \frac{1\text{ms}}{1\mu\text{s}} = (2000)_{10}$$

Step 3 : Control word will be :



Step 4 : The initialization program for 8253 will be :

Instruction	Comments
MVI A,37H	Load control word
OUT B3H	
MVI A,00H	Load LSB
OUT B0H	
MVI A,20	Load MSB
OUT B0H	

Note that this is BCD counting.

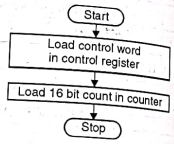


Fig. P. 19.12.3

Ex. 19.12.4 : Design a pulse train generator for a pulse train of frequency 1 kHz and duty cycle of the pulse 25 % using 8254. Assume suitable clock frequency.

Soln. :

Step 1 : Duty cycle of pulse required is 25 % i.e. if 4 parts of wave forms are there it should remain high for 1 part and low for remaining 3 parts i.e. remain high for 1 clock pulse and low for 3 clock pulse, this can be implemented by using mode 2 rate generator and if you wish an inverter at OUT pin will give you the 25 % duty cycle output.

Step 2 : We want 1 kHz output frequency and 4 count pulses to get 25% duty cycle so input frequency will be selected 4 kHz. The counter is used as shown in Fig. P. 19.12.4.

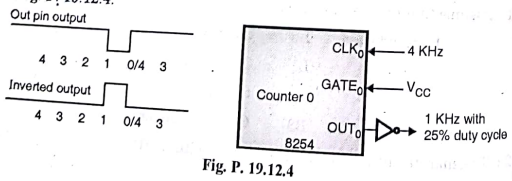
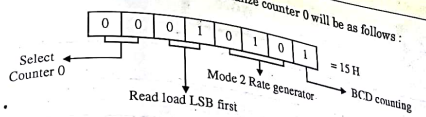


Fig. P. 19.12.4

Step 3 : The control register required to initialize counter 0 will be as follows :



Initialization program will be as follows :

Instruction	Comments
MVI A,15H	CWR
OUT 56H	
MVI A, 04H	Load LSB count value to Counter 0
OUT 50H	

□□□

Note

CHAPTER 20

Interfacing of Data Converters

20.1 Introduction

- Most of the physical quantities such as temperature, pressure, displacement, vibrations etc. are available in analog form. These quantities are represented accurately in analog form but it is difficult to process, store or transmit the analog signal because error gets introduced easily, due to noise.
- Hence to reduce these errors it is always better to express these physical quantities in the digital form.
- The digital representation of a signal makes storage possible, processing simpler and transmission easier.
- Therefore A to D conversion is necessary. Now once the processing, transmission etc. is done the signal should be brought back to its analog form, for which the D to A conversion is essential.
- Both ADC and DAC circuits are called as data converters and they are available in the IC form.

Fig. 20.1.1 shows a A/D and D/A converter application.

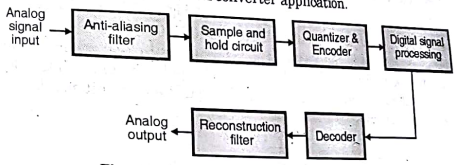


Fig. 20.1.1 : A/D and D/A converter application.

- As shown in the figure the A/D conversion involves bandlimiting the signal, sampling the signal, quantizing it and encoding it into a suitable digital format before transmission. Once the signal is transmitted, it is received and converted back to analog form by the decoder and the reconstruction filter.

20.2 ADC 0801

Q. *Explain in detail ADC 0801 with functional block diagram.*

The ADC 0801 series consists of ADC 0802, ADC 0803, ADC 0804 and ADC 0805, they are CMOS 8 bit ADCs manufactured by the National Semiconductor corporation. It is a 20-pin IC available in dual is line DTP package. It uses successive approximation technique.

The important features of ADC 0801 are as follows :

20.2.1 Features of ADC 0801

(1) Resolution 8 bit.
(2) Conversion time 100 µs.
(3) No zero adjustment required.
(4) On chip clock generator.
(5) Easy interface to all microprocessors.
(6) Operates on single 5 V supply.
(7) The logic inputs and outputs match MOS and TTL voltage level specifications.

20.2.2 Pin Configuration and Functional Block Diagram

- The pin configuration and functional block diagram of ADC 0801 are as shown in Fig. 20.2.1 and Fig. 20.2.2 respectively.
- As shown in Fig. 20.2.2 the ADC contains a circuit equivalent of the 256 R network.
- The analog switches are sequenced by the successive approximation logic to match the analog difference input voltage $[V_{IN}(+) - V_{IN}(-)]$ to a corresponding tap on the R network.
- The most significant bit is tested first and after 8 comparisons an 8 bit binary code is transferred to the output latch and then an interrupt is asserted.

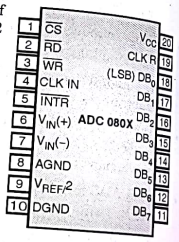


Fig. 20.2.1 : Pin diagram of ADC 0801

- The range of clock frequency is 100 KHz to 800 KHz. The clock may be supplied from an external source or it can be generated by internal clock generator. In case the clock is to be taken from the CPU clock then it is applied to pin 4. If the clock is to be generated by the internal clock generator, then an R – C circuit should be connected to pin 4 and pin 19. The clock frequency will be,

$$f = \frac{1}{1.1\,RC}$$

The value of R varies between 10 K Ω – 50 K Ω.

- For handling the analog input voltage, there are two pins, pin 6 and pin 7. If the input voltage has positive values then the input is to be applied to pin 6 and pin 7 is to grounded. If the input voltage has negative values then the input is to be applied to pin 7 and pin 6 is to be grounded. If the differential input is to be applied both the input pins 6 and 7 are used.

- Pin 9 is for V_{REF}, if it is left open, V_{REF} is equal to supply voltage V_{CC}. If the maximum analog input of + 5 V is required, + 2.5 V is applied to pin 9.
- There are two ground pins, pin 8 and pin 10, both are grounded.

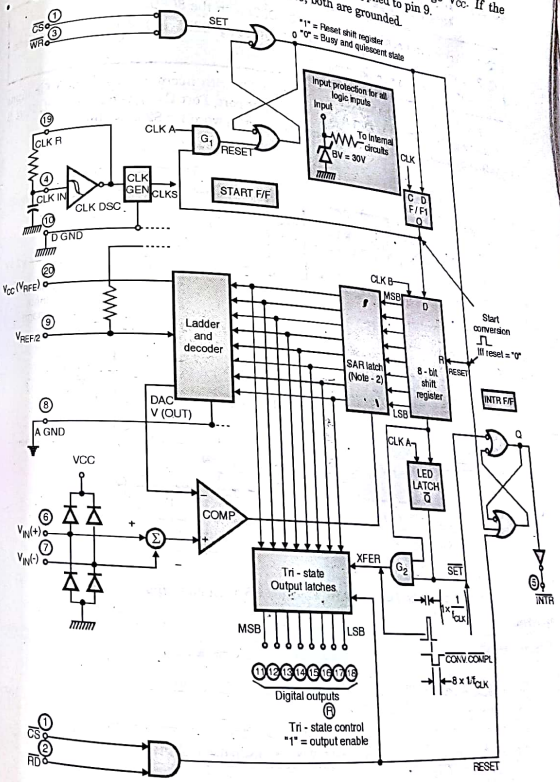


Fig. 20.2.2 : Block diagram

20.2.3 Interfacing ADC 0801 to Microprocessor 8085 for ±5 V Analog Input Voltage

Q. 1 Write short note on interfacing ADC 0801 with the 8085.
Q. 2 Interface 8085 MPU with ADC 0801 using 8255 PPI (Write appropriate assembly code to read from ADC)

Fig. 20.2.3 shows the interfacing of ADC 0804 with microprocessor 8085 for input voltage range of – 5 to + 5 V. Port A is used as input port, Port C_{lower} as output port, Port B as output port and Port C_{upper} as input port. The control word for 8255 is thus, 98 H.

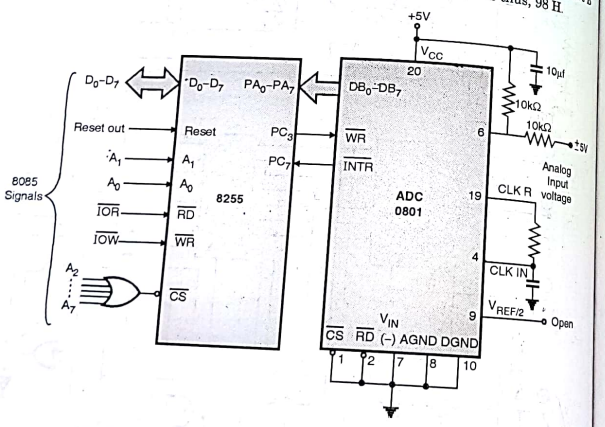


Fig. 20.2.3 : Interfacing of ADC 0801 for ± 5 V analog input voltage

Program

Label	Instruction	Comments
	MVI A, 98 H	Initialise 8255
	OUT 0BH	
	MVI A, 00 H	Send start of conversion pulse to ADC
	OUT 0A H	

Label	Instruction	Comments
	MVI A, 08 H	
	OUT 0A H	
L1 :	IN 0A H	Check for end of conversion
	RAL	Is conversion complete, if not continue
	JC L1	
	IN 08 H	
	STA C00FH	Store the digital value
	HLT	Stop

Result : On applying different analog input voltages, the corresponding digital values can be observed.

Analog Input Voltage	Digital Value
5 V	FF H
4 V	ED H
3 V	C7 H
2 V	B9 H
1 V	93 H
0 V	80 H
– 1 V	64 H
– 2 V	4A H
– 3 V	30 H
– 4 V	1C H
– 5 V	00 H

20.2.4 Interfacing ADC 0804 for Differential Analog Input Voltage

Fig. 20.2.4 shows the interfacing of ADC 0804 for differential analog input voltage.

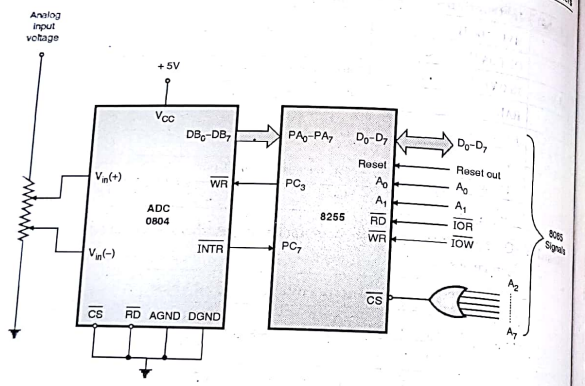


Fig. 20.2.4 : Interfacing ADC 0804 for differential analog input voltage

It was observed that if negative voltage is applied to Vin (–), the circuit did not work. The circuit only works when V_{in} (+) and V_{in} (–) are both positive and V_{in} (+) is greater than V_{in} (–).

Result : On applying different analog input voltages, the corresponding digital values can be observed.

V_{in} (+)	V_{in} (–)	Difference	Digital Output
5 V	1 V	4 V	CE H
5 V	2 V	3 V	9A H
5 V	3 V	2 V	6D H
5 V	4 V	1 V	34 H
4 V	1 V	3 V	9A H
4 V	2 V	2 V	6D H
4 V	3 V	1 V	34 H
3 V	1 V	2 V	6D H
3 V	2 V	1 V	34 H
2 V	1 V	1 V	34 H

20.3 Study and Interfacing of ADC 0808 / 0809

Q. Write short note on : ADC 0809.

20.3.1 Introduction To Monolithic IC ADC 0808 / 0809

- A large number of ADC ICs have been produced by the manufacturing companies like National semiconductors, Motorola, Intersil etc. to meet various demands such as speed of response resolution, compatibility and ease of interfacing with microprocessors etc.
- The National semiconductor produces ADC 0809 which is an 8-bit ADC whereas Intersil produces IC. ICL 7109 which is a 12-bit ADC. Let us discuss the ADC 0809 in details.

20.3.1(A) Principle of A to D Conversion in ADC 0809

- The ADC 0809 operates on the successive approximation technique of A to D conversion.
- It is a CMOS device with 8-analog inputs, an 8 channel multiplexer and microprocessor compatible control logic.
- As the number of bits n = 8, it includes a 256 resistor voltage divider, a group of analog switches and a successive approximation register (SAR).
- As there are 8-analog channels, we can connect upto 8 analog inputs to this IC.
- However due to the use of a multiplexer, at a time only one analog input will be converted into an equivalent 8-bit digital output. The analog input channels can be selected using the three address lines A, B and C.

20.3.1(B) Features of ADC 0809

- Inbuilt 8 analog channels with multiplexer.
- Zero or Full scale adjustment is not required.
- 0 to 5 V input voltage range with a single polarity 5 V supply.
- Output is TTL compatible.
- High speed.
- Low conversion time (100 μs).
- High accuracy.
- 8-bit resolution.
- Low power consumption (less than 15 mW).
- Easy to interface with all microprocessors.
- Minimum temperature dependence.

20.3.1(C) Pin Configuration of ADC 0809

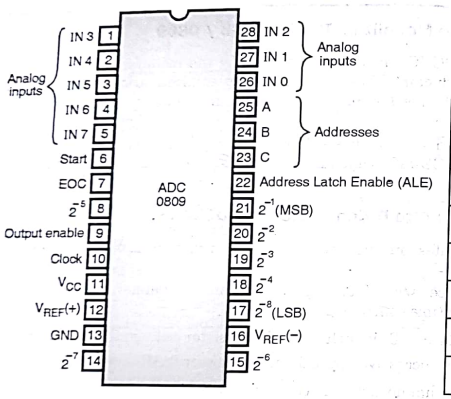


Fig. 20.3.1 : Pin configuration of IC ADC 0809

Table 20.3.1 : Selection of one of the analog inputs using the address lines A, B and C

Selected analog channel	Address		
	C	B	A
IN 0	0	0	0
IN 1	0	0	1
IN 2	0	1	0
IN 3	0	1	1
IN 4	1	0	0
IN 5	1	0	1
IN 6	1	1	0
IN 7	1	1	1

20.3.1(D) Description of ADC 0809

(i) Analog Inputs (IN 0 to IN 7)

Pin numbers 1 to 5 and 26 to 28, designated as IN 0 to IN 7 are the eight analog inputs of this IC. We can connect signals coming from eight different transducers to these inputs. Each one of these inputs will be converted to an 8-bit equivalent digital (binary word). However these inputs are converted into digital form one by one and not all at a time. So one of these eight inputs should be selected for conversion. This selection is done by means of the address pins A, B and C.

(ii) Address Pins A, B, C (Pin 23, 24, 25)

These pins will decide or select one out of the eight analog inputs, for conversion into digital form. For example if CBA = 010 then the "IN.2" is selected and the analog signal at this input is converted to equivalent digital form.

(iii) Reference Voltage [V_{REF} (+) and V_{REF} (–)]

Depending on the desired polarity of the reference voltage, we can connect a positive or negative reference voltage externally to these pins.

(iv) ALE and Output Enable

As shown in the functional block diagram of ADC 0809 (Fig. 20.3.2), the address latch enable (ALE) input is useful in enabling the address latch which stores the

address on lines A, B and C. The output enable pin, when activated will make the digital output available on the output pins, (2^{-1} to 2^{-8}).

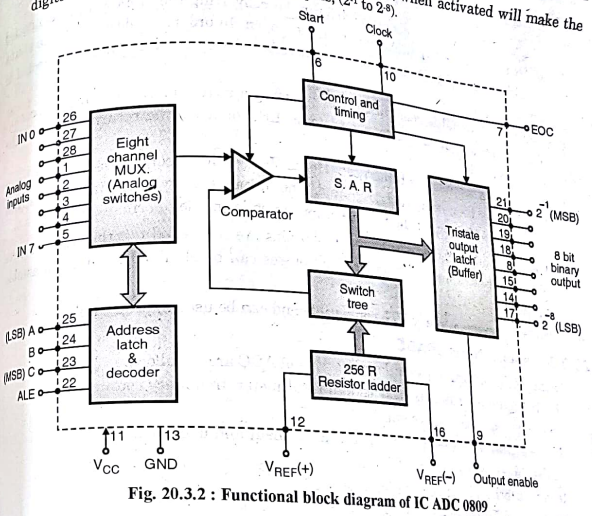


Fig. 20.3.2 : Functional block diagram of IC ADC 0809

(v) Start and EOC

As explained earlier, we have to enable the start input to begin the A to D conversion. The end of conversion is indicated by EOC output.

(vi) Digital Outputs [2^{-1} to 2^{-8}]

The digital output is available to these pins. 2^{-1} represents the MSB and 2^{-8} represents the LSB of digital output.

The functional block diagram of ADC 0809 is as shown in Fig. 20.3.2 and the typical connection diagram is as shown in Fig. 20.3.3.

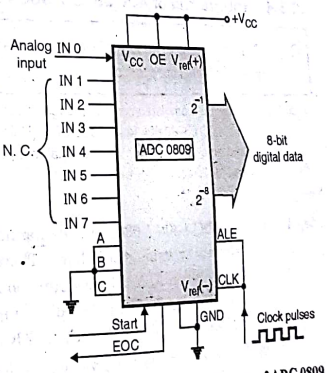


Fig. 20.3.3 : Typical connection diagram of ADC 0809

20.3.2 Typical Connection Diagram of ADC 0809

(i) Fig. 20.3.3 shows the connection diagram for converting the analog input connected to IN 0. All the other analog inputs are left open. In order to select IN 0, the status of address lines ABC should be 000. Therefore these pins have been connected to ground.

(ii) The "start" signal is generated either by using a switch or from a microprocessor. The "EOC" output can be connected to an LED or it can be returned back to the microprocessor.

(iii) The positive reference is being used. Hence V_{REF} (+) has been connected to V_{CC} and V_{REF} (–) has been connected to ground. The output enable (OE) pin also has been connected to V_{CC} so that the tristate output latch is enabled permanently.

(iv) The clock and ALE (address latch enable) pins are connected together to the source of clock pulses. These rectangular clock pulses can be obtained from a 555 astable multivibrator.

(v) The 8-bit binary output is TTL compatible and can be used appropriately.

20.3.3 Applications of ADC

Some of the important general applications of ADC are as follows :

1. In the digital instruments such as digital voltmeter, frequency counter etc.
2. In the data acquisition system.
3. In the digital tachometers for speed measurement and feedback.
4. In digital recording and reproduction.
5. In computerized instrumentation systems.
6. NC and CNC machines.

20.3.4 Interfacing of ADC 0808 / 0809

Q. With the help of a suitable diagram, explain the interfacing of an 8 bit ADC with 8085 microprocessor.

Ex. 20.3.1 : Draw the complete interfacing diagram for interfacing an 8 bit - 8 channel A/D converter like ADC 0808/0809 to an 8085 CPU (Using 8255) Write a program to take samples, one at a time from each channel of analog inputs and display it at a special display Port and wait for 2 seconds for each channel.

Soln.

Step 1 : (i) An ADC 0808 / 0809 is a 8 channel ADC having 8 inputs, all these are selected by using select lines. There are 3 select lines A, B, C and ALE signal. When ALE = 1 depending on A, B, C lines a channel will be selected and that input is connected as input to ADC. The ADC also contains different control signals such as SOC, EOC, OE.

(ii) The SOC is start of conversion, unless a pulse is applied at this input the ADC will not start conversion.

(iii) The ADC will start conversion and when it completes conversion, it gives signal EOC i.e. End of conversion. So microprocessor after giving SOC will go on checking EOC signal. If EOC is present it will enable output buffers by using OE signal which transfers data on output data lines and microprocessor will read the data.

Step 2 : The ADC 0808 / 0809 is to be interfaced to 8085 using 8255. So we decide function of Ports.3

SOC	-	Input to ADC	-	Output for 8255.
EOC	-	Output for ADC	-	Input to 8255
OE	-	Input to ADC	-	Output for 8255
A,B,C	-	Input to ADC	-	Output for 8255
ALE	-	Input to ADC	-	Output for 8255
$D_0 - D_7$	-	Output for ADC	-	Input to 8255

The ALE is used to latch A,B,C inputs to select a channel so ALE and SOC can be shorted and used.

Port A - $D_0 - D_7$ - (input)

Port B - SOC – PB_0, OE – PB_1, A,B,C, – PB_2, PB_3, PB_4 – (Output)

Port C - EOC – PC_0 (Input)

The control word required to initialize 8255 will be as follows :

1	0	0	1	1	0	0	1

= 99 H

Step 3 : The 8255 is interfaced to 8085 in I/O mapped I/O the complete interfacing will be as shown in Fig. P. 20.3.1(a).

Step 4 : The addresses of 8255 Ports and CWR will be 60, 61, 62 and 63 H.

Step 5 : For ADC we decide the sequence and bit pattern required to give control signals. The steps involved will be as follows :

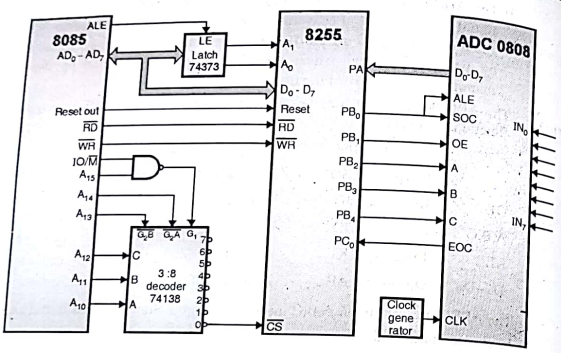


Fig. P. 20.3.1(a) : ADC interfacing using 8255

(a) Give SOC/ALE signal and A, B, C channel select signal -

X	X	X	C	B	A	0	1

A, B, C channel number to be selected.

SOC/ALE = 1
OE = 0

(b) Give SOC = 0, and A, B, C channel select 0E = 0

X	X	X	C	B	A	0	0

SOC/ALE = 1
OE = 0

(c) Check EOC signal i.e bit.

(d) When EOC = 1, make OE = 1

X	X	X	X	X	X	1	0

SOC/ALE = 1
OE = 0

(e) Take data from ADC and Change A, B, C to select next channel.

(f) Display digital data on display Port and go back to step (a)

Step 6 : The flow chart required to implement above steps will be as shown in Fig. P. 20.3.1(b).

Step 7 : Program

Label	Instruction	Operations
	LXI SP, FFFF H	FFFF → SP
	MVI A, 99 H	99 → A
	OUT 63 H	A → CWR
	MVI B, 00 H	00 → B (A, B, C)
loop :	MOV A, B	B → A
	ANI 1C H	Mask other bits except A, B, C
	ORI 01 H	Add SOC to data
	OUT 61 H	A (SOC = 1)
	ANI 1C H	1C A (Remove SOC from data)
	OUT 61	A Port → B (SOC = 0)
Up :	IN 62 H	P_C → A
	ANI 01 H	Mask other bits except
	CPI 01 H	Check bit (EOC)
	JNZ Up	If P C_0 = reset, go to up
	MVI A, 02 H	02 → A
	OUT 61 H	(0E = 1)
	IN 60 H	PA → A
	OUT display	A → Display
	CALL delay	Wait for 2 sec.
	MOV A, B	B → A
	ADI 04 H	A + 04 → A (Change A,B,C to select next ch.)
	MOV B, A	A → B
	JMP loop	Go to loop

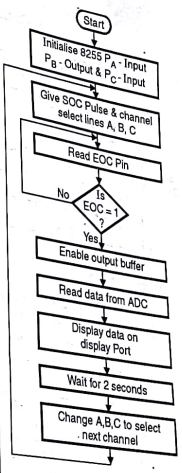


Fig. P. 20.3.1(b)

Ex. 20.3.2 : Write an assembly language program to read samples, one at a time from channel of analog inputs and convert it to digital.

Soln. :

Fig. P.20.3.2 shows the interfacing of ADC 0808 with 8086 using 8255. The analog input 2 is used. So the address lines ABC must be 010. The OE and ALE pins are to be maintained at logic high to select the ADC and enable the outputs. Upper part of port C acts as input while the lower part of port C acts as output port. Port A acts an 8 bit input data port to receive the digital data output from the ADC 0808.

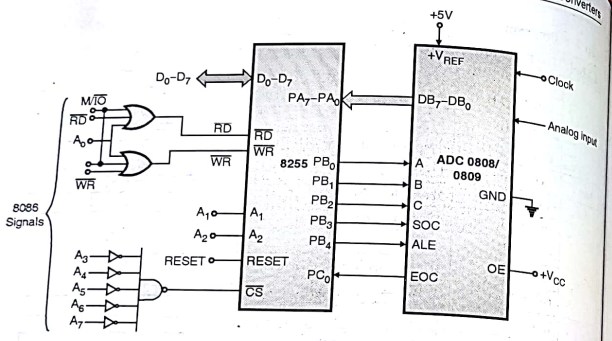


Fig. P. 20.3.2 : Interfacing ADC 0808/0809 with 8086.

The required program is as follows :

Instructions	Comment
. MODEL SMALL	
. CODE	
MOV AL, 98 H	Initialize 8255 port A as input port.
OUT CWR, AL	
MOV AL, 02 H	Select input 2 as analog input.
OUT Port B, AL	
MOV AL, 00 H	Give start of conversion.
OUT Port C, AL	
MOV AL, 01 H	
OUT Port C, AL	
MOV AL, 00 H	
OUT Port C, AL ·	
IN AL, Port C	Check for EOC by reading PCupper.
RCR	rotate carry
IN AL, Port A	If EOC, read digital data in AL
HLT	

20.4 DAC 0800

- The DAC 0800 is an 8 bit high speed current output digital to analog converter.
- It is manufactured by the National Semiconductor Corporation.

- It is a 16 pin IC available in dual in line DIP plastic package. The analog output is available in the form of current I_{OUT}. That means I_{OUT} is proportional to the 8 bit digital input.

20.4.1 Features of DAC 0800

(1) Fast setting time : 100 ns.
(2) Can be interfaced directly with TTL, CMOS, PMOS and other logic levels.
(3) Operates on wide power supply range varying from ± 4.5 V to ± 18 V.
(4) Low cost.
(5) Low power consumption : 33 mWatt ± 5 V.

20.4.2 Pin Diagram and Functional Block Diagram

- The pin diagram and functional block diagram of DAC 0800 are shown in Fig. 20.4.1 and Fig. 20.4.2.
- The internal block diagram shows that the DAC 0808 consists of R-2R ladder along with current switches and reference current amplifier.
- B_1 to B_8 are the 8 digital input lines with B_1 as the most significant bit and B_8 as the least significant bit.
- The analog output is available in the form of current I_{OUT}, therefore we need to use an external current to voltage converter if the analog output in the form of voltage is required
- The DAC 0800 requires a dual polarity (±) supply voltage, typically ± 18 V for its operation. The reference voltage can be either positive or negative.

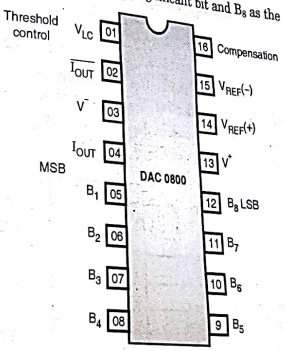


Fig. 20.4.1 : Pin diagram of DAC 0800

- An external reference voltage should be applied to either V_{REF} (+) or V_{REF} (–) depending on the polarity of the reference voltage.

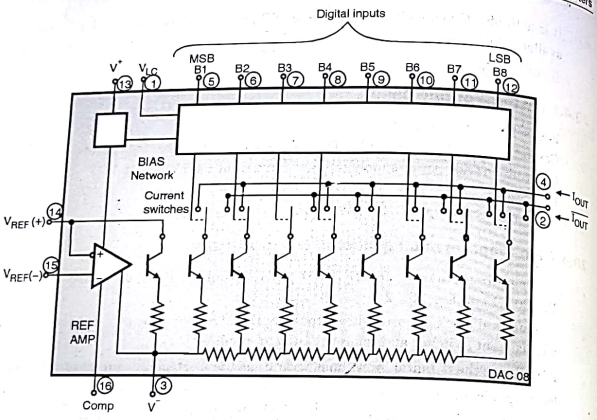


Fig. 20.4.2

Fig. 20.4.3 shows the interfacing of DAC 0800 to microprocessor 8085, using operational amplifier 741. V_{REF} is maintained at + 5V to get an output of + 5V amplitude. The output is observed at the output of operational amplifier 741, the microprocessor 8085 is connected to Port B of 8255.

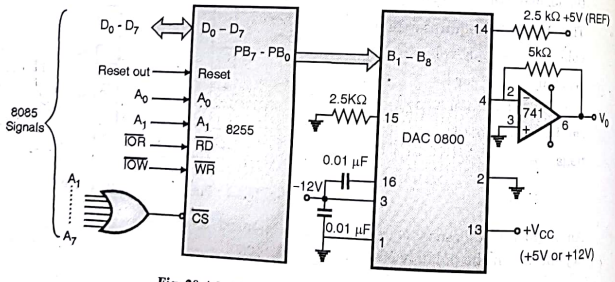


Fig. 20.4.3 : Interfacing DAC 0800 with 8085

20.5 Monolithic IC DAC 0808/0809

The DAC 0808 is an 8-bit current output monolithic DAC manufactured by the National semiconductor corporation. It is a 16 - pin IC available in dual in line DIP plastic package. The analog output is available in the form of current I_o. That means I_o is proportional to the 8-bit digital input. The important features of DAC 0808 are as follows :

Features of DAC 0808

Fast settling time	150 nsec. typically
Power supply voltage range	± 4.5 mW at ± 5V
Low power consumption.	33 mW at ± 5V.
High speed multiplying input slew rate	8 mA / µsec.
Interfaces directly with TTL, DTL and CMOS logic levels.	

20.5.1 Pin Configuration and Functional Block Diagram

- The pin configuration and functional block diagram of DAC 0808 are as shown in Figs. 20.5.1(a) and 20.5.1(b) respectively.
- The internal block diagram shows that DAC 0808 consists of R-2R ladder along with current switches and reference current amplifier.
- A_1 to A_8 are the 8-digital input lines with A_1 as the most significant bit and A_8 as the least significant bit.
- The analog output is available in the form of current I_o, therefore we need to use an external current to voltage converter if the analog output in the form of voltage is required.

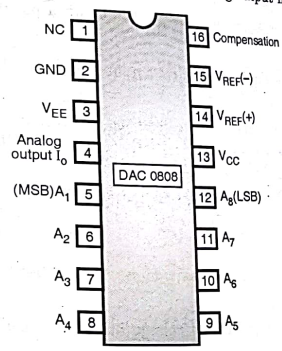


Fig. 20.5.1(a) : Pin configuration of DAC 0808

- DAC 0808 requires a dual polarity (±) supply voltage, typically ± 15V, for its operation. The reference voltage can be either positive or negative.
- An external reference voltage should be applied to either V_{REF} (+) or V_{REF} (–) depending on the polarity of reference voltage.

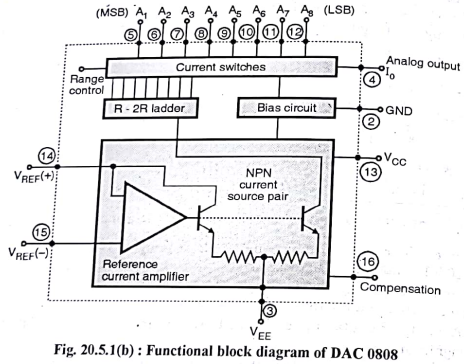


Fig. 20.5.1(b) : Functional block diagram of DAC 0808

20.5.2 Typical Connection Diagram

- The typical connection diagram for DAC 0808 with a positive reference voltage is shown in Fig. 20.5.2.
- This circuit provides a unipolar positive output voltage due to the use of an external I to V converter.
- As the reference voltage has been decided to be positive, an external positive voltage has been applied at pin number 14 (V_{REF} +) of the IC through R_{14}, connecting V_{REF} (–) i.e. pin number 15 to ground through R_{16}.
- Assuming that $R_{14} = R_{16} = R$, the full scale output voltage is given by :

$$V_{FS} = \frac{R_F}{R} \times V_{REF} \qquad ...(20.5.1)$$

- In practice one of these resistors is kept adjustable in order to provide more accurate scaling relationship.
- The reference amplifier provides a voltage at pin number 14 for converting the reference voltage into a current. It also acts as current mirror for feeding the internal R-2R ladder network.

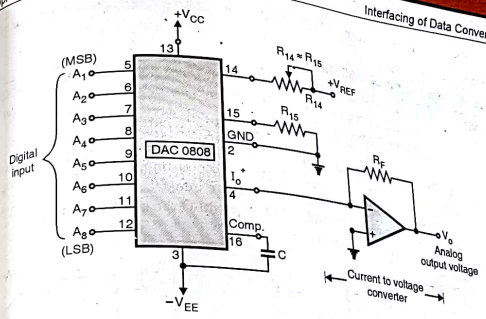


Fig. 20.5.2 : Typical connection diagram for DAC 0808

Note : (i) The reference amplifier input current I_{14} should always flow into pin number 14 irrespective of the reference voltage polarities.

(ii) The value of the compensation capacitor "C" must be increased with increase in the R_{14} to maintain proper phase margin. For R_{14} values of 1 kΩ, 2.5 kΩ and 5 kΩ minimum capacitor values are 15, 37 and 75 pF respectively.

20.5.3 Connection Diagram with a Negative Reference Voltage

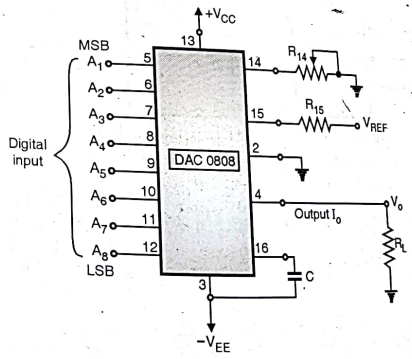


Fig. 20.5.3 : Connection diagram for DAC 0808 with a negative reference voltage

The connection diagram of DAC 0808 with a negative reference voltage is as shown in Fig. 20.5.3.

20.5.4 Interfacing of DAC 0808

Ex. 20.5.1 : Draw the complete interfacing diagram for interfacing 8 bit DAC 0808 to an 8085 CPU (Using 8255) Write a program to generate (i) Square wave output (ii) Triangular wave (iii) Saw tooth wave and (iv) Sine wave.

Soln.

Step 1 : A DAC 0808 is a digital to analog converter having current output. The current output is proportional to digital input applied at input pins. No control signals are involved with DAC, so only data lines are interfaced to 8255.

Step 2 : The 8255 is interfaced to 8085 in I/O mapped I/O the complete interfacing will be as shown in Fig. P. 20.5.1(a).

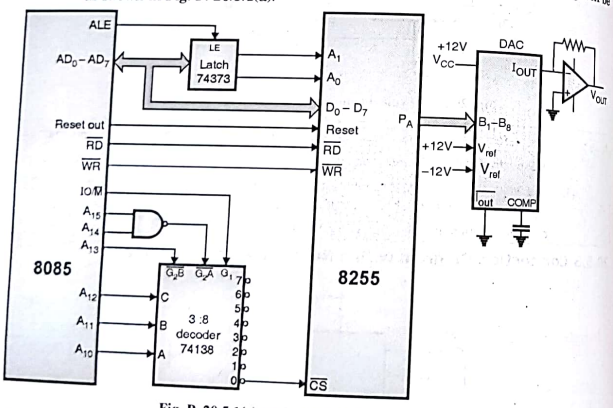


Fig. P. 20.5.1(a) : DAC interfacing using 8255

Step 3 : The addresses of 8255 Ports and control registers will be

A_{15}	A_{14}	A_{13}	A_{12}	A_{11}	A_{10}	A_1	A_0		
1	1	0	0	0	0	0	0	= C0 H	Port A
1	1	0	0	0	0	0	1	= C1 H	Port B
1	1	0	0	0	0	1	0	= C2 H	Port C
1	1	0	0	0	0	1	1	= C3 H	CWR

(1) Square wave

To generate square wave we require two levels of inputs 00 H and F0 H. This is achieved by using output data 00 H wait for some time, then output data F0 H wait for some time. Program (For flowchart refer Fig. P. 20.5.1(b)).

Label	Instructions	Operation
	LXI SP, FFFF H	FFFF → SP
	MVI A, 80 H	80 → A
	OUT C3	A → CWR
UP :	MVI A, 00 H	00 → A
	OUT C0	A → P_A
	CALL delay	Wait for delay
	MVI A, F0	F0 → A
	OUT C0	A → P_A
CALL delay :	Wait for delay	
	JMP UP	Go to UP.

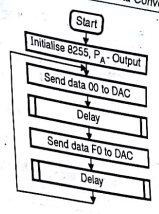


Fig. P. 20.5.1(b)

(2) Triangular wave

To write the program for triangular wave, we first go on increasing the count upto FF H and then decrease it to zero.

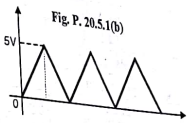


Fig. P. 20.5.1(c) : Triangular wave

Program : (For flowchart refer Fig. P. 20.5.1(d))

Label	Instruction	Comments
	LXI SP, C600 H	Initialize stack
	MVI A, 80H	
	OUT C3 H	Control word
	MVI B, 00H	
BACK :	MOV A, B	
UP :	OUT C1 H	Send rising edge data
	CALL DELAY	If required give delay
	INR B	
	MOV A, B	Store new data/count
	CPI FF H	Check for 5 V.
	JNZ UP	
AGAIN :	DCR B	If 5V, decrement count
	MOV A, B	Save count
	CPI 00 H	Check for 0V
	JZ BACK	If yes, start for rising edge count.
	OUT C1 H	
	CALL DELAY	
	JMP AGAIN	Give delay & continue with falling edge

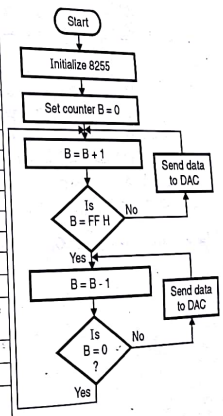


Fig. P. 20.5.1(d)

(3) Sawtooth wave : We initialize to FF H and count down to zero.

Fig. P. 20.5.1(e) : Sawtooth wave

Program : (For flowchart refer Fig. P. 20.5.1(f))

Label	Instructions	Operation
	LXI SP, C600 H	
	MVI A, 80 H	Control word
	OUT C3 H	
BACK :	MVI B, FF H	Count for 5V
UP :	OUT C1 H	
	DCR B	Decrement count
	MOV A, B	
	CPI 00 H	Check if count = 0.
	JZ BACK	
	CALL DELAY	
	JMP UP	

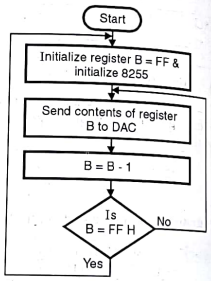


Fig. P. 20.5.1(f)

(4) Sine wave

The 8085 microprocessor cannot perform sine function, hence look up table technique must be used. Assume the look up table starts at memory location 2000 H. At 2000 H the digital value of sin 0 is stored, at 2001 H the digital value of sin 1 is stored and so on.

Program : (For flowchart refer Fig. P. 20.5.1(g))

Label	Instructions	Operation
	LXI SP, FFFF H	
	MVI D, 80 H	8255 in mode 0
	OUT C3	Port A o/p
BACK:	LXI H, 2000 H	Look up table pointer
	LXI D, 0168 H	Byte counter (360 values)
UP:	MOV A, M	Send value to DAC
	OUT C0	

Label	Instructions	Operation
	INX H	Increment look up table pointer.
	DCX D	Decrement byte counter.
	MOV A, E	
	ORA D	
	JNZ UP	
	JMP BACK	

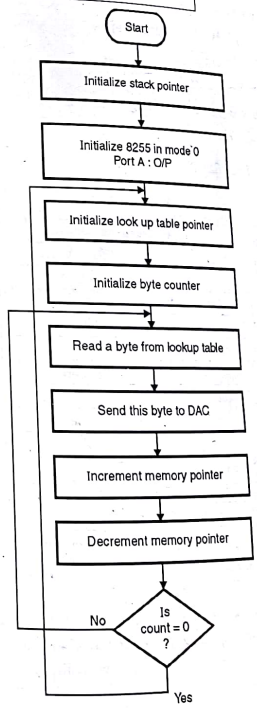


Fig. P. 20.5.1(g)

Ex. 20.5.2 : Write an assembly language program to generate a triangular pulse train using DAC interface.

Soln. :

Algorithm

Step 1 : Initialize 8255 Port A as output port.
Step 2 : Initialize AL = 00 H.
Step 3 : Output the contents of AL through port A.
Step 4 : Increment AL by one.
Step 5 : Check if AL = FF H ? If not, goto step 3.
Step 6 : Decrement AL by one.
Step 7 : Output AL through port A.
Step 8 : Compare AL with 00. If AL = 00, goto step 3.
Step 9 : If not, go to step 6.

Flowchart : [Refer Fig. P. 20.5.2(a)]

Program

Label	Instructions	Comment
	. MODEL SMALL	
	. CODE	
	MOV AL, 80 H	Initialize Port A = Output Port.
	MOV DX, 00 H	Load DX with port address of port A.
	MOV AL, 00 H	
L1 :	OUT DX, AL	Output contents of AL through port A.
	INC AL	Increment AL.
	CMP AL, FF H	Compare AL with FF H, if not continue.
	JNZ L1	
L2 :	DEC AL	Decrement AL.
	OUT DX, DL	Output contents of AL to port A.
	JNZ L2	Decrement till AL = 00.
	JNZ L1	If AL = 00, go to L1 i.e. start from beginning.

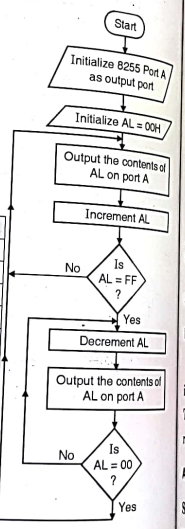


Fig. P. 20.5.2(a)

The above program generates a triangular waveform at the output of DAC. If the speed of system on which you are working is very fast, call delay after incrementing and decrementing AL contents, so that you can see the waveform on CRO.

The Fig. P. 20.5.2(b) shows the interfacing diagram,

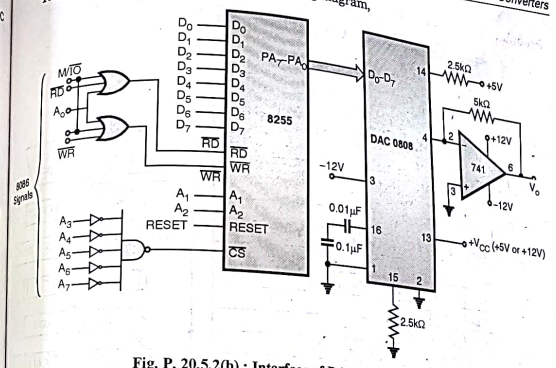


Fig. P. 20.5.2(b) : Interface of DAC 0808 with 8086

Ex. 20.5.3 : Write an assembly language program to generate a rectangular pulse train using DAC interface.

Soln. :

Explanation

We are asked to generate a rectangular pulse train i.e. a square wave using DAC interface. To generate square wave we will output FF H and then 00 H on port A of 8255. The output of 8255 (port A) is connected to DAC 0808. According to the frequency requirement delay is provided in between the two outputs 00 H and FF H.

Algorithm

Step 1 : Initialize 8255 A Port A as output port.
Step 2 : Output FF H to port A of 8255 A.
Step 3 : Call delay.
Step 4 : Output 00 to port A of 8255 A.
Step 5 : Call delay.
Step 6 : Jump to step 2.

Flowchart : [Refer Fig. P. 20.5.3(a)]

Program

Label	Instructions	Comment
	.MODEL SMALL	
	.CODE	
	MOV AL, 80 H	Initialize 8255 with port A as output port.
	MOV DX, 00 H	DX = port A address.
REPEAT :	MOV AL, FF H	AL = FF H
	OUT DX, FE H	Output FF on port A.
	CALL DELAY	
	MOV AL, 00 H	AL = 00 H
	OUT DX, AL	Output 00 on port AL
	CALL DELAY	
	JMP REPEAT	Repeat the process
	DELAY PROC	
	PUSH AX	
	MOV AL, FF H	Load AL = FF H
LOOP :	DEC AL	Decrement AL
	JNZ LOOP	Repeat till AL = 00 H
	POP AX	Pop AX
	RET	Return back to main program.

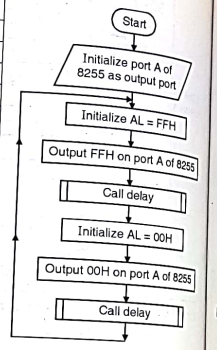


Fig. P. 20.5.3(a)

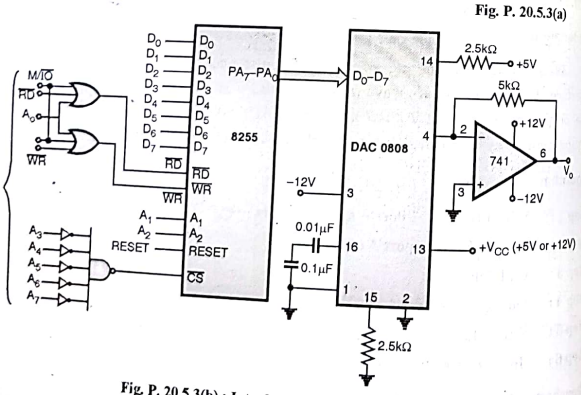


Fig. P. 20.5.3(b) : Interface of DAC 0808 with 8086

20.6 DAC 0830

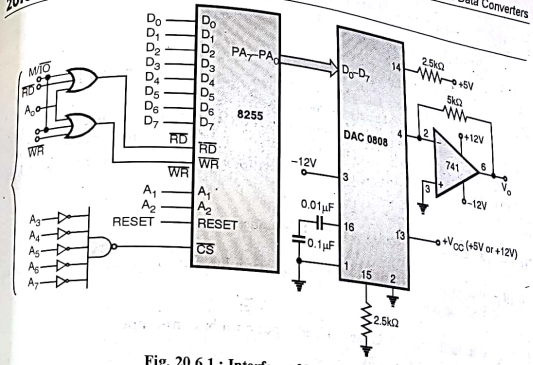


Fig. 20.6.1 : Interface of DAC 0808 with 8086

DAC 0830

It is an 8 bit multiplying DAC to interface directly with the microprocessors like 8085, 8086, 8048. It uses R2R ladder network which divides the reference current and provides an excellent temperature tracking characteristics.

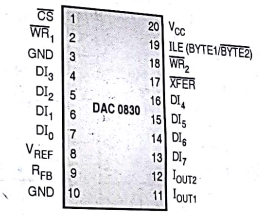


Fig. 20.6.2

Features

(1) Double buffered, single buffered or flow through digital data inputs.
(2) It is pin compatible with 12 bit DAC 1230 series.
(3) Its works with ± 10V reference.
(4) It can be used in the voltage switching mode.
(5) It has logic inputs which meet TTL voltage level specifications.

Fig. 20.6.2 shows the pin diagram.

Pin 1 : $\overline{CS}$: Chip select (active low) : It is in combination with ILE and will enable $\overline{WR_1}$.

Pin 19 : **ILE** (Input Lathe Enable) : It is an active high signal. The ILE is in combination with $\overline{CS}$ and enables the $\overline{WR}$.

Pin 3 : **GND**

$DI_7 - DI_0$: Digital inputs.

Pin 8 : **V_{REF} :** It connects an external precision voltage source to the internal R-2R ladder. It can be selected over the range of + 10 to – 10V.

Pin 9 : **R_{FB}** (feedback resistor) : It is provided on the IC chip for use as shunt feedback resistor for external op-amp which is used to provide an output voltage forDAC.

Pin 10 : It should be at same ground potential as **$I_{OUT\,1}$** and **$I_{OUT\,2}$** for current switching applications.

Pin 17 : **$\overline{XFER}$** (Transfer control signal) : It is an active low signal. It enables $\overline{WR_2}$.

Pin 18 : **$\overline{WR_2}$** (active low) : It causes 8 bit data available in input latch to transfer to DAC register.

Pin 2 : **$\overline{WR_1}$** (active low) : It is used to load input data bits into input latch.

Pin 20 : **V_{CC} :** It can be from + 5V to + 15V.

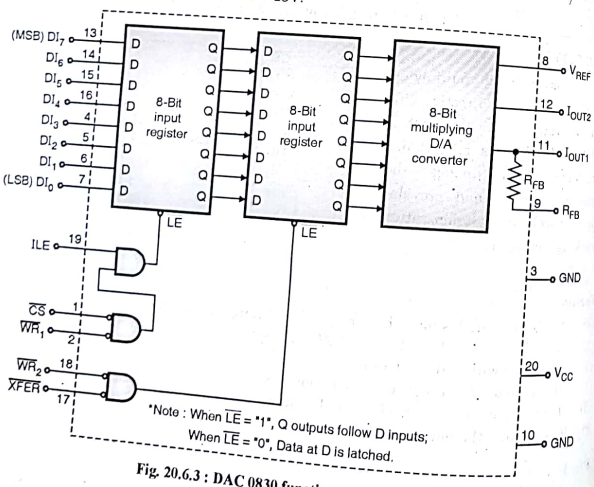


Fig. 20.6.3 : DAC 0830 functional diagram

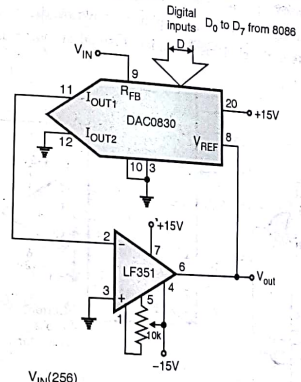
The purpose of any D/A converter to provide an accurate analog output representing the digital input. The output I_{OUT1} is directly proportional to the product of applied reference voltage and the digital input. The output I_{OUT2} is proportional to the complement of digital input.

$$I_{OUT1} = \frac{V_{REF}}{15\ K\Omega} \times \frac{\text{digital Input}}{256}$$

$$I_{OUT2} = \frac{V_{REF}}{15\ K\Omega} \times \frac{255 - \text{Digital Input}}{256}$$

15 KΩ is nominal value of internal resistance.

e.g. 1 : The Fig. 20.6.4 shows interface of DAC 0830 with 8086 for volume control.



- $V_{OUT} = \frac{V_{IN}(256)}{D}$
- When D = 0, the amplifier will go open loop and the output will saturate.
- Feedback impedance from the-input to the output varies from 15kΩ to ∞ as the input code changes from full scale to zero

Fig. 20.6.4 : Interface of DAC 0830 with 8086 for volume control

The I_{OUT1} of DAC 0830 is given to LF 351. LF 351 is a low cost high speed JFET operational amplifier. V_{IN} is the digital input which is to be converted into analog form. The output will saturate when V_{IN} = 0. The input voltage should not exceed the supply

voltage. The LF 351 acts as amplifier and amplifies the analog input. This input will vary from 0 to 5V to the LF 351. The output obtained will be the range of 0-15V.

e.g. 2 : Interface of DAC 0830 with 8086 Microprocessor for driving dc motor.

The Fig. 20.6.4 shows 8086 microprocessor interface to DAC 0830. The microprocessor 8086 is interfaced using the I/O port address 8F00H, 8F001 H. The output of DAC is fed to an operational amplifier. The output of operational amplifier is fed to a driver circuit to power a 12V DC motor. The driver circuit consists of a Darlington amplifier.

$\overline{XFER}$ is provided to the decode logic which goes low for a short during when OUT DX, AL instruction is executed. DX contains the address for selection of DAC i.e. 08F00 H. The voltage required for operation is latched for the conversion. The analog output voltage from operational amplifier, activates the transistors and enables the dc motor to rotate in the clockwise direction. Instead of motor any other device also could be used.

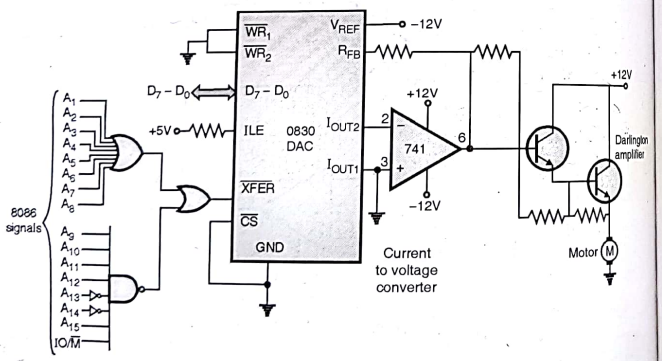


Fig. 20.6.5 : Interface of 0830 with 8086 for driving DC motor

□□□

CHAPTER

21

Interfacing Techniques

21.1 Memory Interfacing

In this section, memory components like EPROM, RAM and DRAM will be interfaced to CPU 8086. Fig. 21.1.1, shows generalised block schematic of memory chip.

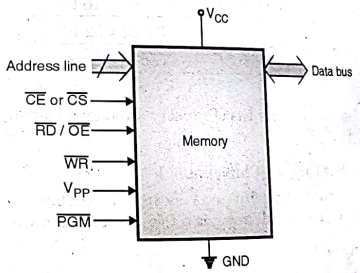


Fig. 21.1.1 : Basic block schematic of memory

Note : (1) V_{PP} and $\overline{PGM}$ pin is only for PROM; not for RAM

(2) $\overline{WR}$ pin is for RAM.

(1) **Address lines**

Number of address lines depends upon size of the memory.