U20ECCJ21

MICROPROCESSOR AND MICROCONTROLLER

UNIT II

Simple Assembly Language Programming, Strings, Procedures, Macros, Assembler Directives- Interrupts and Interrupt Applications. Programs for Digital clock, Interfacing ADC and DAC

Assembler :

Assembler Converts a Souece file (Mnemonics) into machine level language (binary (or) Object code).

We Can use a suitable Editor to type .asm file. By using MASM assemblers (Microsoft macro assembler)(or) TASM (Turbo assembler)

Command :

```
C:\MASM\BIN\> MASM myProg.asm
```

Linker :-

A linker is a Program used to join together several object files into one large object file.

Command :-

```
C:\MASM\BIN\> LINK myProg.obj
```

Modular Programming :-

Programming task is divided into Subtasks, Separate modules (program) are written for performing Subtasks. Each module is tested Separately. The Common routines required in modules are written in Separate module and they are "called" from individual modules as Per Sequence. Programming done in this fashion is called "Modular Programming"

Advantage of Modular Programming:

- Coding is simple, easy and short
- It is easier to design, test and debug.
- It is easier to understand
- It is easy to modify the code.

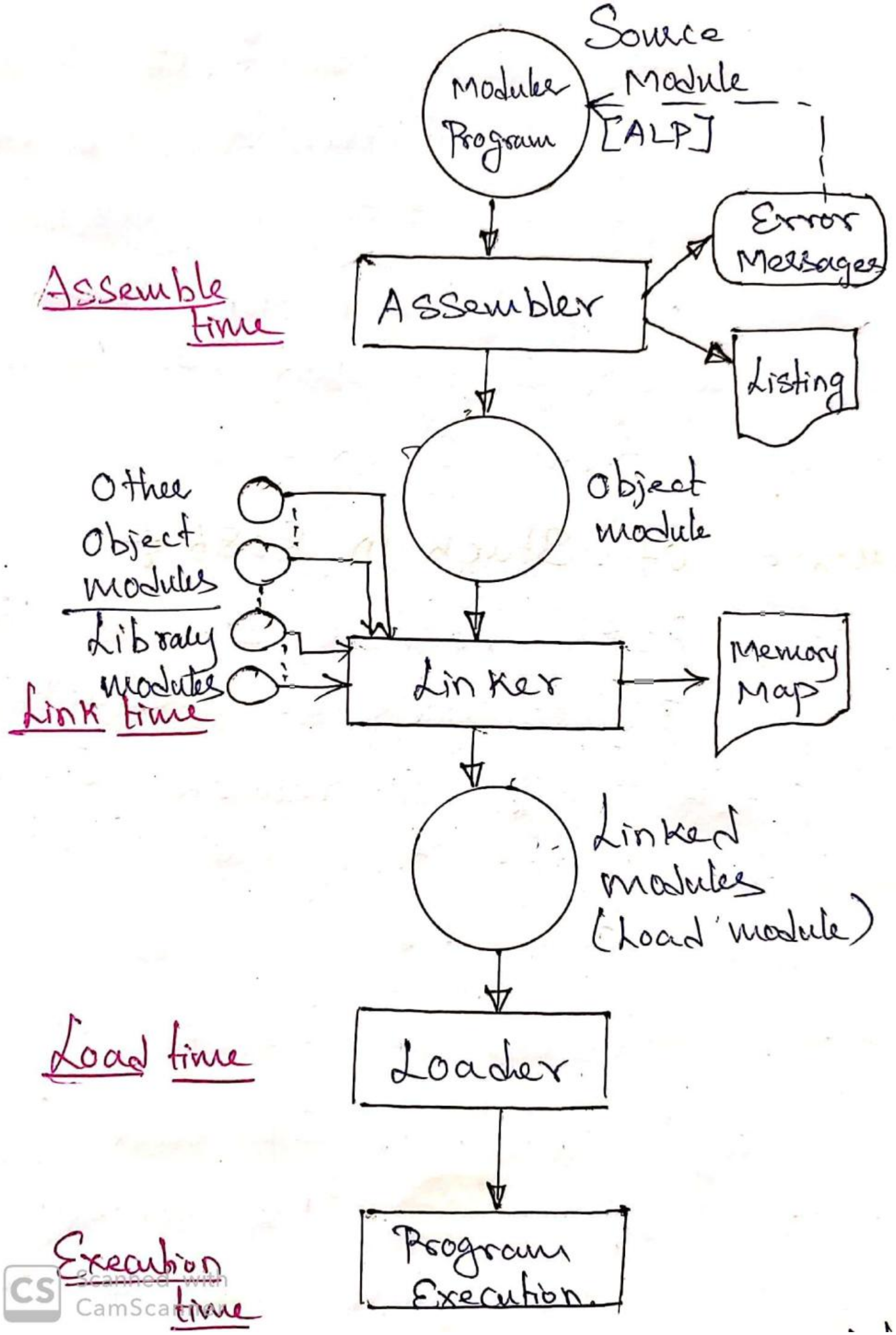
Modular Program used three Components

- Structure
- Procedures (or) Subroutines
- Macros

Linking and Relocation :-

Generating machine Code from assembly language Program is done with the help of assembler, linkee, loader and debugger.

The Steps involved in developing and executing assembly language Program is shown below.


Source Module [ALP]
Module Program
Error Messages
Assemble time
Assembler
Listing
Object module
Other Object modules
Library modules
Link time
Linker
Memory Map
Linked modules (Load module)
Load time
Loader
Execution time
Program Execution.

Stack :-

The Stack is a Portion of read / write memory Set aside by the user for the purpose of Storing information temporarily. PUSH instruction used to write data on Stack and PoP used to read data from Stack.

PUSH and POP operations :-

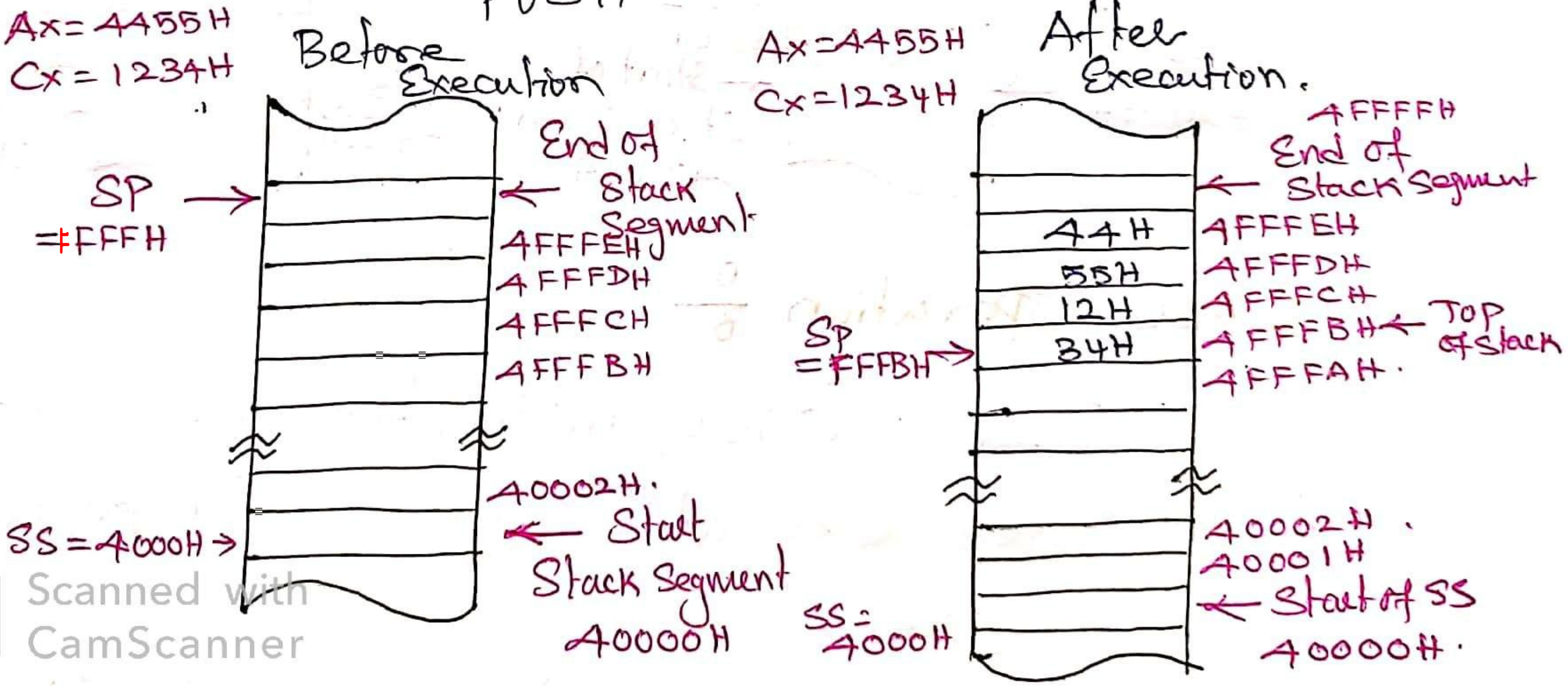
PUSH operation :-

The PUSH instruction decrement Stack Pointer by two and Copies a word from Some Source to the location in the Stack where the Stack Pointer Points.

for Example.

PUSH AX

PUSH CX



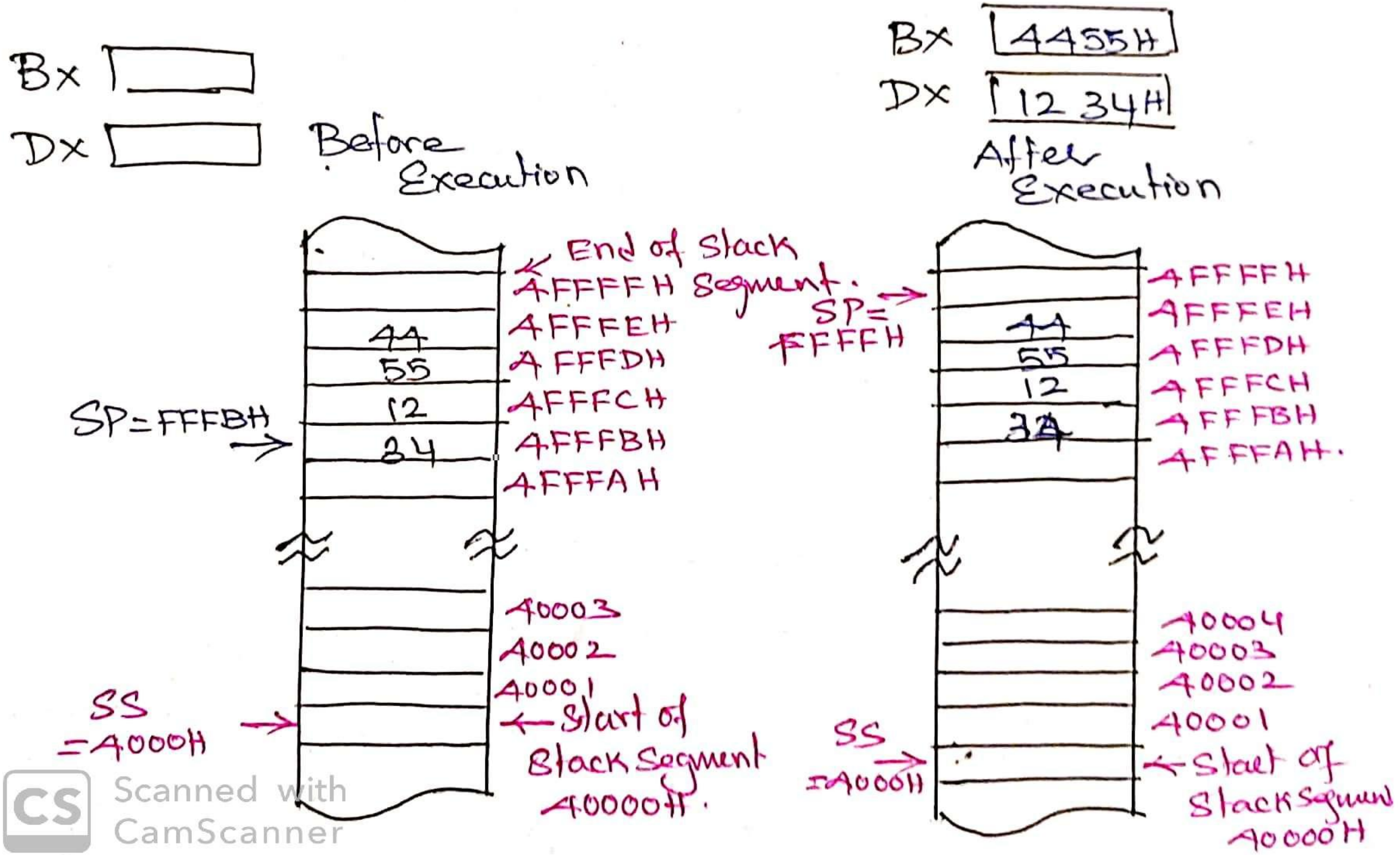
POP Operation :-

The POP instruction Copies a word from the Stack location Pointed by the Stack Pointer to the destination. After the word is Copied, the stack pointer is automatically incremented by 2.

for Example

POP DX

POP BX



CALL operation :-

The CALL instruction is used to transfer execution to a Sub program (or) Procedure.

There are two basic types of CALLs, near and far.

A near CALL is a call to a Subprogram which is in the Same Code Segment. when CALL instruction executes it decrements the stack pointer by two and Copies the offset of the next instruction address.

RET Operation :-

The RET instruction will return the execution from a subprogram to the next instruction affter the CALL instruction if near RET, then the return will be done by replacing the instruction Pointer with a word from the top of the Stack.

if far RET, then instruction Pointer will be replaced by the word at the top of the Stack. The Stack Pointer will then incremented by two. The Code Segment register is then replaced with a word from the new Stack top address.

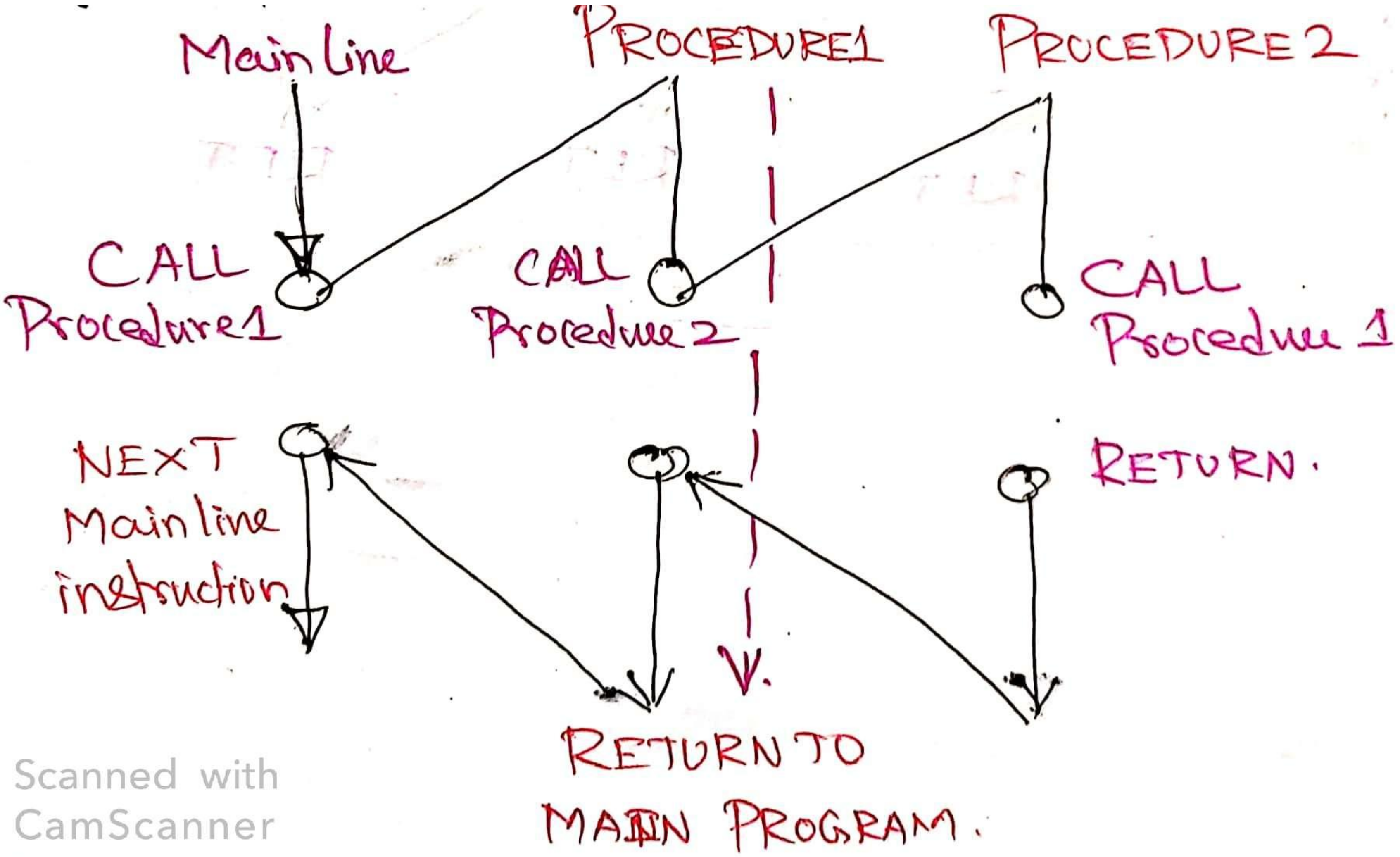
Procedures :-

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 again and again.

One way is to write the group of instructions as a separate Procedure. we can then just CALL the Procedure whenever we need to execute that group of instruction.

Reentrant Procedure :-

In Some Situations it may happen that Procedure 1 is Called from main Program , Procedure 2 is called from Procedure 1 is again called from Procedure 2. In this Situation Program execution flow reenters in the Procedure1

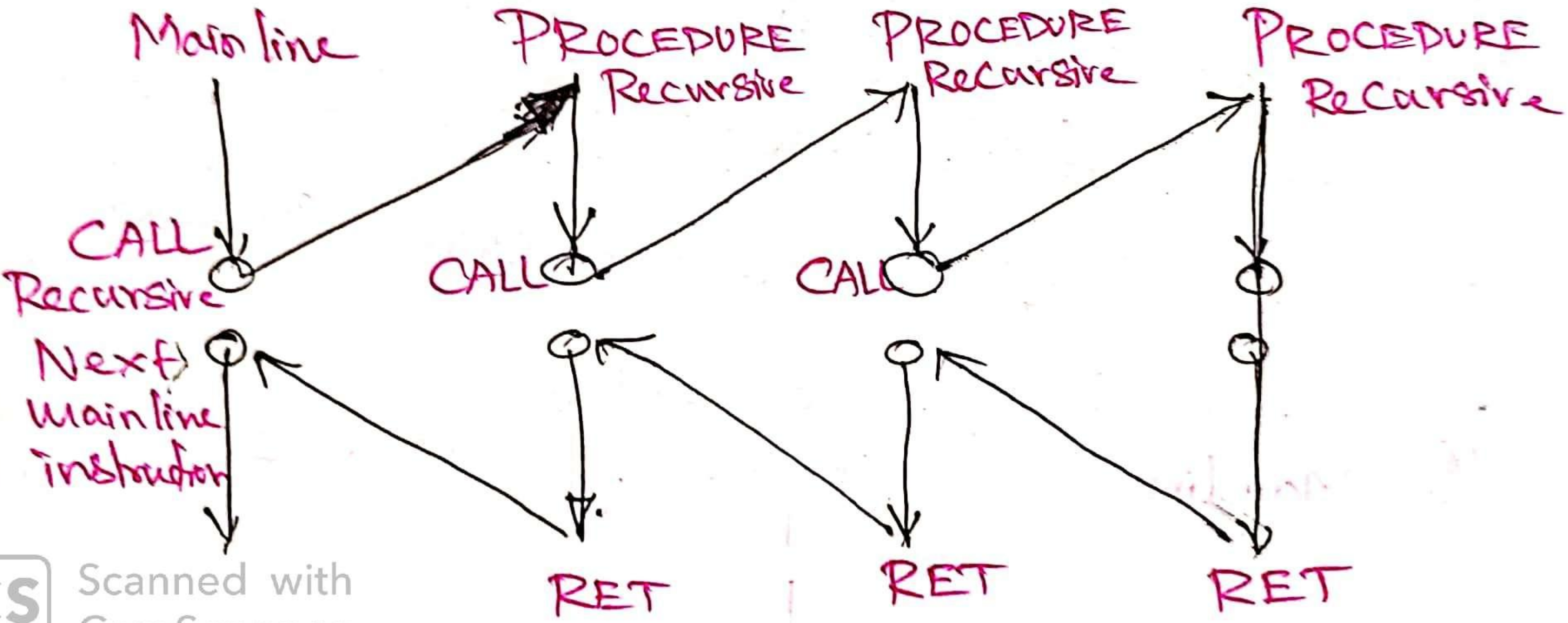

Main Line
PROCEDURE1
PROCEDURE 2
CALL Procedure1
CALL Procedure 2
CALL Procedure 1
NEXT Main line instruction
RETURN.
RETURN TO MAIN PROGRAM.

Recursive Procedure :-

A recursive Procedure is a Procedure which calls itselfs.

Recursive Procedure are used to work with Complex data Structure called trees.

if the Procedures is called with N (recursion depth) =3. Then the n is decremented by one after each Procedure CALL and the Procedure is called until n= 0.



Macros

- A macro inserts a block of statements at various points in a program during assembly
- Are similar to a function
- Are faster than executing a procedure
- Can take parameters

Macros (cont.)

- General Format

```
            MACRO_NAME   MACRO
    Param1,Param2,...,ParamN
                LOCAL  MyLabel
                Your Code ...
                ... Param1 ...
                 ...Param2 ...
                Your Code ...
                JMP MyLabel
                Your Code ...
MyLabel:
                ... ParamN ...
                 Your Code ...
                 ENDM
```

Local Variable(s) in a Macro

- A local variable is one that appears in the macro, but is not available outside the macro

- We use the LOCAL directive for defining a local variable
  - If the label *MyLabel* in the previous example is not defined as local, the assembler will flag it with errors on the second and subsequent attempts to use the macro

- The LOCAL directive must always immediately follow the MACRO directive without any intervening comments or spaces

- Macros can be placed in a separate file
  - use INCLUDE directive to include the file with external macro definitions into a program
  - no EXTERN statement is needed to access the macro statements that have been included

Macros (cont.)

Example:

```
DIV16  MACRO  Result, X, Y

  ; Store into Result the signed result of X / Y
 ; Calculate Result = X / Y
 ; (all 16-bit signed integers)
 ; Destroys Registers AX,DX

MOV   AX, X                ; Load AX with Dividend
CWD                        ; Extend Sign into DX
IDIV  Y                    ; Signed Division
MOV   Result, AX           ; Store Quotient
ENDM
```

Macros vs Procedures

```
Proc_1  PROC    NEAR
        MOV     AX, 0
        MOV     BX, AX
        MOV     CX, 5
        RET
Proc_1  ENDP

Macro_1          MACRO
         MOV    AX, 0
         MOV    BX, AX
         MOV    CX, 5
        ENDM
```

```
CALLProc_1
......
CALLProc_1
......
Macro_1
......
Macro_1
```

Procedures Vs Macros

Procedures	Macros
Accessed by CALL and RET mechanism during program execution	Accessed by name given to macro when defined during assembly
Machine code for instructions only put in memory once	Machine code generated for instructions each time called
Parameters are passed in registers, memory locations or stack	Parameters passed as part of statement which calls macro
Procedures uses stack	Macro does not utilize stack
A procedure can be defined anywhere in program using the directives PROC and ENDP	A macro can be defined anywhere in program using the directives MACRO and ENDM
Procedures takes huge memory for CALL(3 bytes each time CALL is used) instruction	Length of code is very huge if macro's are called for more number of times

Assemble Directives

- Instructions to the Assembler regarding the program being executed.

- Control the generation of machine codes and organization of the program; but no machine codes are generated for assembler directives.

- Also called 'pseudo instructions'

- Used to :
  - › specify the start and end of a program
  - › attach value to variables
  - › allocate storage locations to input/ output data
  - › define start and end of segments, procedures, macros etc..

Assemble Directives

DB

DW

SEGMENT
ENDS

ASSUME

ORG
END
EVEN
EQU

PROC
FAR
NEAR
ENDP

SHORT

MACRO
ENDM

- **Define Byte**
- **Define a byte type (8-bit) variable**
- **Reserves specific amount of memory locations to each variable**
- **Range : 00_H – FF_H for unsigned value;**
  00_H – $7F_H$ for positive value and
  80_H – FF_H for negative value
- **General form : variable DB value/ values**

Example:

LIST DB 7FH, 42H, 35H

Three consecutive memory locations are reserved for the variable LIST and each data specified in the instruction are stored as initial value in the reserved memory location

Assemble Directives

DB

DW

SEGMENT
ENDS

ASSUME

ORG
END
EVEN
EQU

PROC
FAR
NEAR
ENDP

SHORT

MACRO
ENDM

- **Define Word**
- **Define a word type (16-bit) variable**
- **Reserves two consecutive memory locations to each variable**
- **Range : 0000_H – $FFFF_H$ for unsigned value;**
  0000_H – $7FFF_H$ for positive value and
  8000_H – $FFFF_H$ for negative value
- **General form : variable DW value/ values**

Example:

ALIST DW 6512H, 0F251H, 0CDE2H

Six consecutive memory locations are reserved for the variable ALIST and each 16-bit data specified in the instruction is stored in two consecutive memory location.

Assemble Directives

DB

DW

SEGMENT
ENDS

ASSUME

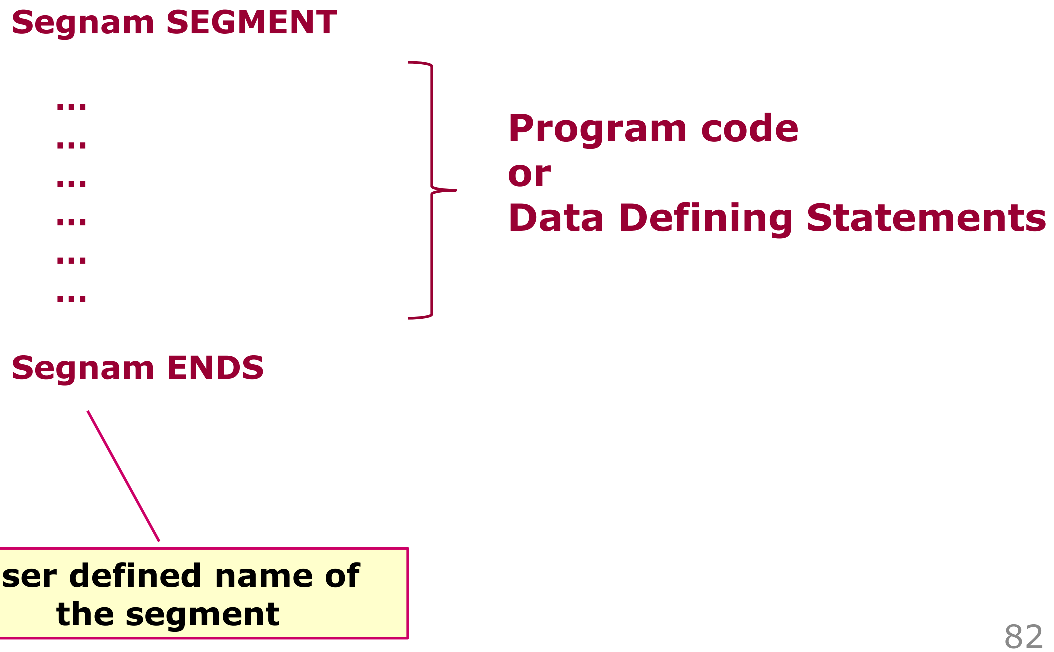
ORG
END
EVEN
EQU

PROC
FAR
NEAR
ENDP

SHORT

MACRO
ENDM

- **SEGMENT : Used to indicate the beginning of a code/ data/ stack segment**
- **ENDS : Used to indicate the end of a code/ data/ stack segment**
- **General form:**



```
Segnam SEGMENT

    …
    …
    …
    …
    …
    …

Segnam ENDS
```

Program code
or
Data Defining Statements

User defined name of the segment

Assemble Directives

DB

DW

SEGMENT
ENDS

ASSUME

ORG
END
EVEN
EQU

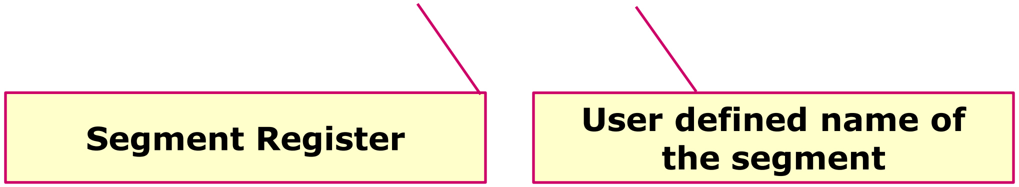
PROC
FAR
NEAR
ENDP

SHORT

MACRO
ENDM

- Informs the assembler the name of the program/ data segment that should be used for a specific segment.
- General form:

ASSUME segreg : segnam, .. , segreg : segnam



Segment Register

User defined name of the segment

Example:

ASSUME CS: ACODE, DS:ADATA	Tells the compiler that the instructions of the program are stored in the segment ACODE and data are stored in the segment ADATA

Assemble Directives

DB

DW

SEGMENT
ENDS

ASSUME

ORG
END
EVEN
EQU

PROC
FAR
NEAR
ENDP

SHORT

MACRO
ENDM

- **ORG** (Origin) is used to assign the starting address (Effective address) for a program/ data segment
- **END** is used to terminate a program; statements after END will be ignored
- **EVEN** : Informs the assembler to store program/ data segment starting from an even address
- **EQU** (Equate) is used to attach a value to a variable

Examples:

ORG 1000H	Informs the assembler that the statements following ORG 1000H should be stored in memory starting with effective address 1000_H
LOOP EQU 10FEH	Value of variable LOOP is $10FE_H$
_SDATA SEGMENT ORG 1200H A DB 4CH EVEN B DW 1052H _SDATA ENDS	In this data segment, effective address of memory location assigned to A will be 1200_H and that of B will be 1202_H and 1203_H.

Assemble Directives

DB

DW

SEGMENT
ENDS

ASSUME

ORG
END
EVEN
EQU

**PROC
ENDP
FAR
NEAR**

SHORT

MACRO
ENDM

- **PROC** **Indicates the beginning of a procedure**
- **ENDP** **End of procedure**
- **FAR** **Intersegment call**
- **NEAR** **Intrasegment call**
- **General form**

```
procname PROC[NEAR/ FAR]

            …
            …
            …

            RET

procname ENDP
```

Program statements of the procedure

Last statement of the procedure

User defined name of the procedure

Assemble Directives

DB

DW

SEGMENT
ENDS

ASSUME

ORG
END
EVEN
EQU

**PROC
ENDP
FAR
NEAR**

SHORT

MACRO
ENDM

Examples:

ADD64 PROC NEAR … … … RET ADD64 ENDP	The subroutine/ procedure named ADD64 is declared as NEAR and so the assembler will code the CALL and RET instructions involved in this procedure as near call and return
CONVERT PROC FAR … … … RET CONVERT ENDP	The subroutine/ procedure named CONVERT is declared as FAR and so the assembler will code the CALL and RET instructions involved in this procedure as far call and return

Assemble Directives

DB

DW

SEGMENT
ENDS

ASSUME

ORG
END
EVEN
EQU

PROC
ENDP
FAR
NEAR

SHORT

MACRO
ENDM

- **Reserves one memory location for 8-bit signed displacement in jump instructions**

Example:

JMP SHORT AHEAD	The directive will reserve one memory location for 8-bit displacement named AHEAD

Assemble Directives

DB

DW

SEGMENT
ENDS

ASSUME

ORG
END
EVEN
EQU

PROC
ENDP
FAR
NEAR

SHORT

MACRO
ENDM

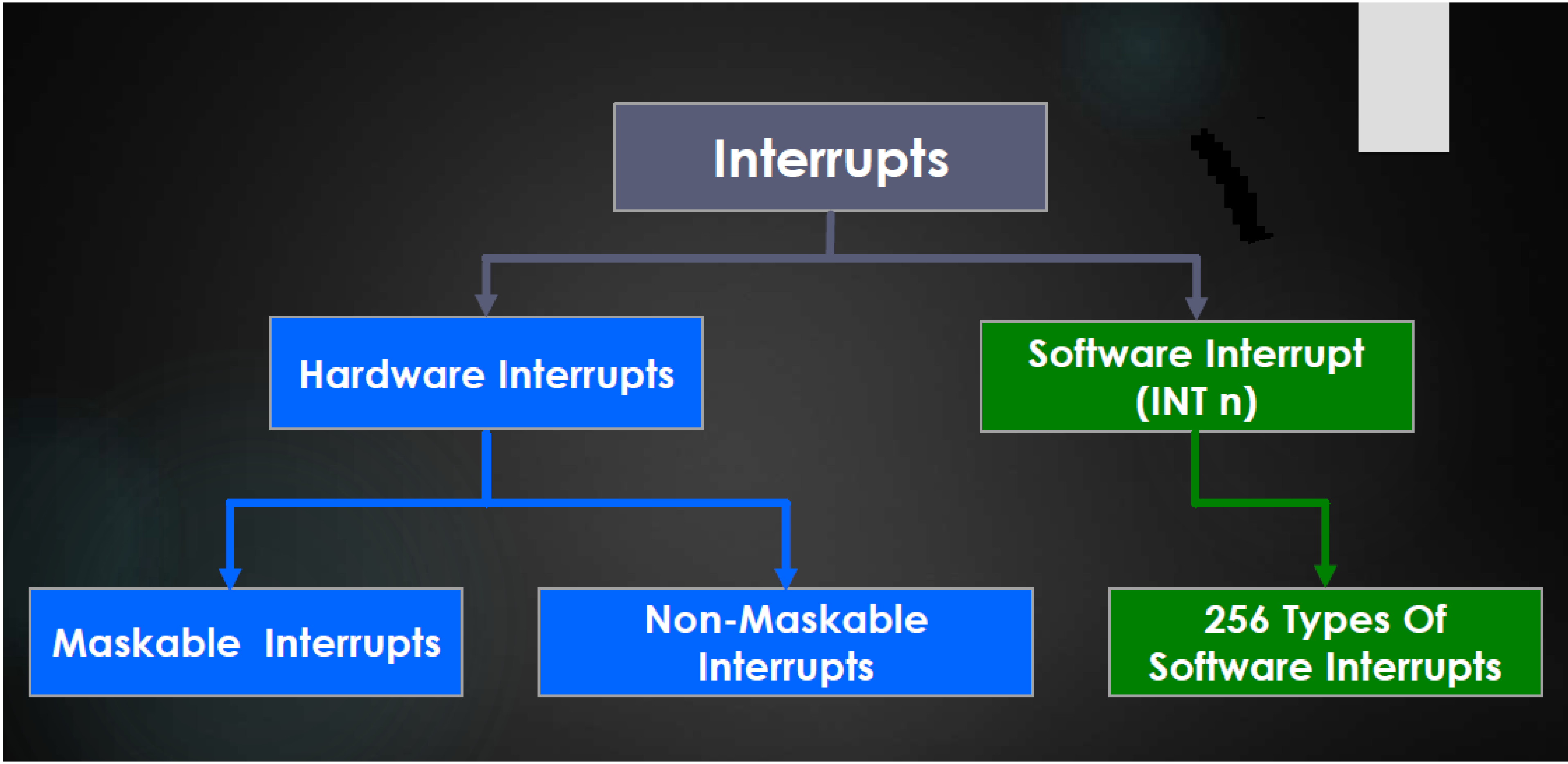
- **MACRO Indicate the beginning of a macro**
- **ENDM End of a macro**
- **General form:**

```
macroname MACRO[Arg1, Arg2 ...]

          ...
          ...
          ...

macroname ENDM
```

Program statements in the macro

User defined name of the macro

8086 Interrupts



Interrupts

Hardware Interrupts

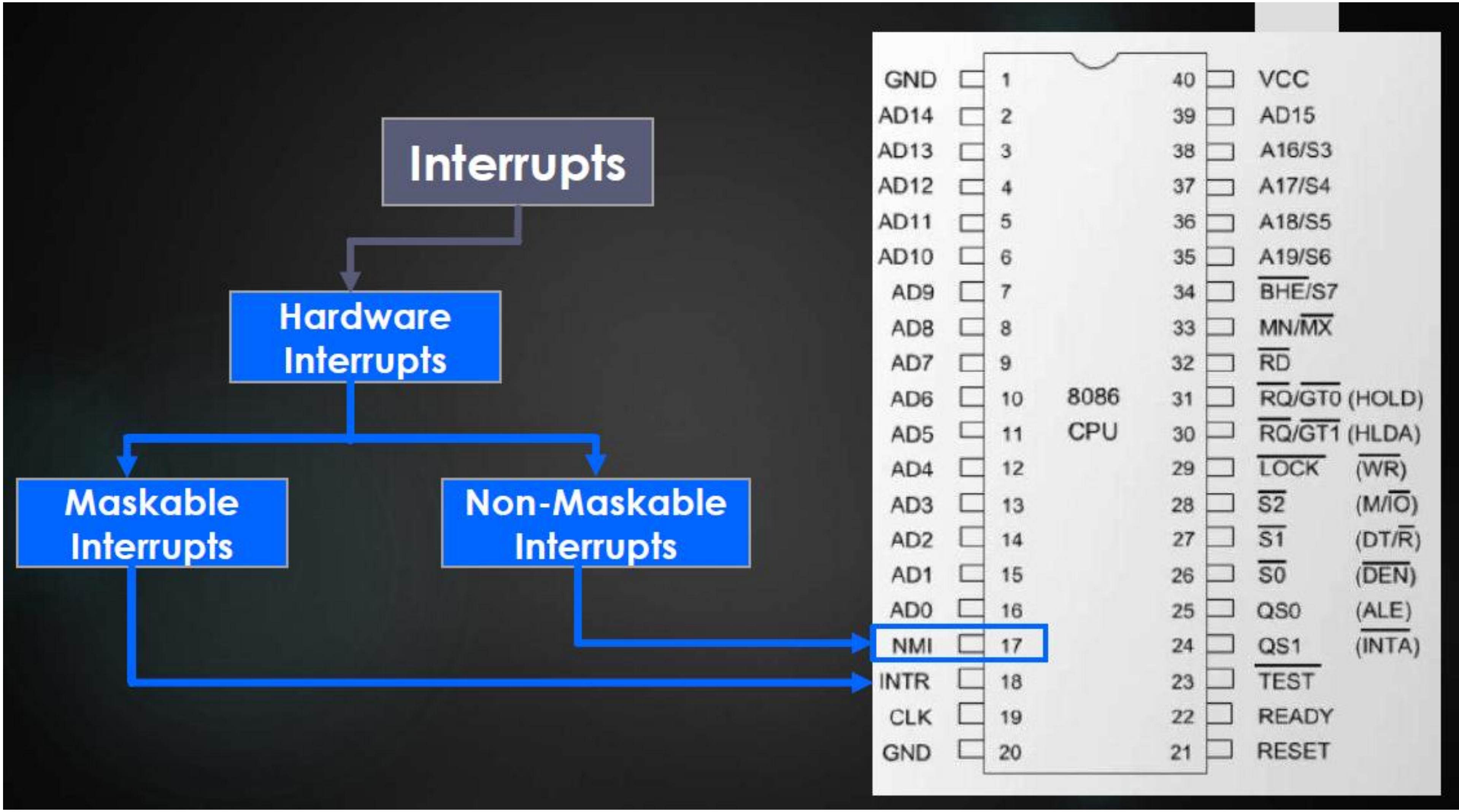
Software Interrupt (INT n)

Maskable Interrupts

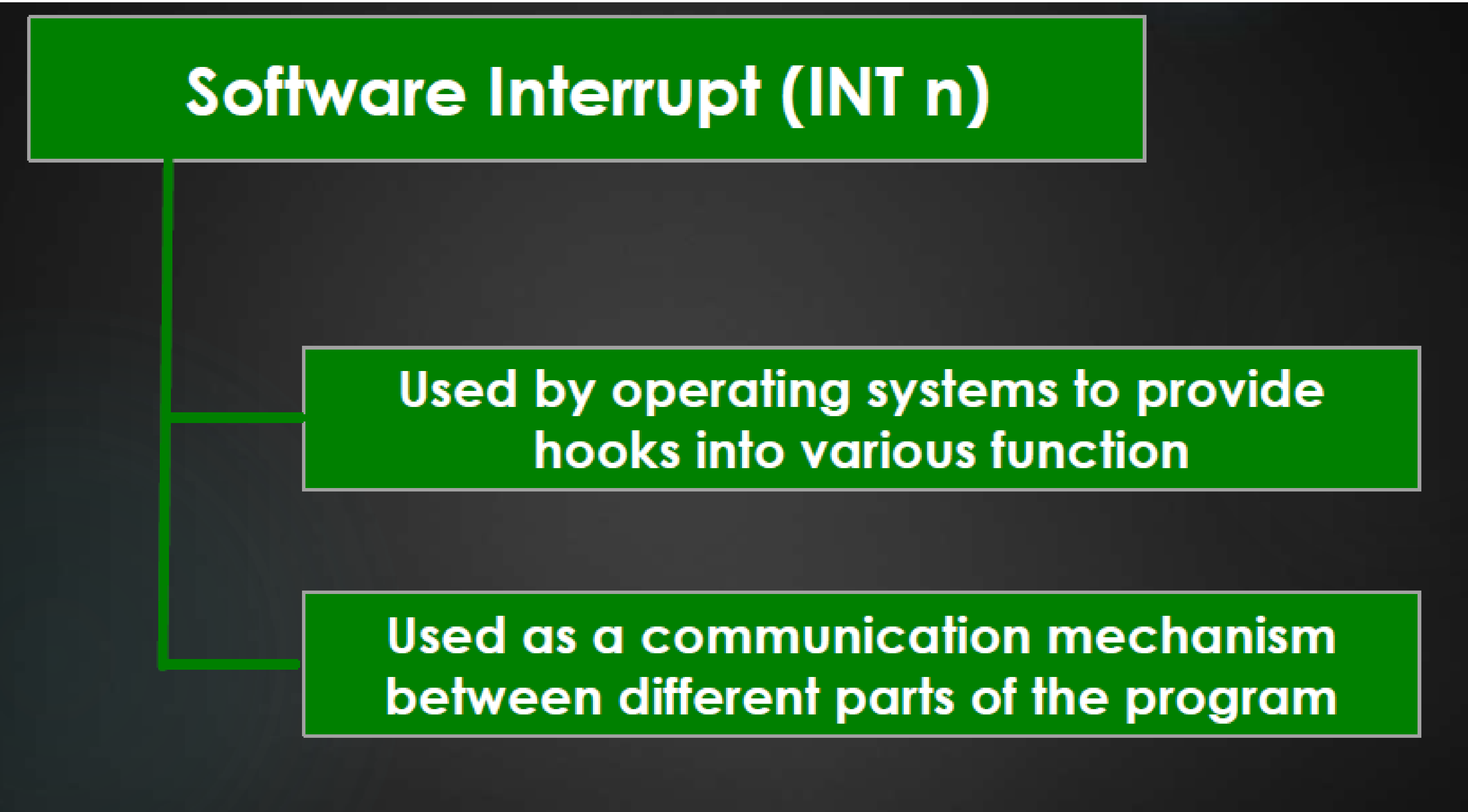
Non-Maskable Interrupts

256 Types Of Software Interrupts

8086 Interrupts



8086 Interrupts

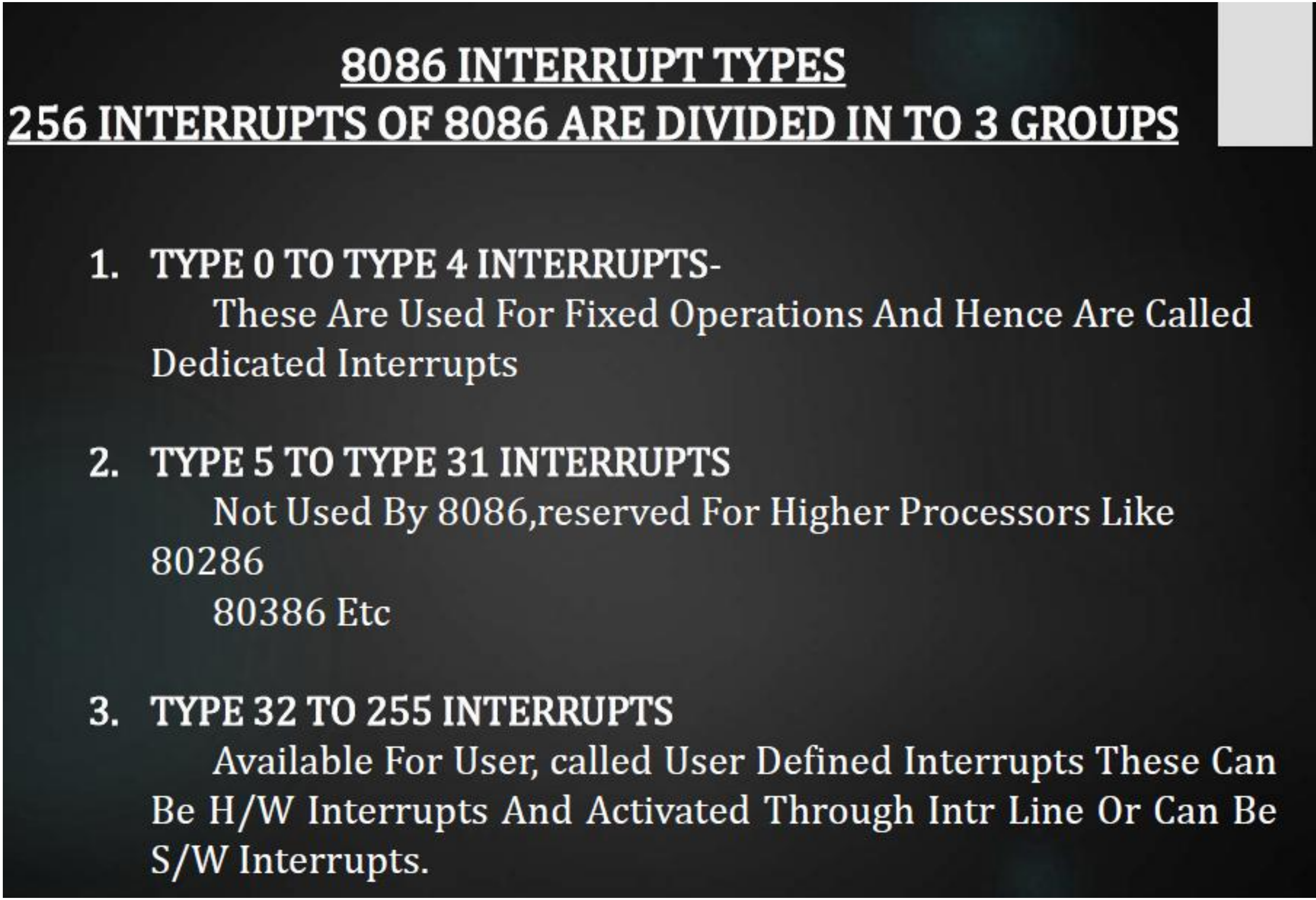


Software Interrupt (INT n)

Used by operating systems to provide hooks into various function

Used as a communication mechanism between different parts of the program

8086 Interrupts



8086 INTERRUPT TYPES

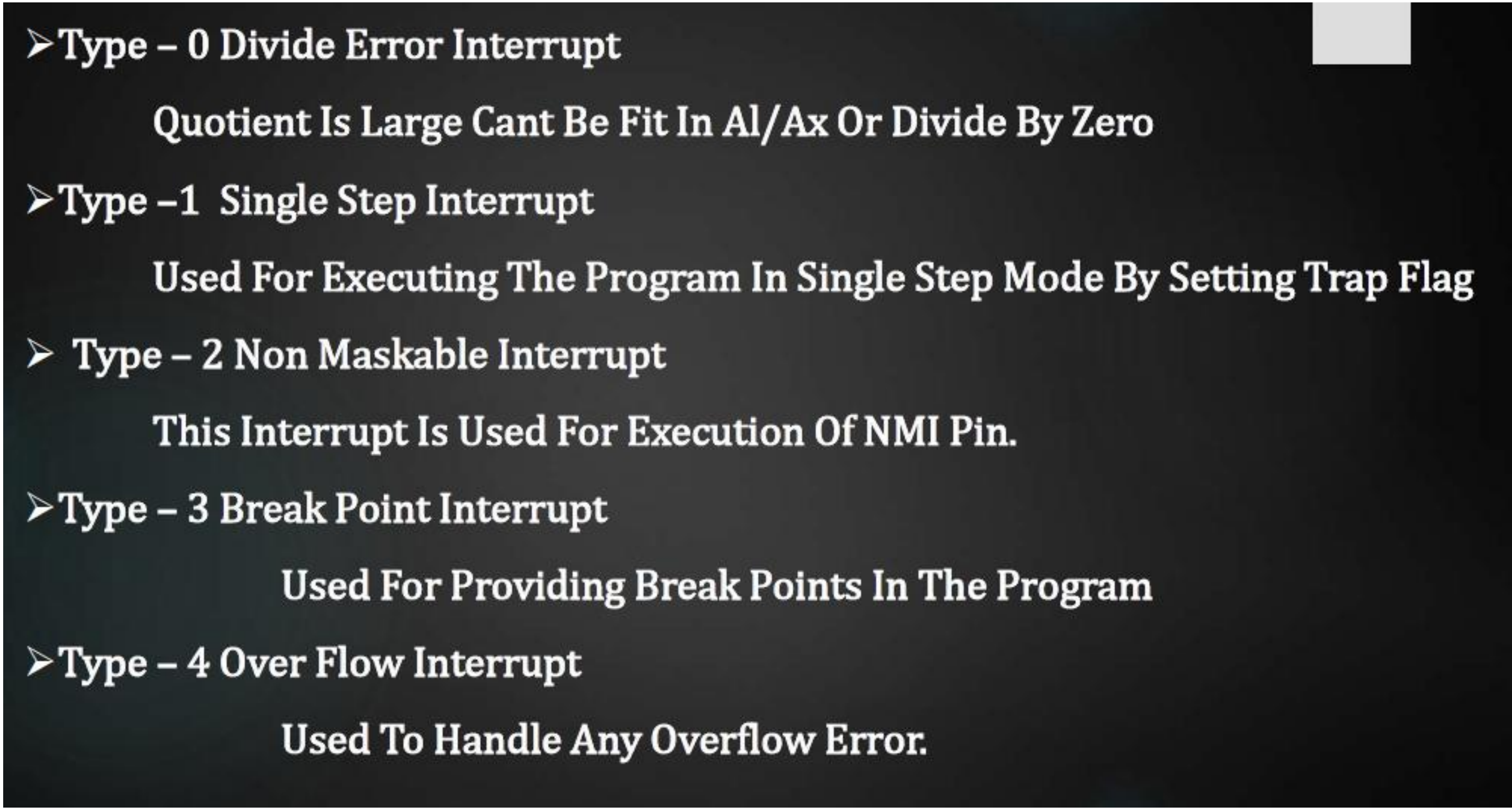
256 INTERRUPTS OF 8086 ARE DIVIDED IN TO 3 GROUPS

1. TYPE 0 TO TYPE 4 INTERRUPTS-
   These Are Used For Fixed Operations And Hence Are Called Dedicated Interrupts

2. TYPE 5 TO TYPE 31 INTERRUPTS
   Not Used By 8086,reserved For Higher Processors Like 80286
   80386 Etc

3. TYPE 32 TO 255 INTERRUPTS
   Available For User, called User Defined Interrupts These Can Be H/W Interrupts And Activated Through Intr Line Or Can Be S/W Interrupts.

8086 Interrupts



- Type – 0 Divide Error Interrupt

  Quotient Is Large Cant Be Fit In Al/Ax Or Divide By Zero

- Type –1 Single Step Interrupt

  Used For Executing The Program In Single Step Mode By Setting Trap Flag

- Type – 2 Non Maskable Interrupt

  This Interrupt Is Used For Execution Of NMI Pin.

- Type – 3 Break Point Interrupt

  Used For Providing Break Points In The Program

- Type – 4 Over Flow Interrupt

  Used To Handle Any Overflow Error.

String Manipulation Instructions

- ❑ String : Sequence of bytes or words

- ❑ 8086 instruction set includes instruction for string movement, comparison, scan, load and store.

- ❑ REP instruction prefix : used to repeat execution of string instructions

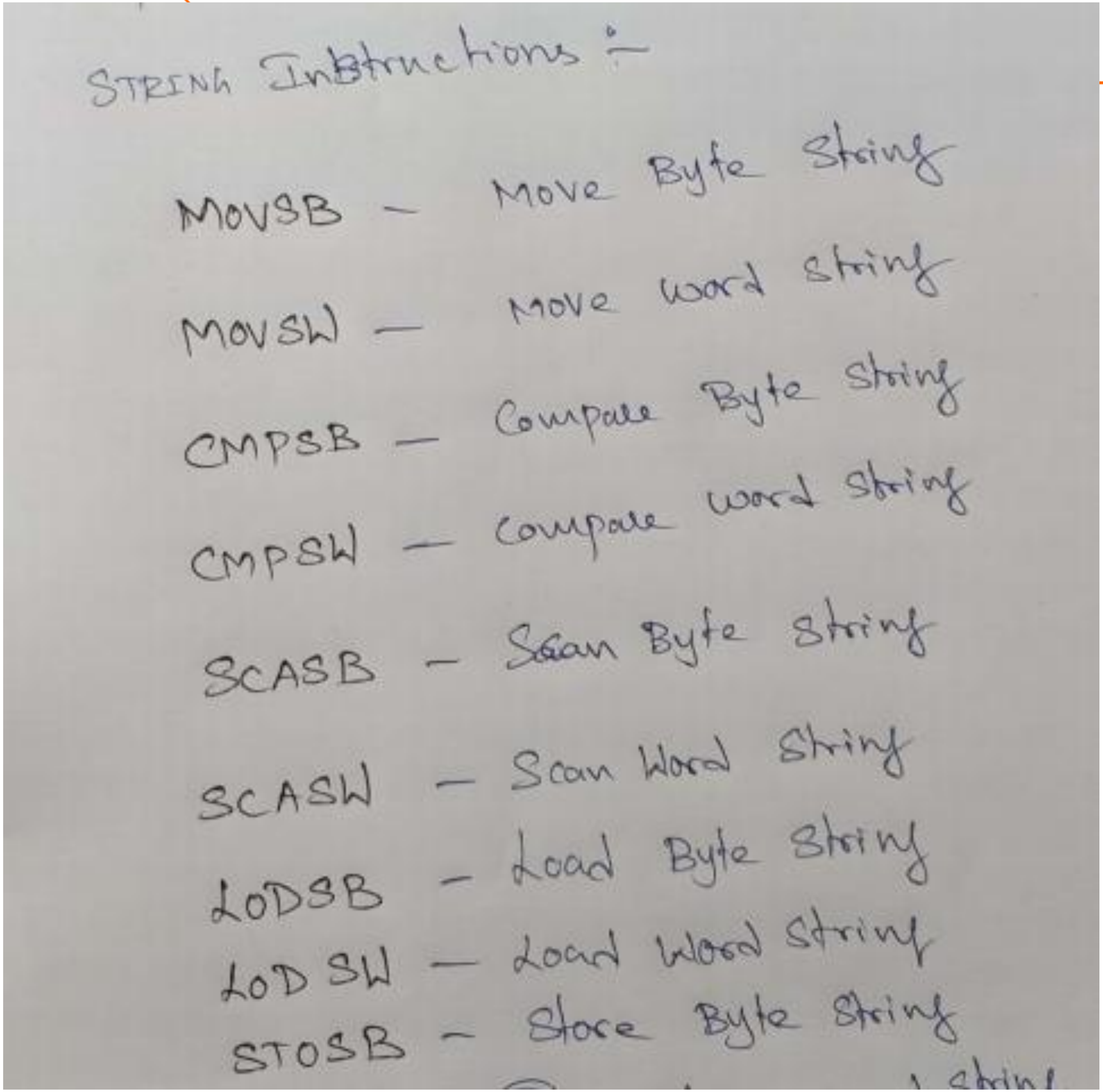
- ❑ String instructions end with S or SB or SW.
  S represents string, SB string byte and SW string word.

- ❑ Offset or effective address of the source operand is stored in SI register and that of the destination operand is stored in DI register.

- ❑ Depending on the status of DF, SI and DI registers are automatically updated.

- ❑ DF = 0 ⇒ SI and DI are incremented by 1 for byte and 2 for word.

- ❑ DF = 1 ⇒ SI and DI are decremented by 1 for byte and 2 for word.



STRING Instructions :-

MOVSB – Move Byte String

MOVSW – Move word string

CMPSB – Compare Byte String

CMPSW – compare word String

SCASB – Scan Byte string

SCASW – Scan Word String

LODSB – Load Byte String

LODSW – Load Word String

STOSB – Store Byte String

string

PROGRAM FOR DIGITAL CLOCK DESIGN USING 8086.

AIM: To write an ALP program for displaying the system clock.

APPARATUS: 1.MASM
2. PC

PROGRAM:

```
ASSUME CS: CODE,DS:DATA
        EXTERN GET_TIME: NEAR
DATA SEGMENT
 TIME_BUF DB ‟00:00:00$‟
DATA ENDS

CODE SEGMENT
MAIN PROC
START:
        MOV AX,DATA
        MOV DS, AX
        LEA BX, TIME_BUF
        CALL GET_TIME
        LEA DX, TIME_BUF
        MOV AH, 09H
        INT 21H
        MOV AH, 4CH
        INT 21H
        MAIN ENDP
CODE ENDS
END MAIN
```

RESULT: Program for displaying the system clock performed .