SARASWATI Education Society's

SARASWATI College of Engineering

Learn Live Achieve and Contribute

Kharghar, Navi Mumbai - 410 210.

NAAC Accredited

Department of Computer Engineering

Name: Shaikh Adnan Shaukat Ali

Rno: 55

Subject: Microprocessor (MP)

Class/Sem: SE/IV

SARASWATI Education Society's
SARASWATI College of Engineering
Learn Live Achieve and Contribute
Kharghar, Navi Mumbai - 410 210.
NAAC Accredited

Department of Computer Engineering

Subject :Microprocessor **Class/Sem: SE/IV**

INDEX

Expt. No.	Date	Name of the Experiment	Page No
1	02-02-2021	WAP to perform 16-bit addition , subtraction, multiplication of two numbers and division of 16-bit number by 8-bit number and display the contents of flag register	1-14
2	01-03-2021	WAP to display Fibonacci series	15-17
3	08-03-2021	WAP to find factorial of number using procedure.	18-22
4	15-03-2021	WAP based on string instruction to transfer a block and to find string length	23-25
5	15-02-2021	WAP to perform to addition, subtraction, multiplication & division using mixed language programming	26-30
6	12-04-2021	WAP to find GCD/LCM of 2 nos.	31-32
7	20-04-2021	WAP to find minimum. from given array.	33-34
8	27-04-2021	WAP to using int 10H	35-41
9	12-04-2021	WAP to find factorial of number using macro	42-43
10	11-05-2021	WAP to display string using int 21h	44-48

Suhasini Parvatikar

Subject I/C

SARASWATI Education Society's
SARASWATI College of Engineering
Learn Live Achieve and Contribute
Kharghar, Navi Mumbai - 410 210.
NAAC Accredited

Department of Computer Engineering

Subject :Microprocessor **Class/Sem:SE/IV**

INDEX

Expt. No.	Date	Name of the Experiment	Page No
1	30/02/2021	Assignment 1	49-64
2	15/05/2021	Assignment 2	65-79

Suhasini Parvatikar
Subject I/C

EXPERIMENT NO- 1

AIM: WAP to perform 8bit/16 bit arithmetic operations and display the contents of flag register.

Resource Required: P-IV and above RAM 128MB, Dot Matrix Printer, Emu 8086, MASM 611/ TASM, Turbo C/C++, Printer, Printout Stationary.

THEORY:

Assemble language has two types of statements:

1. **Executable**: Instruction that are translated into machine code by the assembler
2. **Assembler Directives**:
    - Statements that direct the assembler to do some special task.
    - No machine language code is produced for these statements.
    - Their main task is to inform the assembler about the start/ end of a segment, procedure or program, procedure or program, to reserve appropriate space from data storage etc.
    - Some of the assembler directives are listed below:

.DB (define byte): used to define a byte variable. Ex SUM DB 0. Assembler reserves 1 byte of memory for the variable SUM and initializes it to 0.

.DW (Define word, 16 bit): used to define a word type variable.

.DD (Define double word, 32 bit): used to define a double word type variable.

.DQ (Quad Word): used to define a quad word type variable.

Instructions:

MOV Destination Source

Move a byte/word from the source to the destination specified in the instruction.

Source: Register, Memory location, immediate number

Destination: Register, Memory location

Both source & destination cannot be memory locations.

MOVE Register, Register

MOV Memory location, Register

MOV Register, Memory location

MOV Register data.

ADD Destination, Source

Adds the source to the destination & stores the result back in the destination.

Source: Register, Memory Location, Immediate number

Destination: Register

Both source & destination have to be of the same size.

ADD Register, Register

ADD Memory location, Register

ADD Register, Memory location

ADD Register, data.

ADC Destination, Source
Adds the source to the destination & stores the result the with carry back in the destination.

Source: Register, Memory location, immediate number.

Destination: Register

Both source & destination have to be of the same size

ADD Register, Register

ADD Memory location, Register

ADD Register, Memory location

ADD Register, data.

SUB/SBB Destination, source

It is similar to ADD/ADC expect that it does subtraction.

DAA (Decimal adjust for addition)

It makes the result in BCD from after BCD addition is performed.

It works only on AL register.

If D3-DO (Lower 4-bit) > 9 then AF is set, Add 06h to AL.

2

If D7-D4 (Upper 4-bit) > 9 then CF is set, Add 60h to AL.

DAS (Decimal adjust for subtraction)

It makes the result in packed BCD from after BCD subtraction is performed

It works only on AL register.

If D3-DO (Lower 4-bit) > 9 then AF is set, Add 06h to AL.

If D7-D4 (Upper 4-bit) > 9 then CF is set, Add 60h to AL.

MUL source (Unsigned 8/16-bit registers)

Source: Register, Memory Location

If the source is 8-bit it is multiplied with AL & result is stored in AX

(AH- higher byte, AX- lower byte)

If the source is 16-bit, it is multiplied with AX & result is stored in

DX –AX register (DX-higher byte, AX-lower byte)

MUL affects AF, PF, SF, & ZF.

DIV source (Unsigned 8/16-bit register-divisior)

This instruction is used for unsigned division.

Divides a word by a byte or a double word by word.

If divisior is 8-bit then the dividend is in AX register.

After division the quotient is in AL & reminder is in DX

If divisior is 16-bit then the dividend is in DX-AX

<u>Algorithm</u> <u>:</u>

1. **An algorithm for addition of two 16-bit numbers.**

   Step 1: Start

   Step 2: Initialize data segment

3

Step 3: Declare two variables that hold the actual data.

Step 4: Initialize code segment

Step 5: Initialize DS register to program

Step 6: move first no. in register (bx)

Step 7: move second no. in register (cx)

Step 8: Perform the addition

Step 9: Stop

2. **An algorithm for subtraction of two 16-bit numbers**.

Step 1: Start

Step 2: Initialize data segment

Step 3: Declare two variables that hold the actual data.

Step 4: Initialize code segment

Step 5: Initialize DS register to program

Step 6: move first no. in register (bx)

Step 7: move second no. in register (cx)

Step 8: Perform the subtraction

Step 9: Stop

Assemble language has two types of statements:

1. **Executable**: Instruction that are translated into machine code by the assembler
2. **Assembler Directives**:
   - Statements that direct the assembler to do some special task.
   - No machine language code is produced for these statements.
   - Their main task is to inform the assembler about the start/ end of a segment, procedure or program, procedure or program, to reserve appropriate space from data storage etc.
   - Some of the assembler directives are listed below:

.DB (define byte): used to define a byte variable. Ex SUM DB 0. Assembler reserves 1

4

byte of memory for the variable SUM and initializes it to 0.

.DW (Define word, 16 bit): used to define a word type variable.

.DD (Define double word, 32 bit): used to define a double word type variable.

.DQ (Quad Word): used to define a quad word type variable.

Instructions:

MOV Destination Source

Move a byte/word from the source to the destination specified in the instruction.

Source: Register, Memory location, immediate number

Destination: Register, Memory location

Both source & destination cannot be memory locations.

MOV Register, Register

MOV Memory location, Register

MOV Register, Memory location

MOV Register data.

ADD Destination, Source

Adds the source to the destination & stores the result back in the destination.

Source: Register, Memory Location, Immediate number

Destination: Register

Both source & destination have to be of the same size.

ADD Register, Register

ADD Memory location, Register

ADD Register, Memory location

ADD Register, data.

ADC Destination, Source

Adds the source to the destination & stores the result the with carry back in the destination.

Source: Register, Memory location, immediate number.

5

Destination: Register

Both source & destination have to be of the same size

ADD Register, Register

ADD Memory location, Register

ADD Register, Memory location

ADD Register, data.

SUB/SBB Destination, source

It is similar to ADD/ADC expect that it does subtraction.

DAA (Decimal adjust for addition)

It makes the result in BCD from after BCD addition is performed.

It works only on AL register.

If D3-DO (Lower 4-bit) > 9 then AF is set, Add 06h to AL.

If D7-D4 (Upper 4-bit) > 9 then CF is set, Add 60h to AL.

DAS (Decimal adjust for subtraction)

It makes the result in packed BCD from after BCD subtraction is performed

It works only on AL register.

If D3-DO (Lower 4-bit) > 9 then AF is set, Add 06h to AL.

If D7-D4 (Upper 4-bit) > 9 then CF is set, Add 60h to AL.

MUL source (Unsigned 8/16-bit registers)

Source: Register, Memory Location

If the source is 8-bit it is multiplied with AL & result is stored in AX

(AH- higher byte, AX- lower byte)

If the source is 16-bit, it is multiplied with AX & result is stored in

DX –AX register (DX-higher byte, AX-lower byte)

MUL affects AF, PF, SF, & ZF.

6

IMUL source (Signed 8/16-bit registers)

Sam as MUL expect that the source is a signed number.

DIV source (Unsigned 8/16-bit register-divisior)

This instruction is used for unsigned division.

Divides a word by a byte or a double word by word.

If divisior is 8-bit then the dividend is in AX register.

After division the quotient is in AL & reminder is in DX

If divisior is 16-bit then the dividend is in DX-AX

Register.

IDIV source (Signed 8/16-bit register-divisior)

Same as DIV except that the source is a signed number.

<u>Algorithm</u> <u>:</u>

1. **An algorithm for addition of two 16-bit numbers.**

Step 1: Start

Step 2: Initialize data segment

Step 3: Declare two variables that hold the actual data.

Step 4: Initialize code segment

Step 5: Initialize DS register to program

Step 6: move first no. in register (bx)

Step 7: move second no. in register (cx)

Step 8: Perform the addition

Step 9: Stop

2. **An algorithm for addition of two 16-bit numbers**.

Step 1: Start

Step 2: Initialize data segment

7

Step 3: Declare two variables that hold the actual data.

Step 4: Initialize code segment

Step 5: Initialize DS register to program

Step 6: move first no. in register (bx)

Step 7: move second no. in register (cx)

Step 8: Perform the subtraction

Step 9: Stop

3. **An algorithm for multiplication of two 16-bit numbers.**

Step 1 : Start

Step 2: Initialize data segment

Step 3 : Declare two variables that hold the actual data.

Step 4 : Initialize code segment

Step 5 : Initialize DS register to program

Step 6 : move first no. in register (ax)

Step 7 : move second no. in register (cx)

Step 8 : multiply both the numbers by using word pointer & transfer the result in particular variable.

Step 9 : Stop

4. **An algorithm for division of 16-bit number by 8-bit number.**

Step 1 : Start

Step 2: Initialize data segment

Step 3 : Declare two variables that hold the actual data.

Step 4 : Initialize code segment

Step 5 : Initialize DS register to program

Step 6 : move 16-bit no.in register (al)

Step 7 : move 8-bit no.in register (bl)

Step 8 : perform division operation & store the result in particular variable

8

Step 9 : Stop

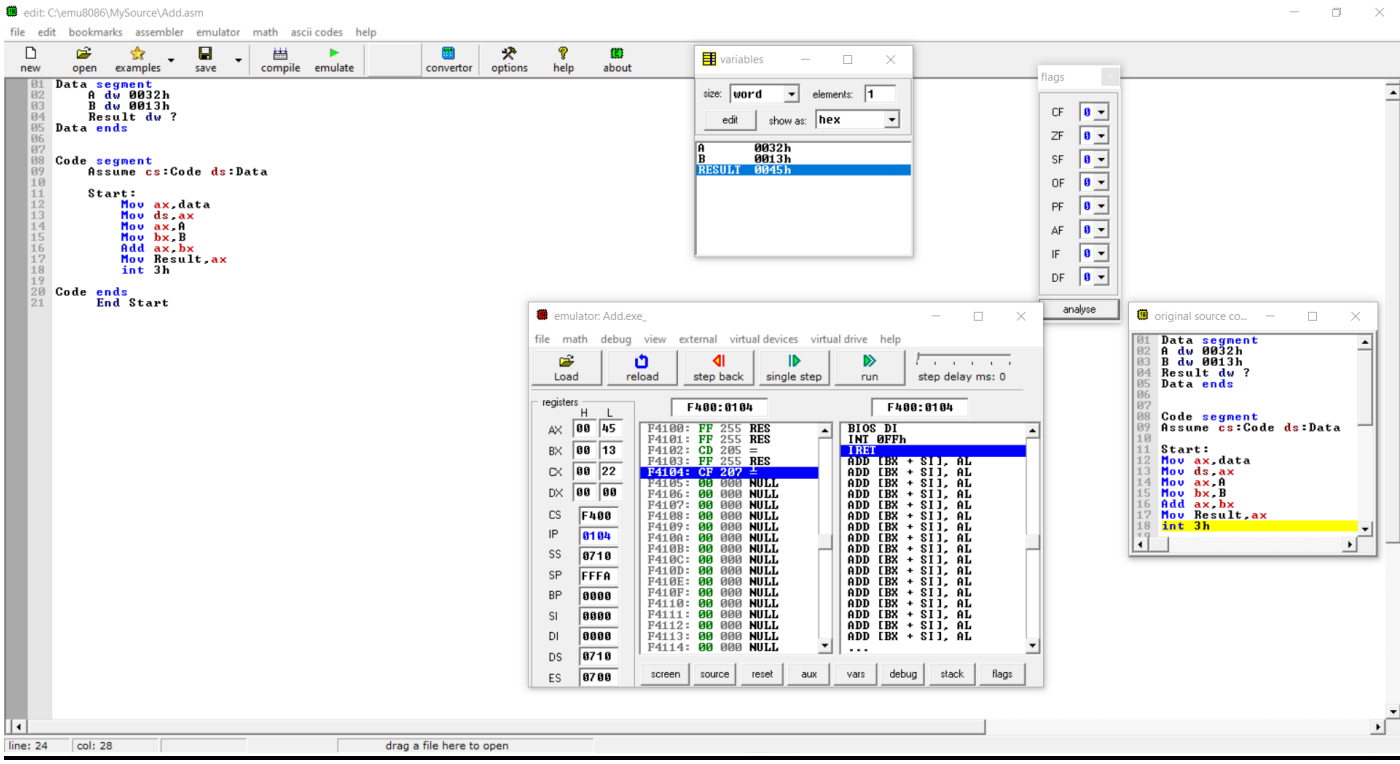
5.

CONCLUSION: **Hence we have done arithmetic operation on 2 8/16 bits number in emu8086 using assembly and display the output and flag contents**

<u>1.Addition of two 16bits</u>:

```
Data segment
   A dw 0032h
   B dw 0013h
   Result dw ?
Data ends
Code segment
   Assume cs:Code ds:Data
   Start:
      Mov ax,data
      Mov ds,ax
      Mov ax,A
      Mov bx,B
      Add ax,bx
      Mov Result,ax
      int 3h
Code ends
    End Start
```

Output:



2. **Subtraction of two 16bits number:**

Data segment

A dw 0032h

B dw 0013h

Result dw ?

Data ends

Code segment

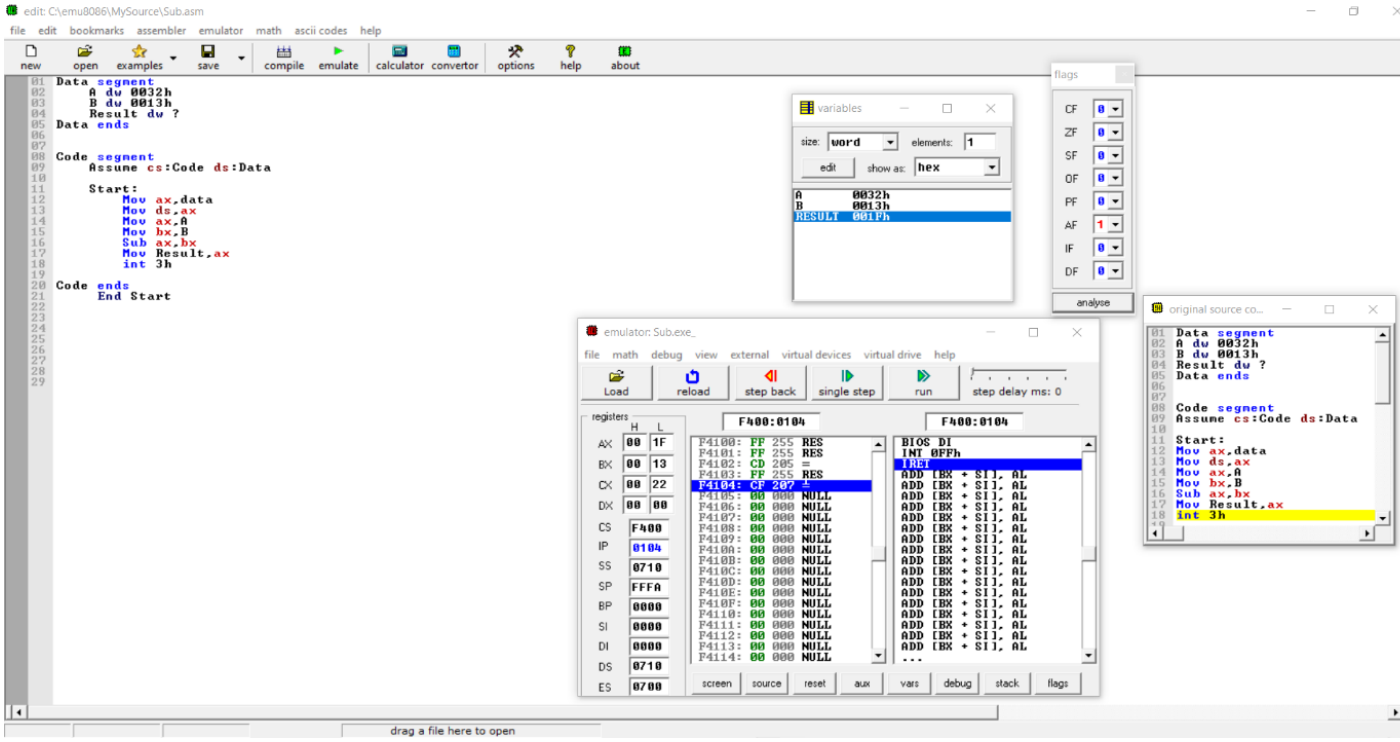
Assume cs:Code ds:Data

Start:

10

```
    Mov ax,data
    Mov ds,ax
    Mov ax,A
    Mov bx,B
    Sub ax,bx
    Mov Result,ax
    int 3h
Code ends
  End Start
```

Output :



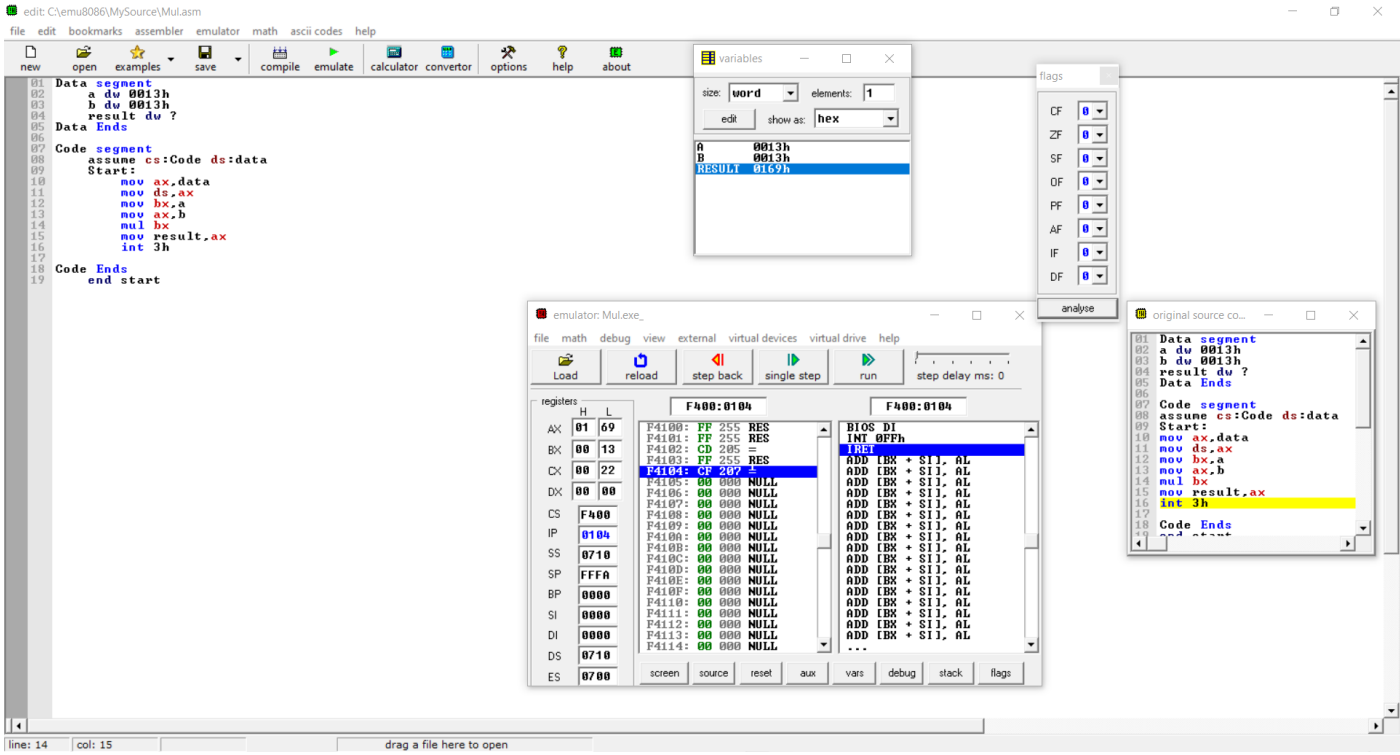
11

3. **Multiplication of two 16bits**:

```
Data segment
   a dw 0013h
   b dw 0013h
   result dw ?
Data Ends
Code segment
   assume cs:Code ds:data
   Start:
      mov ax,data
      mov ds,ax
      mov bx,a
      mov ax,b
      mul bx
      mov result,ax
      int 3h
Code Ends
   end start
```


12

Output:



4. **Division of 16bits/8bits**:

Data segment

a dw 0011h

b db 05h

quotient db ?

remainder db ?

Data Ends

Code segment

Assume cs:code ds:data

start:

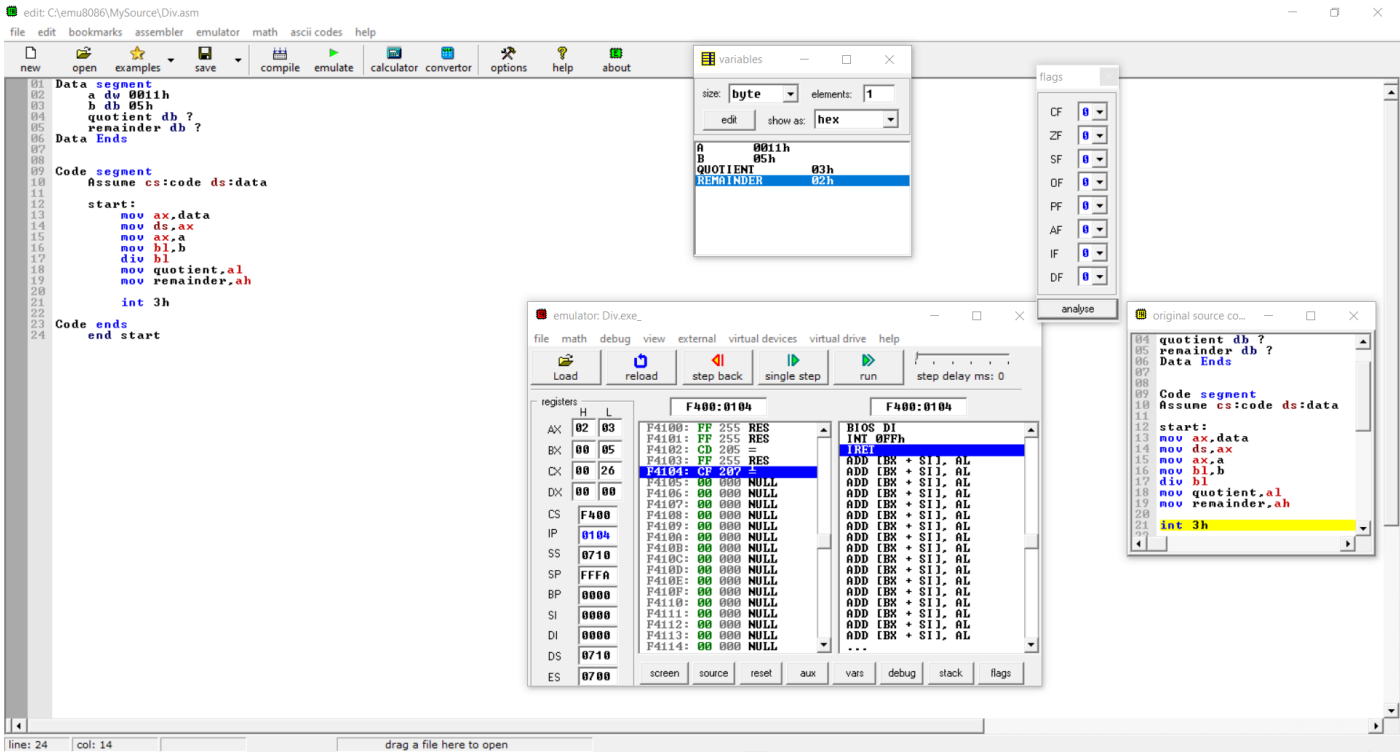
mov ax,data

mov ds,ax

13

```
        mov ax,a
        mov bl,b
        div bl
        mov quotient,al
        mov remainder,ah
        int 3h
Code ends
    end start
```

Output:



14

EXPERIMENT NO- 2

AIM: Write an assembly language program to find Fibonacci series of N terms

Resource Required: P-IV and above RAM 128MB, Dot Matrix Printer, Emu 8086, MASM 611/ TASM, Turbo C/C++, Printer, Printout Stationary.

THEORY:

The Fibonacci numbers are the numbers in the following integer sequence.

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ……..

In mathematical terms, the sequence Fn of Fibonacci numbers is defined by the recurrence relation

Fn = Fn-1 + Fn-2

ALGORITHM:

Step I : Initialize the data segment

Step II : Initialize the counter =10 i.e 0A h

Step III : Initialize SI to starting address

Step IV : Store the 1st term at the location where SI is pointing

Step V : Increment SI to point next location

Step VI : Store next term 1 to location where SI is pointing

Step VII : Increment Si to point to next location

Step VIII : Next term =[SI-2]+[SI-1]

Step IX : Store the result to location pointed by SI

Step X : Increment SI

Step XI Decrement counter

Step XII Check if count=0,if nt the go to step VIII

Step XIII : Display the result.

Step XIV : Stop.

CONCLUSION: We have successfully calculated Fibonacci sequence of desired 'N' numbers using assembly language programming.

Program:

```
Data segment
   count dw 000Ch
   array db count DUP(?)
Data ends
Code segment
   assume cs:Code ds: Data
   Start:
      mov ax,Data
      mov ds,ax
      mov cx,count
      lea si,array
      mov al,00h
      mov [si],al
      inc si
      dec cx
      mov bl,01h
      mov [si],bl
      inc si
      dec cx

      up:
         mov al,[si-1]
         mov bl,[si-2]
         add al,bl
         mov [si],al
         inc si
```

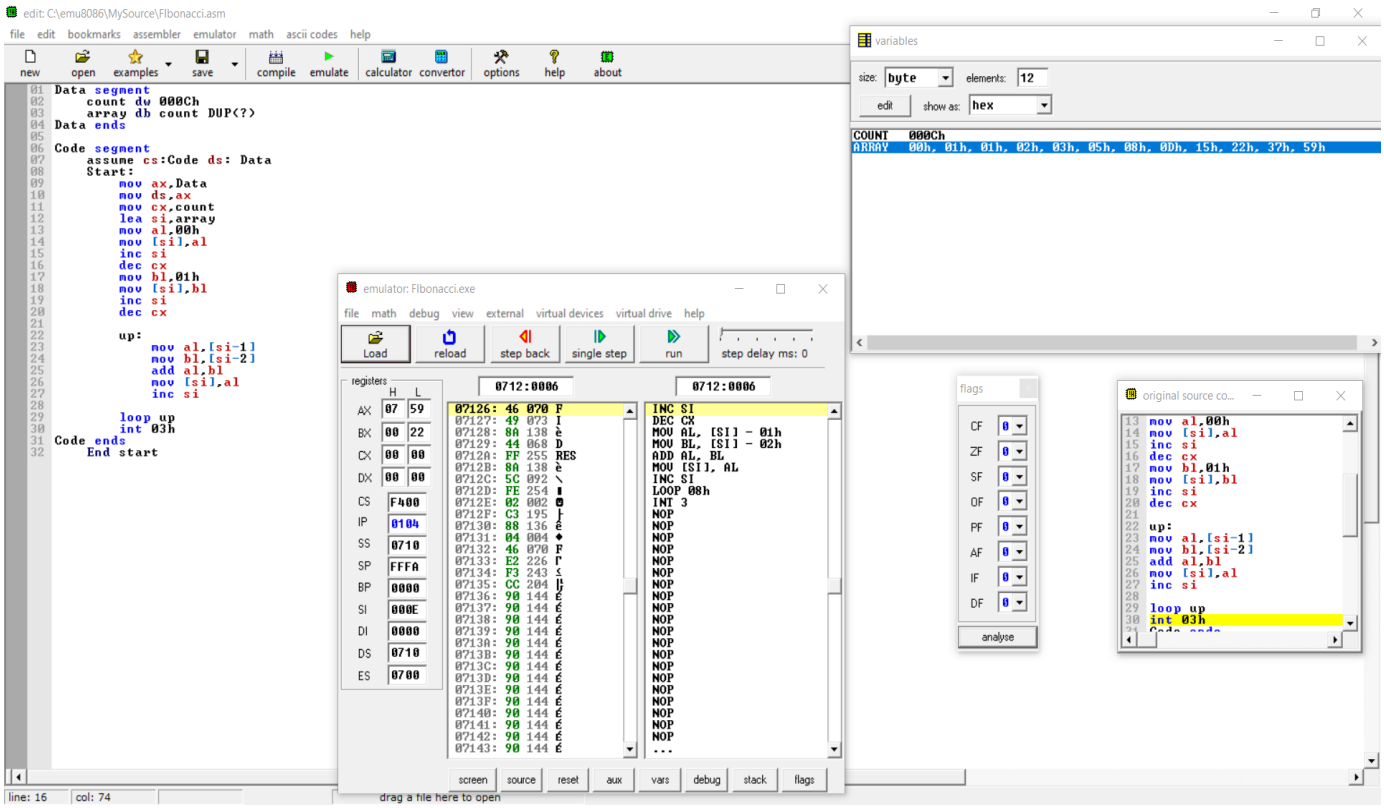
loop up

int 03h

Code ends

End start

Output:



EXPERIMENT NO- 3

AIM: WAP to find factorial of number using procedure.

Resource Required: P-IV and above RAM 128MB, Dot Matrix Printer, Emu 8086, MASM 611/ TASM, Turbo C/C++, Printer, Printout Stationary.

THEORY:

Instructions used in this program are:

1) JNZ (Jump if not Zero): This is conditional Jump. This instruction will jump to specified label when zero flag is not set.

2) DEC: DEC decrements the source by one

Syntax: DEC source

3) CMP: Compare the numerical value of the destination with the source and set flags appropriately. This comparison is carried out in the form of a subtraction to determine which of the operands has a greater value. After a CMP instruction, OF, SF, ZF and CF are set appropriately. For example, if the operands have equal values, then ZF if set.

Syntax:

CMP destination, source

4) CALL AND RET : These instructions interrupts the flow of a program by passing control to an internal or external subroutine .The return instruction returns the control from a subroutine back to a calling program .CALL passes the control to a label specified after the call keyword. When the subroutine ends with return instruction, the instruction following CALL are processed.

ALGORITHM:

Step I : Initialize the data segment

Step II : Initialize the variable A to number

Step III : Move the contents of variable A to AX

Step IV : Call procedure factorial

Step V : Decrement the value of variable A

Step VI : Multiply A with AX

Step VII : Copy the value of A to CX

Step VIII : Compare contents of CX with 01

Step IX : If equal go to step X else step V

Step X : Return to calling program

Step XI : Display the value in Fact variable

Step XII : Stop

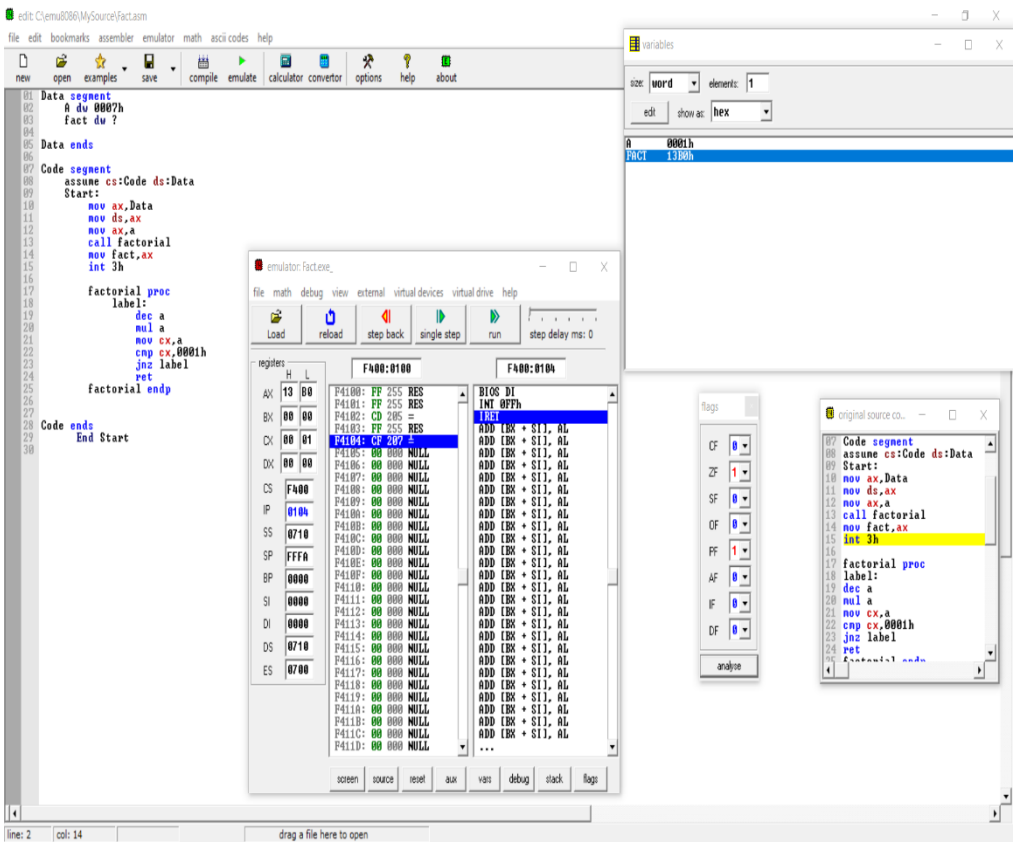
CONCLUSION: We have successfully calculated Factorial of desired number using Procedural as well as normal programming approach in Assembly language using EMU 8086.

<u>Procedural Code</u>:

```
Data segment
   A dw 0007h
   fact dw ?

Data ends

Code segment
   assume cs:Code ds:Data
   Start:
      mov ax,Data
      mov ds,ax
      mov ax,a
      call factorial
      mov fact,ax
      int 3h

      factorial proc
```

```
        label:
            dec a
            mul a
            mov cx,a
            cmp cx,0001h
            jnz label
            ret
    factorial endp

Code ends
    End Start
```

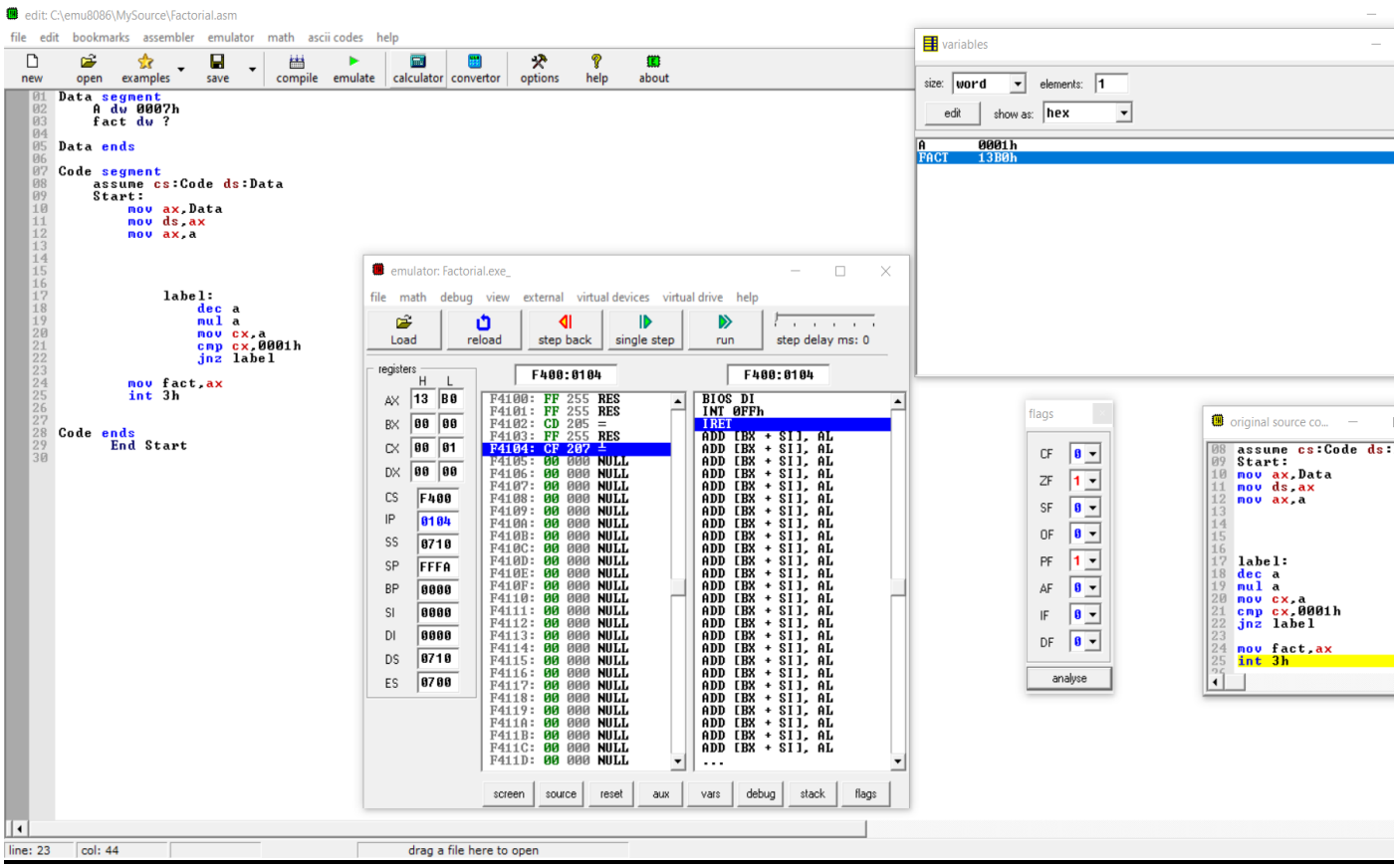
Output:



<u>Normal code</u>:

```
Data segment
   A dw 0007h
   fact dw ?

Data ends

Code segment
   assume cs:Code ds:Data
   Start:
      mov ax,Data
      mov ds,ax
      mov ax,a
         label:
            dec a
            mul a
            mov cx,a
            cmp cx,0001h
            jnz label

      mov fact,ax
      int 3h

Code ends
     End Start
```

Output:


edit: C:\emu8086\MySource\Factorial.asm
Data segment
A dw 0007h
fact dw ?
Data ends
Code segment
assume cs:Code ds:Data
Start:
mov ax,Data
mov ds,ax
mov ax,a
label:
dec a
mul a
mov cx,a
cmp cx,0001h
jnz label
mov fact,ax
int 3h
Code ends
End Start
emulator: Factorial.exe_
variables
A 0001h
FACT 13B0h
flags
original source co...
line: 23
col: 44
drag a file here to open

EXPERIMENT NO- 4

AIM: WAP based on string instruction to transfer a block and to find string length.

Resource Required: P-IV and above RAM 128MB, Dot Matrix Printer, Emu 8086, MASM 611/ TASM, Turbo C/C++, Printer, Printout Stationary.

THEORY:

String Instructions:

String is a group of bytes/words and their memory is always allocated in a sequential order.

Following is the list of instructions under this group −

1. REP − Used to repeat the given instruction till CX ≠ 0.
2. MOVS/MOVSB/MOVSW − Used to move the byte/word from one string to another.
3. COMS/COMPSB/COMPSW − Used to compare two string bytes/words.
4. SCAS/SCASB/SCASW − Used to scan a string and compare its byte with a byte in AL or string word with a word in AX.
5. LODS/LODSB/LODSW − Used to store the string byte into AL or string word into AX.

ALGORITHM:

a) To transfer a block of data (non-overlapping block)

Step I : Initialize the data in the source and destination memory

Step II : Initialize SI and DI with source and destination address

Step III : Initialize CX with count

Step IV : Initialize DF flag to zero

Step V : Transfer the data byte from source to destination

Step VI : Repeat above step till CX!=0

Step VII : Display the result.

Step VIII : Stop.

b) To find the length of String

Step I : Initialize the data segment

Step II : Initialize the len variable to 0

Step III : Point the starting element of the list by SI register

Step IV : Compare [si] with '$'

Step V : If equal go to VIII step else step V

Step VI : Increment len

Step VII : Increment SI

Step VIII : Display the length

Step IX End

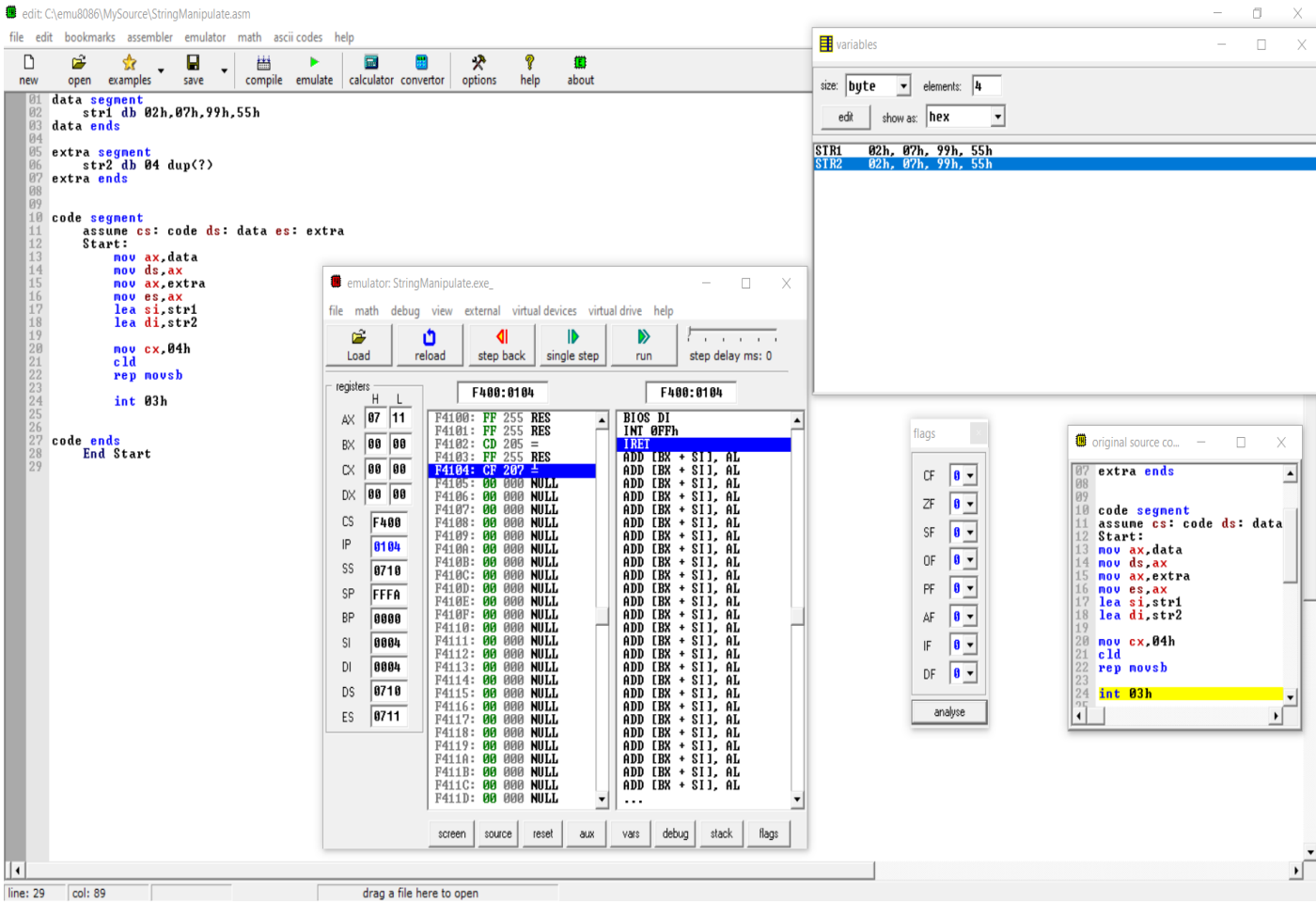
<u>CONCLUSION</u>: We have successfully a)Transfer data from one block to another

b)Calculated length of a string

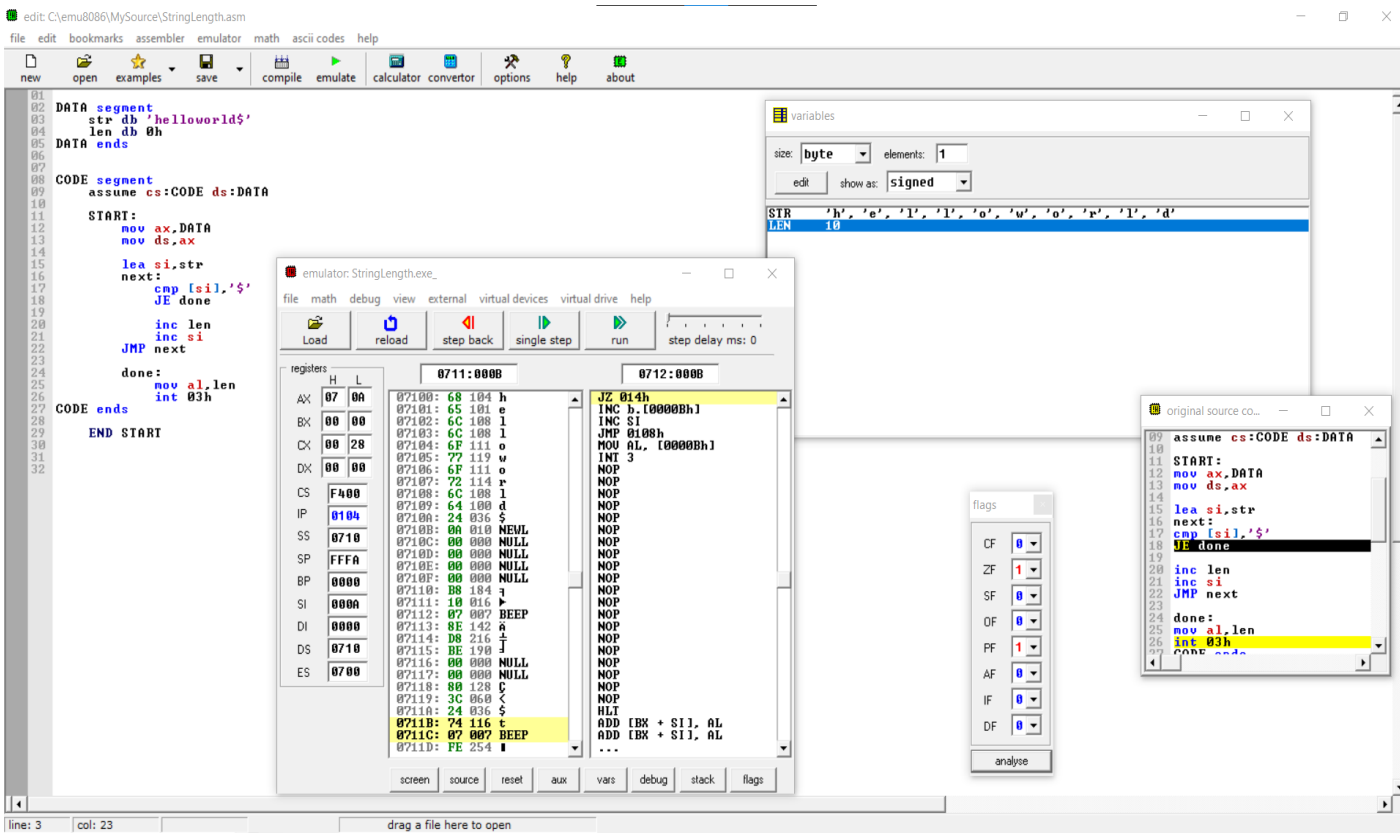
in Assembly Language using EMU8086.

<u>Code and Output</u>:

a)<u>Transferring block of data</u>:



b) String Length:



EXPERIMENT NO- 5

AIM: WAP to perform addition, subtraction, multiplication and division using assembly language in C++ (Mixed language programming)

Resource Required: P-IV and above RAM 128MB, Dot Matrix Printer, Emu 8086, MASM 611/ TASM, Turbo C/C++, Printer, Printout Stationary.

THEORY:

Assembly is useful for time-critical or real-time processes, because unlike with high-level languages, there is no ambiguity about how the code will be compiled**.**

Inline Assembly

One of the most common methods for using assembly code fragments in a C programming project is to use a technique called **inline assembly**. Inline assembly is invoked in different compilers in different ways. Also, the assembly language syntax used in the inline assembly depends entirely on the assembly engine used by the C compiler. Microsoft C++, for instance, only accepts inline assembly commands in MASM syntax, while GNU GCC only accepts inline assembly in GAS syntax(also known as AT&T syntax) .

- Microsoft C Compiler
- GNU GCC Compiler
- Borland C Compiler

a. Linked assembly

When an assembly source file is assembled by an assembler, and a C source file is compiled by a C compiler, those two **object files** can be linked together by a **linker** to form the final executable. The only disadvantages of mixing assembly and C in this way are that

a) both the assembler and the compiler need to be run, and

b) those files need to be manually linked together by the programmer.

These extra steps are comparatively easy, although it does mean that the programmer needs to learn the command-line syntax of the compiler, the assembler, and the linker.

- **Inline Assembly vs. linked assembly**

  Advantages of inline assembly:

  Short assembly routines can be embedded directly in C function in a C code file. The mixed-language file then can be completely compiled with a single command to the C compiler (as opposed to compiling the assembly code with an assembler, compiling the C code with the C Compiler, and then linking them together). This method is fast and easy.

  Advantages of linked assembly:

  If a new microprocessor is selected, all the assembly commands are isolated in a ".asm" file. The programmer can update just that one file -- there is no need to change any of the ".c" files (if they are portably written).

CONCLUSION: We have successfully performed arithmetic operation using mixed language programming in C/C++.

Code:

```
#include<stdio.h>

#include<conio.h>

void main()

{

int a, b, c, q, r;

int ch, ans;

clrscr();

do

{
```

```
printf("\n****MENU**");
printf("\n1.Addition.\n2.Subtraction.\n3.Multiplication.\n4.Division.\n");
printf("\tEnter your choice: ");
scanf("%d", &ch);
printf("\tEnter first number: ");
scanf("%d", &a);
printf("\tEnter second number: ");
scanf("%d", &b);
switch(ch)
{
        case 1:
                asm mov ax, a;
                asm mov bx, b;
                asm add ax, bx;
                asm mov c, ax;
                printf("\n\tAddition result is: %d", c);
                break;
        case 2:
                asm mov ax, a;
                asm mov bx, b;
                asm sub ax, bx;
                asm mov c, ax;
                printf("\n\tSubtraction result is: %d", c);
```

```
            break;
        case 3:
            asm mov ax, a;
            asm mov bx, b;
            asm mul bx;
            asm mov c, ax;
            printf("\n\tMultiplication result is: %d", c);
            break;
        case 4:
            asm mov ax, a;
            asm mov bx ,b;
            asm div bx;
            asm mov q, ax;
            asm mov r, dx;
            printf("\n\tDivision reault is: %d,%d", q, r);
            break;
        default:
            printf("\nINVALID INPUT!!");
}
printf("\nDo you want to continue?(1/0)");
scanf("%d", &ans);
}while(ans == 1);
getch();
```

}

Output:

```
****MENU**
1.Addition.
2.Subtraction.
3.Multiplication.
4.Division.
        Enter your choice: 1
        Enter first number:25
        Enter second number: 34

        Addition result is: 59
Do you want to continue?(1/0)1

****MENU**
1.Addition.
2.Subtraction.
3.Multiplication.
4.Division.
        Enter your choice: 4
        Enter first number: 56
        Enter second number: 8

        Division reault is: 7,0
Do you want to continue?(1/0)
```

```
****MENU**
1.Addition.
2.Subtraction.
3.Multiplication.
4.Division.
        Enter your choice: 2
        Enter first number: 44
        Enter second number: 22

        Subtraction result is: 22
Do you want to continue?(1/0)1

****MENU**
1.Addition.
2.Subtraction.
3.Multiplication.
4.Division.
        Enter your choice: 3
        Enter first number: 23
        Enter second number: 2

        Multiplication result is: 46
Do you want to continue?(1/0) _
```

EXPERIMENT NO- 6

AIM: WAP to find GCD of two numbers

Resource Required: P-IV and above RAM 128MB, Dot Matrix Printer, Emu 8086, MASM 611/ TASM, Turbo C/C++, Printer, Printout Stationary.

THEORY: GCD stands for Greatest Common Divisor. GCD is largest number that divides the given numbers. The instructions used in this program are:

1) MOV Destination Source

Move a byte/word from the source to the destination specified in the instruction.

Source: Register, Memory location, immediate number

Destination: Register, Memory location

Both source & destination cannot be memory locations.

Example: MOV Register, Register

MOV Memory location, Register

MOV Register, Memory location

MOV Register data.

2) JUMP Label/address: This is unconditional jump .It transfers the control of execution to the specified address

3) JE label/address: This is conditional jump. These instructions only execute when the ZF = 1 is true.

4) JB label/address: This is conditional jump. These instructions only execute when the CF = 1 is true.

ALGORITHM:

Step I : Initialize the data segment

Step II : Load AX and BX registers with the operands.

Step III : Check if the two numbers are equal. If yes goto step X, else goto step IV.

Step IV : Is number 1 > number 2? If yes goto step VI else goto step V.

Step V : Exchange the contents of AX and BX register, such that AX contains the bigger number.

Step VI : Initialize DX register with 00H.

Step VII : Perform the division operation (contents of AX / contents of BX).

Step VIII : Check if there is remainder. If yes goto step IX, else goto step X.

Step IX : Move the remainder into AX register and goto step IV

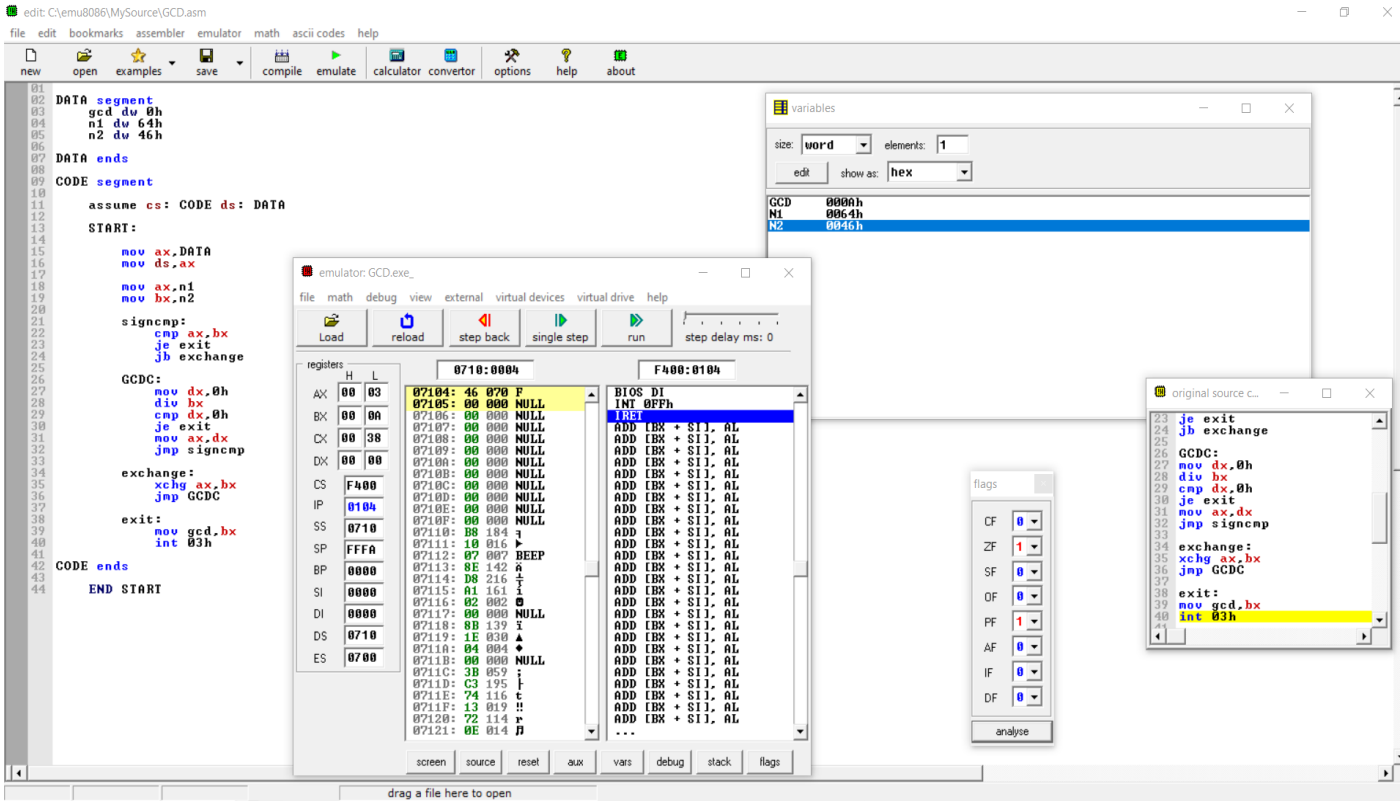
Step X : Save the contents of BX as GCD.

Step XI : Display the result.

Step XII : Stop.

CONCLUSION: We have successfully calculated GCD of two numbers using Euclidean algorithm in Assembly language using EMU 8086

<u>Code and Output</u>:



EXPERIMENT NO- 7

AIM: WAP to find Maximum number from an array

Resource Required: P-IV and above RAM 128MB, Dot Matrix Printer, Emu 8086, MASM 611/ TASM, Turbo C/C++, Printer, Printout Stationary.

THEORY:

Instructions used in this program are:

1) **LEA:** Load Effective Address loads the specified register with the offset of a memory location.

   Syntax: LEA register, memory

2) **JC (Jump if carry):** This is conditional Jump. This instruction will jump to specified label when carry flag is set.

3) **JNZ (Jump if not Zero):** This is conditional Jump. This instruction will jump to specified label when zero flag is not set.

4) **DEC:** DEC decrements the source by one

   Syntax: DEC source

5) **CMP:** Compare the numerical value of the destination with the source and set flags appropriately. This comparison is carried out in the form of a subtraction to determine which of the operands has a greater value. After a CMP instruction, OF, SF, ZF and CF are set appropriately. For example, if the operands have equal values, then ZF if set.

   Syntax:
   ```
   CMP destination, source
   ```

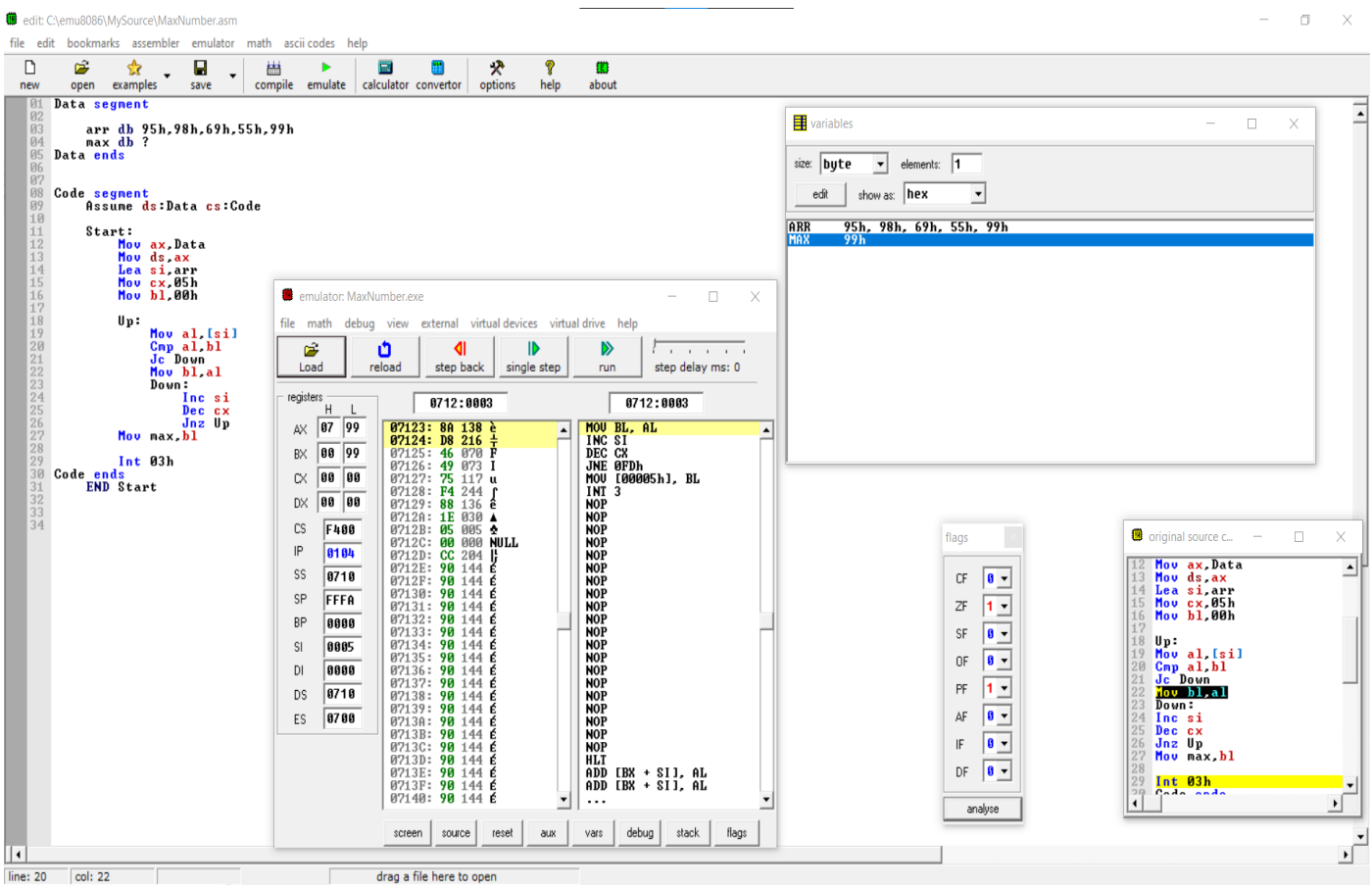
ALGORITHM:

Step I : Initialize the data segment
Step II : Move the length to CX
Step III : Initialize BL register to 00h

Step IV	:	Load the first number to SI register.
Step V	:	Move contents pointed by SI register to AL register
Step VI	:	Compare the contents of AL and BL register.
Step VII	:	If AL< BL, then go to Step IX else go to Step VIII
Step VIII	:	Move the contents of AL to BL register
Step IX	:	Increment SI
Step X	:	Decrement CX contents by 1
Step XI	:	Repeat till CX=0
Step XII	:	Display the value in max variable
Step XIII	:	Stop

<u>CONCLUSION</u>: We have successfully found the maximum number from an array in Assembly Language using EMU8086.

<u>Code and Output</u>:



EXPERIMENT NO- 8

AIM: WAP to draw rectangle using int10H

Resource Required: P-IV and above RAM 128MB, Dot Matrix Printer, Emu 8086, MASM 611/ TASM, Turbo C/C++, Printer, Printout Stationary.

THEORY:

Instructions used in this program are:

INT 10h / **AH = 0** - set video mode.

input:

AL = desired video mode.

these video modes are supported:

00h - text mode. 40x25. 16 colors. 8 pages.

03h - text mode. 80x25. 16 colors. 8 pages.

13h - graphical mode. 40x25. 256 colors. 320x200 pixels. 1 page.

example:

```
mov al, 13h
mov ah, 0
int 10h
```

INT 10h / **AH = 01h** - set text-mode cursor shape.

input:

CH = cursor start line (bits 0-4) and options (bits 5-7).

CL = bottom cursor line (bits 0-4).

when bit 5 of CH is set to **0**, the cursor is visible. when bit 5 is **1**, the cursor is not visible.

```
; hide blinking text cursor:
mov ch, 32
mov ah, 1
int 10h
```

```
; show standard blinking text cursor:
        mov ch, 6
        mov cl, 7
        mov ah, 1
        int 10h

; show box-shaped blinking text cursor:
        mov ch, 0
        mov cl, 7
        mov ah, 1
        int 10h

;       note: some bioses required CL to be >=7,
;       otherwise wrong cursor shapes are displayed.
```

INT 10h / AH = 2 - set cursor position.

input:

DH = row.

DL = column.

BH = page number (0..7).

example:

```
        mov dh, 10
        mov dl, 20
        mov bh, 0
        mov ah, 2
        int 10h
```

INT 10h / AH = 03h - get cursor position and size.

input:

BH = page number.

return:

DH = row.

DL = column.

CH = cursor start line.

CL = cursor bottom line.

INT 10h / **AH = 05h** - select active video page.

input:

AL = new page number (0..7).

the activated page is displayed.

INT 10h / **AH = 06h** - scroll up window.

INT 10h / **AH = 07h** - scroll down window.

input:

AL = number of lines by which to scroll (00h = clear entire window).

BH = attribute used to write blank lines at bottom of window.

CH, CL = row, column of window's upper left corner.

DH, DL = row, column of window's lower right corner.

INT 10h / **AH = 08h** - read character and attribute at cursor position.

input:

BH = page number.

return:

AH = attribute.

AL = character.

INT 10h / AH = 09h - write character and attribute at cursor position.

input:

AL = character to display.

BH = page number.

BL = attribute.

CX = number of times to write character.

INT 10h / AH = 0Ah - write character only at cursor position.

input:

AL = character to display.

BH = page number.

CX = number of times to write character.

INT 10h / AH = 0Ch - change color for a single pixel.

input:

AL = pixel color

CX = column.

DX = row.

example:

```
mov al, 13h
mov ah, 0
int 10h     ; set graphics video mode.
mov al, 1100b
```

```
mov cx, 10
mov dx, 20
mov ah, 0ch
int 10h     ; set pixel.
```

INT 10h / AH = 0Dh - get color of a single pixel.

input:

CX = column.

DX = row.

output:

AL = pixel color

INT 10h / AH = 0Eh - teletype output.

input:

AL = character to write.

this functions displays a character on the screen, advancing the cursor and scrolling the screen as necessary. the printing is always done to current active page.

example:

```
mov al, 'a'
mov ah, 0eh
int 10h

; note: on specific systems this
; function may not be supported in graphics mode.
```

INT 10h / AH = 13h - write string.

input:

AL = write mode:

bit 0: update cursor after writing;

bit 1: string contains attributes.

BH = page number.

BL = attribute if string contains only characters (bit 1 of AL is zero).

CX = number of characters in string (attributes are not counted).

DL,DH = column, row at which to start writing.

ES:BP points to string to be printed.

example:

```
mov al, 1
mov bh, 0
mov bl, 0011_1011b
mov cx, msg1end - offset msg1 ; calculate message size.
mov dl, 10
mov dh, 7
push cs
pop es
mov bp, offset msg1
mov ah, 13h
int 10h
jmp msg1end
msg1 db " hello, world! "
msg1end:
```

INT 10h / AX = 1003h - toggle intensity/blinking.

input:

BL = write mode:

0: enable intensive colors.

1: enable blinking (not supported by the emulator and windows command prompt).

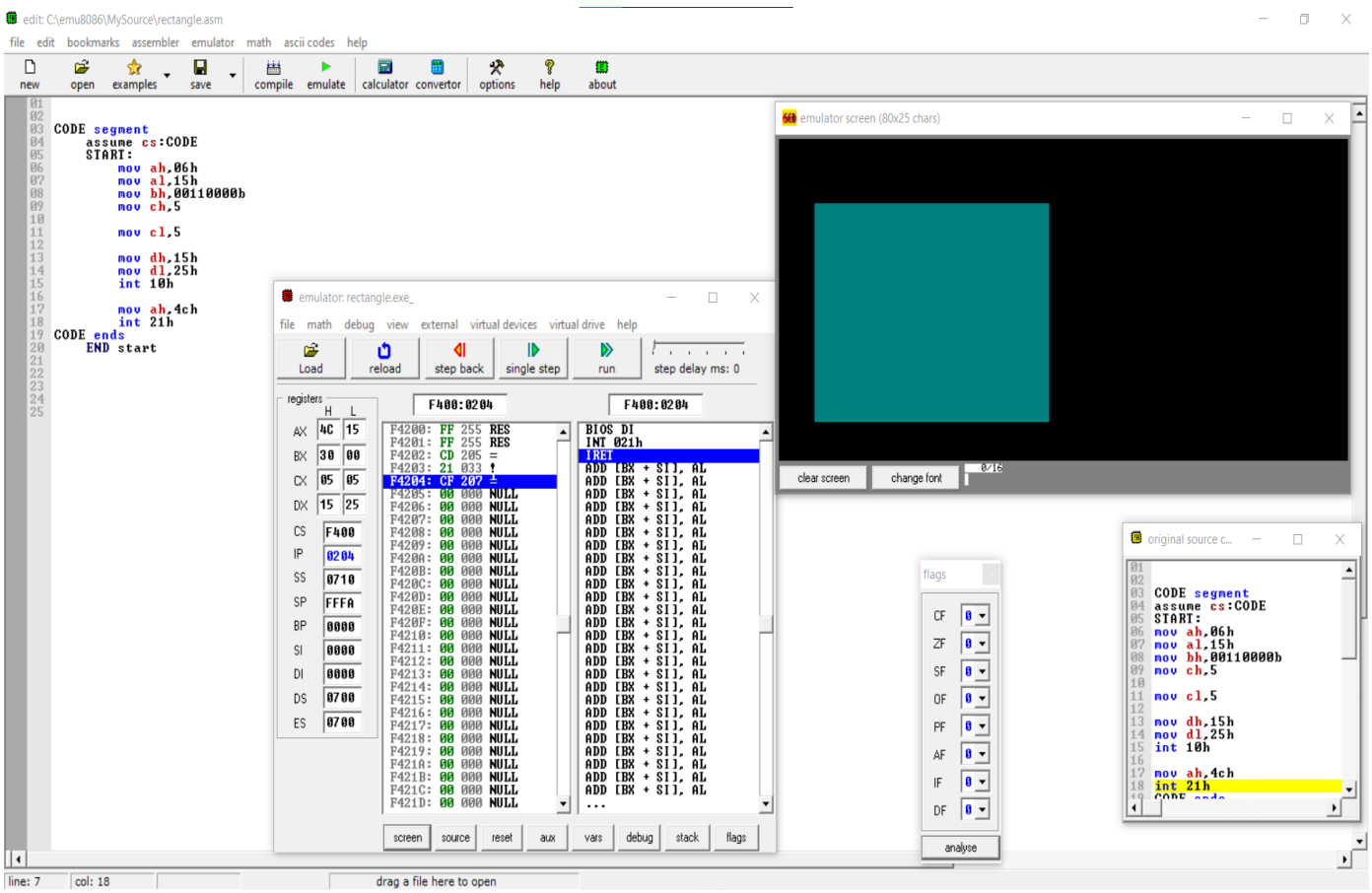
BH = 0 (to avoid problems on some adapters).

example:

```
mov ax, 1003h
mov bx, 0
int 10h
```

CONCLUSION: We have successfully drawn rectangle using INT 10H instruction in assembly language using EMU8086.

<u>Code and Output</u>:



EXPERIMENT NO- 9

AIM: WAP to find Factorial of a number using Macro

Resource Required: P-IV and above RAM 128MB, Dot Matrix Printer, Emu 8086, MASM 611/ TASM, Turbo C/C++, Printer, Printout Stationary.

THEORY:

A **Macro** is a set of instructions grouped under a single unit. It is another method for implementing modular programming in the 8086 microprocessors (The first one was using Procedures).

The **Macro** is different from the Procedure in a way that unlike calling and returning the control as in procedures, the processor generates the code in the program every time whenever and wherever a call to the **Macro** is made.

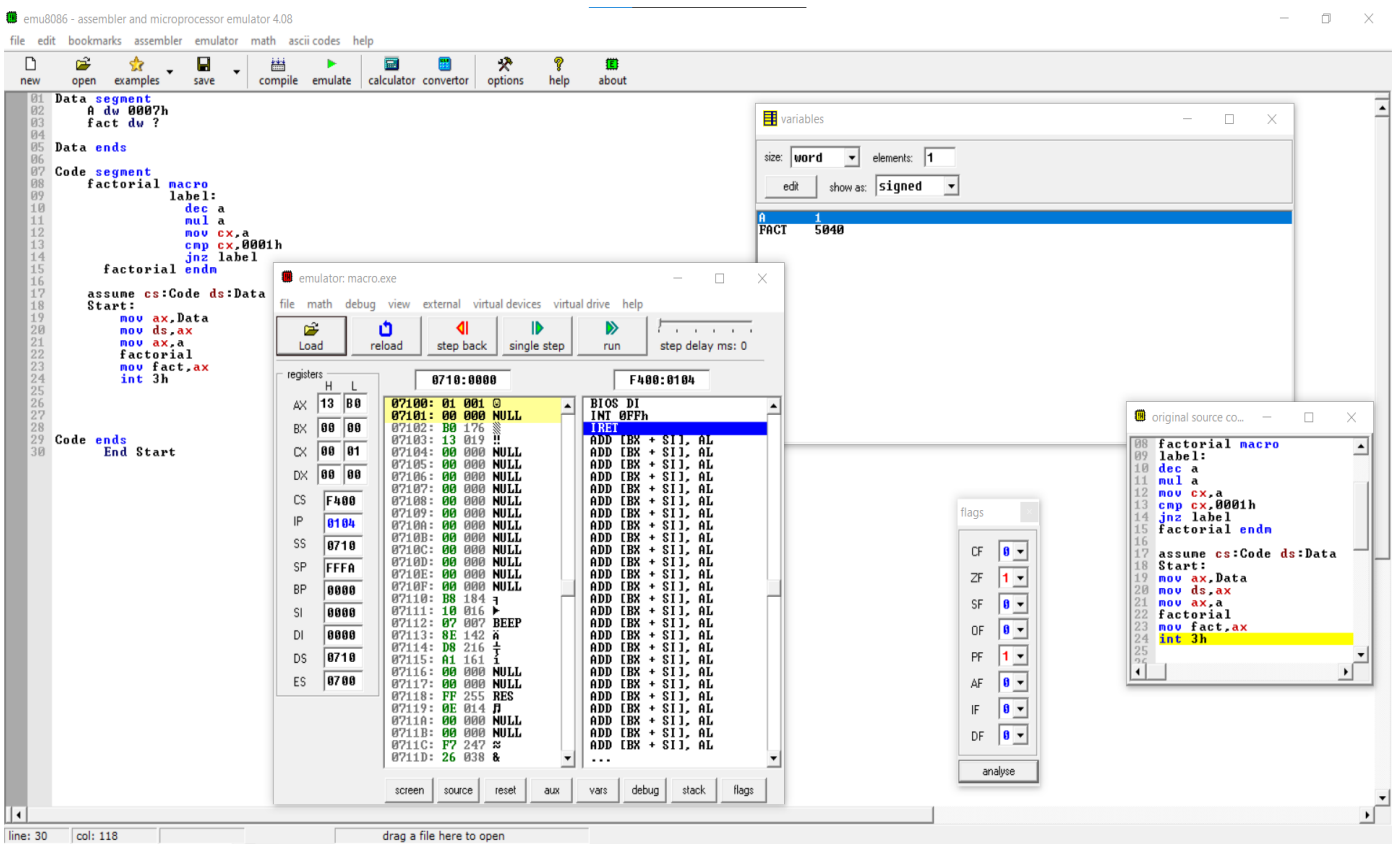
A **Macro** can be defined in a program using the following assembler directives: **MACRO** (used after the name of Macro before starting the body of the Macro) and **ENDM** (at the end of the Macro). All the instructions that belong to the Macro lie within these two assembler directives.

The following is the syntax for defining a **Macro in the 8086 Microprocessor**:

```
Macro_name  MACRO  [ list of parameters ]
    Instruction 1
    Instruction 2
    - - - - - - - - - - -
    - - - - - - - - - - -
    - - - - - - - - - - -
    Instruction n
ENDM
```

CONCLUSION: We have successfully calculated factorial of a number using macro in assembly language using EMU8086.

Code and Output:



EXPERIMENT NO- 10

AIM: WAP for device driver (Accept the input from keyboard and display it)

Resource Required: P-IV and above RAM 128MB, Dot Matrix Printer, Emu 8086, MASM 611/ TASM, Turbo C/C++, Printer, Printout Stationary.

THEORY:

DOS Interrupts: MS-DOS provides many common services through INT 21h. Entire books have been written about the variety of functions available.

Most basic ones for console input and output here.

1) Input a character.

```
MOV AH, 01h
INT 21h
```

After the interrupt, AL contains the ASCII code of the input character. The character is echoed (displayed on the screen). Use function code 8 instead of 1 for no echo.

2) Input a string.

```
SECTION .data
Buffer DB BUFSIZE ;BUFSIZE is max number of chars to read, <= 255
RESB BUFSIZE + 1
SECTION .text
MOV DX, Buffer
MOV AH, 0Ah
INT 21h
```

After the interrupt, BYTE [Buffer + 1] will contain the number of characters read, and the characters themselves will start at Buffer + 2. The characters will be terminated by a carriage return (ASCII code 13), although this will not be included in the count

3) Output a character.

```
MOV DL, ...
MOV AH, 02h
```

INT 21h

Load the desired character into DL, then call the interrupt with function code 2 in AH.

4) Output a string.

MOV DX, ...

MOV AH, 09h

INT 21h

Load the address of a '$'-terminated string into DX, then call the interrupt with function code 9

in AH.

5) Exit.

MOV AL, ...

MOV AH, 4Ch

INT 21h

Load the return code (0 for normal exit, non-zero for error) into AL, then call the interrupt with function code 4Ch in AH.

ALGORITHM:

Step I : Accept input from keyboard

Step II : Check if input=’0’.if yes goto Step IV otherwise continue next line

Step III : Jump to step 1

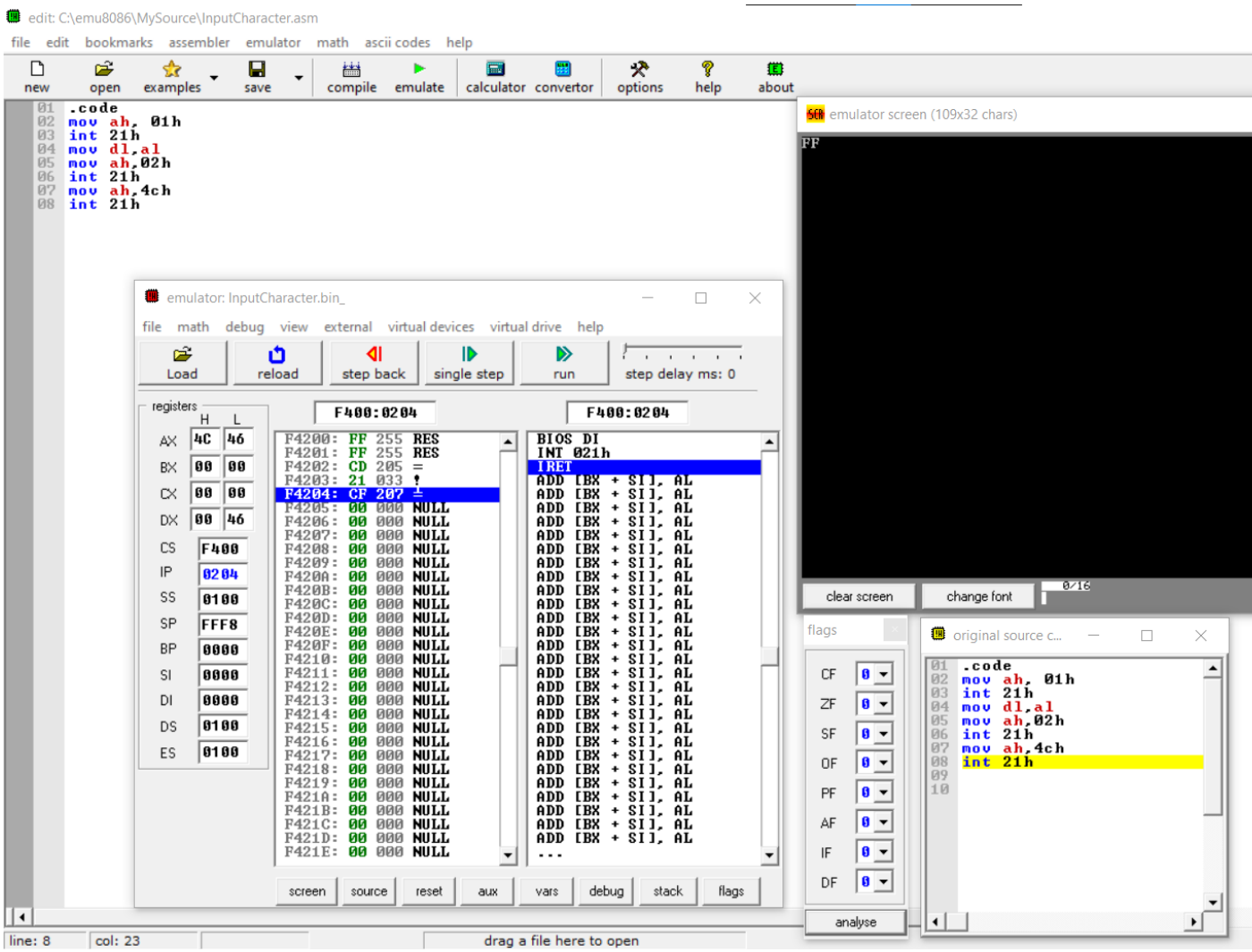
Step IV : Stop

CONCLUSION: We have successfully:

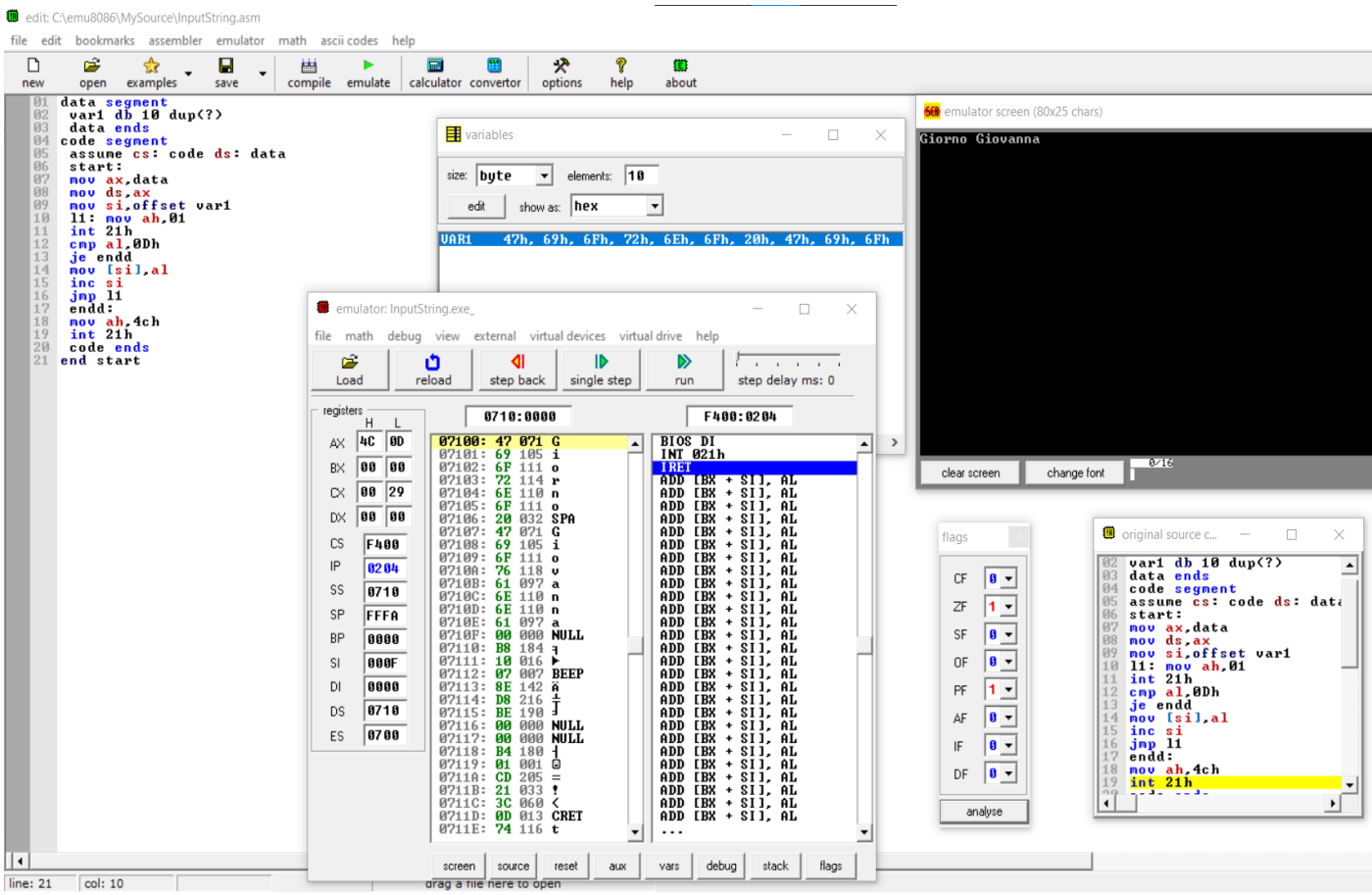
- **I) Accepted input of character from the keyboard and display it on the terminal using INT 21h Instruction in Assembly Language using EMU8086.**
- **II) Accepted input of string from the keyboard using INT 21h Instruction in Assembly Language using EMU8086.**
- **III) Display string on the terminal using INT 21h Instruction in Assembly Language using EMU8086.**

Code and Output:

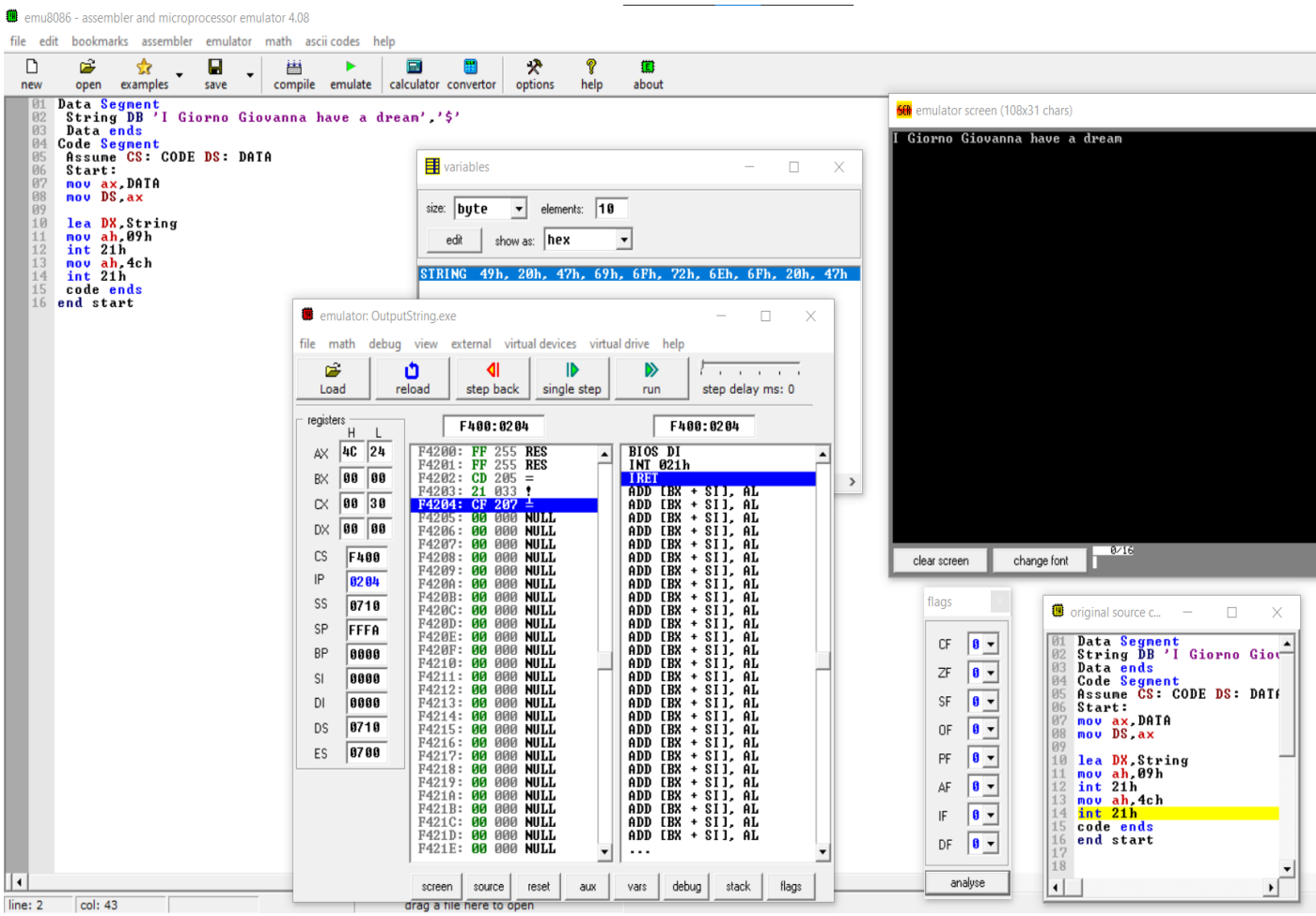
I) Accepting character and displaying it:



II) Accepting String:



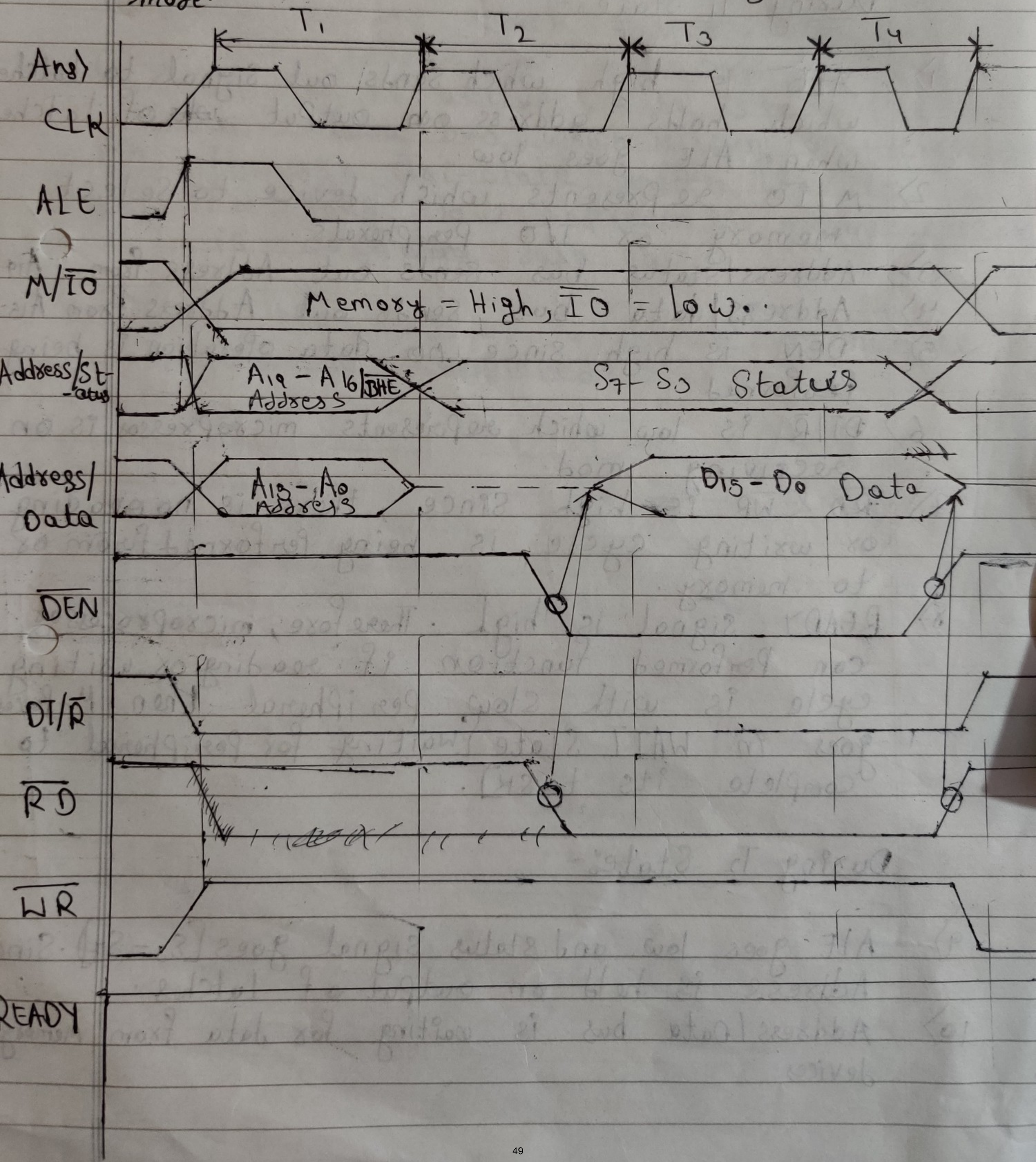
III) Displaying String:



Assignment-1 M.P

Q.>> Draw and explain the read cycle in minimum mode.

Ans>



During T_1 State:-

1) ALE is high which sends out signal to Latches which holds address on output ~~the~~ of Latches when ALE goes low.
2) M/$\overline{IO}$ represents which device to select memory or I/O peripherals.
3) Address/Status bus sends out Address from A_{19}-A_{16}
4) Address/data bus sends out Address from A_{15}-A_0
5) $\overline{DEN}$ is high since no data operation is being performed
6) DT/$\overline{R}$ is low which represents microprocessor is on receiving mod.
7) $\overline{RD}$ & $\overline{WR}$ is high since, there is no reading or writing cycle is being performed from or to memory.
8) READY signal is high. Therefore, microprocessor can performed function if reading or writing cycle is with slow peripheral then μ-Processor goes in WAIT State [waiting for peripheral to complete its task].

During T_2 State:-

9) ALE goes low and status signal goes [S_3-S_7]. Since, Address is held on output of latches.
10) Address/Data bus is waiting for data from Memory/IO devices

During T_3 State:-

1. $\overline{DEN}$ is low which Data bus is available for Transmission of receiving of Data
2. $D_{15}-D_0$ is active, Data is ~~being~~ present on data bus, which is given to microprocessor through Tri-State buffer [8286].
3. $\overline{RD}$ goes low represents microprocessor is receiving data now.
4. At the End of T_4 State a machine cycle is completed.

During -

Q.>> Explain IVT Table.

Ans> ① The Event that causes the Interruption is called interrupt and special routine executed to service the Interrupt is called ISR (Interrupt service Routine).

② IVT - Interrupt vector table of 8086 uses 1st 1 KB of memory from 0000 to 003FFH.

③ This table gives starting-address to 256 ISR

④ Type 0 to 4: Predefined /dedicated interrupts

⑤ Type 5 to 31: Reserved for higher processor

⑥ Type 32 to 255: Available for users.

⑦ To calculate PA of ISR, microprocessor need base address and offset address. ~~For~~ i.e for each ISR four memory location are used

⑧ Therefore, for 256 ISR 1024 memory location are used.

⑨ These ISR are known as type 0 ISR, type 1 ISR, type 2 ISR, ..., type 255 ISR.

II Interrupt Sequence:-

① Push current content of flag CS, IP into Stack memory pointed by S.P. Now S.P is decremented by -2, -2, -2 i.e 6

(256 ISR)	0000	IP_L	Type 1	Type 255 … Type 32 — user available
	0001	IP_H		
	0002	CS_L		
	0003	CS_H		
	↓			Type 31 … Type 5 — Reserved Interrupts
	003FC	IP_L	Type 255	Type 4, Type 3, Type 2, Type 1, Type 0 — dedicated Interrupts
	003FD	IP_H		
	003FE	CS_L		
	003FF	CS_H		

2.) Trap flag and interrupt enable flag both are cleared.

3.) Microprocessor calculate PA of IVT as follows:

$PA = 4 \times 64 = 256 = (00100)_{16}$

4.) Now content of $(00100)_{16}$ & $(00101)_{16}$ is placed in I.P and content of $(00102)_{16}$ & $(00103)_{16}$ is placed in C.S

5.) Microprocessor calculates startup physical address of type 64 ISR by using current content of CS and IP

6) It executes type 64 ISR and performs a particular task.

7.) At the end of ISR, microprocessor ~~execut~~ executes instruction IRET. Now content is returned to main program by popping contents of stack memory pointed by SP into IP

8

8.) Now microprocessor continues the main program.

(III) Interrupt types:-

Type 0 to Type 4 are known as dedicated Interrupts. The reason is these ISR always perform same task.

Dedicated Interrupts:-

1) Type 0 (Divide by zero):- After performing division, if microprocessor is enable to store quotient in destination register, the μP executes type 0 ISR.

2) Type 1 (single step interrupt):- After executing one instruction μP checks Trap flag If Trap flag is set, the μP executes type 1 ISR. Then μP returns to main program and executes next instruction then μP executes type 1 ISR. This action ~~is~~ continues till Trap flag is reset.

3) Type 2 (NMI):- When μP receives request at NMI, then μP executes type 2 ISR. This ISR is used to answer the request at NMI. NMI is used for critical events such as power failure and emergency shut off.

4) Type 3 (Break Point):-

Whenever μP executes instruction INT then μP executes type 3 ISR. This ISR is used for ~~debbage~~ debugging it is used replaced single step whenever single step is time consuming.

5) Type 4 (Over flow):-

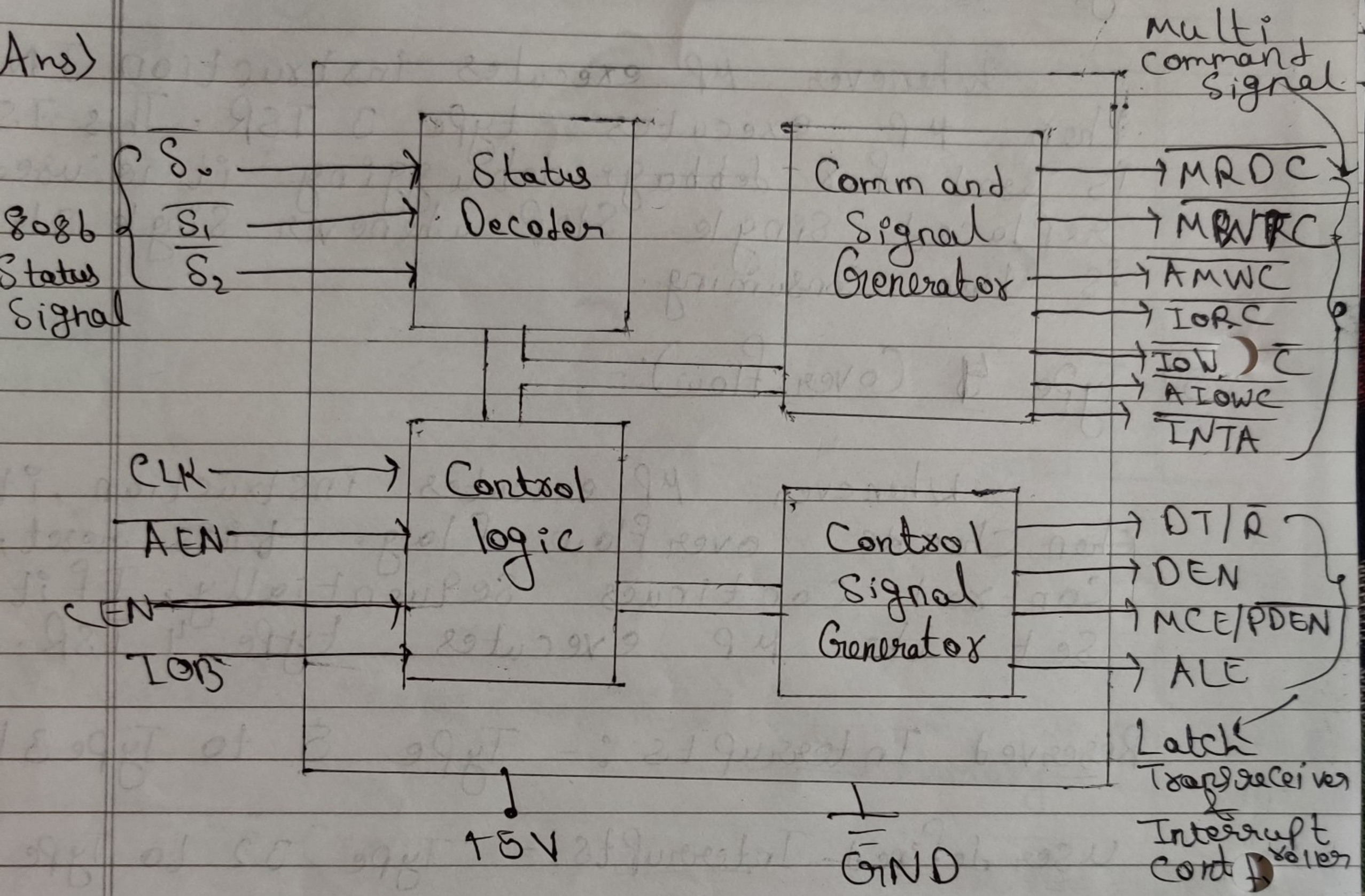
Whenever μP executes instruction, it ~~then~~ checks over flow flag. If it's reset, control continues sequentially. If it is set the μP executes type 4 ISR.

Reserved Interrupts :- Type 5 to Type 31

User defined Interrupts :- Type 32 to Type 255

Q.) Explain 8288 bus controller.

Ans)



① 8288 bus controller are used to produce Control signals.

② The value of control signals depends on $\overline{S_2}, \overline{S_1}, \overline{S_0}$

	$\overline{S_2}$	$\overline{S_1}$	$\overline{S_0}$		Bus cycle
IO	0	0	0		$\overline{INTA}$
	0	0	1		$\overline{IORC}$
	0	1	0		$\overline{IOWC}$ & $\overline{AIOWC}$
	0	1	1		None
M	1	0	0	Instⁿ fetch	$\overline{MRDC}$
	1	0	1		$\overline{MRDC}$
	1	1	0		$\overline{MWTC}$ & $\overline{AMWTC}$
	1	1	1		None.

1.) $\overline{MRDC}$ (Memory Read):- It is used to read data from memory.

2) $\overline{MWTC}$ (Memory write):- It is used to write data into memory.

3.) $\overline{IORC}$ (IO Read):- It is used to read data from IO.

4.) $\overline{IOWC}$ (IO Write):- It is used to write data into IO

5.) $\overline{AMWTC}$ (Advanced Memory write):- Similar to $\overline{MWTC}$ except one function i.e, it is activated one clock cycle earlier.

6.) $\overline{AIOWC}$ (Advanced IO write):- Similar to $\overline{IOWC}$ except one function i.e, it is activated one cycle earlier

7.) ALE :- Enable Latch.

8.) DT/$\overline{R}$:- It is given to transreceiver it is used to control direction of data flow.

9.) DEN :- It is given to transreceiver through NOT gate. It is used to enable data buffer of transreceiver.

10.) $\overline{AEN}$, IOB & CEN :- They are I/P signals used in multiprocessor environment.
If IOB = 0 & $\overline{AEN}$ = 0, then 8288 is in System Bus Mode.
If IOB = 0 & $\overline{AEN}$ = 1, then 8288 is in IO BUS mode.
When CEN = 0, 8288 is disabled.

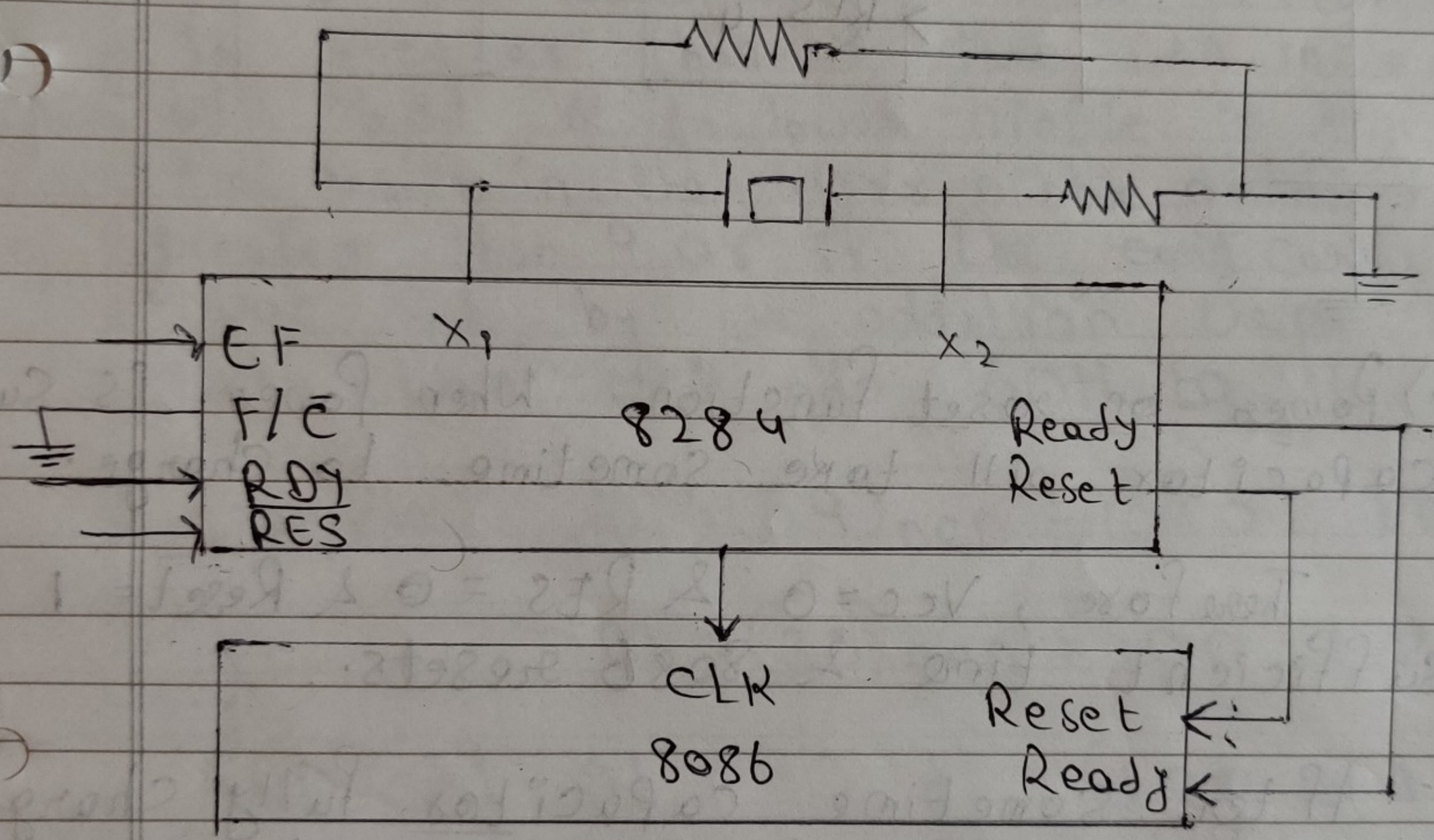
11.) MCE/$\overline{PDEN}$:- It is O/P signal in System Bus mode, MCE = 1 (Master Cascade Enable).
In IO Bus mode, it is known as $\overline{PDEN}$ (peripheral data enable).

In single processor system, 8288 is operated in System bus mode.
CEN = 1, IOB = 0, $\overline{AEN}$ = 0.

Q.) Explain how microprocessor resets.

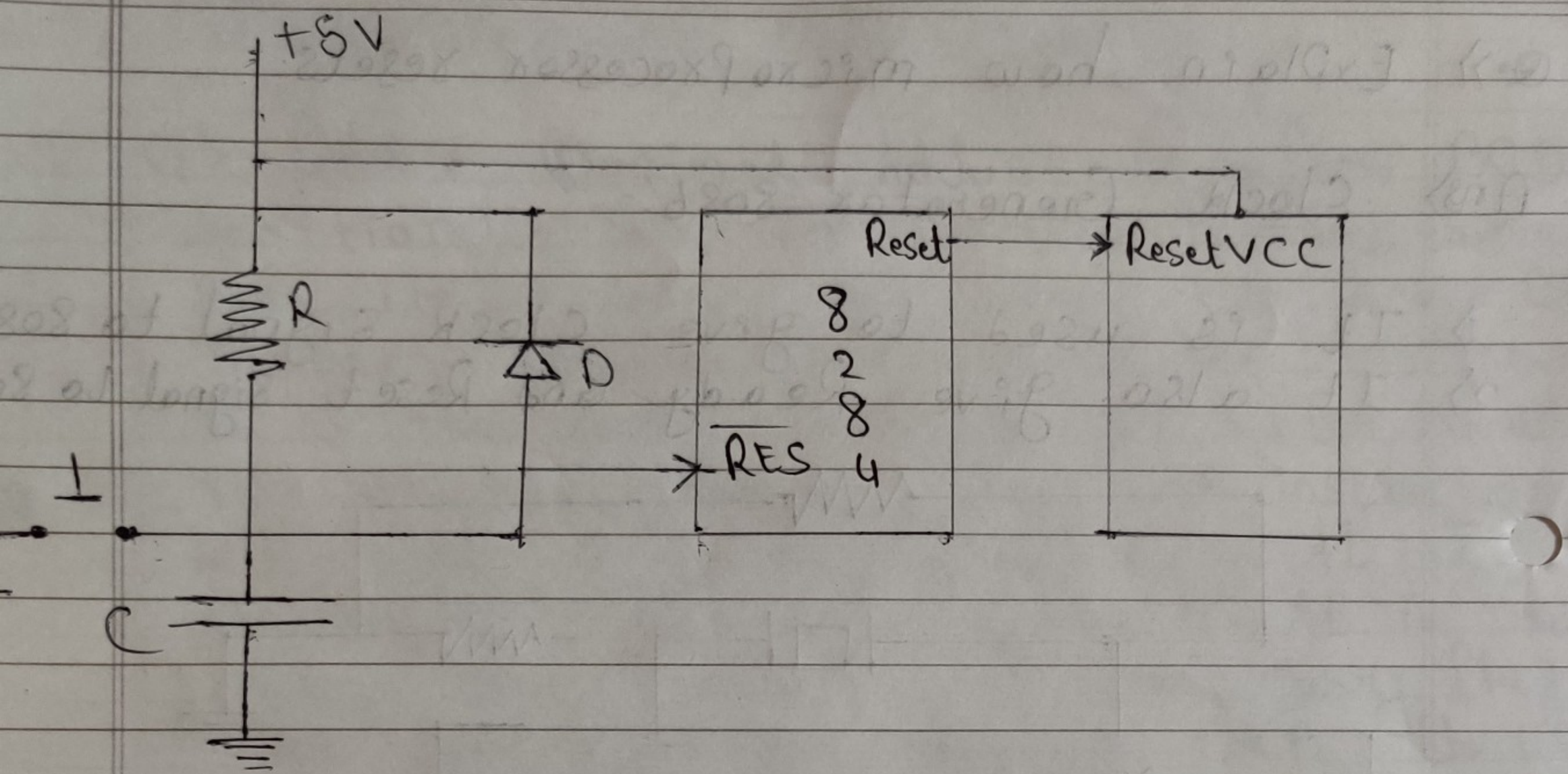
Ans) Clock Generator 8086:-

1) It is used to give clock signal to 8086.
2) It also give Ready and Reset Signal to 8086.



External circuits to Reset 8086:-

8086 can be reset in 2 ways:-
1) Power on reset
2) Manual Reset



1.) Power on reset function:- When Power is supplied, capacitor will take sometime to charge.

Therefore, Vcc = 0 & $\overline{RES}$ = 0 & Reset = 1 for sufficient time & 8086 resets.

- After sometime capacitor fully charges and so Vcc = 1 & so $\overline{RES}$ = 1 & so reset = 0 so 8086 comes out of reset state.

Q.)) Explain following instructions :-

(i) DAA :- (Decimal Adjust AL after BCD Addition) :- This instruction is used to make sure the result of adding two packed BCD numbers is adjusted tobe a legal BCD number. If the lower nibble in A.L after an addition is greater than 9 then DAA instruction will add 6 to lower nibble in AL. If the result in the upper nibble of AL is now greater than 9 or if the ~~cart~~ carry flag was set by the addition then the DAA instruction will add 60H to AL.

Example :- $AL = (59)_{BCD} = (0101\ 1001)_2$

$BL = (35)_{BCD} = (0011\ 0101)_2$

ADD AL, BL $\therefore AL = (1000\ 1110)_2$

$AL = (8E)_{BCD}$

DAA : Add 06 $\because E > 09$

$\therefore AL = (1001\ 0100)_2 = (94)_{BCD}$

$AL = (94)_{BCD}$ which is correct BCD.

(iii) XLAT :- XLAT / XLATB instruction is used to translate a byte from one code to another code. The instruction replaces a byte in the AL register with a byte pointed to By BX in a lookup table in a memory. Before the XLAT instruction can be executed, the lookup table containing the values for new code must be put in memory, and the offset of the starting of the lookup table must be loaded in BX. The code byte to be translated is put in AL. To point to the desired byte in the look up table, the XLAT instruction add the byte in AL to ~~Sta~~ the offset BX. It then copies the Byte from the address pointed by (BX+AL) back into AL.

Example:-

```
MOV BX, OFFSET EBCDIC_TABLE
; Point BX at Start of EBCDIC
; table in DS.

XLAT B
; Replace content in AL with
; EBCDIC from table.
```

The XLAT instruction can be used to convert any code of 8 bits or less to any other code of 8 bits or less.

(iii) CMP :- This instruction compares a byte from the specified source with a byte from the specified destination. [word can also be compare].
The comparison is actually done by subtracting the source byte or word from the destination byte or word. The source and the destination are not changed, but the flags are set to indicate the results of the comparison. AF, OF, SF, ZF, PF and CF are updated by CMP instruction.

example: CMP CX, BX

	CF	ZF	SF	
CX = BX	0	1	0	; Result of ; subtraction is 0
CX > BX	0	0	0	; No borrow required, so CF = 0
CX < BX	1	0	1	; Subtraction ; required borrow ; so, CF = 1

Source can be register, memory location or immediate number.
destination can be register & memory location

2.>> Identify the addressing mode of the following instructions:

i) MOV ax, bx :- Register addressing mode Data is stored in register.

ii) In al, [2000] :- Displacement or direct addressing mode. Content of Port location 2000 is translated is AL.

iii) DAA :- Implied addressing mode correct BCD number is form in AL.

iv) mov ax, [bx+si+4] :- Base indexed addressing Mode. After adding bx, si & 4 result of memory location is placed in ax.

v) mov al, 05h :- Immediate addressing mode. Place 05h in al.

Page :
Date : / /

Assignment - 2

Q.1) Flag register of 80386

Ans)

	31	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
F	Reserved		V	R	0	N	I O P L		O	D	I	T	S	Z	0	A	0	P	1	C
L	for INTEL		M	F		T			F	F	F	F	F	F		F		F		F
A																				
GS																				

FLAG REGISTER OF 80386.

Flags:-

① C (Carry) :- It holds the carry after calculations.

② P (Parity) :- Parity is a logic 0 for odd parity and a logic 1 for even parity. Parity is a count of ones in a number expressed as even or odd.

③ A (Auxiliary Carry) :- Carry occurs bits positions 3 and 5 of the results.

④ Z (Zero) :- The zero flag shows that the result of an arithmetic or logical operation is zero. When Z = 1, the result is zero, when Z = 0, the result was non-zero

⑤ T (Trap) :- Enables trapping through an on-chip debugging facility.

Page:
Date: / /

⑥ S(Sign):- The sign flag holds the arithmetic sign after an arithmetic or a logical operation. If S=1 the sign bit is set and the result is negative. If S=0, the sign bit is not set & the result is positive.

⑦ I (Interrupt):- The interrupt flag controls the operations of the INTR (Interrupt request) input pin. If I=1, the INTR pin is enabled; if I=0, the INTR pin is disabled.

⑧ VM (Virtual mode):- If this flag set, the 80386 enters the virtual mode within the protected mode. In this mode, if any privileged instruction is executed, an exception 13 is generated.

⑨ RF (Resume):- Used by the debug registers DR6 and DR7. It enables you to ~~run~~ turn off certain exceptions while debugging code.

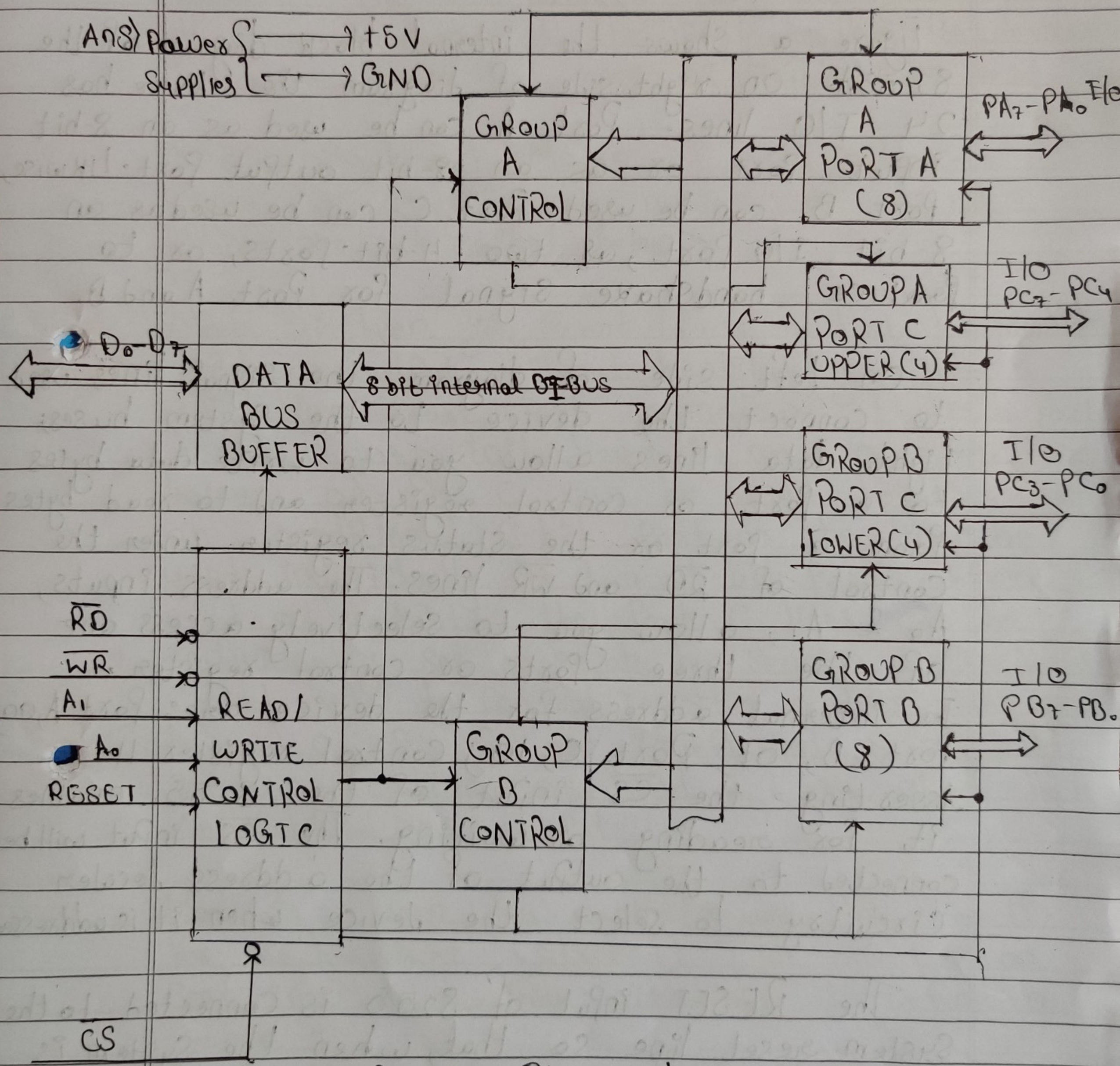
⑩ NT (nested task):- The nested task flag is used to indicated that the current task is nested within another task in protected mode operation. This flag can be manipulated for multitasking.

⑪ IOPL (IO/Privilege level):- IOPL is used in in protected mode operation to select the privilege level for I/O devices. If the current privilege level is higher or more trusted than the IOPL, I/O executed without hindrance. If the IOPL is lower than the current privilege level, an interrupt occurs.

Page :
Date : / /

Q.2) 8255 block diagram.

Ans)



8255 Block diagram

Figure a.

Page:
Date: / /

Internal Block diagram and System Connections of 8255:-

Figure a shows the internal block diagram of the 8255A. On right side of diagram the device has 24 I/O lines. Port A can be used as an 8-bit input port or as an 8-bit output port. Likewise, Port B can be used. Port C can be used as an 8-bit I/O port, as two 4-bit ports, or to produce handshake signal for Port A and B.

On left side of diagram, the signal lines used to connect the device to the system buses: Eight data lines allow you to write data bytes to a port or control register and to read bytes from a port or the status register under the control of $\overline{RD}$ and $\overline{WR}$ lines. The address inputs, A_0 & A_1, allow you to selectively access one of the three ports or control register. The internal address for the device are: Port A, 00; Port B, 01; Port C, 10; Control register 11. Asserting the $\overline{CS}$ input of the 8255 enables it for reading or writing. The $\overline{CS}$ input will be connected to the output of the address decoder circuitry to select the device when it is addressed.

The RESET input of 8255 is connected to the system reset line so that, when the system is reset, all the port lines are intialized as ~~outputs~~ ~~after a power-up or reset~~ input lines. If port lines were initialized as outputs after a power-up or reset, the port might try to output to the output

Page:
Date: / /

of a device connected to the port. The possible argument between the two outputs might destroy one or both of them. Therefore, all the programmable port devices initialize their port lines as inputs when reset.

8255 Operational Modes and Initialization:-
There are 3 different modes of 8255 which are described below:-

Mode 0:- When you want to use a port for simple input or output without handshaking, you initialize that port in mode 0. If both port A and B are initialized in mode 0, then two halves of port C can be used together as an additional 8-bit port, or they can be used individually as two 4-bit ports. When used as outputs, the port C lines can be individually set or reset by sending a special control word to control register address. The two halves of port C are independent, so one half can be initialized as input, & the other half initialized as output.

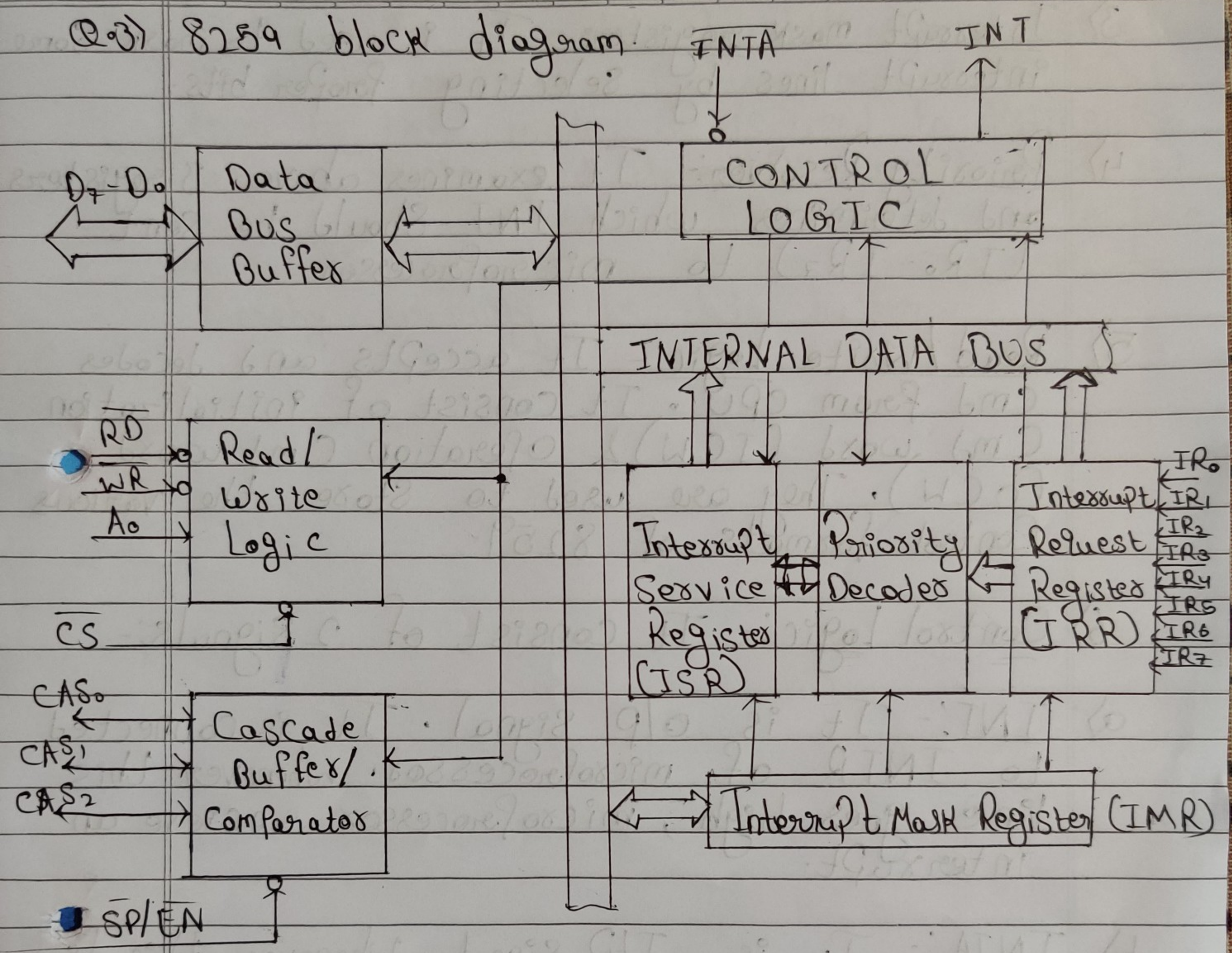
Mode 1:- When you want to use port A or B for a handshake (strobed) I/O operation as seen in Mode 0, you initialized that port in mode 1.
In thise mode, some of the pins of port C function as handshake lines. Pins PC_0, PC_1 & PC_2 function as handshake lines for port B if it is initialized in mode 1. If port A is initialized as

a handshake (mode 1) input port, then pins PC_3, PC_4 and PC_5 function as handshake signals. Pins PC_6 and PC_7 are available for use as input lines or output lines. If Port A is initialized as a handshake ~~signals. Port C~~ output port, then port C pins PC_3, PC_6, and PC_7 function as handshake signals. Port C pins PC_4 and PC_5 are available for use as input or output lines.

MODE 2:- Only port A can be initialized in mode 2. In mode 2, port A can be used for bidirectional handshake data transfer. This means that data can be output or input on the same eight lines. The 8255 might be used in this mode to extend the system bus to a slave microprocessor or to transfer data bytes to and from a floppy disk controller board. If port A is initialized in mode 2, then pins PC_3 through PC_7 are used as handshake lines for port A. The other three pins, PC_0 through PC_2, can be used for I/O if portB is in mode 0. The three pins will be used for port B handshake lines if port B is initialized in mode 1.

Page :
Date : / /

Q.3) 8259 block diagram.



1) Interrupt Request Register:- It has 8 input lines $IR_7 - IR_0$. Peripheral devices are connected to these line. When these lines goes high the requests are are stored in this register.

2) Interrupt Service Register:- It is used to store all the lines which are being serviced.

Page:
Date: / /

3) Interrupt mask register:- It is used to mask some interrupt lines by selecting proper bits.

4) Priority Resolver:- It examines above 3 registers and determines which INT should be sent ($IR_0 - IR_7$) to microprocessor

5) Read/Write Logic:- It accepts and decodes cmd from CPU. It consist of initialization cmd word (ICW) & operation cmd word (OCW). They are used to store the various control formats of 8259.

6) Control Logic:- It consist of 2 signals:-

a) INT:- It is o/p signal. It is connected to INTR of microprocessor. Whenever this line goes high, microprocessor recieve an interrupt.

b) INTA:- It is I/P signal. Whenever this line goes active low microprocessor acknowledge the arrival of interrupt request to 8259.

7) Data bus buffer:- It is 8-bit bidirectional buffer. It is used to interface the 8259 to system bus.

8) Cascade buffer/comparator:- It is used in cascade mode. It is used to increase no. of interrupt levels by cascading many 8259. It uses cascade lines CAS_2, CAS_1, CAS_0 & $\overline{SP}/\overline{EN}$.

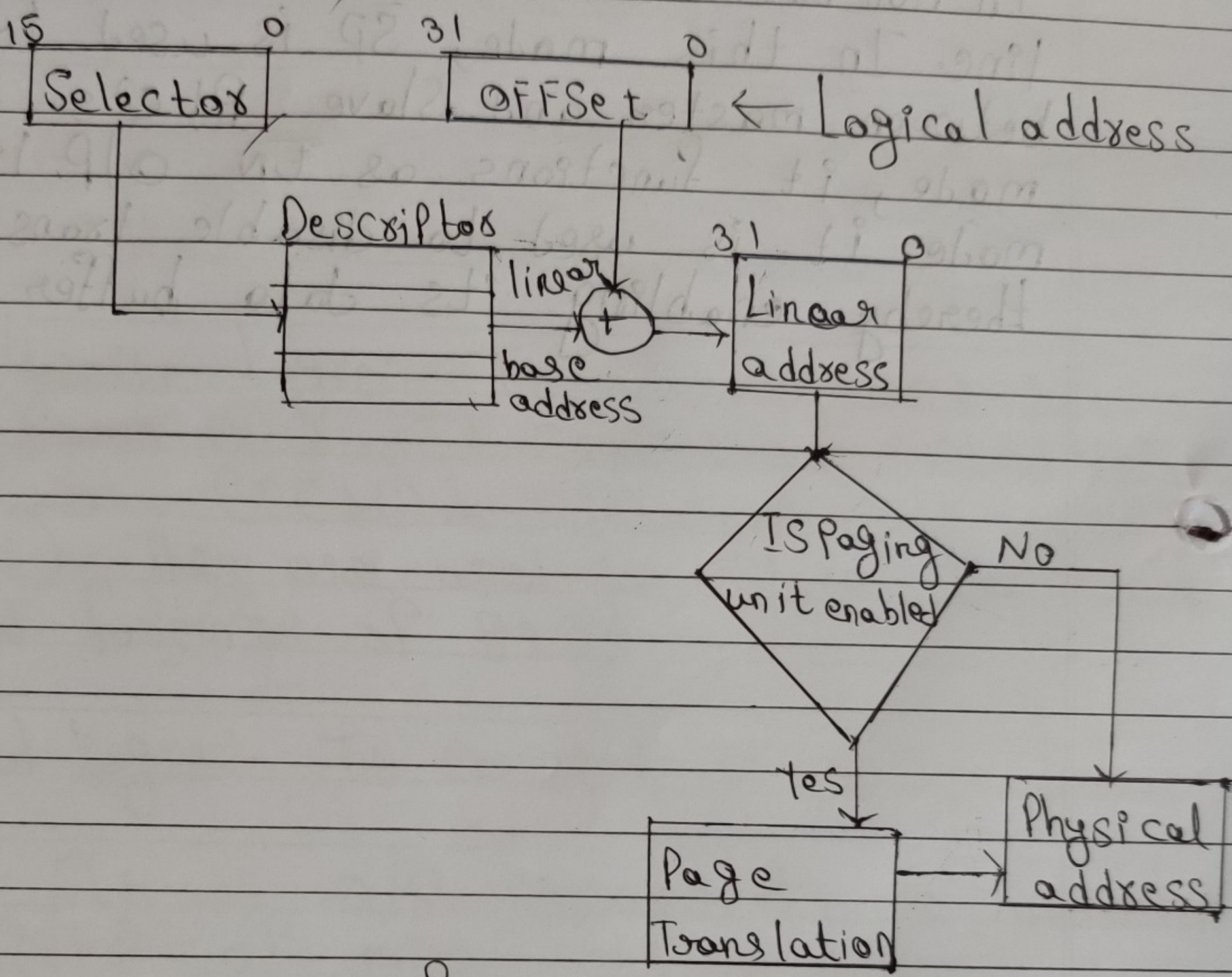
Page :
Date : / /

9) $\overline{SP}$ / $\overline{EN}$:- It is active low bidirectional 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. In buffered mode, it functions as $\overline{EN}$ o/p line. In this mode it is used to disable transreceiver & thereby enabling its data buffer.

Page:
Date: / /

4.) Address translation of 80386.

Ans)



Overview of address translation.

① The 80386 has three distinct address:-
i) logical
ii) Linear
iii) Physical

② Logical address consists of a selector & offset.

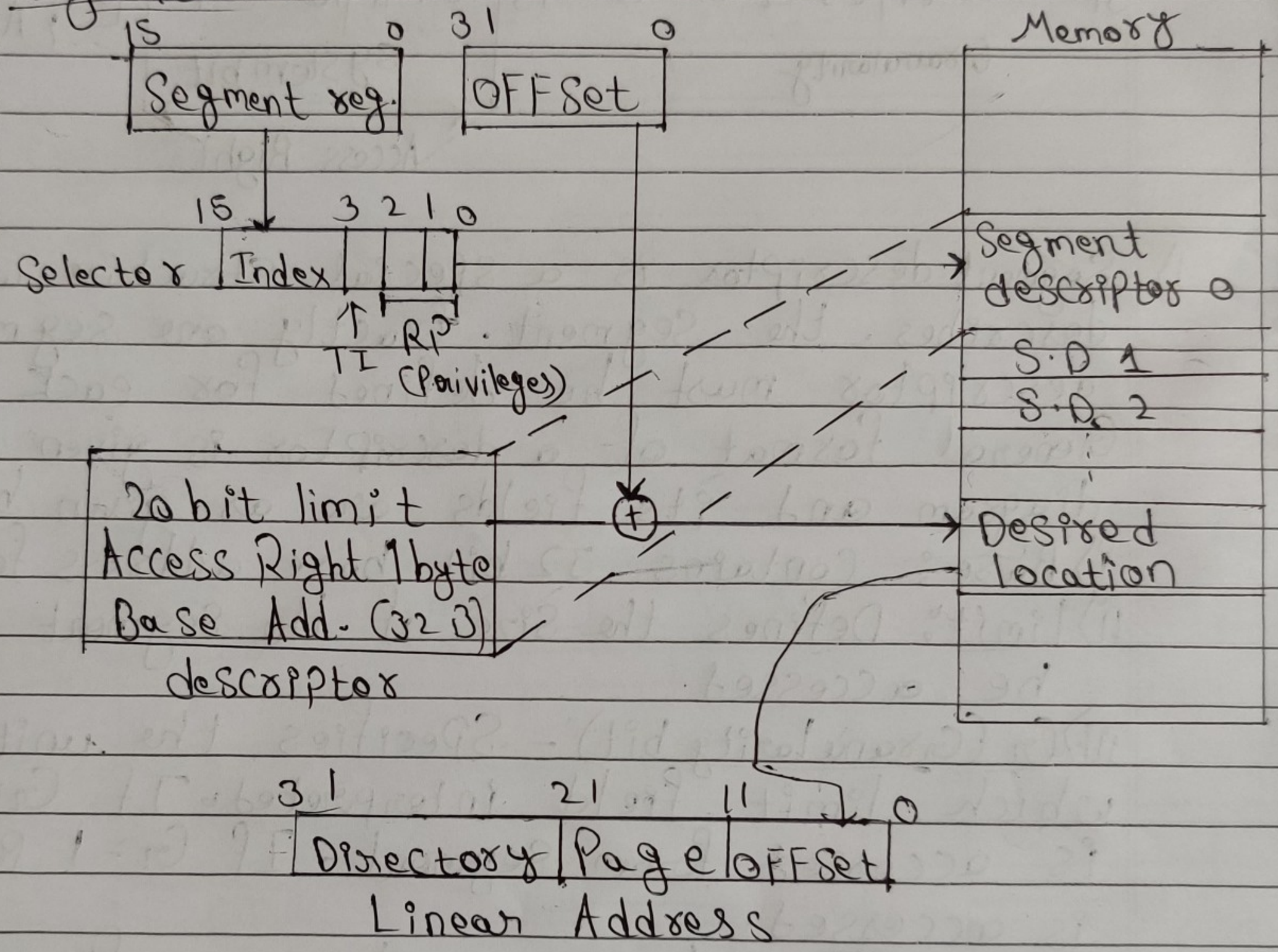
③ In real mode, the segmentation unit adds the selector with offset to form linear address while in protected mode, every segment has linear base address which is stored in segment descriptor.

4) The linear base address from the descriptor is added to offset address to generate 32-bit linear address. This process is called segmentation.

Page :
Date : / /

④ If paging unit is enabled, paging mechanism translates linear address into physical address.

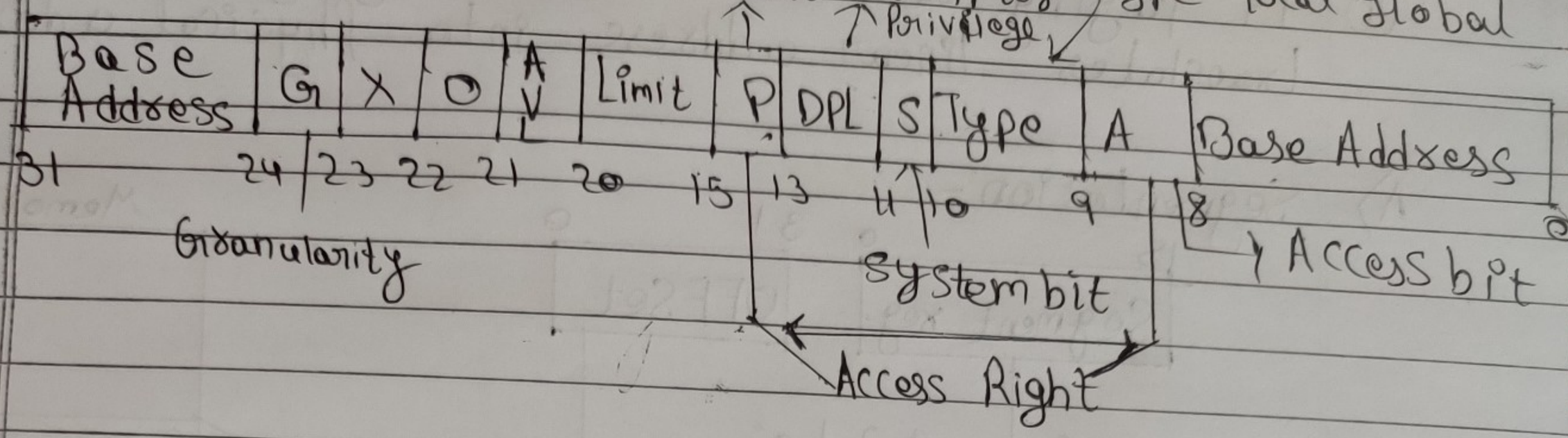
A) Segmentation:-



1) The selector component of each logical address contains 2 bit of privilege level of a program.
2) The descriptor of each segment also contains 2 bit of privilege level of that segment.
~~3) If the privilege of each segment also contain~~
3) If the privilege level in selector is greater than privilege level of descriptor, then MMU allows the program to access the segment.
4) If TI = 0, then global descriptor table is used, else local descriptor is used.

Page :
Date : / /

I.) **Descriptor Format:-** Present Descriptor, Privilege, Type - local global



1) Segment descriptor is a special structure which describes the segment. Exactly one segment descriptor must be defined for each program. General format of a descriptor is given in above diagram and its fields are as given below:

i) Base: Contains 32-bit base address for a segment.

ii) Limit: Defines the size of the segment that can be accessed.

iii) G (Granularity bit):- Specifies the unit with which limit field interpreted. If G=0, byte is accessed from segment. If G=1 page (4KB) is accessed.

iv) AVL:- This bit is completely undefined & 80386 ignores it.

v) Access Right bytes:-

a) P: Present bit If P=1, segment is loaded in physical memory.

b) DPL: Descriptor privilege level - Defines the level of privilege associated with descriptor.
DPL = 00 (most privileged)
DPL = 11 (least privileged)

c) System bit: S :- If S=0, then system segment is accessed
S=1, Code/data segment is accessed.

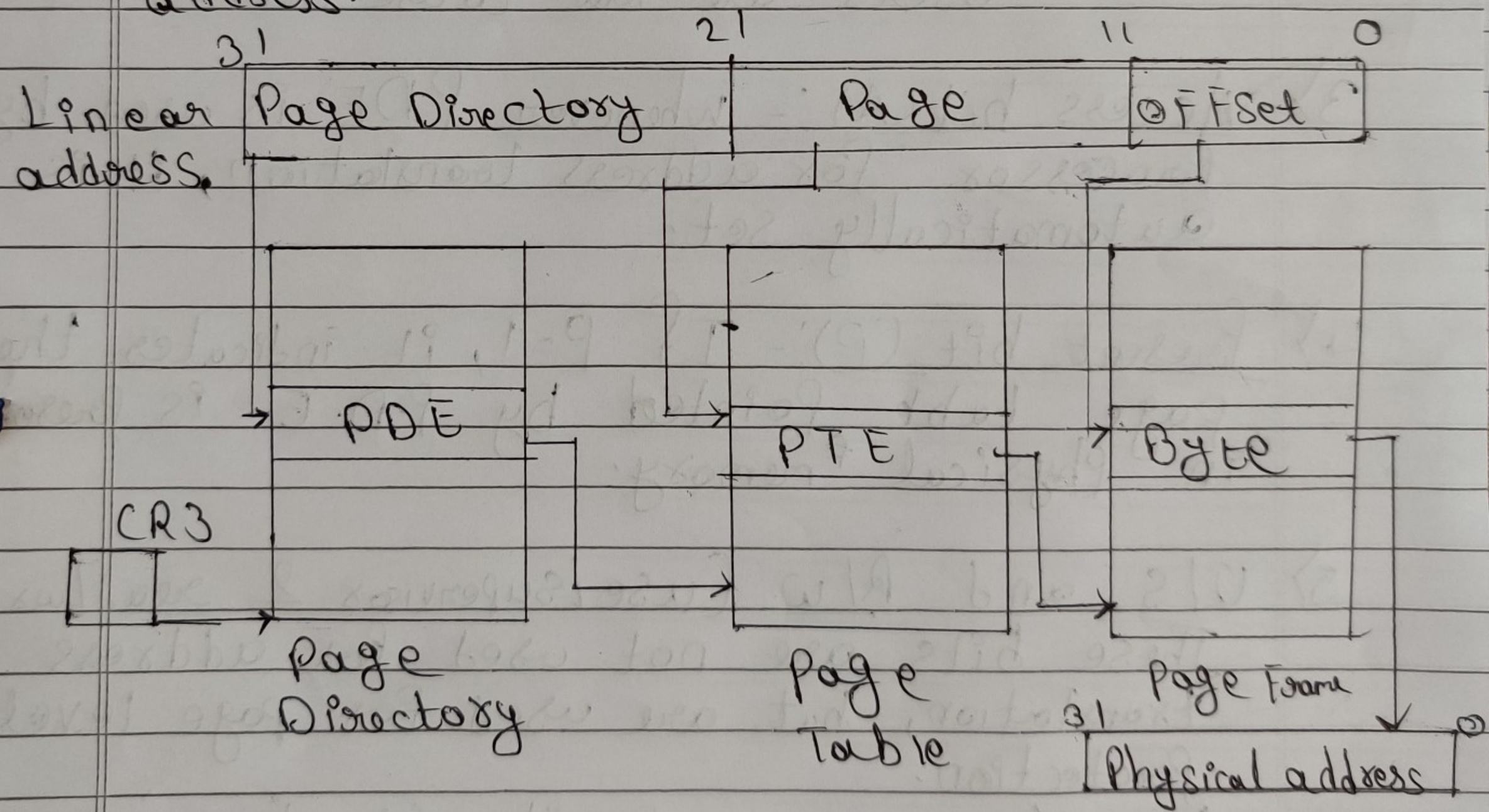
Page :
Date : / /

d) Type : It defines the type of descriptor being used.

e) A (Accessbit) :- It is automatically set to 1 when descriptor is accessed.

B) Paging :-

This is the second phase of address translation. The
In this phase, 80386 transforms linear address generated by segmentation to physical address.



Page Directory Table format :-

1. PDE Format :

31	12	9	5	4	3	2	1	0
Page table base address	User		A		U/S user/super	R/W read/write	P	

U/S	R/W	Privilege All	Privilege to PL→0,1,2
0	0	None	R/W
0	1	None	R/W
1	0	Readonly	R/W
1	1	Read only	R/W

1) Page Table Address
Specifies physical starting address or base address of page table.

2) User:- Bit 9-11 are not used by 80386
Users are free to use them.

3) Access bit (A):- whenever PDE is used by processor for address translation, it is automatically set.

4) Present bit (P):- If P=1, it indicates the page table pointed by P.D.E is present in physical memory.

5) U/S and R/W (use/supervisor & read/write):-
These bits are not used for address translation, but are used for page level protection.
When U/S=0, only privilege levels 0,1,2 are ~~are~~ accessible to data.
When U/S=1, then all privilege levels can access the data.

Page :
Date : / /

PTE (Page Table format):-

Page ~~table~~ Frame	User		D	A		U/S	R/W	P

6) Dirty bit (D):- The dirty bit is ~~set~~ automatically set whenever page frame is written into. From this, user can keep track of most often written pages of memory.