Microprocessors

Course Code
22415

Course Work Journey

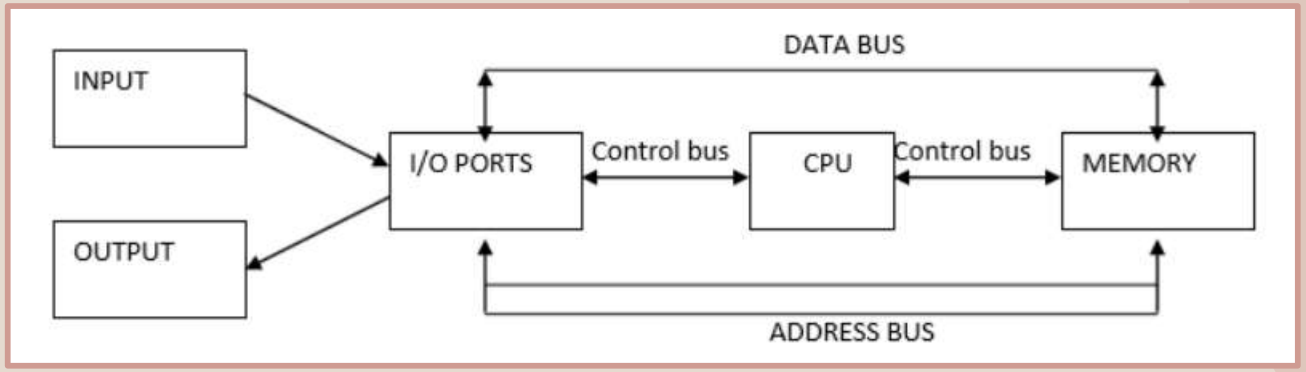
		Teaching Hours	Progress Bar
1	8086 – 16 Bit Microprocessor	**8 Hours**	**0 %**
2	The Art of Assembly Language Programming	**12 Hours**	**0 %**
3	Instruction Set of 8086 Microprocessor	**16 Hours**	**0 %**
4	Assembly Language Programming	**16 Hours**	**0 %**
5	Procedure and Macros	**12 Hours**	**0 %**

UNIT-1
8086 – 16 Bit Microprocessor

Objective

- Develop a foundational understanding of the 8086 microprocessor.
- Examine the internal architecture of the 8086 microprocessor, delving into block diagram and register organization

A Simple Computer



Major Part of micro computer

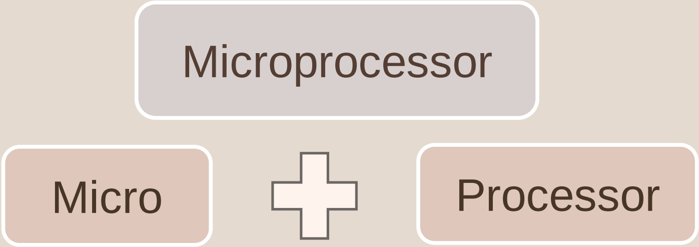
- Central Processing Unit (CPU)
- Memory
- Input – Output Circuitry

CPU controls the operation of the computer. The CPU contains an arithmetic logic Which can perform add, subtract, OR, AND, invert, or exclusive-OR operations on binary words when instructed to do so.

The memory section usually consists of a mixture of RAM and ROM. The first purpose is to store the binary codes for the sequence of instructions you want the computer to carry out. The second purpose of the memory is to store the binary-coded data with which the computer is going to be working.

I/O section allows the computer to take in data from the outside world or send data to the outside world. These allow the user and the computer to communicate with each other

What is Microprocessor**?**



Microprocessor

Micro

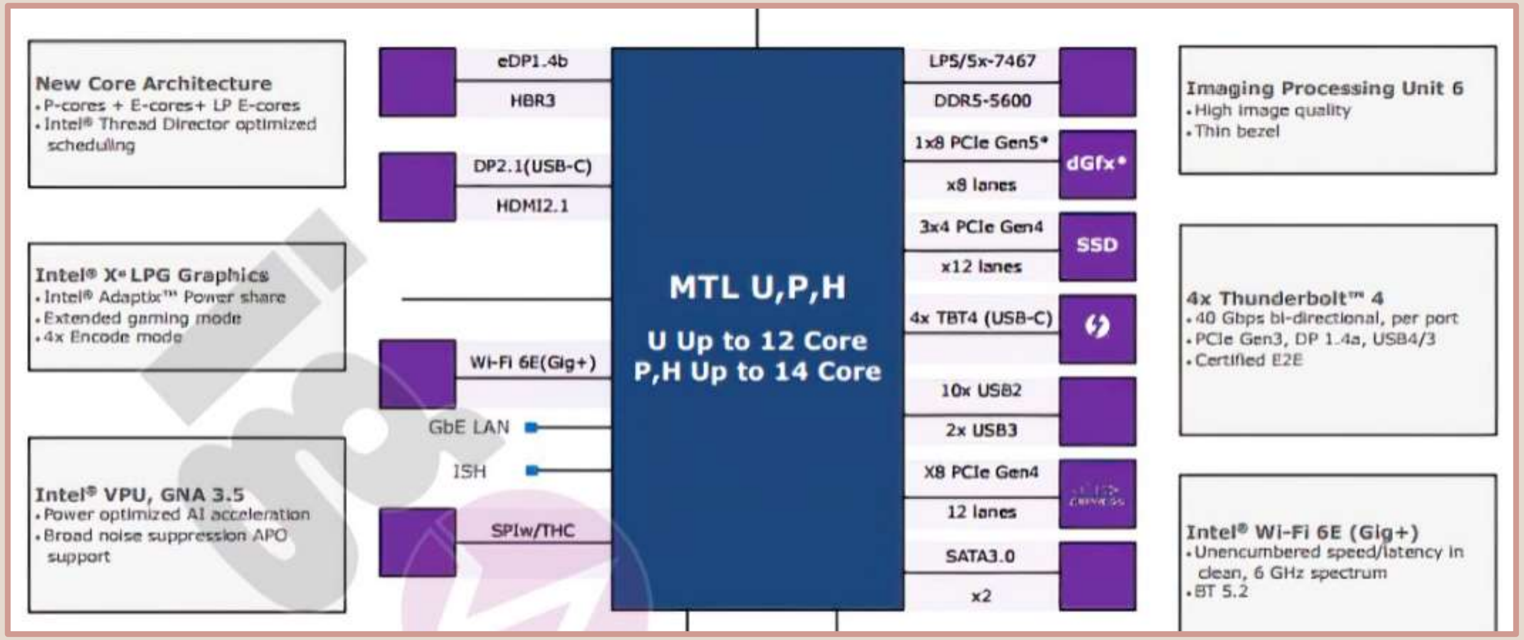
Processor

- Processor means a device that processes numbers, specifically binary numbers, 0’s and 1’s.
- In the early 1970’s the microchip was invented. All the components that made up the processor were now placed on a single piece of silicon. The size became several thousand times smaller, and the speed became several hundred times faster which gave birth to the “Microprocessor”

Definition

Microprocessor is a multipurpose, programmable device that accepts digital data as input, processes it according to instructions stored in its memory, and provides results as output.

Intel Core 14th Gen processors



New Core Architecture
- P-cores + E-cores+ LP E-cores
- Intel® Thread Director optimized scheduling

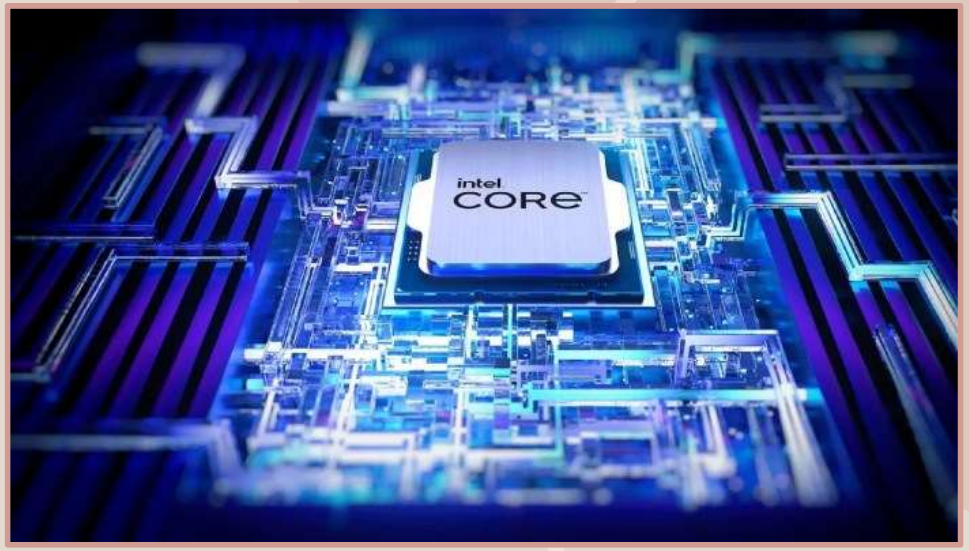
Intel® Xe LPG Graphics
- Intel® Adaptix™ Power share
- Extended gaming mode
- 4x Encode mode

Intel® VPU, GNA 3.5
- Power optimized AI acceleration
- Broad noise suppression APO support

Imaging Processing Unit 6
- High image quality
- Thin bezel

4x Thunderbolt™ 4
- 40 Gbps bi-directional, per port
- PCIe Gen3, DP 1.4a, USB4/3
- Certified E2E

Intel® Wi-Fi 6E (Gig+)
- Unencumbered speed/latency in clean, 6 GHz spectrum
- BT 5.2



8086 Microprocessor - Salient features, Pin description

Overview

- Intel 8086 is 16- bit microprocessor that is intended to be used as the CPU in a microcomputer.
- The term 16-bit means that its arithmetic logic unit, its internal registers, and most of the instructions are designed to work with 16-bit binary words

Now, when we say it's a "16-bit" processor, think of it as having 16 switches that can either be turned on or off. These switches help the processor handle information, much like how a light switch controls the flow of electricity.

Salient Features

16-Bit Architecture

- The 8086 processes information in 16-bit units, enhancing its computational efficiency and capacity.

Powerful Instruction Set

- Offers a diverse set of instructions for various operations, including data transfer, arithmetic and logical operations, and control flow.

Segmented Memory Model

- Utilizes a segmented memory model, allowing efficient memory organization and addressing by dividing the memory into segments.

Comprehensive Address Bus

- Incorporates a 20-bit address bus, capable of addressing up to 1 MB of memory, facilitating the handling of large amounts of data.

Multiprogrammable

- 8086 supports multiprogramming, a technique where the code for two or more processes resides in memory simultaneously, and the processor executes them in a time-multiplexed fashion.

Multiple Operating Modes

- 8086 is designed to operate in two modes, namely the minimum mode and the maximum mode.

Salient Features

Multiple Operating Modes

Minimum mode is suitable for single-processor microcomputer systems, where the 8086 CPU directly handles control signals.

Maximum mode is employed in multiprocessor systems, utilizing an external bus controller for coordinated control signal generation when multiple 8086 processors are involved.

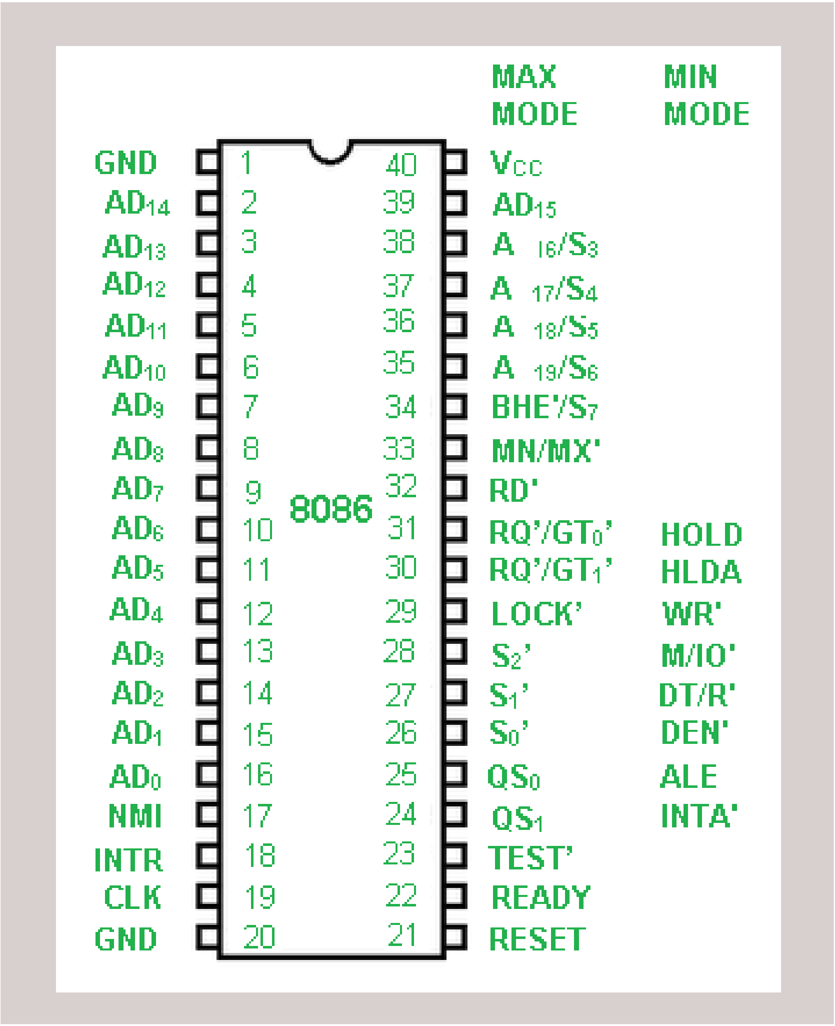
Variable Clock Speeds

- Operating at different clock speeds, the 8086 offers a range of frequency options, allowing for customization based on performance requirements.

Pipelining

- It uses two stages of pipelining, i.e. Fetch Stage and Execute Stage, which improves performance.

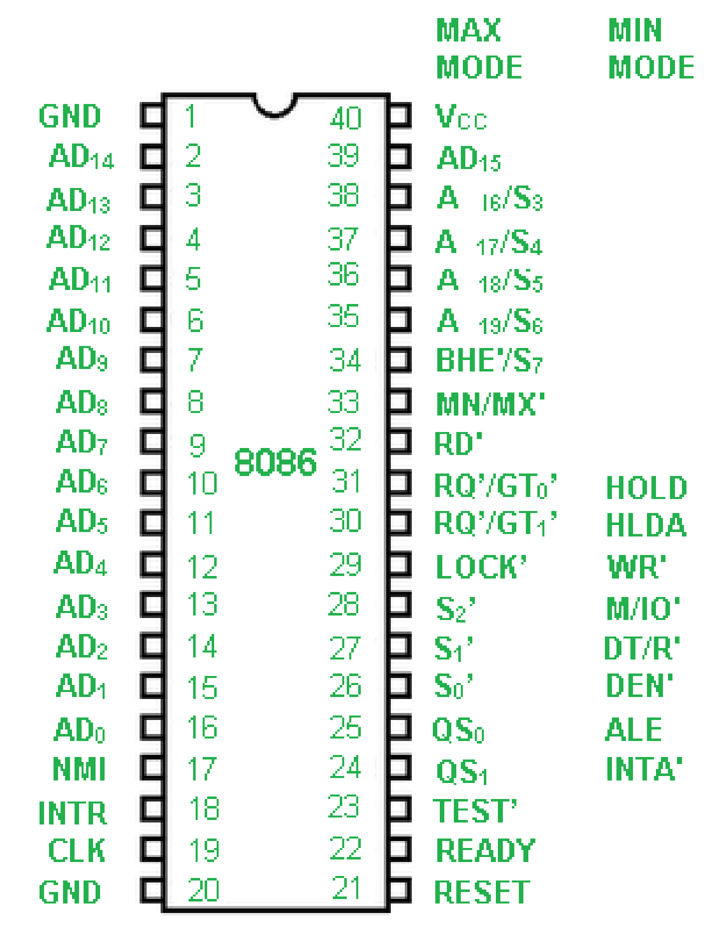
Pin Description



8086 was the first 16-bit microprocessor available in 40-pin DIP (Dual Inline Package) chip.

- It has 5V DC supply at VCC pin 40 and uses ground at VSS pin 1 and 20 for its operation.
- AD0-AD15. These are 16 address/data bus. AD0-AD7 carries low order byte data and AD8-AD15 carries higher order byte data. During the first clock cycle, it carries 16-bit address and after that it carries 16-bit data.
- A16-A19/S3-S6. These are the 4 address/status buses. During the first clock cycle, it carries 4-bit address and later it carries status signals.
- Clock signal is provided through Pin-19. It provides timing to the processor for operations. Its frequency is different for different versions, i.e. 5MHz, 8MHz and 10MHz.
- NMI stands for non-maskable interrupt and is available at pin 17. It is an edge triggered input, which causes an interrupt request to the microprocessor.

Pin Description



8086 was the first 16-bit microprocessor available in 40-pin DIP (Dual Inline Package) chip.

- INTR is available at pin 18. It is an interrupt request signal, which is sampled during the last clock cycle of each instruction to determine if the processor considered this as an interrupt or not.

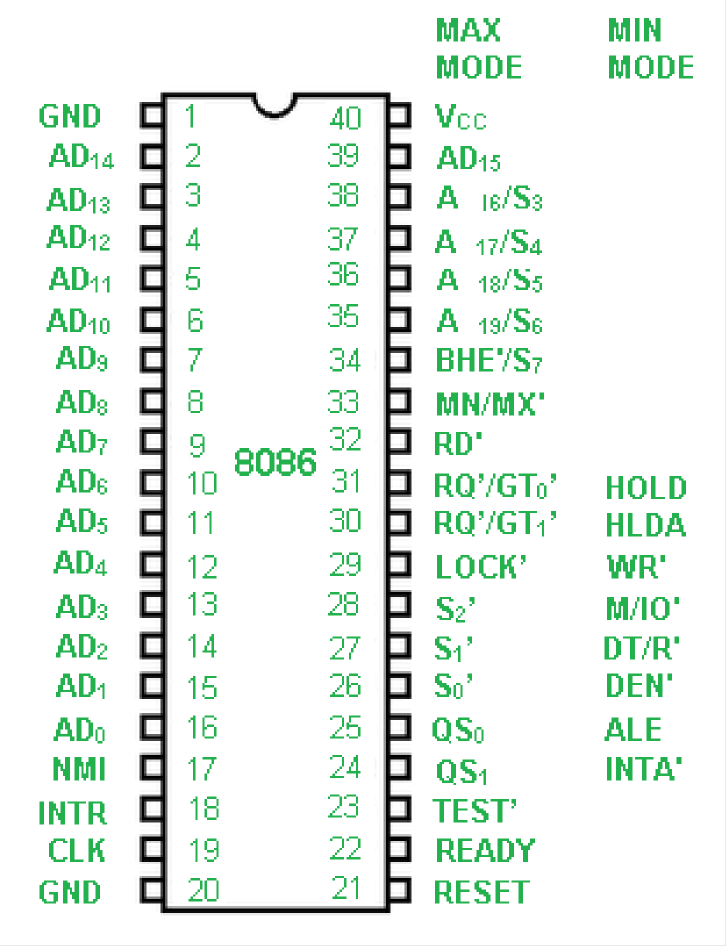
- Reset is available at pin 21 and is used to restart the execution. It causes the processor to immediately terminate its present activity. This signal is active high for the first 4 clock cycles to RESET the microprocessor.

- Ready is available at pin 22. It is an acknowledgement signal from I/O devices that data is transferred. It is an active high signal. When it is high, it indicates that the device is ready to transfer data. When it is low, it indicates wait state.

- TEST' signal is like wait state and is available at pin 23. When this signal is high, then the processor has to wait for IDLE state, else the execution continues.

- INTA is an interrupt acknowledgement signal and id available at pin 24. When the microprocessor receives this signal, it acknowledges the interrupt.

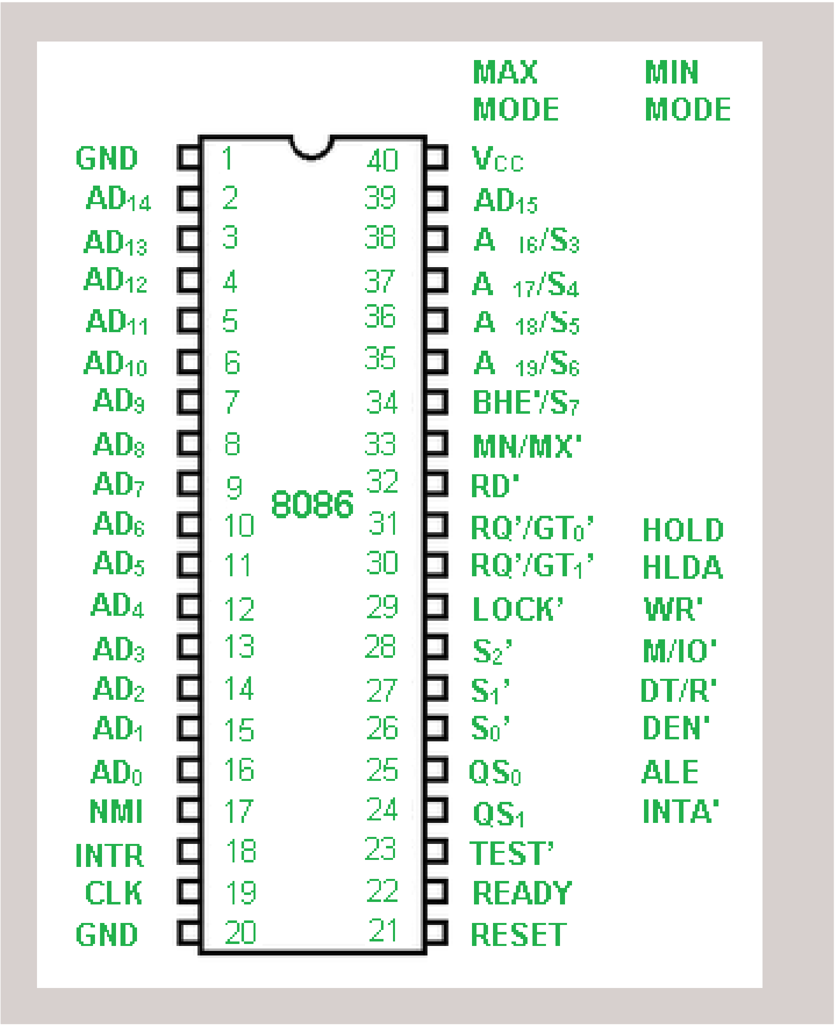
Pin Description



8086 was the first 16-bit microprocessor available in 40-pin DIP (Dual Inline Package) chip.

- ALE stands for address enable latch and is available at pin 25. A positive pulse is generated each time the processor begins any operation. This signal indicates the availability of a valid address on the address/data lines.
- DEN' stands for Data Enable and is available at pin 26. It is used to enable Transreceiver 8286. The transreceiver is a device used to separate data from the address/data bus.
- DT/R' stands for Data Transmit/Receive signal and is available at pin 27. It decides the direction of data flow through the transreceiver. When it is high, data is transmitted out and vice-versa.
- M/IO' signal is used to distinguish between memory and I/O operations. When it is high, it indicates I/O operation and when it is low indicates the memory operation. It is available at pin 28.
- WR' stands for write signal and is available at pin 29. It is used to write the data into the memory or the output device depending on the status of M/IO signal.

Pin Description



8086 was the first 16-bit microprocessor available in 40-pin DIP (Dual Inline Package) chip.

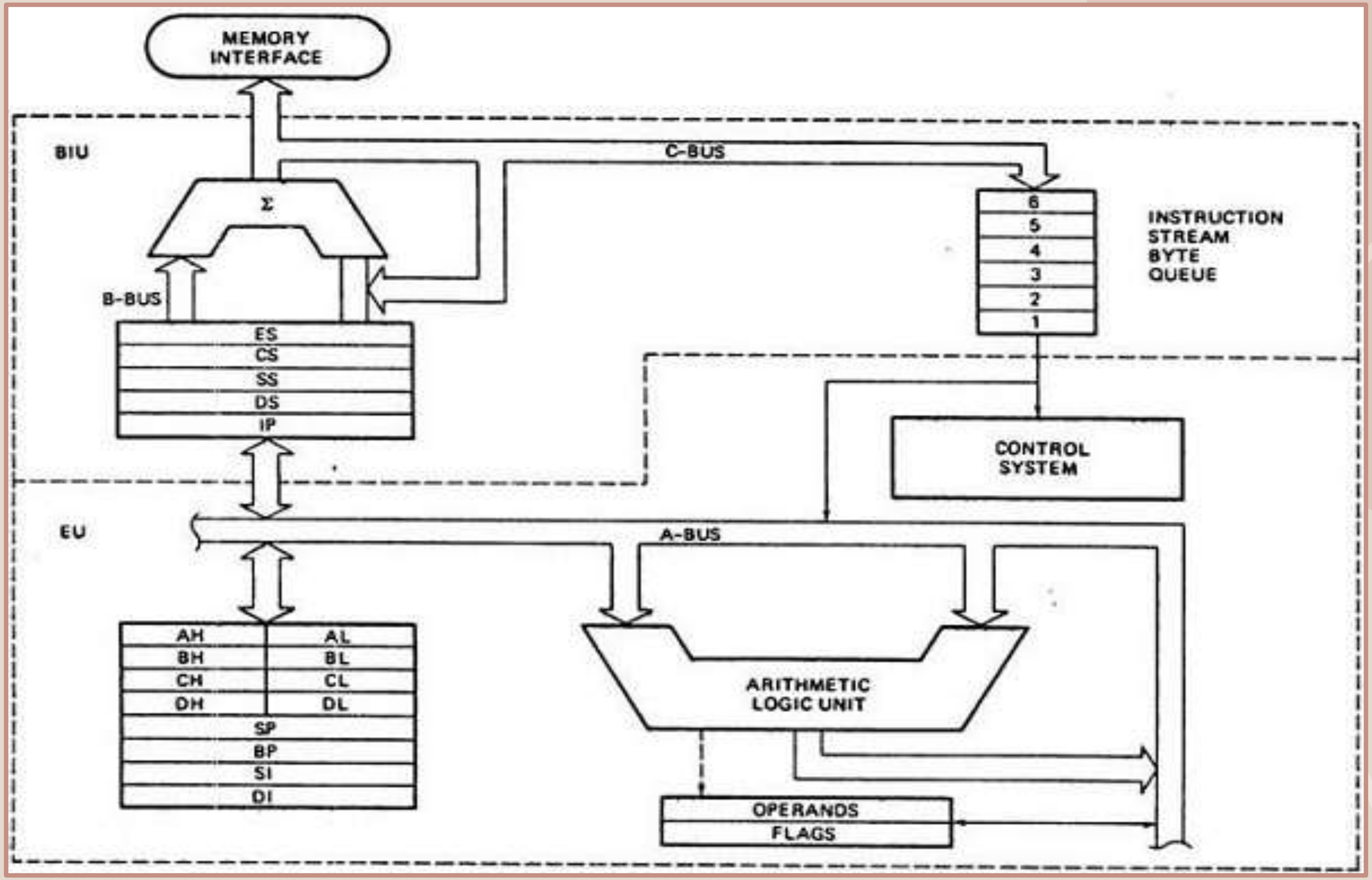
- HLDA stands for Hold Acknowledgement signal and is available at pin 30. This signal acknowledges the HOLD signal.
- HOLD signal indicates to the processor that external devices are requesting to access the address/data buses. It is available at pin 31.
- RD’ is available at pin 32 and is used to read signal for Read operation.
- MN/MX’ stands for Minimum/Maximum and is available at pin 33. It indicates what mode the processor is to operate in; when it is high, it works in the minimum mode and vice-versa.
- BHE stands for Bus High Enable. It is available at pin 34 and used to indicate the transfer of data using data bus AD8-AD15. This signal is low during the first clock cycle, thereafter it is active.
- When LOCK signal is active, it indicates to the other processors not to ask the CPU to leave the system bus. It is activated using the LOCK prefix on any instruction and is available at pin 29.

Architecture of 8086 : Functional Block Diagram, Register Organization

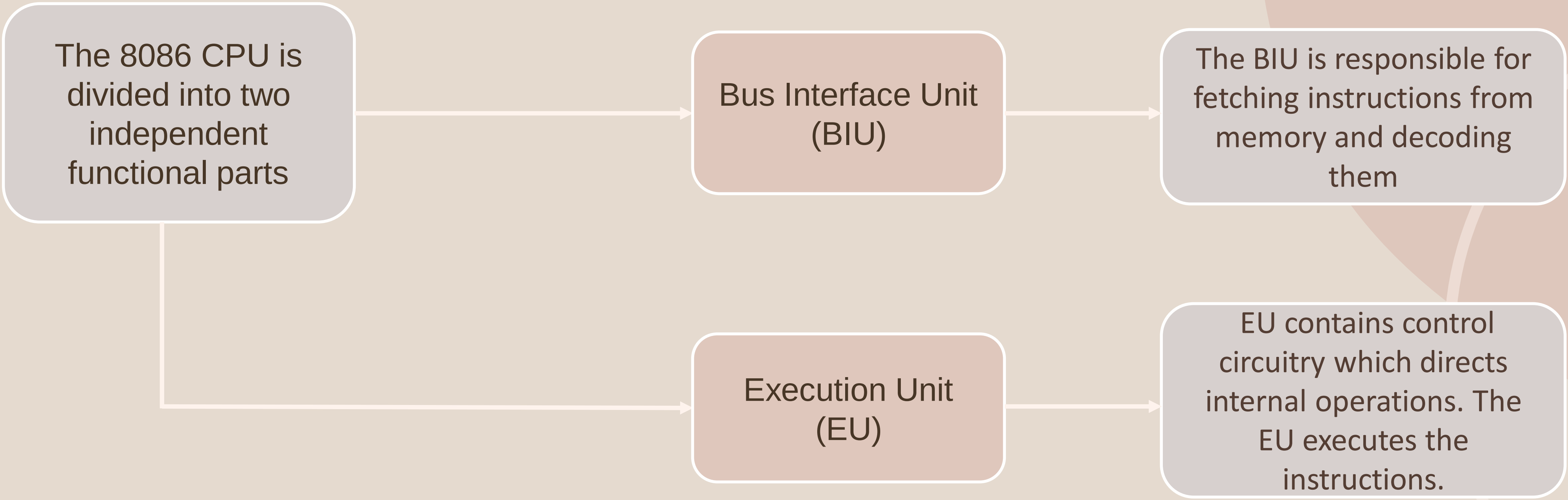
Overview

- The architecture of the 8086 microprocessor is rooted in a Complex Instruction Set Computer (CISC) architecture.
- This design enables the 8086 to support an extensive set of instructions, many of which can execute multiple operations in a single instruction. This instruction set contributes to the versatility and efficiency of the 8086, making it suitable for a broad range of applications.

Functional Block Diagram



Functional Block Diagram



The 8086 CPU is divided into two independent functional parts

Bus Interface Unit (BIU)

The BIU is responsible for fetching instructions from memory and decoding them

Execution Unit (EU)

EU contains control circuitry which directs internal operations. The EU executes the instructions.

BUS Interface Unit (BIU)

Components of BIU

Instruction Queue :
It hold the instruction byte of the next instruction to be executed by EU

Segment Registers :
Four segment registers of 16-bit provide powerful memory management mechanism

ES(Extra Segment), **CS**(Code Segment), **SS**(Stack Segment) and **DS**(Data Segment) are 64kb registers

Instruction Pointer (IP) :
It is a 16 bit register which acts as a counter and points to the address of next instruction to be executed

Address Generation and Bus Control :
Generation of 20-Bit physical address

Tasks Performed

- Fetch instruction from memory
- Read instruction from the memory
- Write instruction to the memory
- Input the data from the peripheral ports
- Output the data to the peripheral ports
- Address generation for the memory references
- Transferring instruction bytes to the instruction queue
- BUI handles all transfer of data and address on the buses for execution unit

Execution Unit (EU)

Components of EU

Arithmetic Logic Unit (ALU) :
It contains a 16-Bit ALU, that performs addition-subtraction, increment-decrement, complement, binary shift, AND, OR XOR, etc.

Control Unit (CU) :
The Control Unit helps to improve the performance of the 8086 microprocessor by managing the flow of instructions and data through the microprocessor, ensuring that the microprocessor operates correctly and efficiently.

Flag Register :
It has 9 flag register of 16-Bit.

General Purpose Registers (GPR):
EU has 4 GPR of 16 –Bit namely AX, BX, CX, DX. Each register is combination of two registers of 8-Bit. AH, AL, BH, BL, CH, CL, DH, DL, where ‘L’ is lower byte and ‘H’ is higher byte.

Index Register :
There are two 16-Bit index registers that are Source Index(SI) and Destination Index(DI). Both registers are used for string related operations and for moving block of memory from one location to another.

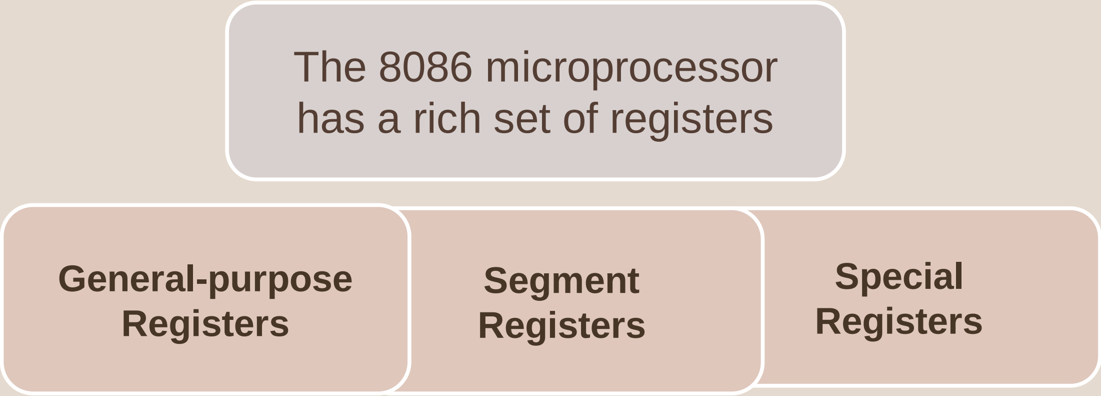
Pointers :
Pointers are 16-Bit registers namely Stack Pointer(SP) and Base Pointer(BP). SP points to the topmost item of the stack whereas BP is primarily used in accessing parameters passed by the stack. Offset address of both the registers is relative to the stack segment

Execution Unit (EU)

Tasks Performed

- Decodes the instruction
- Executes decoded instruction
- Commands BIU to fetch instruction from location
- Performs the operation on data
- EU also known as execution heart of the processor

Functional Block Diagram



The 8086 microprocessor has a rich set of registers

General-purpose Registers

Segment Registers

Special Registers

- The general-purpose registers can be used to store data and perform arithmetic and logical operations

- The segment registers are used to address memory segments.

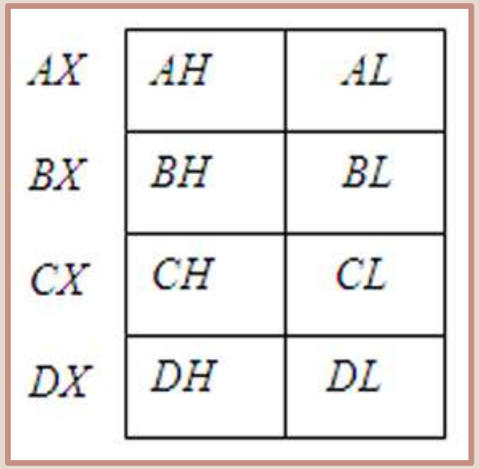
- The special registers include the flags register, which stores status information about the result of the previous operation, and the instruction pointer (IP), which points to the next instruction to be executed. Also includes index registers.

Registers

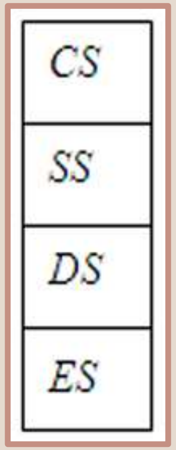
General-purpose Registers

Segment Registers

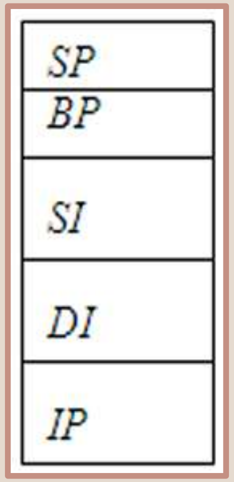
Special Registers



AX	AH	AL
BX	BH	BL
CX	CH	CL
DX	DH	DL



CS
SS
DS
ES



SP
BP
SI
DI
IP

Registers

General-purpose Registers

AX : Accumulator Register consists of two 8-bit registers AL and AH.
AX works as an intermediate register in memory and I/O operations
Accumulator is used for the instruction such as MUL and DIV

BX : Base Register consists of two 8-bit registers BL and BH.
BX register usually contains data pointers base, It is used to hold the address of a procedure or variable.

CX : Count Register consists of two 8-bit registers CL and CH.
Counter register can be used in loop, shift and rotate instructions. It is also used in string manipulation

DX : Data Register consists of two 8-bit registers DL and DH.
Data registers can be used together with AX registers for MUL and DIV instructions. This register can be used as port number in I/O operations.

Registers

Segment Registers

CS : Code Segment
The CS register is used for addressing a memory location in the code segment of the memory where the executable program is stored

DS : Data Segment
The DS contains most data used by program. Data is accessed in the Data Segment by the offset address or the content of other register that holds the offset address.

SS : Stack Segment
It is usually used to store information about the memory segment that stores the call stack of currently executed program.

ES : Extra Segment
This segment register to gain access to segments when it is difficult or impossible to modify the other segment registers.

Registers

Pointer Registers

SP : Stack Pointer
SP is 16-Bit register pointing to program stack in stack segment

BP : Base Pointer
BP is 16-Bit register pointing to data in stack segment

IP : Instruction Pointer
IP is 16-Bit register pointing to next instructions to be executed

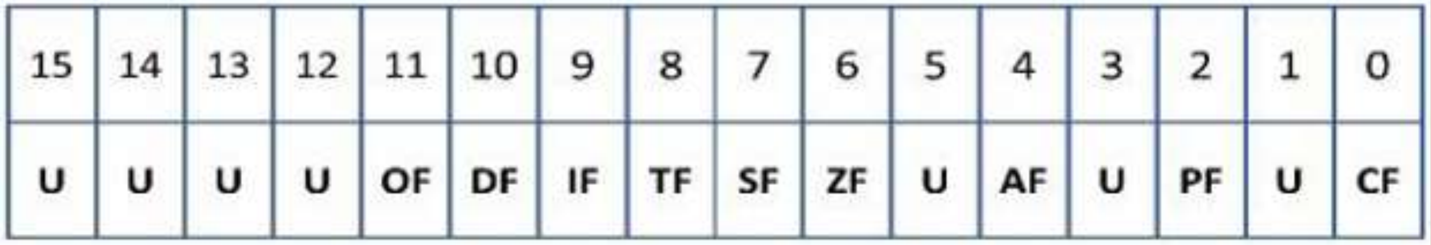
Indexed Registers

SI : Source Index
SI is 16-Bit register. It is used in the pointer addressing of data and as a source in some string-related operations. Its offset is relative to the data segment.

DI : Destination Index
DI is 16-Bit register. It is used in the pointer addressing of data and as a destination in some string-related operations. Its offset is relative to the extra segment.

Registers

Flag Registers



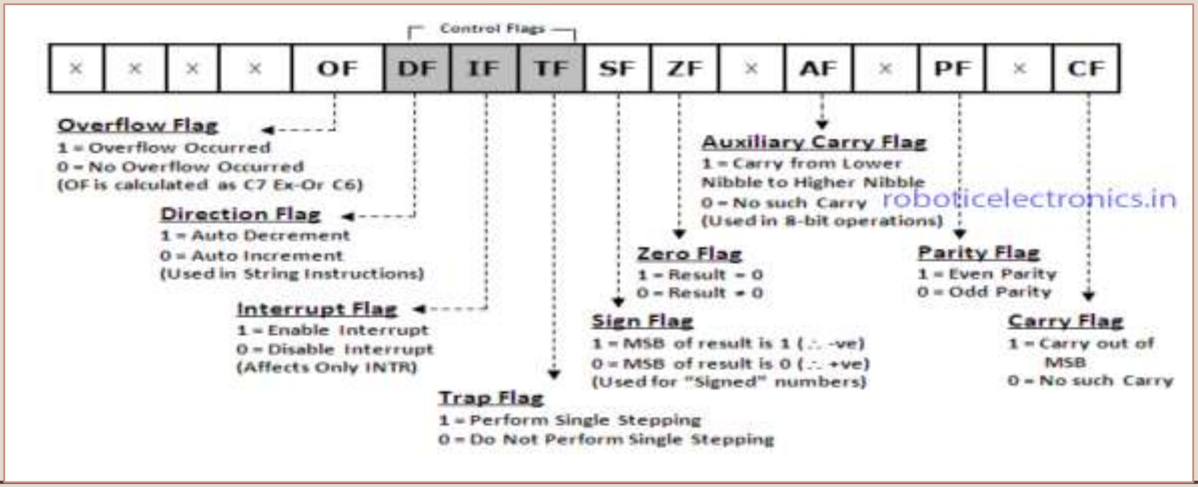
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
U	U	U	U	OF	DF	IF	TF	SF	ZF	U	AF	U	PF	U	CF

The 16-Bit Flag Register contains 9 active flag and remaining flags are undefined

Flag bits are divided in two sections

- Status Flags
- Control Flags

Out of 9 flags, 6 flags are status flags, and 3 flags are control flags



Overflow Flag
1 = Overflow Occurred
0 = No Overflow Occurred
(OF is calculated as C7 Ex-Or C6)

Direction Flag
1 = Auto Decrement
0 = Auto Increment
(Used in String Instructions)

Interrupt Flag
1 = Enable Interrupt
0 = Disable Interrupt
(Affects Only INTR)

Trap Flag
1 = Perform Single Stepping
0 = Do Not Perform Single Stepping

Sign Flag
1 = MSB of result is 1 (∴ -ve)
0 = MSB of result is 0 (∴ +ve)
(Used for "Signed" numbers)

Zero Flag
1 = Result = 0
0 = Result ≠ 0

Auxiliary Carry Flag
1 = Carry from Lower Nibble to Higher Nibble
0 = No such Carry
(Used in 8-bit operations)

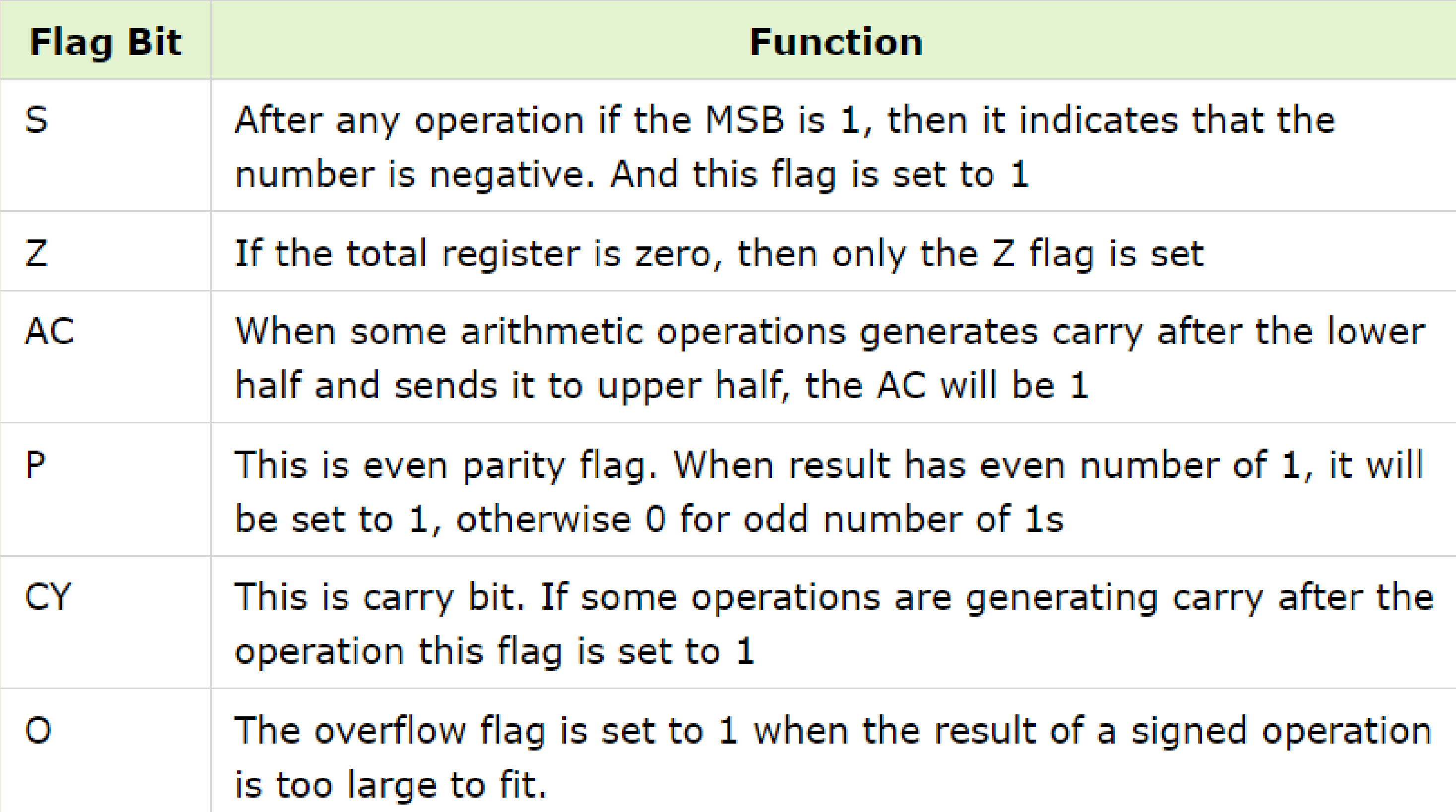
Parity Flag
1 = Even Parity
0 = Odd Parity

Carry Flag
1 = Carry out of MSB
0 = No such Carry

Registers

Status Flag

It indicates certain condition that arises during the execution. They are controlled by the processor.

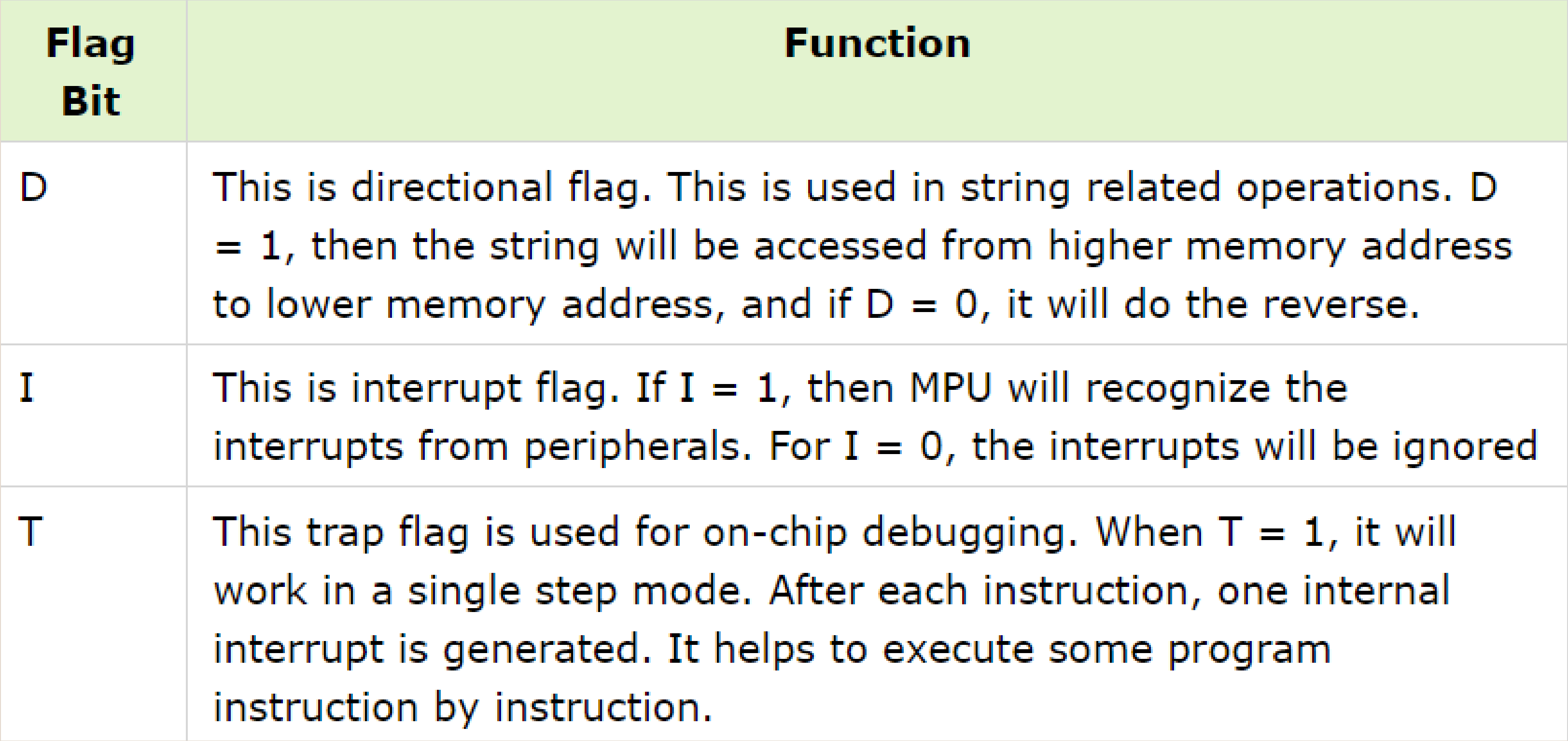


Flag Bit	Function
S	After any operation if the MSB is 1, then it indicates that the number is negative. And this flag is set to 1
Z	If the total register is zero, then only the Z flag is set
AC	When some arithmetic operations generates carry after the lower half and sends it to upper half, the AC will be 1
P	This is even parity flag. When result has even number of 1, it will be set to 1, otherwise 0 for odd number of 1s
CY	This is carry bit. If some operations are generating carry after the operation this flag is set to 1
O	The overflow flag is set to 1 when the result of a signed operation is too large to fit.

Registers

Control Flag

It controls certain operations of the processor. They are deliberately set/ reset by the user.

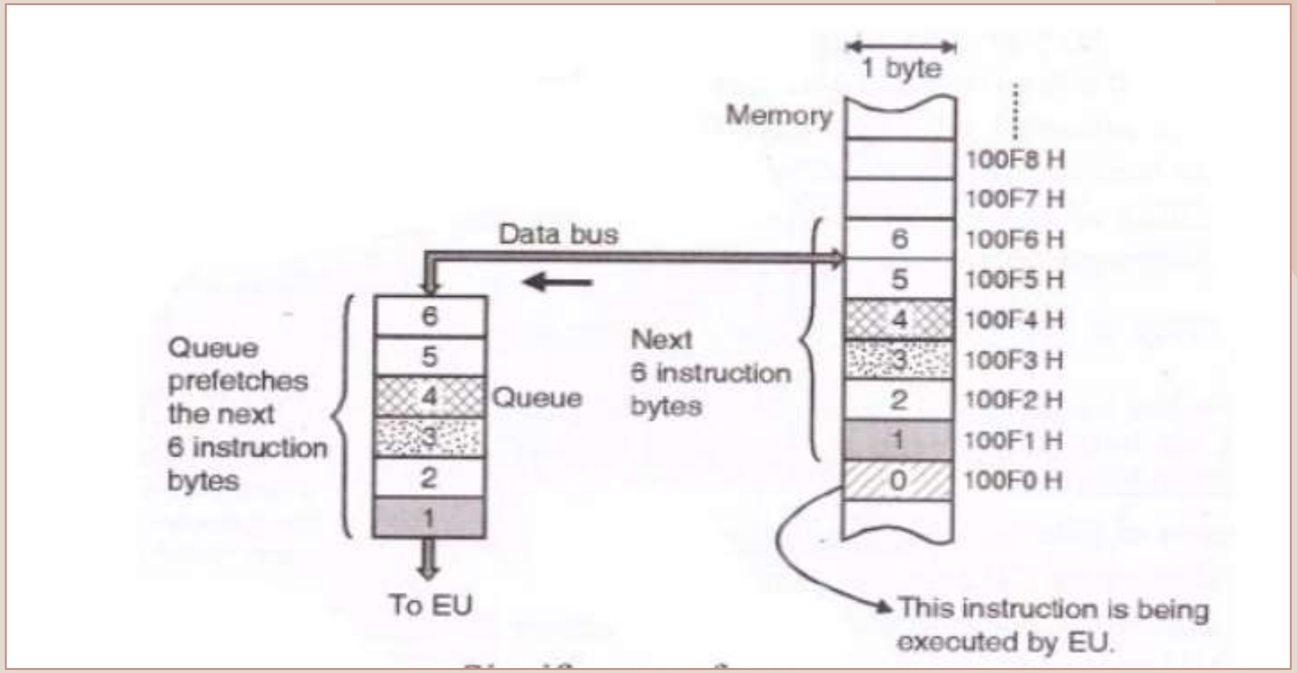


Flag Bit	Function
D	This is directional flag. This is used in string related operations. D = 1, then the string will be accessed from higher memory address to lower memory address, and if D = 0, it will do the reverse.
I	This is interrupt flag. If I = 1, then MPU will recognize the interrupts from peripherals. For I = 0, the interrupts will be ignored
T	This trap flag is used for on-chip debugging. When T = 1, it will work in a single step mode. After each instruction, one internal interrupt is generated. It helps to execute some program instruction by instruction.

Architecture of 8086 : Pipelining

- When the EU is decoding an instruction or executing an instruction, which does not require use of the buses, the BIU fetches up to six instruction bytes for the execution.
- The BIU stores these pre-fetched bytes in a **first-in-first-out** register set called a queue.
- When the EU is ready for its next instruction from the queue in the BIU. This is much faster than sending out an address to the system memory and waiting for memory to send back the next instruction byte or bytes.
- Except in the case of **JMP** and **CALL** instructions, where the queue must be dumped and then reloaded starting from a new address, this pre-fetch and queue scheme greatly speeds up processing.
- Fetching the next instruction while the current instruction executes is called **pipelining**.

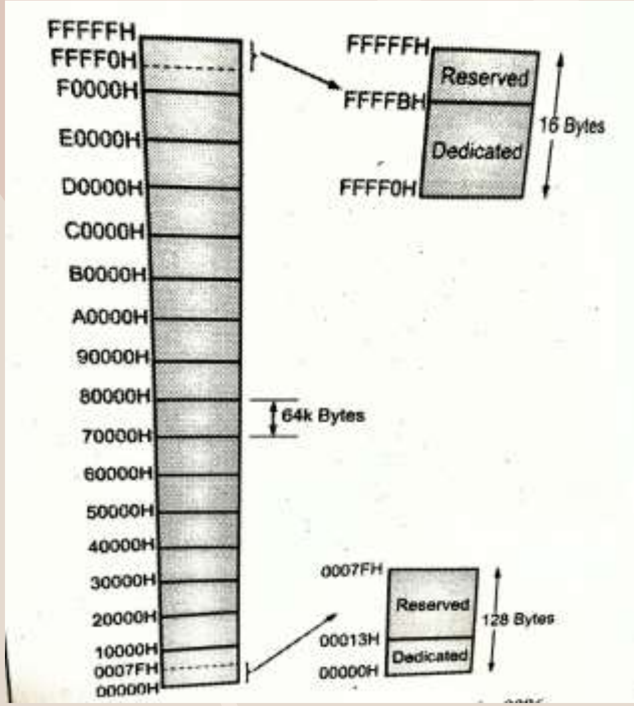
Architecture of 8086 : Pipelining



For example, while the EU is busy in decoding the instruction corresponding to memory location 100F0, the BIU fetches the next six instruction bytes from locations 100F1 to 100F6 numbered as 1 to 6.

Memory Segmentation

- The memory in an 8086 based system is organized as segmented memory.
- The CPU 8086 is able to access IMB of physical memory. The complete 1MB of memory can be divided into 16 segments.
- Each of 64KB size and is addressed by one of the segment register.
- The 16-bit contents of the segment register point to the starting location of a particular segment. The address of the segments may be assigned between 0000H to F000H respectively.
- To address a specific memory location within a segment, we need an offset address. The offset address values are from 0000H to FFFFH so that the physical addresses range from 00000H to FFFFFH.



Memory Segmentation

Mathematically,

Segment = Total memory available/size of each
Segment = 1MB/64KB
= 1024KB/64KB
Segment = 16 segments

8086 has 20 lines address bus.
With 20 address lines, the memory that can be addressed is 2^{20} bytes.

2^{20} = 1,048,576 bytes
= FFFFF H

How 20-Bit Physical Address is Formed

The content of segment register (segment address) is shifted left bit-wise four times.

The content of an offset register (offset address) is added to the result of the previous shift operation.

These two operations together produce a 20-bit physical address.

Physical Address Calculation

Formula - Physical address = Segment address * 10H + Offset address.

Example –
Segment Address = 1005H
Offset Address = 5555H

1005H = 0001 0000 0000 0101
5555H = 0101 0101 0101 0101

Segment address * 10H = Shift left by 4 positions
= 0001 0000 0000 0101 0000

Physical address = 0001 0000 0000 0101 0000 **+** 0101 0101 0101 0101
= 0001 0101 0101 1010 0101

Physical Address Calculation

1. Calculate the physical address for the given CS = 3420H, IP = 689AH
2. Describe physical address generation in 8086. If CS = 2135H and IP = 3478H Calculate the physical address
3. Calculate the physical address if CS = 2308H and IP is equal to 76A9H
4. Describe physical address generation in 8086. If CS = 2000H and IP = 1122H Calculate the physical address
5. Describe physical address generation in 8086. If CS = 69FAH and IP = 834CH Calculate the physical address
6. Calculate the physical address for given-

- DS = 73A2H, SI = 3216H
- CS = 7370H, IP = 561EH

Microprocessors

Course Code
22415

Course Work Journey

		Teaching Hours	Progress Bar
1	8086 – 16 Bit Microprocessor	**8 Hours**	**100%**
2	The Art of Assembly Language Programming	**12 Hours**	**0 %**
3	Instruction Set of 8086 Microprocessor	**16 Hours**	**0 %**
4	Assembly Language Programming	**16 Hours**	**0 %**
5	Procedure and Macros	**12 Hours**	**0 %**

UNIT-2
The Art of Assembly Language Programming

Objective

- Learn to systematically approach problem-solving by writing algorithms, creating flowcharts, and utilizing an initialization checklist to ensure a structured and efficient development process.
- Familiarize yourself with essential programming tools, including text editors for code creation, assemblers for converting assembly code to machine code, linkers for combining multiple code modules, and debuggers for identifying and resolving programming errors.
- Explore assembler directives in-depth, understanding their role in guiding the assembler during the compilation process.

Assembly Language

WHAT IS IT

Assembly language is a low-level programming language that acts as a bridge between human-readable code and the machine language that a computer's processor understands.

Each type of computer processor has its own set of instructions for tasks like taking input or displaying information. These instructions are in machine language, which consists of 1s and 0s and is hard for humans to use.

Assembly language makes it easier for programmers by using symbolic codes that represent those machine instructions in a more understandable way. It's specific to a particular type of processor and provides a simpler way for humans to write programs that the computer's processor can execute.

Assembly Language Program Development Steps

1. **Defining the problem:** The first step in writing program is to think very carefully about the problem that the program must solve.

2. **Algorithm:** It is defined as an ordered set of unambiguous steps that produces a result and terminate in a finite time.

3. **Flowchart:** The flowchart is a graphical representation of the program operation or task.

4. **Initialization checklist:** Initialization task is to make the checklist of entire variables, constants, all the registers, flags and programmable ports

5. **Choosing instructions:** Choose those instructions that make program smaller in size and more importantly efficient in execution.

6. **Converting algorithms to assembly language program:** Every step in the algorithm is converted into program statement using correct and efficient instructions or group of instructions.

Algorithm

An algorithm is a set of instructions designed to perform a specific task.

Task - Write a program to add two integer numbers

```
Step 1 : Start
Step 2 : Declare the variable Num1,Num2 ,Sum
Step 3 : Read value for Num1,Num2
Step 4 : Add two numbers & assign the result to Sum.
            $Sum = Num1 + Num2$
Step 5 : Display the value of Sum
Step 6 : Stop
```

Algorithm

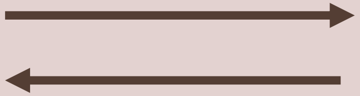
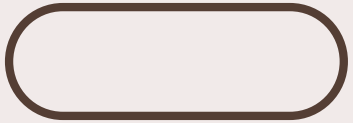
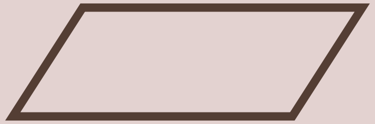
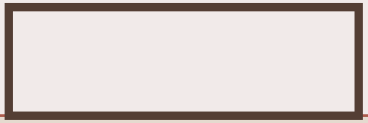
Task - Write a program to calculate area of circle

```
Step 1 : Start
Step 2 : Declare the variable r, area ; Pi = 3.14
Step 3 : Read value for r
Step 4 : Calculate area of circle
             area  = Pi * r * r
Step 5 : Display the value of area
Step 6 : Stop
```

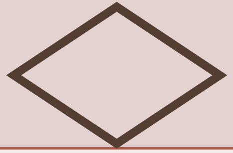
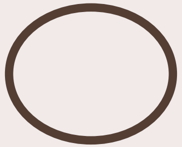
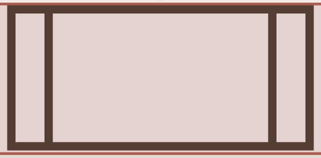
Flowchart

Flowchart is the graphical representation of steps for performing the task.

It shows steps in sequential order and is widely used in presenting the flow of algorithms, workflow or processes.

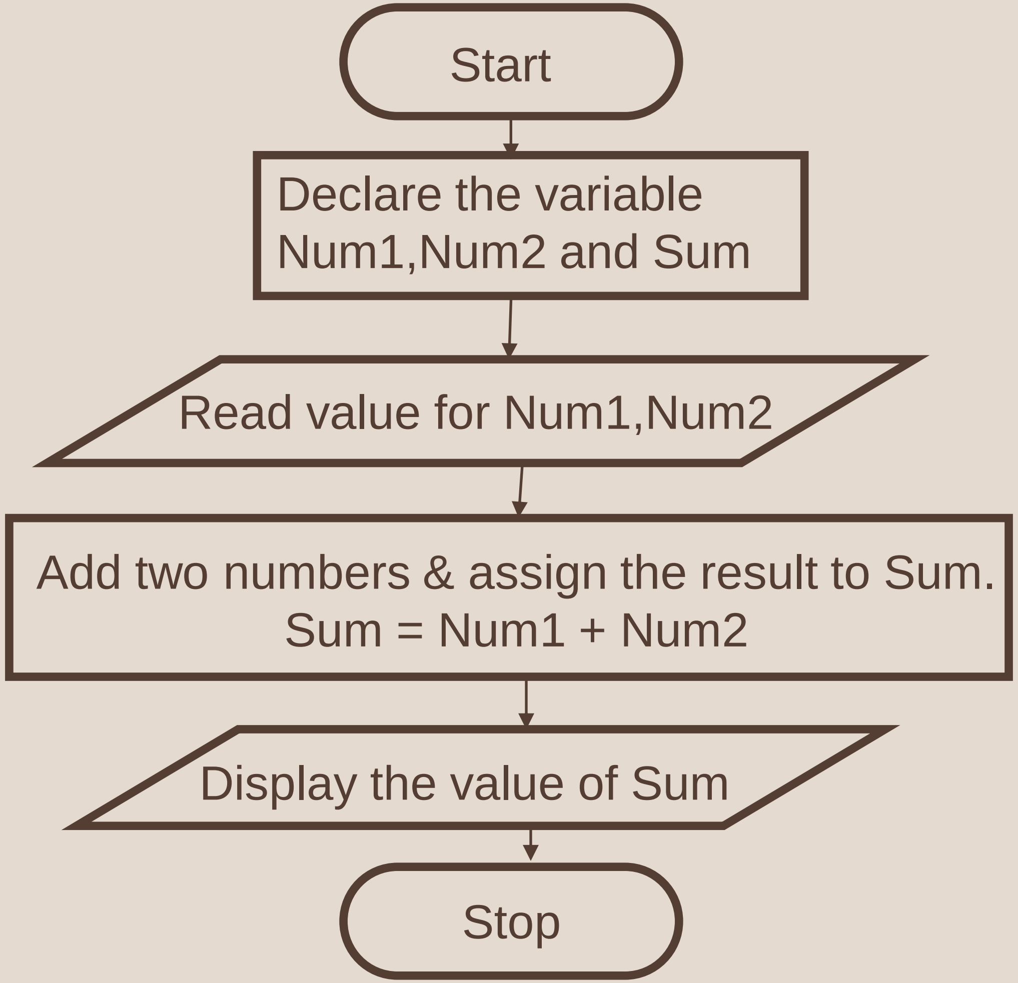
Symbol	Symbol Name	Function
	Flow Lines	Used to connect symbols
	Terminal	Used to start, pause or halt in the program logic
	Input/Output	Represents the information entering or leaving the system
	Processing	Represents arithmetic and logical instructions

Flowchart

Symbol	Symbol Name	Function
	Decision	Represents a decision to be made
	Connector	Used to join different flow lines
	Sub Function	Used to call function

Flowchart

Task - Write a program to add two integer numbers


Start
Declare the variable Num1,Num2 and Sum
Read value for Num1,Num2
Add two numbers & assign the result to Sum.
Sum = Num1 + Num2
Display the value of Sum
Stop

Flowchart

Task - Write a program to calculate area of circle

Introduction to Assembly Language Tools

Tools	Function	Software
Assembler	An assembler is a program that converts source code program written in assembly language into object files in machine language	MASM, TASM, NASM, GNU Assembler
Linker	A linker is a program that combines object file created by the assembler with other object files and link libraries and produces a single executable program	TLINK for TASM, LINK.EXE and LINK32.EXE for MASM
Debugger	A debugger is a program that allows you to trace the execution of a program and examine the content of registers and memory	Turbo Debugger for TASM, CodeView for MASM
Editor	An editor is used to create assembly language source files	Notepad, ConTEXT

Assembler Directives

Assembler directives are the commands to the assembler that direct the assembly process.

Assembly Language Program consist two type of statements.

- The instructions which are translated to machine codes by assembler.
- The directives that direct the assembler during assembly process, for which no machine code is generated.

Assembler Directives

ASSUME

Function - Used to inform the assembler about the logical segments to be assumed for different segments used in the program.

Syntax – ASSUME segreg : segname

Example – ASSUME CS:CODE, DS:DATA, SS:STACK

DB
Defined Byte

Function - The DB directive is used to reserve byte or bytes of **memory locations** in the available memory.

Syntax – Name of variable **DB** initialization value.

Example – MARKS DB 35H,30H,35H,40H
NAME DB “VARDHAMAN”

Assembler Directives

DW
Defined Word

Function - The DW directive is used to define a variable of type word or to reserve storage location of type word in memory

Syntax – Name of variable **DW** initialization value.

Example – MULTIPLIER DW 437AH

DD
Define Double

Function - The directive DD is used to define a double word (4bytes) variable.

Syntax – Name of variable **DD** initialization value.

Example – Data1 DD 12345678H

DQ
Define
Quad Word

Function - This directive is used to direct the assembler to reserve 4 words (8 bytes) of memory for the specified variable

Syntax – Name of variable **DQ** initialization value.

Example – Data1 DQ 123456789ABCDEF2H

Assembler Directives

DT
Define Ten Bytes

Function - The DT directive directs the assembler to define the specified variable requiring 10 bytes for its storage and initialize the 10-bytes with the specified values.

Syntax – Name of variable **DT** initialization value.

Example – Data1 DT 123456789ABCDEF34567H

END
End of Program

Function - The END directive marks the end of an ALP. The statement after the directive END will be ignored by the assembler.

ENDP
End of Procedure

Function - The ENDP directive is used to indicate the end of procedure. In the Assembly Language programming the subroutines are called procedures.

ENDS
End of Segment

Function - The ENDP directive is used to indicate the end of segment.

Example – DATA SEGMENT

```
                .
                        .
                        DATA ENDS
```

Assembler Directives

EQU
Equate

Function - The EQU directive is similar to equal. It has to be used in the beginning of the program.

Example – FACTOR EQU 03H
ADD AL, FACTOR *** Assembler will code it as ADD AL, 03H ***

ORG
Originate

Function – The ORG directive allows you to set the location counter to a desired value at any point in the program

Example – ORG 2000H
Tells the assembler to set the location counter to 2000H

SEGMENT

Function –It is used to define segments

Example –
```
CODE SEGMENT
              Code instructions go here
        CODE ENDS
```

Thank you

Microprocessors

Course Code
22415

Course Work Journey

		Teaching Hours	Progress Bar
1	8086 – 16 Bit Microprocessor	**8 Hours**	**100%**
2	The Art of Assembly Language Programming	**12 Hours**	**100 %**
3	Instruction Set of 8086 Microprocessor	**16 Hours**	**0 %**
4	Assembly Language Programming	**16 Hours**	**0 %**
5	Procedure and Macros	**12 Hours**	**0 %**

UNIT-3
Instruction Set of 8086 Microprocessor

Objective

- Learn about the addressing modes along with the operation performed by them
- Understand various instruction set available in 8086 microprocessor

Machine Language Instruction Format

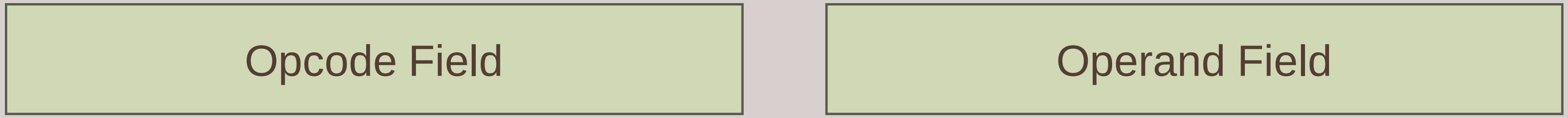
Instruction is a command given to the microprocessor to perform a specific task on specified data.

Each instruction has two parts –

1. Opcode
2. Operand

Opcode : Task to be performed
Operand : The data field to be operated on

Instruction Format :



Opcode Field

Operand Field

Machine Language Instruction Format

Instruction Format (1 Byte) :

- The first byte always consists of opcode
- In this byte, first 6 bits are of Opcode which defines the operation to be carried out by an instruction

Opcode	**D**	**W**
6-Bit	1-Bit	1-Bits

D stands for direction.

If D=0, then the direction is from the register

If D=1, then the direction is to the register

W stands for word.

If W=0, then only a byte is being transferred, i.e. 8 bits

If W=1, them a whole word is being transferred, i.e. 16 bits

Machine Language Instruction Format

Instruction Format (2 Byte) :

- This format is two byte long
- In this format, first byte consists of opcode and width of an operand specified by the D and W.

Opcode	**D**	**W**
6-Bit	1-Bit	1-Bits

- Whereas Second byte consists of MOD, REG and R/M field

Opcode	**D**	**W**	**MOD**	**REG**	**R/M**
6-Bit	1-Bit	1-Bits	2-Bits	3-Bits	3-Bits

Machine Language Instruction Format

Opcode 6-Bit	**D** 1-Bit	**W** 1-Bits

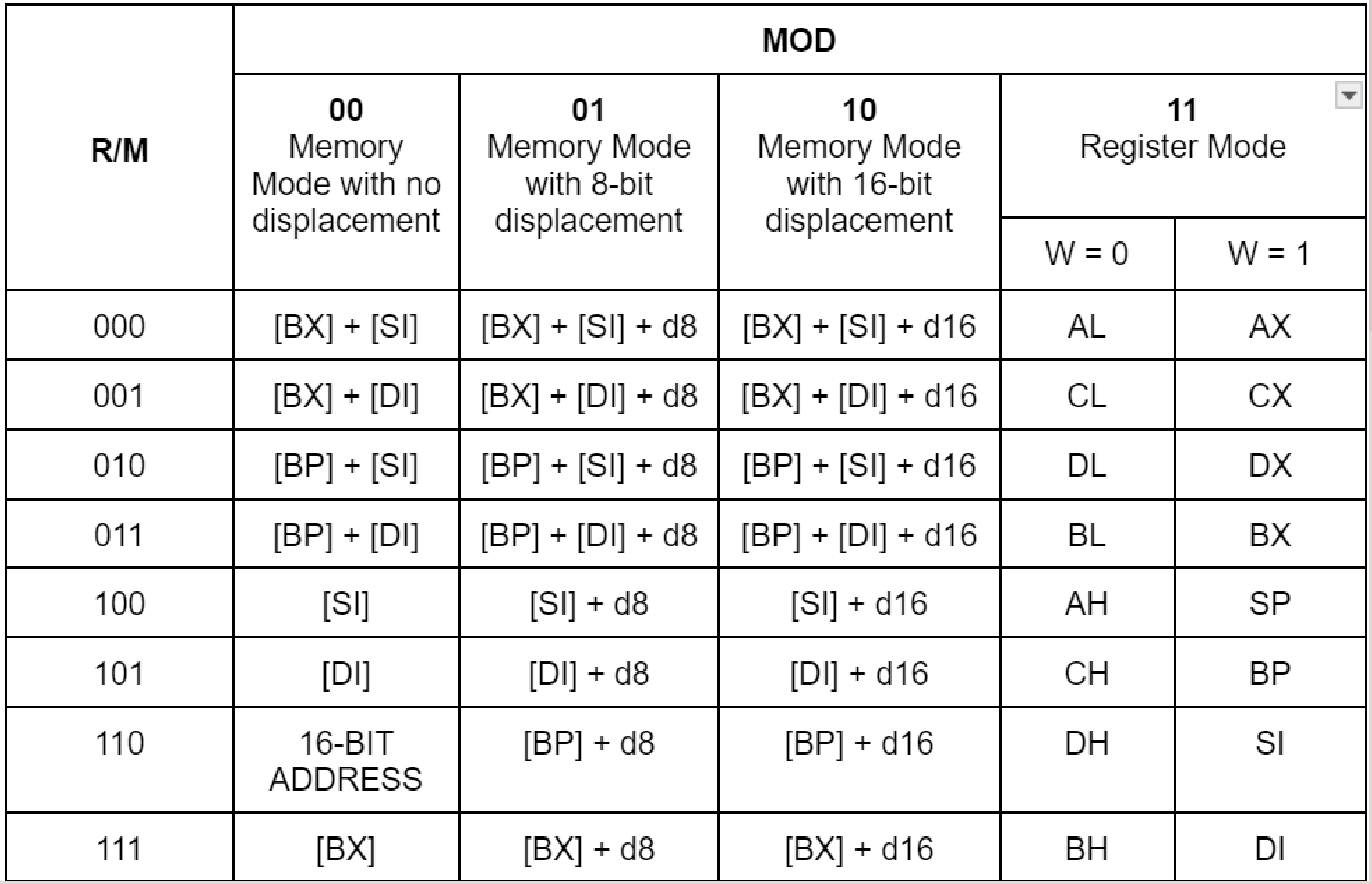
Opcode 6-Bit	**D** 1-Bit	**W** 1-Bits	**MOD** 2-Bits	**REG** 3-Bits	**R/M** 3-Bits

Instruction Format (2 Byte) :

- MOD indicates the displacement is present or not. If present, then it is 8-bit or 16-bit.
- REG indicates the name of the register that is source or destination
- R/M indicates source or destination operand located in register

Machine Language Instruction Format

The MOD and R/M together is calculated based upon the addressing mode and register being used in it. This is calculated as follows:



R/M	MOD				
	00 Memory Mode with no displacement	**01** Memory Mode with 8-bit displacement	**10** Memory Mode with 16-bit displacement	**11** Register Mode	
				W = 0	W = 1
000	[BX] + [SI]	[BX] + [SI] + d8	[BX] + [SI] + d16	AL	AX
001	[BX] + [DI]	[BX] + [DI] + d8	[BX] + [DI] + d16	CL	CX
010	[BP] + [SI]	[BP] + [SI] + d8	[BP] + [SI] + d16	DL	DX
011	[BP] + [DI]	[BP] + [DI] + d8	[BP] + [DI] + d16	BL	BX
100	[SI]	[SI] + d8	[SI] + d16	AH	SP
101	[DI]	[DI] + d8	[DI] + d16	CH	BP
110	16-BIT ADDRESS	[BP] + d8	[BP] + d16	DH	SI
111	[BX]	[BX] + d8	[BX] + d16	BH	DI

Machine Language Instruction Format

The general Instruction format that most of the instructions of the 8086-microprocessor follow is:

Opcode 6-Bit	**D** 1-Bit	**W** 1-Bits	**MOD** 2-Bits	**REG** 3-Bits	**R/M** 3-Bits	**Lower Order bits of displacement**	**Higher Order bits of displacement**

- The low order displacement and high order displacement are optional, and the instruction format contains them only if there exists any displacement in the instruction.

- If the displacement is of 8 bits, then only the cell of low order displacement is filled and if the displacement is of 16 bits, then both the cells of low order and high order are filled, with the exact bits that the displacement number represents.

Addressing Modes

The way of specifying data to be operated by an instruction is known as addressing modes. This specifies that the given data is an immediate data or an address.

TYPES OF ADDRESSING MODES

Register Mode: In this type of addressing mode both the operands are registers. Operands are stored in 16-bit general purpose register.

```
Example - MOV AX, BX
          XOR AX, DX
          ADD AL, BL
```

Immediate mode – In this mode, the operand is contained in the instruction itself. Operand can be 8-bit or 16-bit in length.

```
Example -MOV AX, 2000
         MOV CL, 0A
         ADD AL, 45
         AND AX, 0000
```

Remember,

To initialize the value of segment register a register is required.

```
Example - MOV AX, 2000
          MOV CS, AX
```

Types of Addressing Modes

Direct Mode: In this mode, the effective address is directly given in the instruction as displacement.

Example - MOV AX, [4321H]
MOV AX, [0500H]

Register Indirect mode – In this addressing mode the effective address resides in either Index Register (SI or DI) or Base Register BX.

Example: **Physical Address = Segment Address + Effective Address**
MOV AX, [DI]
ADD AL, [BX]
MOV AX, [SI]

Based indexed mode: It is the combination of based and indexed addressing modes. The physical memory address is calculated according to the base register.

Example - MOV AL, [BP+SI]
MOV AX, [BX+DI]

Instruction Set of 8086 Microprocessor

Instruction is a command to perform a specific task

The 8086 microprocessor supports 8 types of instructions −

- Data Transfer Instructions
- Arithmetic Instructions
- Bit Manipulation Instructions
- String Instructions
- Program Execution Transfer Instructions (Branch & Loop Instructions)
- Processor Control Instructions
- Iteration Control Instructions
- Interrupt Instructions

Data Transfer Instruction

Data transfer instructions are used to transfer the data from the source operand to the destination operand.

Following are the list of instructions under this group to transfer the data :

- **MOV**
- **PUSH**
- **POP**
- **XCHG**
- **IN**
- **OUT**
- **XLAT**
- **LEA**
- **LDS/LES**
- **LAHF**
- **SAHF**
- **PUSHF**
- **POPF**

Data Transfer Instruction

Instruction	Example	Explanation
MOV	MOV AX, BX	Move the contents of BX into AX.
PUSH	PUSH AX	Push the value in AX onto the stack.
POP	POP BX	Pop the top value from the stack into BX.
XCHG	XCHG AX, BX	Exchange the contents of AX and BX.
IN	IN AL, 60h	Input a byte from port 60h into AL.
OUT	OUT 70h, AL	Output the value in AL to port 70h.
XLAT	XLAT	Translate a byte using the AL register.
LEA	LEA SI, [BX+2]	Load SI with the effective address of BX+2.
LDS/LES	LDS SI, [1234]	Load DS:SI with the 32-bit pointer at 1234H.
LAHF	LAHF	Load AH with the flags in the FLAGS register.
SAHF	SAHF	Store the flags in AH into the FLAGS register.
PUSHF	PUSHF	Push the flags onto the stack.
POPF	POPF	Pop the top value from the stack into flags.

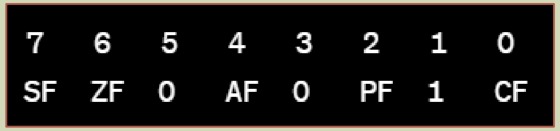
Data Transfer Instruction

- **XLAT :**

Example - Table DB 'ABCDEFGH’
MOV AL, 3
XLAT

- **LAHF :**

Example - LAHF loads the lower byte of the AX register with the contents of the flags register (FLAGS).



7	6	5	4	3	2	1	0
SF	ZF	0	AF	0	PF	1	CF

- **SAHF :**

Example - SAHF does the reverse of LAHF. It takes the values stored in the higher byte of the AX register (AH) and updates the status flags in the FLAGS register accordingly.

Arithmetic Instruction

Arithmetic Instructions are the instructions which perform basic arithmetic operations such as addition, subtraction and a few more.

Following are the list of instructions under this group to perform arithmetic operations:

- **ADD, ADC**
- **INC**
- **DEC**
- **SUB, SBB**
- **CMP**
- **DAA, DAS**
- **NEG**
- **MUL, IMUL**
- **CBW**
- **CWD**
- **DIV, IDIV**

Arithmetic Instruction

Instruction	Example	Explanation
ADD	ADD AX, BX	Adds the contents of BX to AX.
ADC	ADC AX, BX	Adds the contents of BX and the carry flag to AX.
INC	INC CX	Increments the value in CX by 1.
DEC	DEC DX	Decrements the value in DX by 1.
SUB	SUB AX, BX	Subtracts the contents of BX from AX.
SBB	SBB AX, BX	Subtracts the contents of BX and the borrow flag from AX.
DAA	DAA	Decimal Adjust after Addition. Changes the content of register from binary to 4-bit BCD
DAS	DAS	Decimal Adjust after Subtraction.
CMP	CMP AX, BX	Compares AX and BX without changing AX.
NEG	NEG AX	Negates (changes the sign of) the value in AX.
MUL	MUL BX	Multiplies AX by BX (unsigned).
IMUL	IMUL CX, DX	Multiplies CX by DX (signed).
CBW	CBW	Convert Byte to Word (sign-extend AL into AX).

Arithmetic Instruction

Instruction	Example	Explanation
CWD	CWD	Convert Word to Double Word (sign-extend AX into DX:AX).
DIV	DIV CX	Divides DX:AX by CX, result in AX, remainder in DX.
IDIV	IDIV BX	Divides DX:AX by BX (signed), result in AX, remainder in DX.

Logical Instruction and Bit Manipulation Instruction

Logical instructions are the instructions that perform basic logical operations.

These instructions are used to perform operations where data bits are involved which performs operations like logical, shift, etc.

Following are the list of instructions under this group to perform arithmetic operations:

- **AND (Logical AND)**
- **OR (Logical OR)**
- **NOT (Logical Invert)**
- **XOR (Logical Exclusive OR)**
- **TEST (Logical Compare Instruction)**
- **SHL/SAL (Shift Logical/ Arithmetic Left)**
- **SHR (Shift Logical Right)**
- **SAR (Shift Arithmetic Right)**
- **ROR (Rotate Right without carry)**
- **ROL (Rotate Left without carry)**
- **RCR (Rotate Right through carry)**
- **RCL (Rotate Left through carry)**

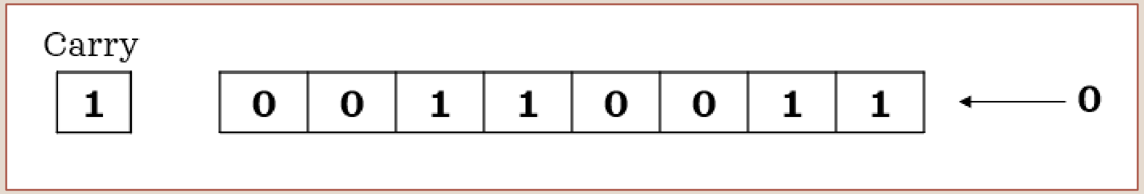
Logical Instruction

Instruction	Example	Explanation
AND	AND AX, BX	Performs a bitwise AND operation between AX and BX.
OR	OR AX, BX	Performs a bitwise OR operation between AX and BX.
NOT	NOT AX	Performs a bitwise NOT operation on the contents of AX.
XOR	XOR AX, BX	Performs a bitwise XOR operation between AX and BX.
TEST	TEST AX, BX	Performs a bitwise AND operation between AX and BX (no result). Updates the carry flag with the performed operation
SHL/SAL	SHL AX, 1	Shifts the bits in AX left by 1 position.
SHR	SHR AX, 1	Shifts the bits in AX right by 1 position.
SAR	SAR AX, 1	Arithmetic right shift of the bits in AX by 1 position.
ROR	ROR AX, 1	Rotates the bits in AX right by 1 position.
ROL	ROL AX, 1	Rotates the bits in AX left by 1 position.
RCR	RCR AX, 1	Rotates the bits in AX right through carry by 1 position.
RCL	RCL AX, 1	Rotates the bits in AX left through carry by 1 position.

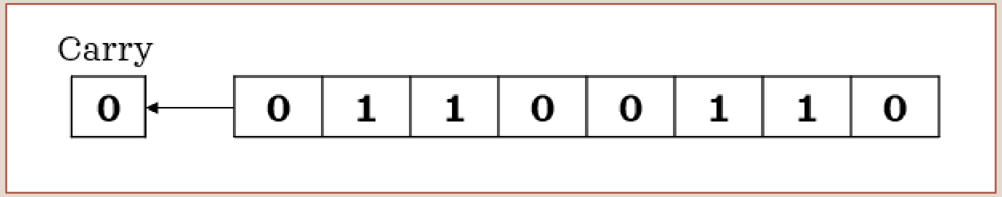
Logical Instruction

SHL : Shift Logical Left

Example : SHL BL, 1
It suggests that left-shift BL bits once
Assume that,
Before operation,
BL = 00110011 and CF = 1



After Operation,

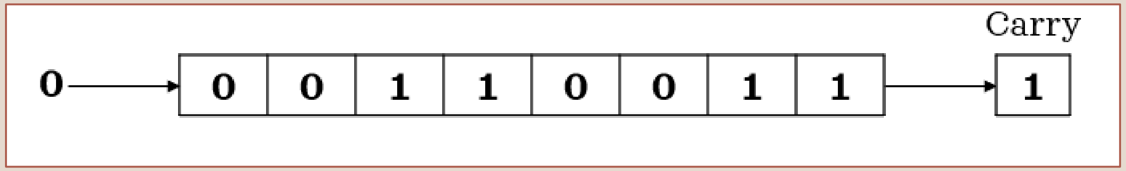


Logical Instruction

SHR : Shift Logical Right

Example : SHR BL, 1
It suggests that right-shift BL bits once
Assume that,
Before operation,

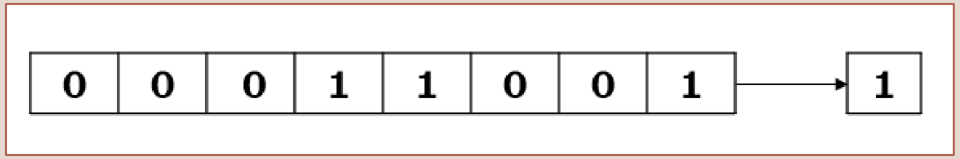
BL = 00110011 and CF = 1



Carry

0 → 0 0 1 1 0 0 1 1 → 1

After Operation,

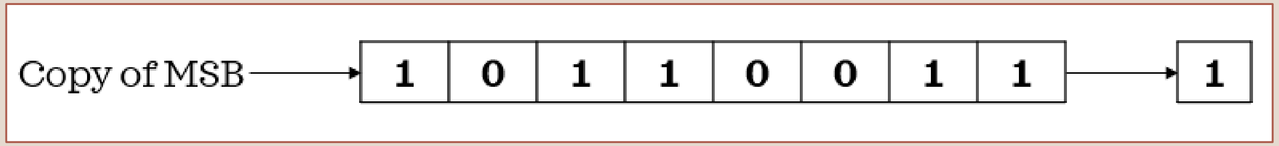


0 0 0 1 1 0 0 1 → 1

Logical Instruction

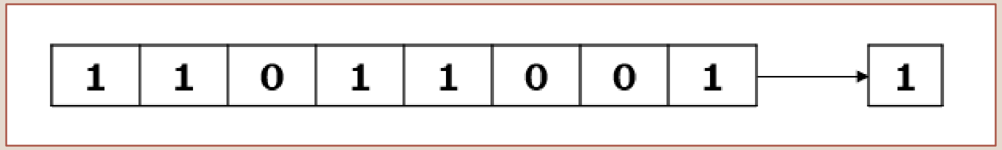
SAR : Shift Arithmetic Right

Example : SAR BL, 1
It suggests that right-shift BL bits once
Assume that,
Before operation,
BL = 10110011   and CF = 1



Copy of MSB → 1 0 1 1 0 0 1 1 → 1

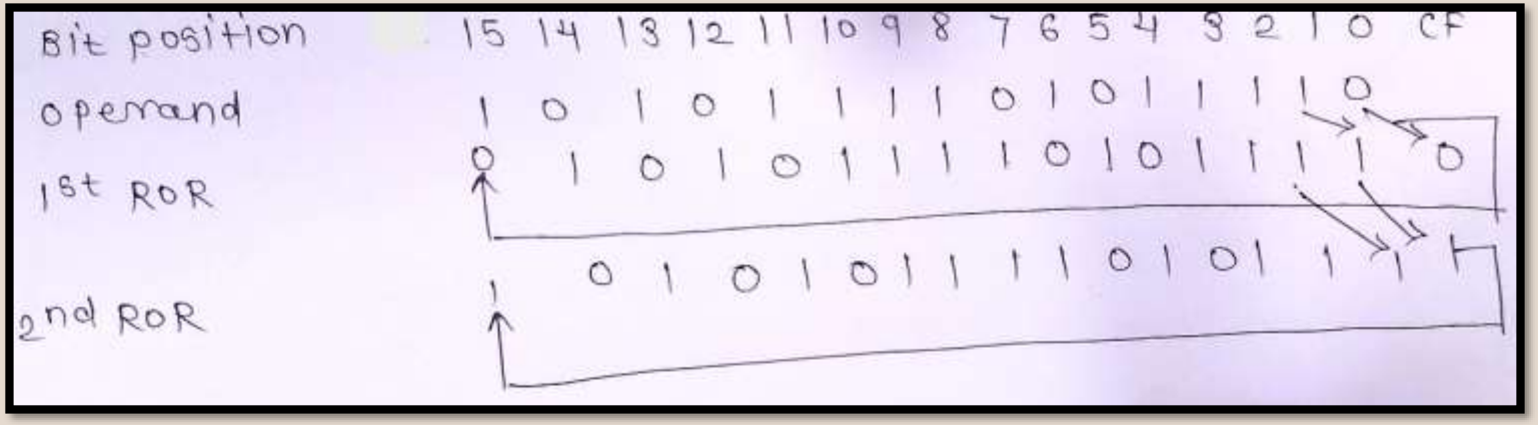
After Operation,



1 1 0 1 1 0 0 1 → 1

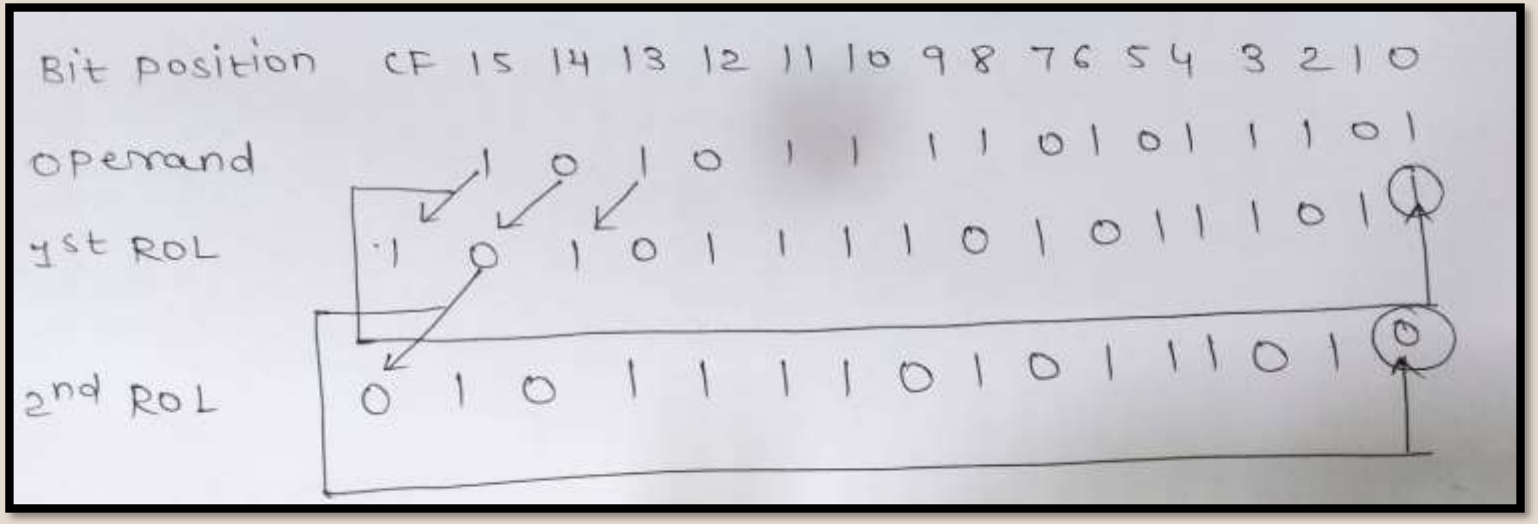
Logical Instruction

ROR : Rotate Right



Bit position	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	CF
operand	1	0	1	0	1	1	1	1	0	1	0	1	1	1	1	0	
1st ROR	0	1	0	1	0	1	1	1	1	0	1	0	1	1	1	1	0
2nd ROR	1	0	1	0	1	0	1	1	1	1	0	1	0	1	1	1	1

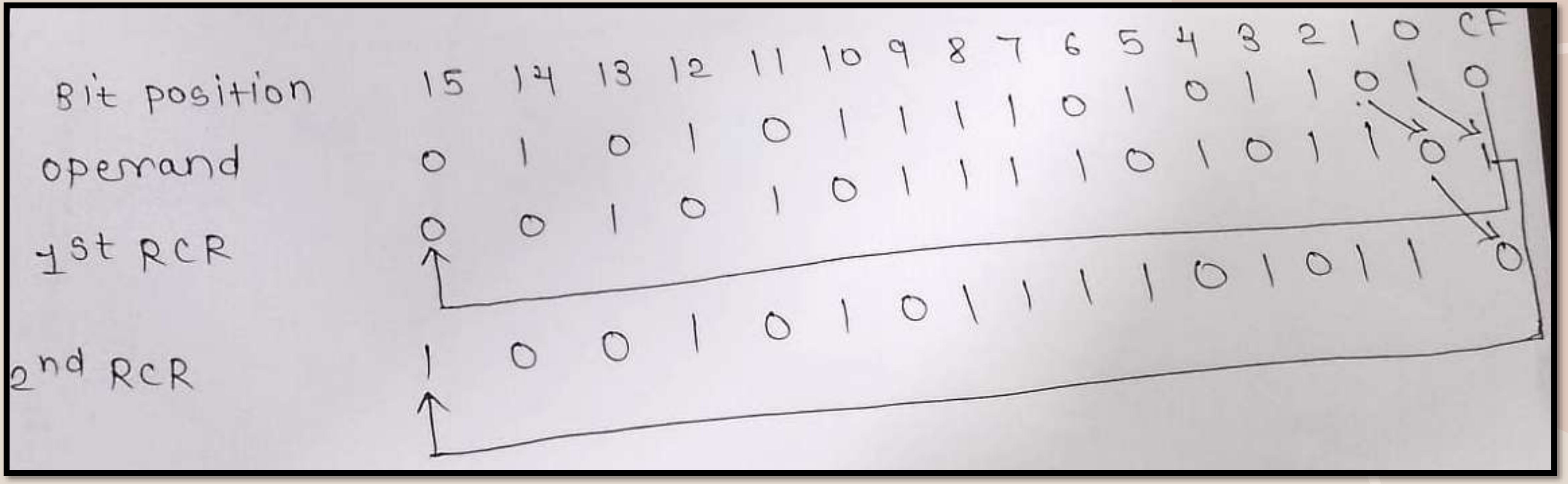
ROL : Rotate Left



Bit position	CF	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
operand		1	0	1	0	1	1	1	1	0	1	0	1	1	1	0	1
1st ROL	1	0	1	0	1	1	1	1	0	1	0	1	1	1	0	1	1
2nd ROL	0	1	0	1	1	1	1	0	1	0	1	1	1	0	1		0

Logical Instruction

RCR : Rotate Right with Carry



Bit position	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	CF
operand	0	1	0	1	0	1	1	1	1	0	1	0	1	1	0	1	0
1st RCR	0	0	1	0	1	0	1	1	1	1	0	1	0	1	1	0	1
2nd RCR	1	0	0	1	0	1	0	1	1	1	1	0	1	0	1	1	0

String Manipulation Instruction

String is a group of bytes/words, and their memory is always allocated in a sequential order. String is either referred as byte string or word string.

Following are the list of instructions under this group to perform string manipulation:

- **REP**
- **OR (Logical OR)**
- **NOT (Logical Invert)**
- **XOR (Logical Exclusive OR)**
- **TEST (Logical Compare Instruction)**
- **SHL/SAL (Shift Logical/ Arithmetic Left)**
- **SHR (Shift Logical Right)**
- **SAR (Shift Arithmetic Right)**
- **ROR (Rotate Right without carry)**
- **ROL (Rotate Left without carry)**
- **RCR (Rotate Right through carry)**
- **RCL (Rotate Left through carry)**

String Manipulation Instruction

Instruction	Example	Explanation
REP	REP MOVSB	It will continue to copy string bytes until the number bytes loaded into CX has been copied
REPE/REPZ	REPE CMPSB	Repeat if Equal and Repeat if Zero, Compare String Bytes until end of string or until strings are not equal
REPNE/REPNZ	REPNE SCASW	Repeat if Not Equal and Repeat if Not Zero, Scan a string of words until a word in string matches the word in AX
MOVS/MOVSB	MOVS DI, SI	Moves a byte from DS:SI to ES:DI
CMPS/CMPSB	CMPSB	Compares the byte at DS:SI with the byte at ES:DI.
INS/INSB	INSB DX	Inputs a byte from the I/O port specified by DX to the memory location addressed by ES:DI.
OUTS/OUTSB	OUTSB DX	Outputs the byte at DS:SI to the I/O port specified by DX.
SCAS/SCASB	SCASB	Scans the byte at ES:DI and compares it with AL.
LODS/LODSB	LODSB	Loads a byte from DS:SI into AL or AX

Program Control Transfer and Branching Instruction

These instructions are used to transfer/branch the instructions during an execution.

There are two types of branching instructions –

- Unconditional branch
- Conditional branch

The Unconditional Program execution control transfer instructions are as follows

- **CALL**
- **RET**
- **JMP**
- **LOOP**

Program Control Transfer and Branching Instruction

Instruction	Example	Explanation
CALL	CALL subroutine	Calls a subroutine or procedure at the specified address.
RET	RET	Returns control from a subroutine to the calling program.
JMP	JMP label	Jumps to the specified label or memory address unconditionally.
LOOP	LOOP destination	Decrements CX by 1 and jumps to the destination if CX ≠ 0.

Program Control Transfer and Branching Instruction

These instructions are used to transfer/branch the instructions during an execution.

There are two types of branching instructions –

- Unconditional branch
- Conditional branch

The Conditional Program execution control transfer instructions are as follows

- **JC**
- **JNC**
- **JE/JZ**
- **JNE/JNZ**
- **JO**
- **JNO**
- **JP/JPE**
- **JNP/JPO**
- **JS**
- **JNS**
- **JA/JNBE**
- **JAE/JNB**
- **JB/JNAE**
- **JBE/JNA**
- **JG/JNLE**
- **JGE/JNL**
- **JUJNGE**
- **JLE/JNG**
- **LOOPE/LOOPZ**
- **LOOPNE/LOOPNZ**

Program Control Transfer and Branching Instruction

Instruction	Example	Explanation
JC	JC address	Jump to the specified address if the carry flag (CY) is set (CY = 1).
JNC	JNC address	Jump to the specified address if the carry flag (CY) is not set (CY = 0).
JE/JZ	JE/JZ address	Jump to the specified address if the zero flag (ZF) is set (ZF = 1).
JNE/JNZ	JNE/JNZ address	Jump to the specified address if the zero flag (ZF) is not set (ZF = 0).
JO	JO address	Jump to the specified address if the overflow flag (OF) is set (OF = 1).
JNO	JNO address	Jump to the specified address if the overflow flag (OF) is not set (OF = 0).
JP/JPE	JP/JPE address	Jump to the specified address if the parity flag (PF) is set (PF = 1).
JNP/JPO	JNP/JPO address	Jump to the specified address if the parity flag (PF) is not set (PF = 0).
JS	JS address	Jump to the specified address if the sign flag (SF) is set (SF = 1).

Program Control Transfer and Branching Instruction

Instruction	Example	Explanation
JNS	JNS address	Jump to the specified address if the sign flag (SF) is not set (SF = 0).
JA/JNBE	JA/JNBE address	Jump to the specified address if above/not below/equal (CF = 0 and ZF = 0).
JAE/JNB	JAE/JNB address	Jump to the specified address if above or equal/not below (CF = 0).
JB/JNAE	JB/JNAE address	Jump to the specified address if below/not above or equal (CF = 1).
JBE/JNA	JBE/JNA address	Jump to the specified address if below or equal/not above (CF = 1 or ZF = 1).
JG/JNLE	JG/JNLE address	Jump to the specified address if greater/not less than or equal (ZF = 0 and SF = OF).
JGE/JNL	JGE/JNL address	Jump to the specified address if greater or equal/not less than (SF = OF).
JL/JNGE	JL/JNGE address	Jump to the specified address if less than/not greater than or equal (SF ≠ OF).
JLE/JNG	JLE/JNG address	Jump to the specified address if less than or equal/not greater than (ZF = 1 or SF ≠ OF).

Program Control Transfer and Branching Instruction

Instruction	Example	Explanation
JCXZ	JCXZ address	Jump to the specified address if the CX register is zero.
LOOPE/LOOPZ	LOOPE/LOOPZ address	Loop until ZF = 1 and CX = 0.
LOOPNE/LOOPNZ	LOOPNE/LOOPNZ address	Loop until ZF = 0 and CX = 0.

Process Control Instruction

These instructions are used to control the processor action by setting/resetting the flag values.

The Conditional Program execution control transfer instructions are as follows

- **STC**
- **CLC**
- **CMC**
- **STD**
- **CLD**
- **STI**
- **CLI**

Process Control Instruction

Instruction	Example	Explanation
STC	STC	Sets the carry flag (CF) to 1.
CLC	CLC	Clears or resets the carry flag (CF) to 0.
CMC	CMC	Complements the state of the carry flag (CF).
STD	STD	Sets the direction flag (DF) to 1.
CLD	CLD	Clears or resets the direction flag (DF) to 0.
STI	STI	Sets the interrupt enable flag to 1, enabling interrupts (INTR input).
CLI	CLI	Clears the interrupt enable flag to 0, disabling interrupts (INTR input).

Thank you

Microprocessors

Course Code
22415

Course Work Journey

		Teaching Hours	Progress Bar
1	8086 – 16 Bit Microprocessor	**8 Hours**	**100%**
2	The Art of Assembly Language Programming	**12 Hours**	**100 %**
3	Instruction Set of 8086 Microprocessor	**16 Hours**	**100 %**
4	Assembly Language Programming	**16 Hours**	**0 %**
5	Procedure and Macros	**12 Hours**	**0 %**

UNIT-4
Assembly Language Programming

Objective

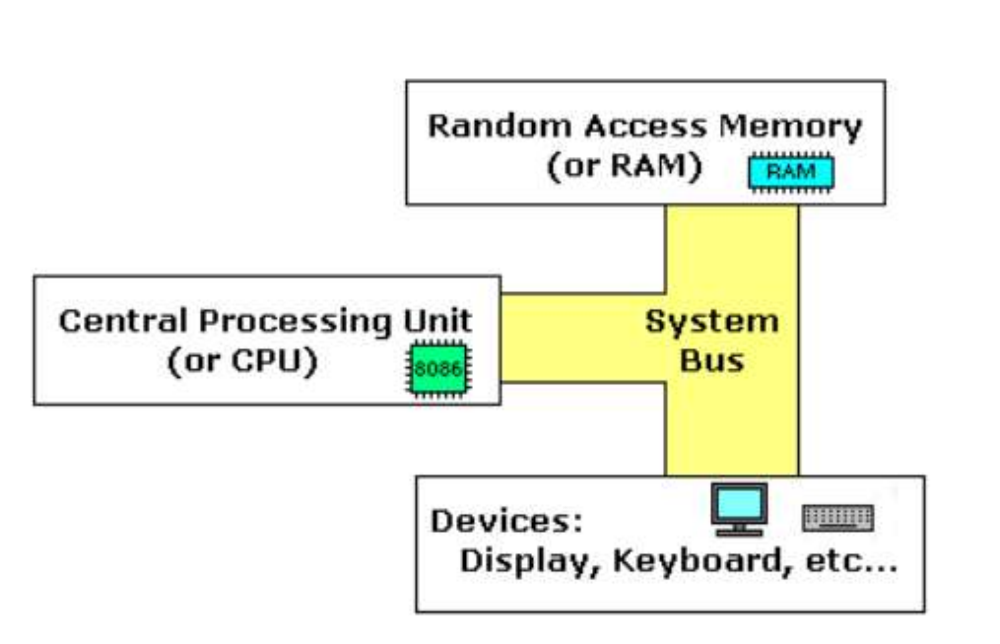
- Develop relevant program for the given problem.
- Perform block transfer and string manipulation operations.

Overview

What is Assembly Language ?
Assembly language is a low-level programming language.

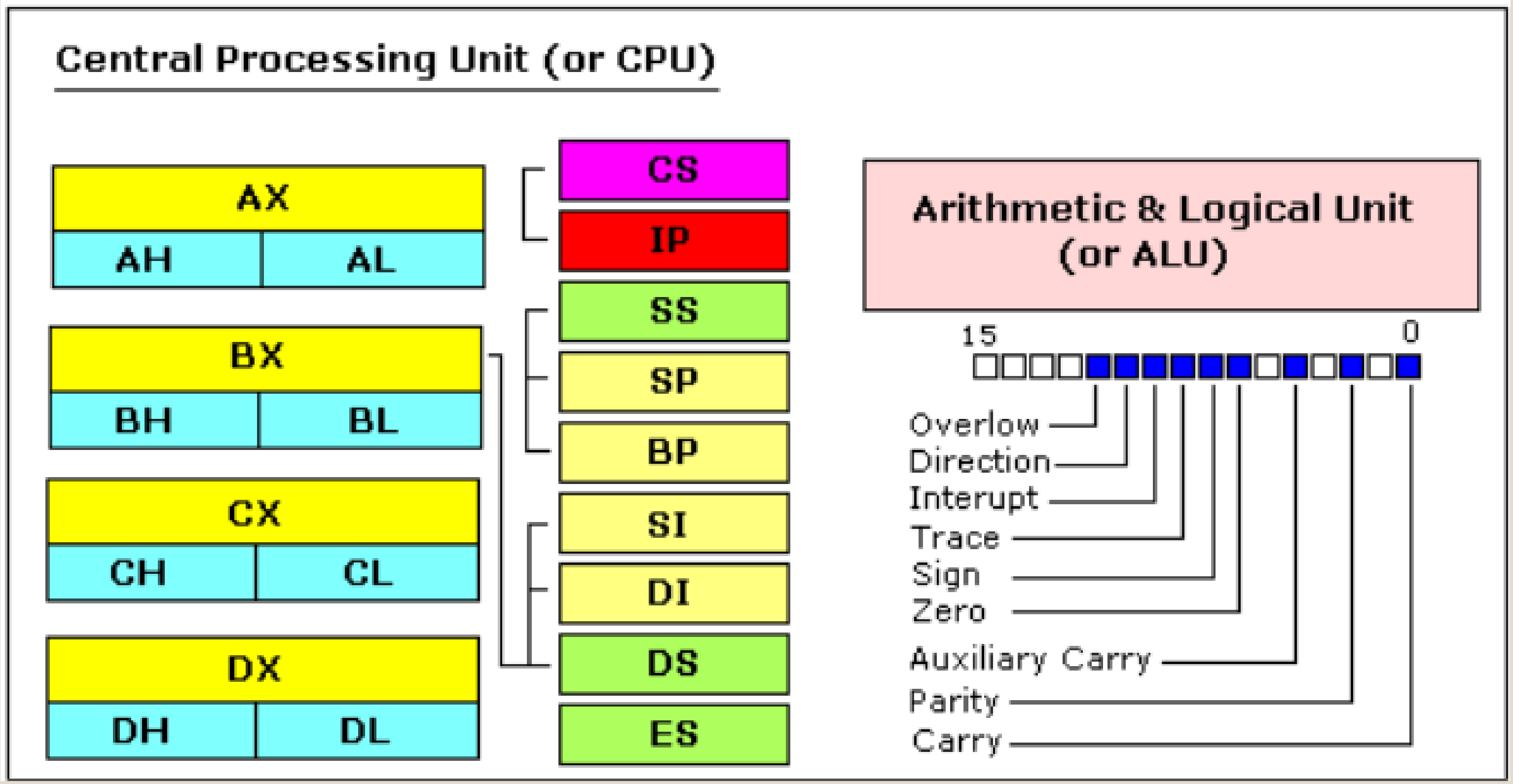
Computer Architecture -

- A simple computer has a **system bus** which connects various components of the computer.
- **CPU** is the heart of the computer, most of the computational tasks are performed in CPU
- **RAM** is the memory unit in which programs are loaded in order to be executed.



Overview

What's There Inside?



Note :

As registers are located inside the CPU, they are much faster than memory.

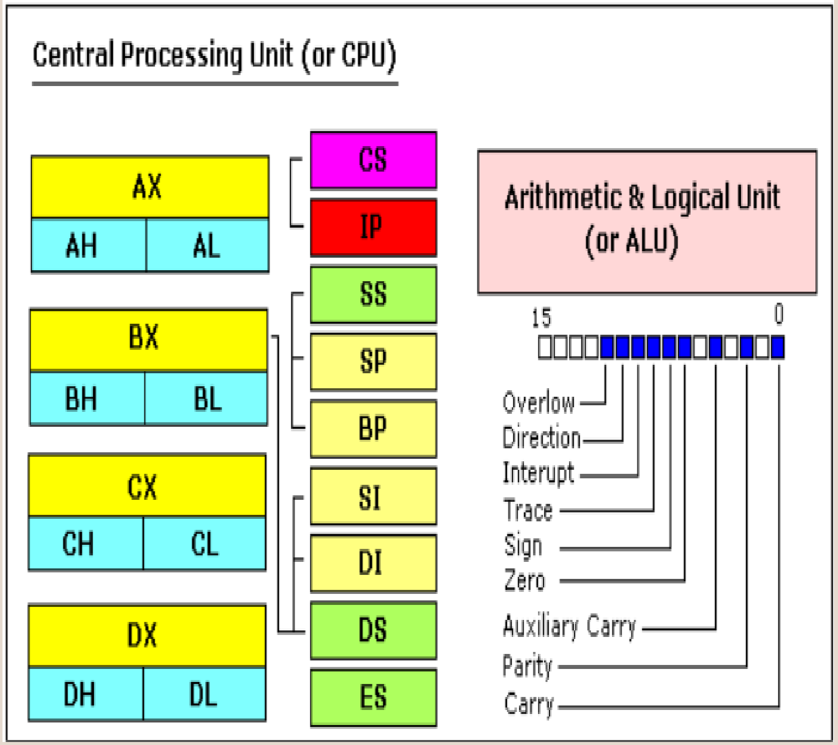
Accessing a memory location requires the use of a system bus, so it takes much longer.

Therefore, you should try to keep variables in the registers.

Register sets are very small and most registers have special purposes which limit their use as variables, but they are still an excellent place to store temporary data of calculations.

Overview

What’s There Inside?



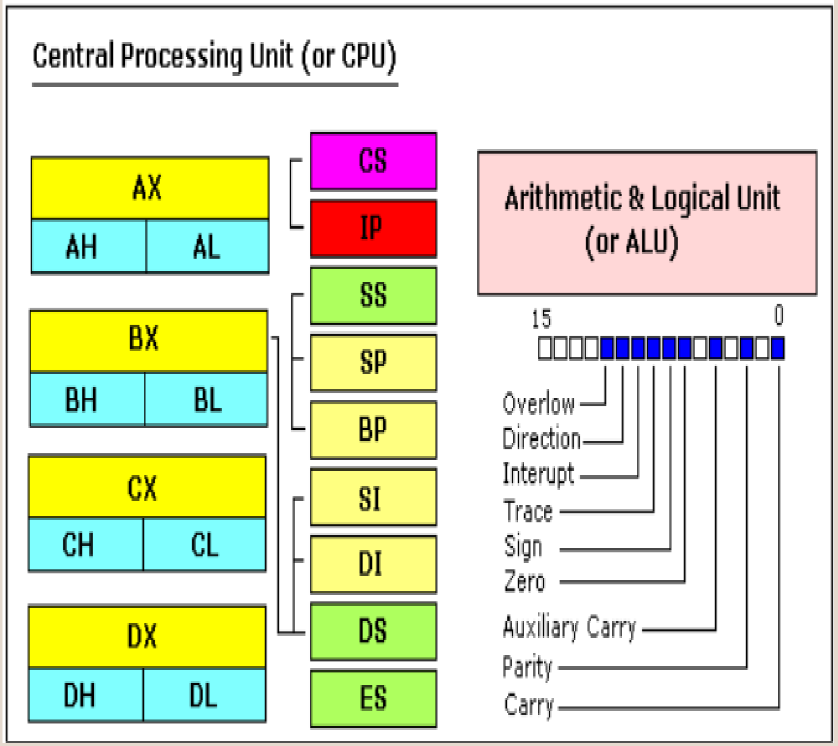
Segment Register –
Although it is possible to store any data in the segment registers, this is never a good idea. The segment registers have a very special purpose - pointing at accessible blocks of memory.

- CS - points at the segment containing the current program.
- DS - generally points at segment where variables are defined.
- ES - extra segment register, it's up to a coder to define its usage.
- SS - points at the segment containing the stack.

Note : By default, BX, SI and DI registers work with DS register
BP and SP work with SS register.

Overview

What's There Inside?



Segment Register –
Although it is possible to store any data in the segment registers, this is never a good idea. The segment registers have a very special purpose - pointing at accessible blocks of memory.

- CS - points at the segment containing the current program.
- DS - generally points at segment where variables are defined.
- ES - extra segment register, it's up to a coder to define its usage.
- SS - points at the segment containing the stack.

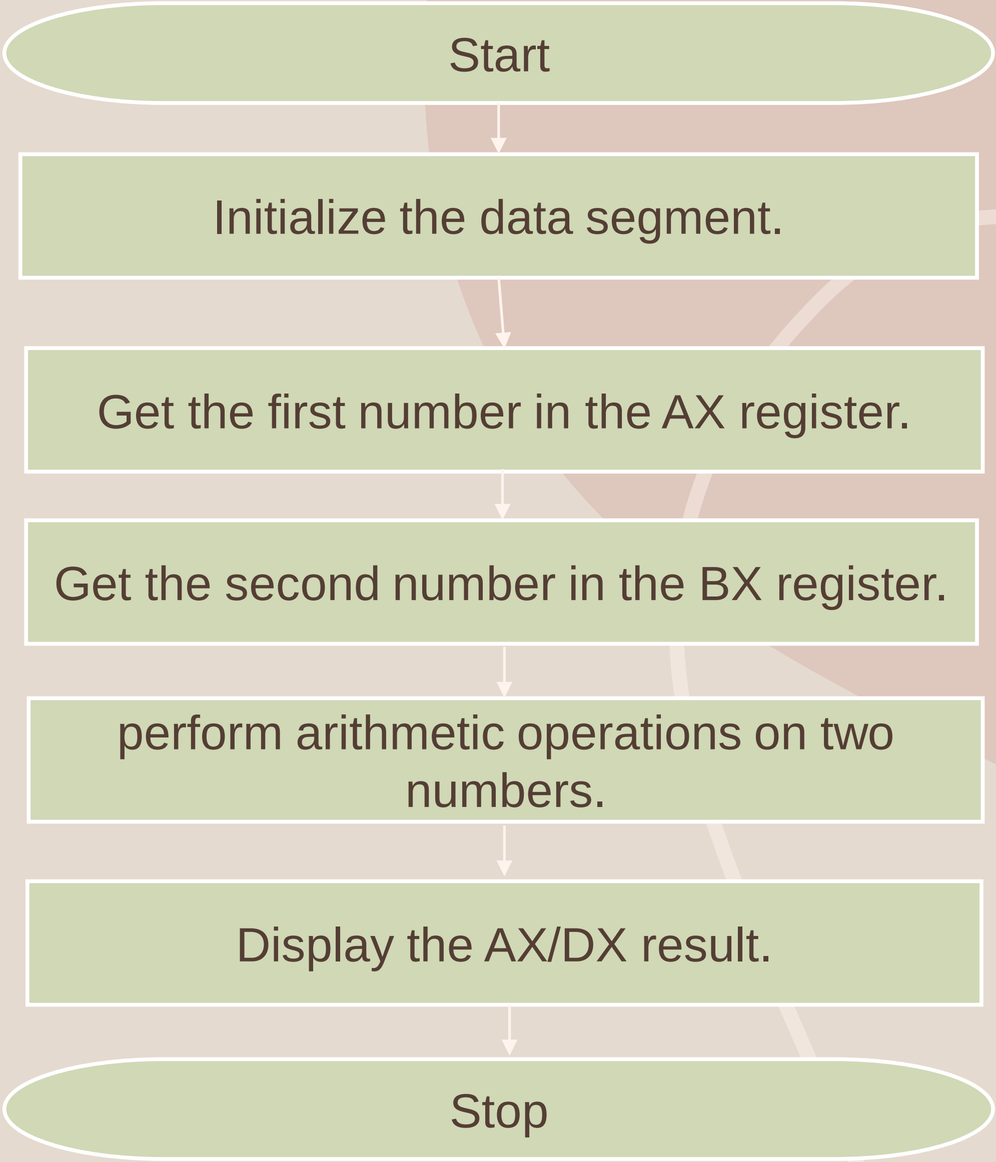
Note : By default, BX, SI and DI registers work with DS register
BP and SP work with SS register.

Assembly Language Programming

Code : ALP to add two 8-bit numbers

Flowchart -

```
DATA SEGMENT
    A DB 02H
    B DB 05H
    C DB ?
DATA ENDS

CODE SEGMENT
    ASSUME CS:CODE, DS:DATA
    START:
    MOV AX, DATA
    MOV DS, AX
    MOV AL, A
    MOV BL, B
    ADD AL, BL
    MOV C, AL
    INT 03H
CODE ENDS
END START
```



Start

Initialize the data segment.

Get the first number in the AX register.

Get the second number in the BX register.

perform arithmetic operations on two numbers.

Display the AX/DX result.

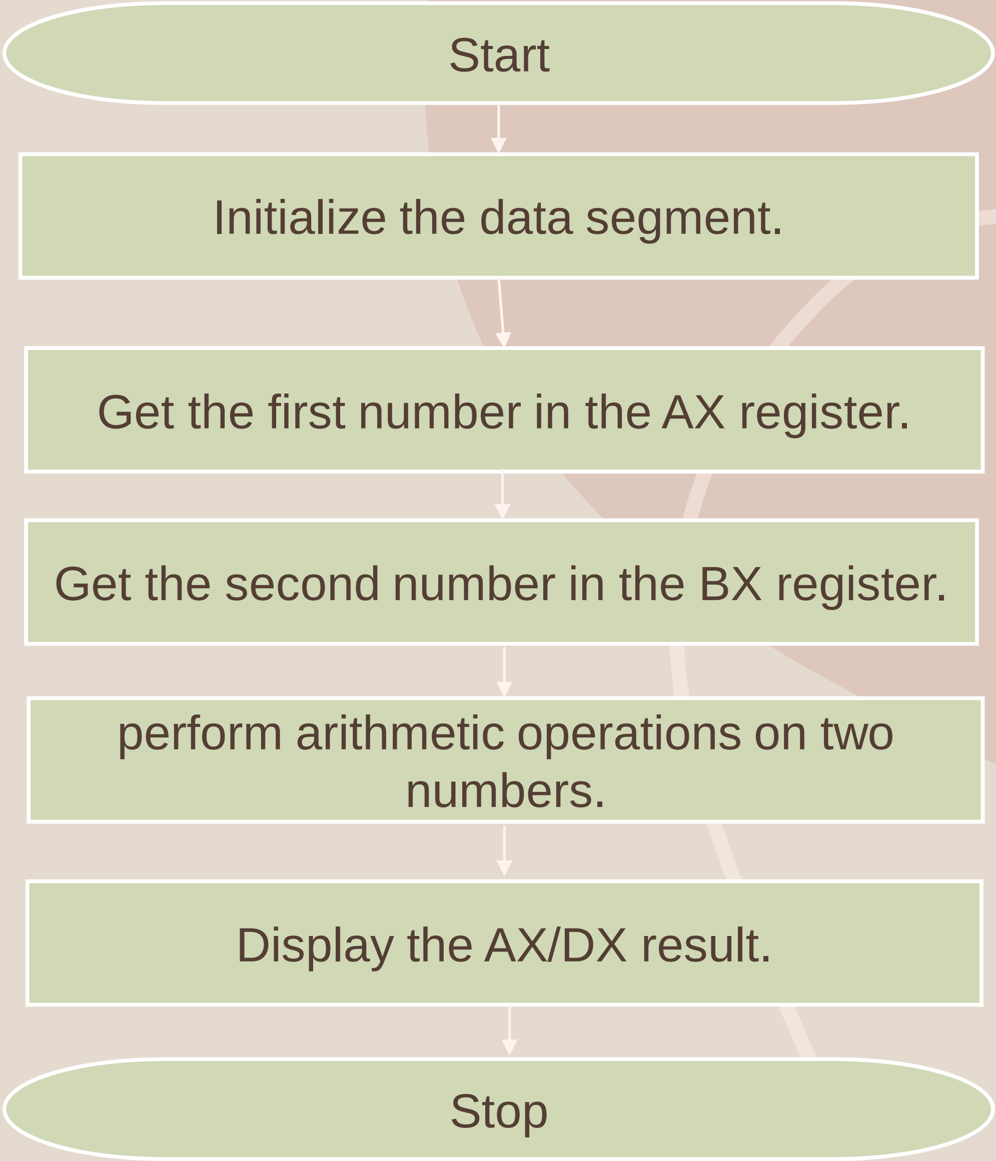
Stop

Assembly Language Programming

Code : ALP to add two 16-bit numbers

Flowchart -

```
DATA SEGMENT
   A DW 0202H
   B DW 0408H
   C 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 C, AX
   INT 03H
CODE ENDS
END START
```



Start

Initialize the data segment.

Get the first number in the AX register.

Get the second number in the BX register.

perform arithmetic operations on two numbers.

Display the AX/DX result.

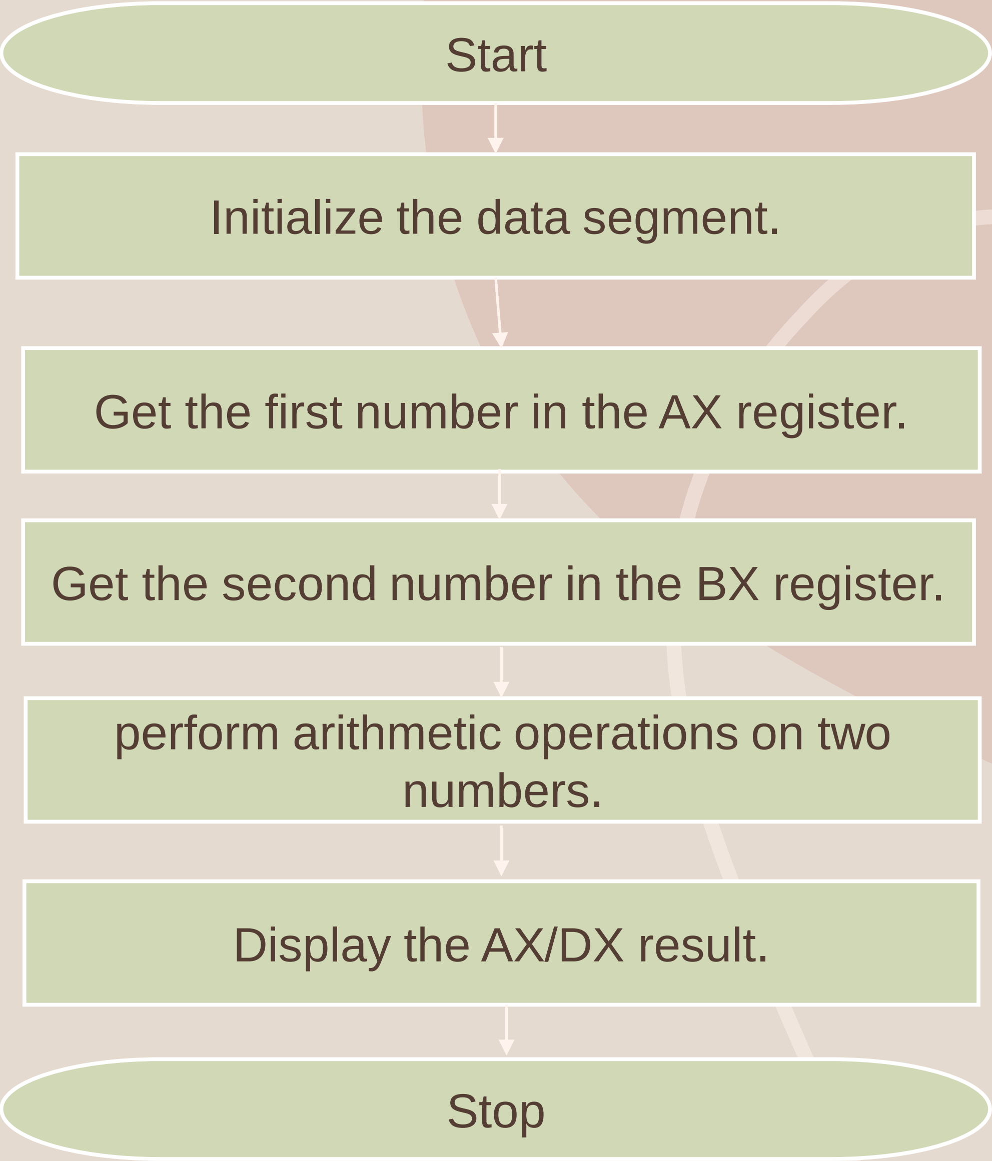
Stop

Assembly Language Programming

Code : ALP to subtract two 8-bit numbers

Flowchart -

```
DATA SEGMENT
   A DB 08H
   B DB 03H
   C DW ?
DATA ENDS

CODE SEGMENT
   ASSUME CS:CODE, DS:DATA
   START:
   MOV AX, DATA
   MOV DS, AX
   MOV AL, A
   MOV BL, B
   SUB AL, BL
   MOV C, AX
   INT 03H
CODE ENDS
END START
```



Start

Initialize the data segment.

Get the first number in the AX register.

Get the second number in the BX register.

perform arithmetic operations on two numbers.

Display the AX/DX result.

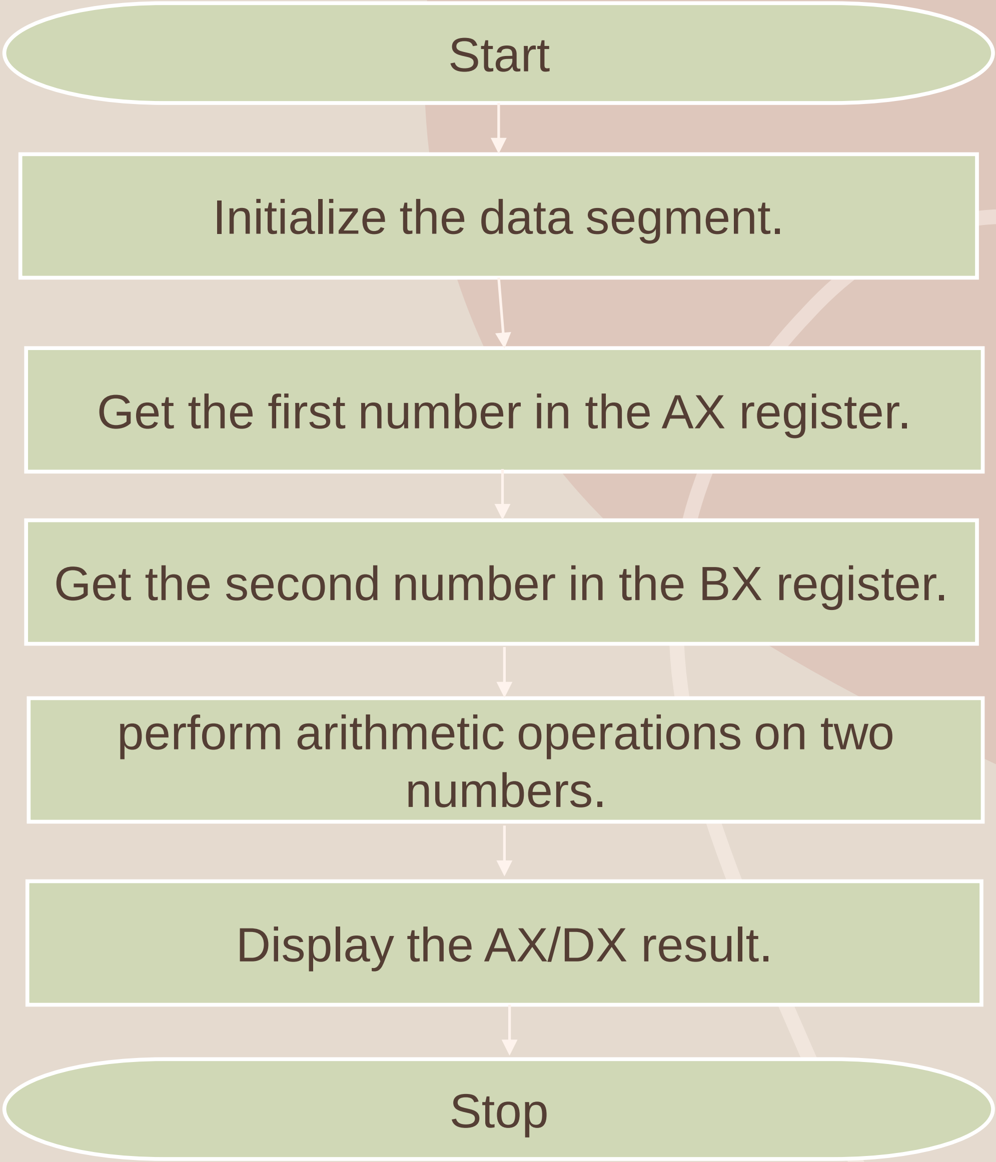
Stop

Assembly Language Programming

Code : ALP to subtract two 16-bit numbers

Flowchart -

```
DATA SEGMENT
   A DW 9A88H
   B DW 8765H
   C DW ?
DATA ENDS

CODE SEGMENT
   ASSUME CS:CODE, DS:DATA
   START:
   MOV AX, DATA
   MOV DS, AX
   MOV AX, A
   MOV BX, B
   SUB AX, BX
   MOV C, AX
   INT 03H
CODE ENDS
END START
```



Start

Initialize the data segment.

Get the first number in the AX register.

Get the second number in the BX register.

perform arithmetic operations on two numbers.

Display the AX/DX result.

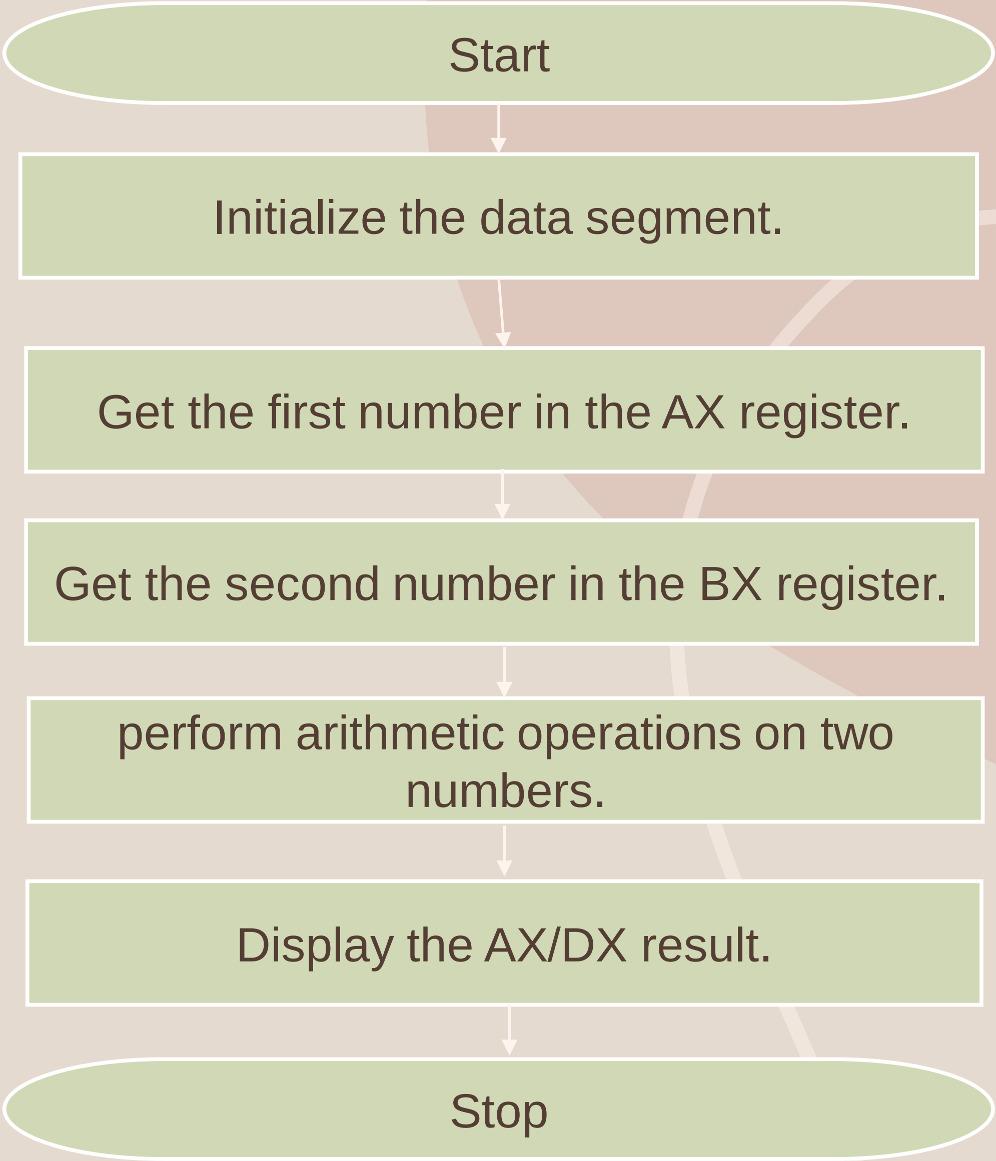
Stop

Assembly Language Programming

Code : ALP to multiply two 8-bit numbers

Flowchart -

```
DATA SEGMENT
   A DB 08H
   B DB 03H
   C DW ?
DATA ENDS

CODE SEGMENT
   ASSUME CS:CODE, DS:DATA
   START:
   MOV AX, DATA
   MOV DS, AX
   MOV AX, 0000H
   MOV BX, 0000H
   MOV AL, A
   MOV BL, B
   MUL BL
   MOV C, AX
   INT 03H
CODE ENDS
END START
```

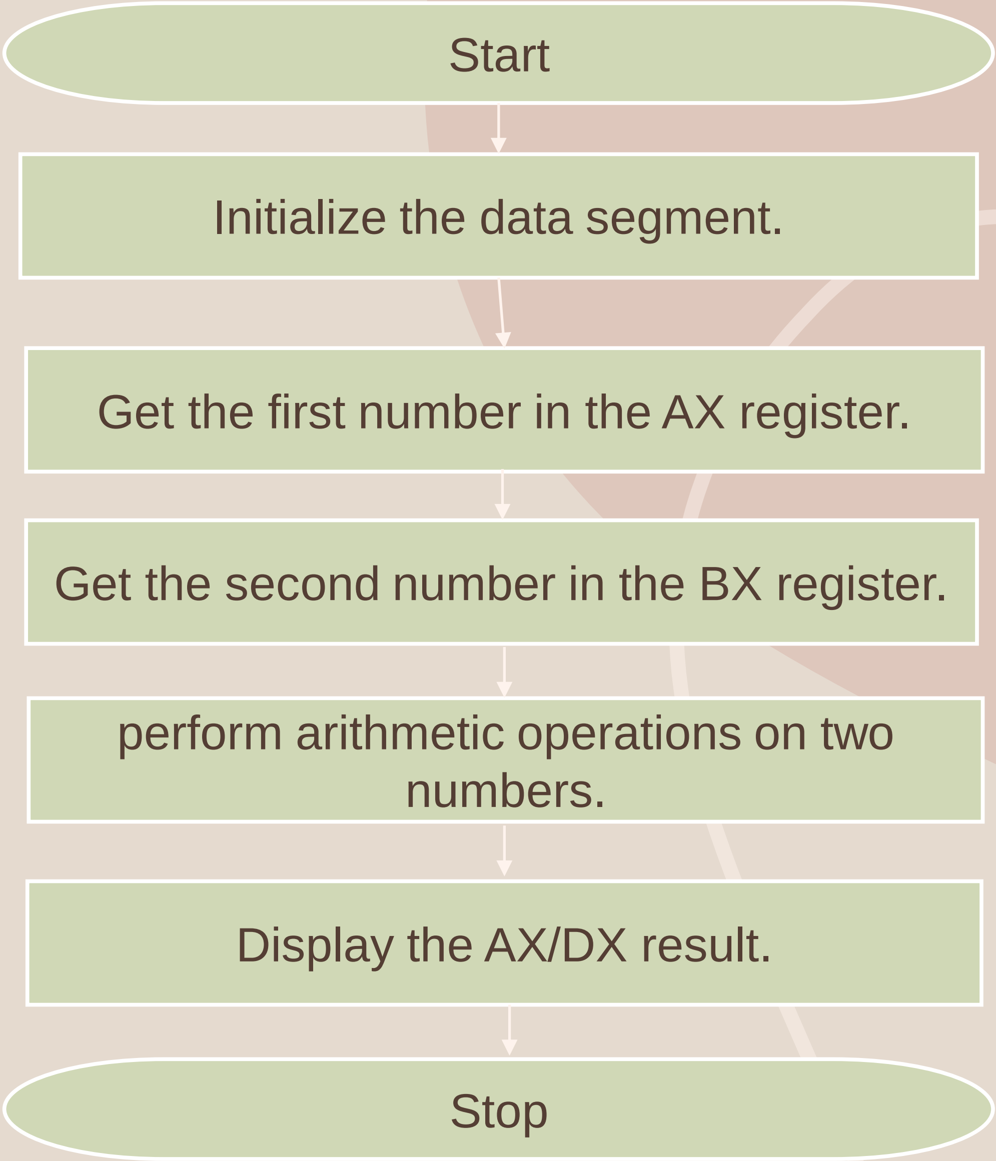

Start
Initialize the data segment.
Get the first number in the AX register.
Get the second number in the BX register.
perform arithmetic operations on two numbers.
Display the AX/DX result.
Stop

Assembly Language Programming

Code : ALP to multiply two 16-bit numbers

Flowchart -

```
DATA SEGMENT
A DW 5555H
B DW  1001H
C DD ?
DATA ENDS

CODE SEGMENT
ASSUME DS:DATA, CS:CODE
START:
MOV AX,DATA
MOV DS,AX
 MOV AX, 0000H
 MOV BX, 0000H
MOV AX,A
MOV BX,B
MUL BX
MOV WORD PTR C, AX
MOV WORD PTR C+2, DX
INT 3
CODE ENDS
END START
```



Start

Initialize the data segment.

Get the first number in the AX register.

Get the second number in the BX register.

perform arithmetic operations on two numbers.

Display the AX/DX result.

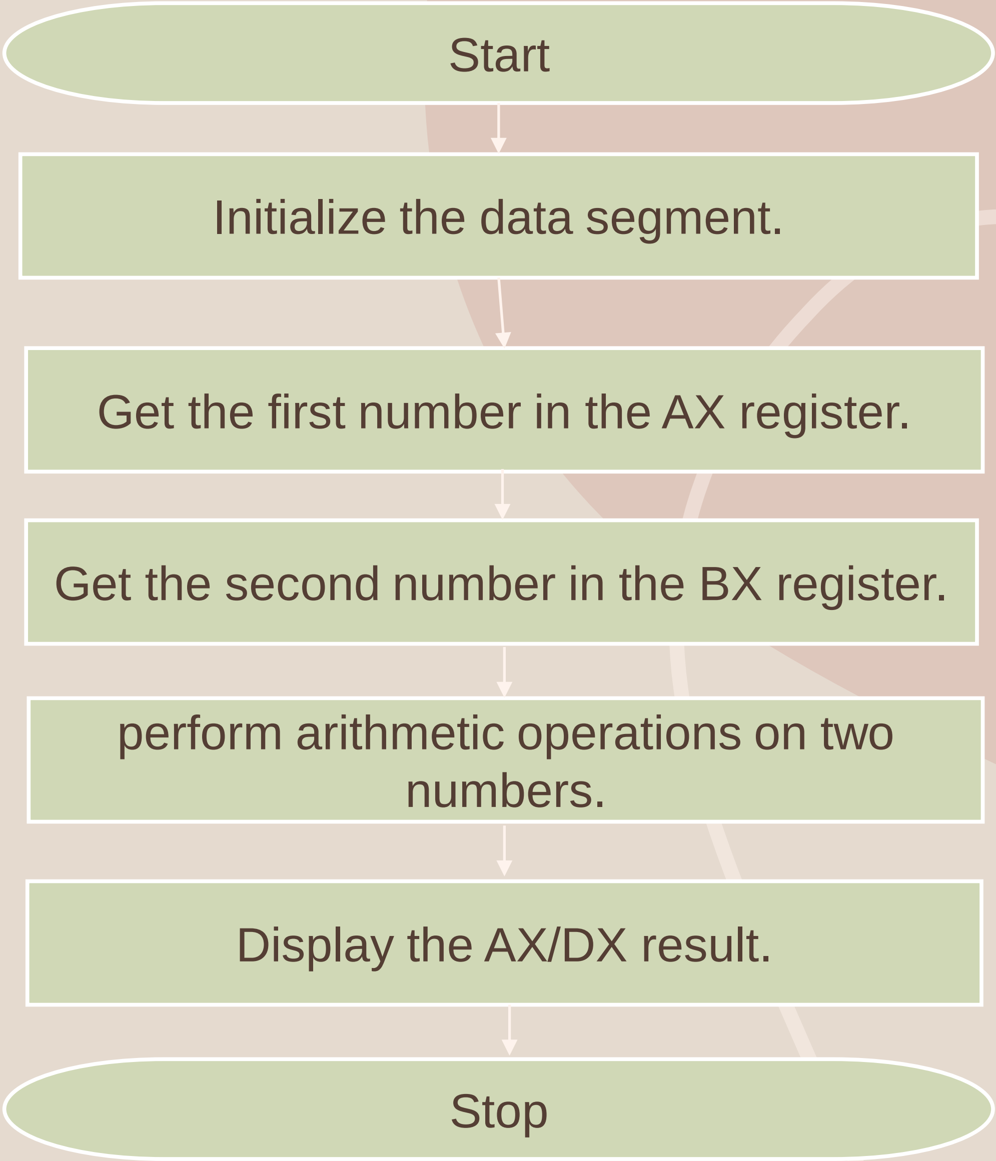
Stop

Assembly Language Programming

Code : ALP to divide two 8-bit numbers

Flowchart -

```
DATA SEGMENT
   A DB 28H
   B DB 02H
   C DW ?
DATA ENDS

CODE SEGMENT
   ASSUME CS:CODE, DS:DATA
   START:
   MOV AX, DATA
   MOV DS, AX
   MOV AX, 0000H
   MOV BX, 0000H
   MOV AL, A
   MOV BL, B
   DIV BL
   MOV C, AX
   INT 03H
CODE ENDS
END START
```



Start

Initialize the data segment.

Get the first number in the AX register.

Get the second number in the BX register.

perform arithmetic operations on two numbers.

Display the AX/DX result.

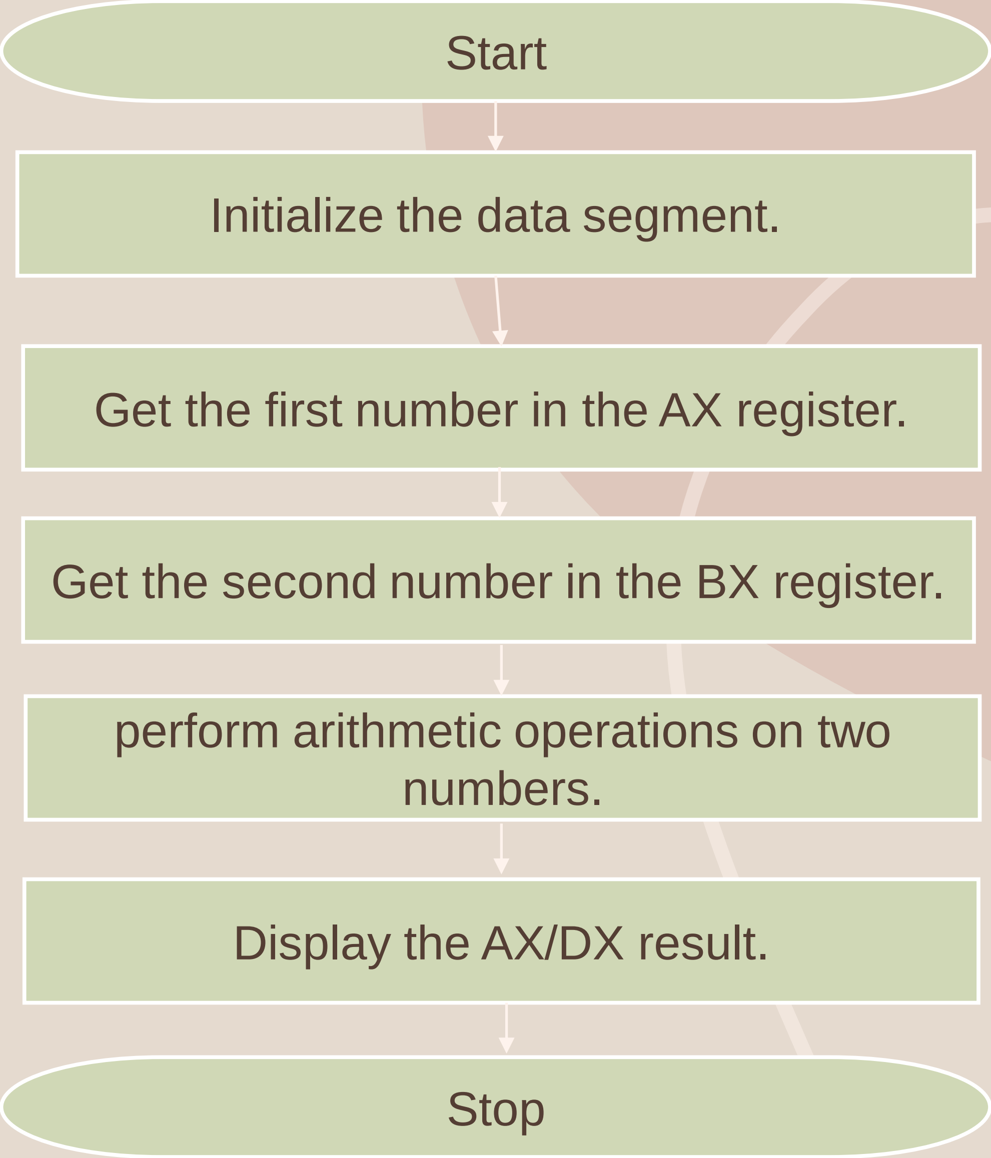
Stop

Assembly Language Programming

Code : ALP to divide two 16-bit numbers

Flowchart -

```
DATA SEGMENT
A DW 4444H
B DW  2002H
C DW ?
DATA ENDS

CODE SEGMENT
ASSUME DS:DATA, CS:CODE
START:
MOV AX,DATA
MOV DS,AX
 MOV AX, 0000H
 MOV BX, 0000H
MOV AX,A
MOV BX,B
DIV BX
MOV C, AX
INT 3
CODE ENDS
END START
```



Start

Initialize the data segment.

Get the first number in the AX register.

Get the second number in the BX register.

perform arithmetic operations on two numbers.

Display the AX/DX result.

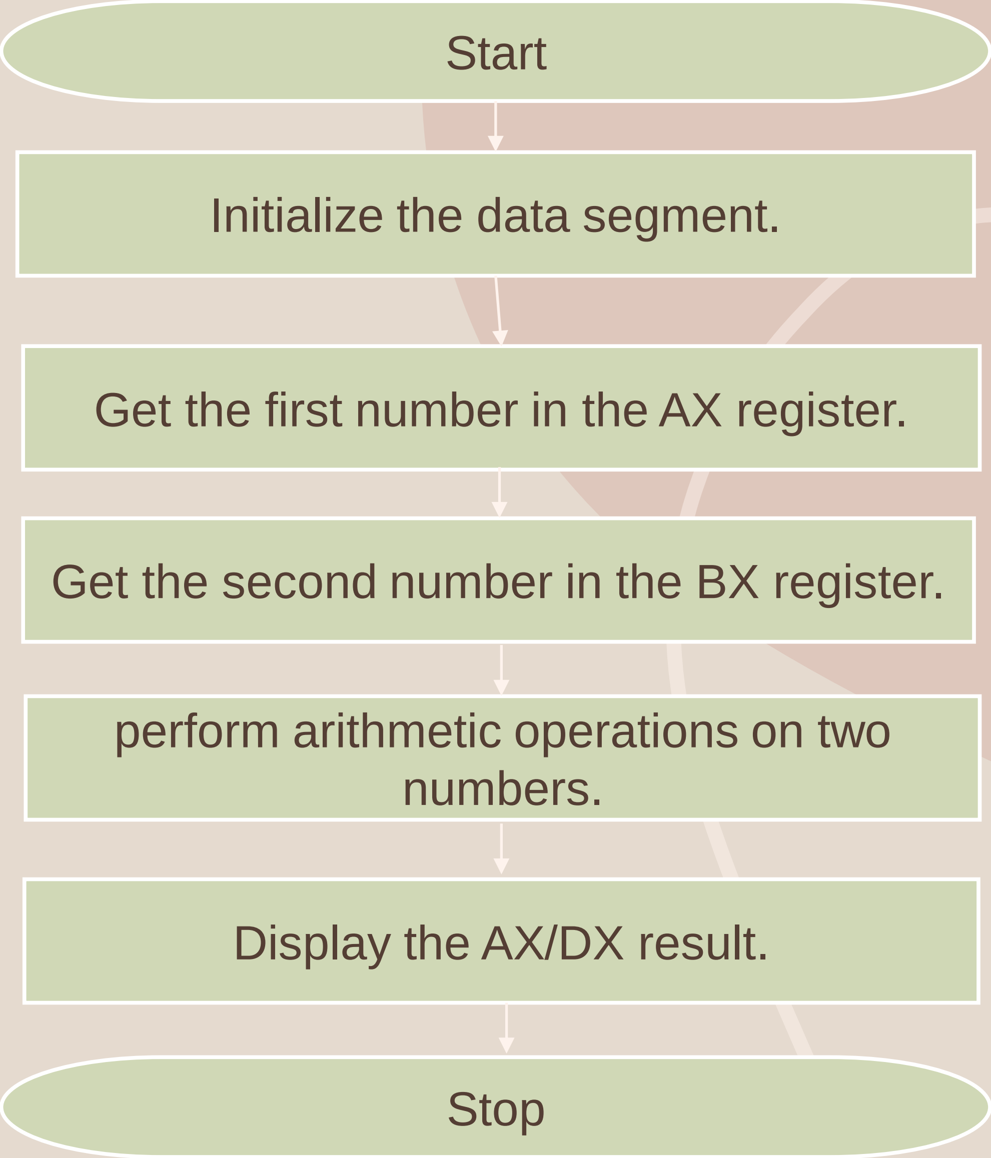
Stop

Assembly Language Programming

Code : ALP for signed multiplication of two 8-bit numbers

Flowchart -

```
DATA SEGMENT
   A DB 0F2H
   B DB 09H
   C DW ?
DATA ENDS

CODE SEGMENT
   ASSUME CS:CODE, DS:DATA
   START:
   MOV AX, DATA
   MOV DS, AX
   MOV AX, 0000H
   MOV BX, 0000H
   MOV AL, A
   MOV BL, B
   IMUL BL
   MOV C, AX
   INT 03H
CODE ENDS
END START
```



Start

Initialize the data segment.

Get the first number in the AX register.

Get the second number in the BX register.

perform arithmetic operations on two numbers.

Display the AX/DX result.

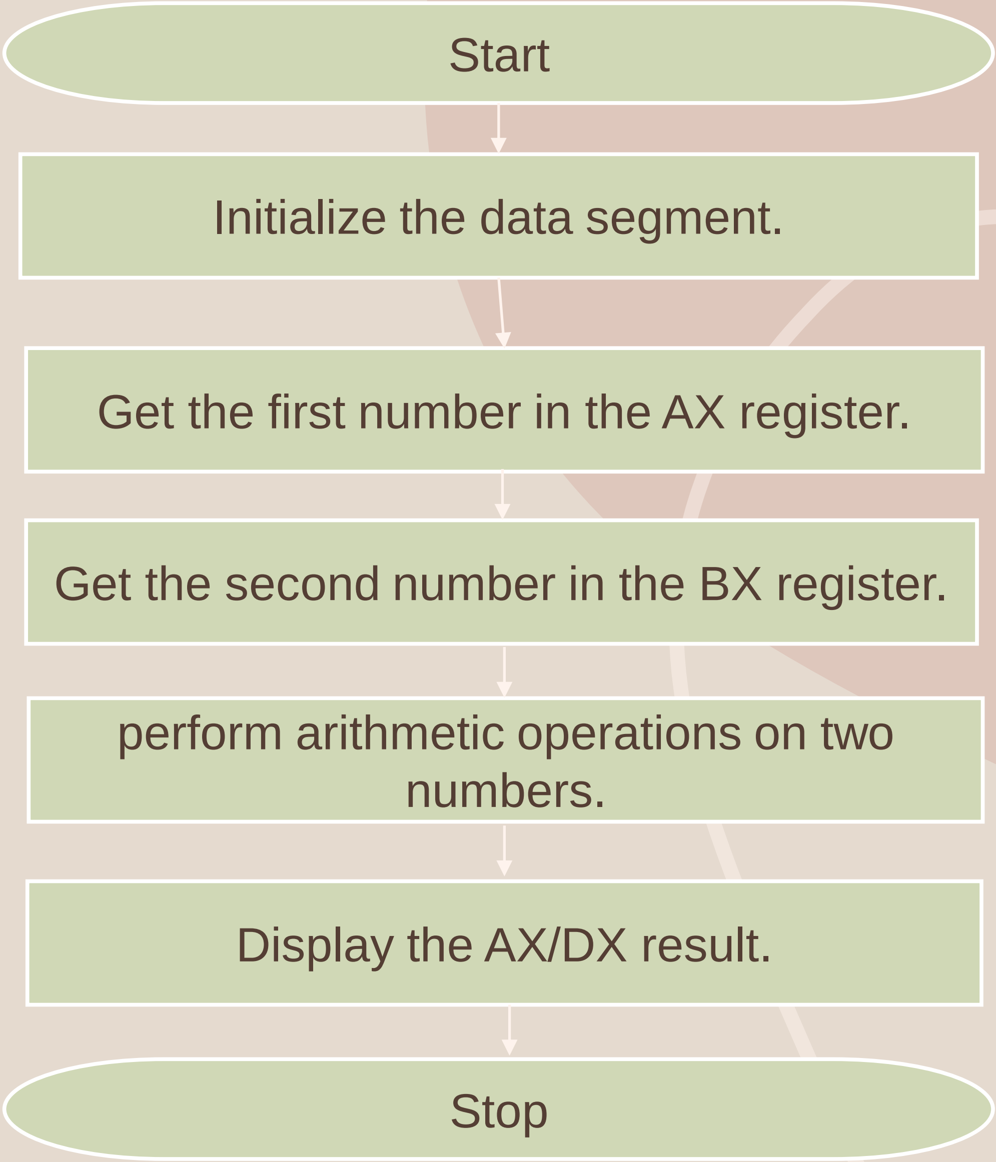
Stop

Assembly Language Programming

Code : ALP for signed division of two 8-bit numbers

Flowchart -

```
DATA SEGMENT
   A DB 0F2H
   B DB 09H
   C DW ?
DATA ENDS

CODE SEGMENT
   ASSUME CS:CODE, DS:DATA
   START:
   MOV AX, DATA
   MOV DS, AX
   MOV AX, 0000H
   MOV BX, 0000H
   MOV AL, A
   MOV BL, B
   IDIV BL
   MOV C, AX
   INT 03H
CODE ENDS
END START
```

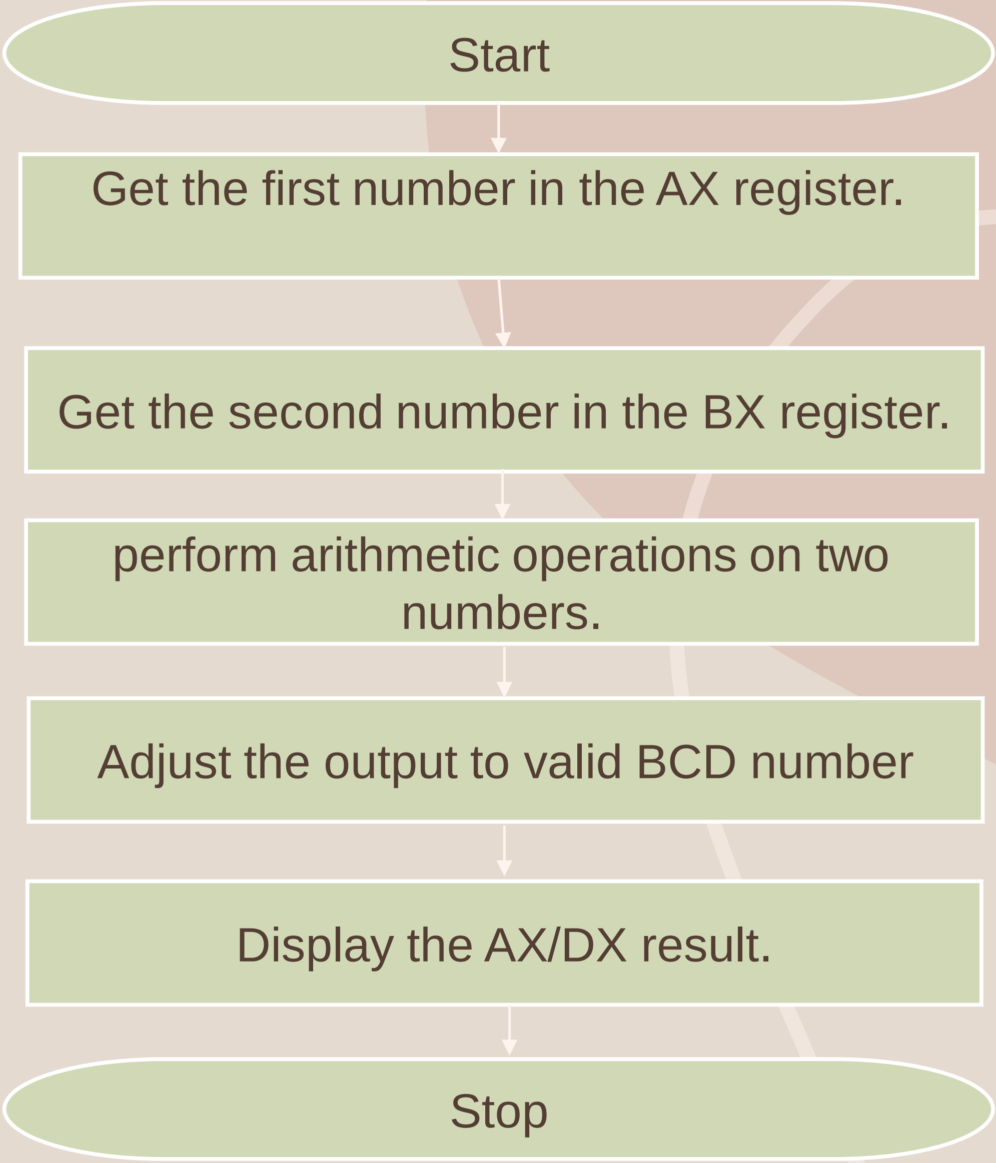

Start
Initialize the data segment.
Get the first number in the AX register.
Get the second number in the BX register.
perform arithmetic operations on two numbers.
Display the AX/DX result.
Stop

Assembly Language Programming

Flowchart -

Code :
ALP to add two 8-bit BCD numbers

```
DATA SEGMENT
   A DB 80H
   B DB 26H
   RES_LSB DB 0H
   RES_MSB DB 0H
DATA ENDS

CODE SEGMENT
   ASSUME CS:CODE, DS:DATA
   START:
   MOV AX, DATA
   MOV DS, AX
   MOV AL, A
   MOV BL, B
   ADD AL, BL
   DAA
   JNC NEXT
   INC RES_MSB
   NEXT:
   MOV RES_LSB , AL
   INT 03H
CODE ENDS
END START
```


Start

Get the first number in the AX register.

Get the second number in the BX register.

perform arithmetic operations on two numbers.

Adjust the output to valid BCD number

Display the AX/DX result.

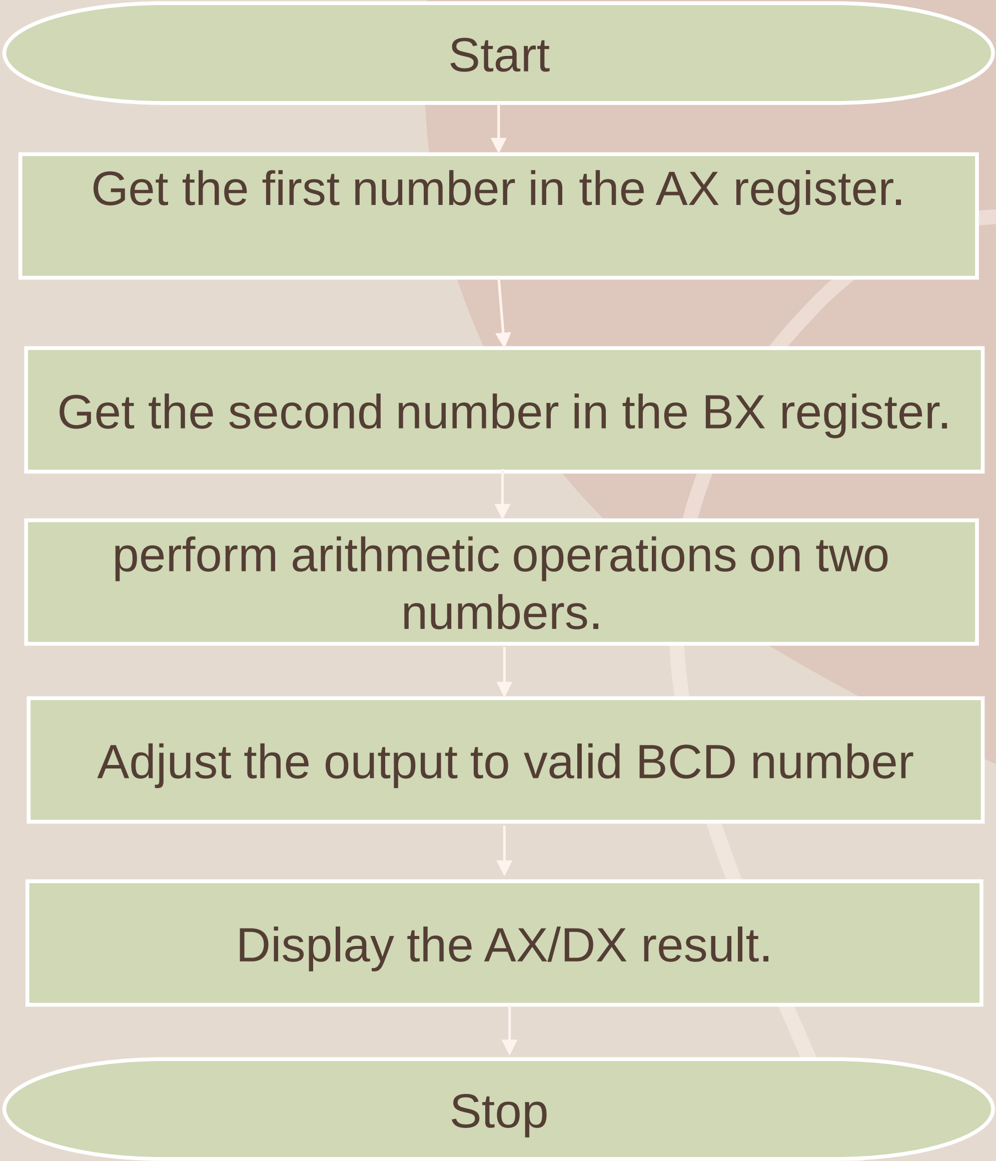
Stop

Assembly Language Programming

Flowchart -

Code :
ALP to subtract two 8-bit BCD numbers

```
DATA SEGMENT
   A DB 80H
   B DB 26H
   RES_LSB DB 0H
DATA ENDS

CODE SEGMENT
   ASSUME CS:CODE, DS:DATA
   START:
   MOV AX, DATA
   MOV DS, AX
   MOV AL, A
   MOV BL, B
   SUB AL, BL
   DAS
   MOV RES_LSB , AL
   INT 03H
CODE ENDS
END START
```



Start

Get the first number in the AX register.

Get the second number in the BX register.

perform arithmetic operations on two numbers.

Adjust the output to valid BCD number

Display the AX/DX result.

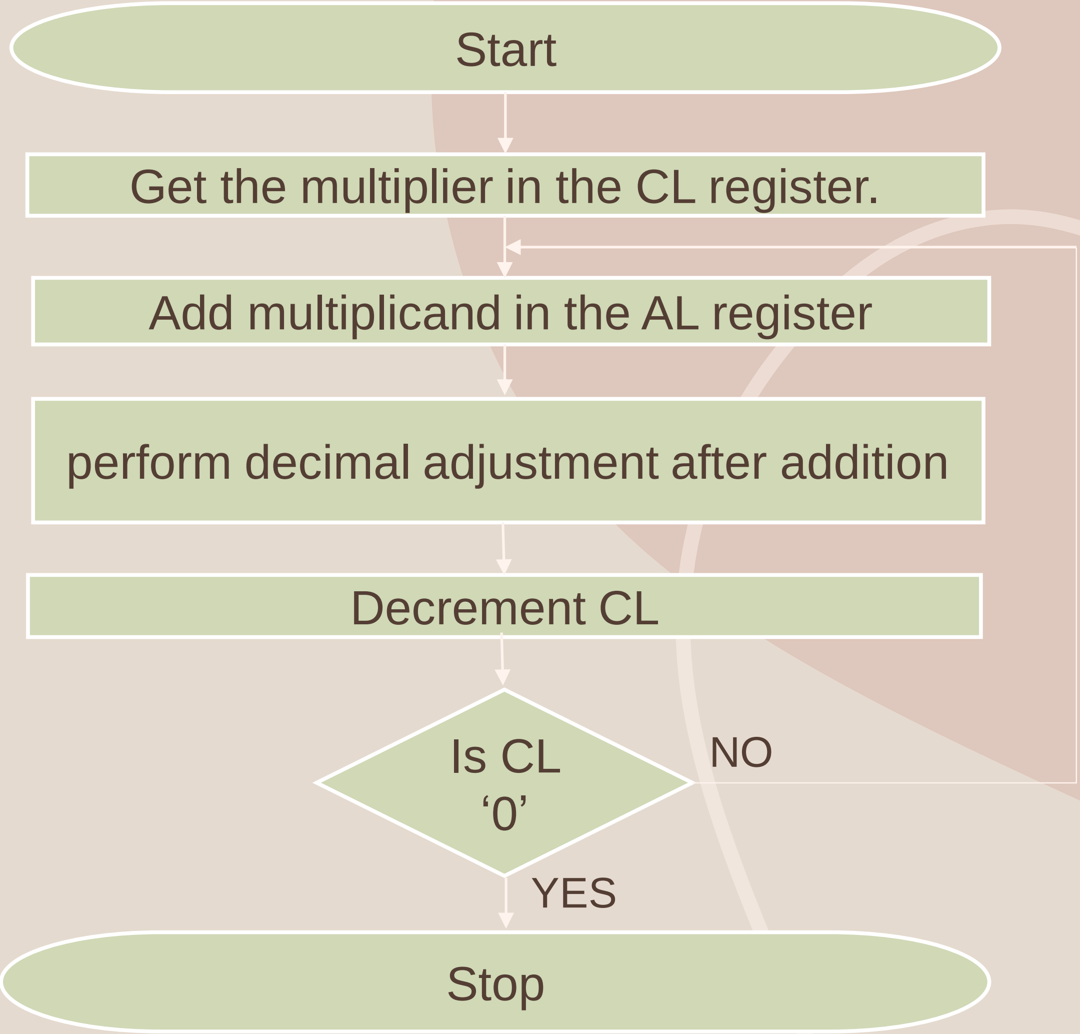
Stop

Assembly Language Programming

Flowchart -

Code :
ALP to multiply two 8-bit BCD numbers

```
DATA SEGMENT
   A DB 12H
   B DB 03H
DATA ENDS

CODE SEGMENT
   ASSUME CS:CODE, DS:DATA
   START:
   MOV AX, DATA
   MOV DS, AX
   MOV AX, 0000H
   MOV CL, B
   UP:
   ADD AL, A
   DAA
   DEC CL
   JNZ UP
   INT 03H
CODE ENDS
END START
```

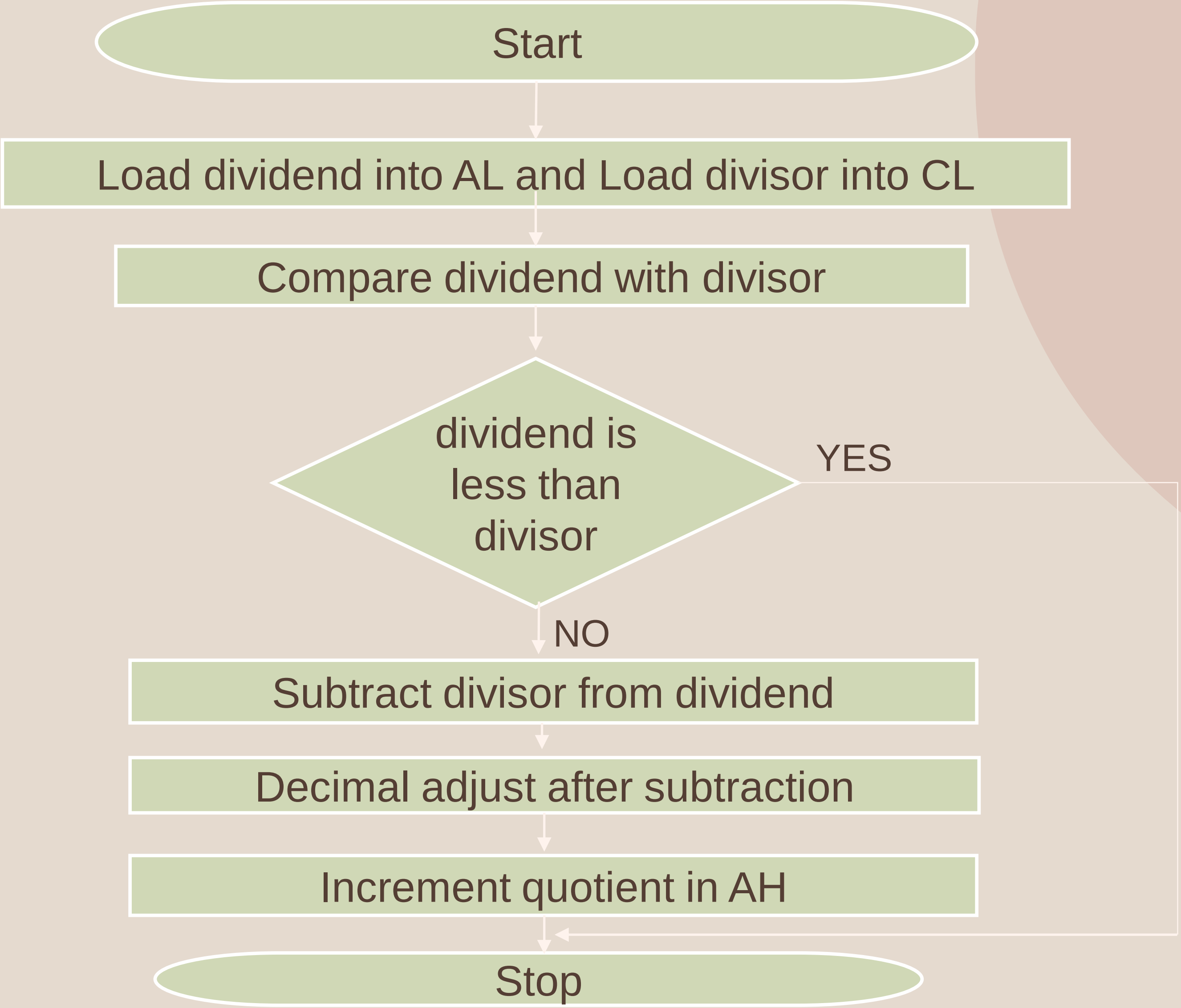


Start
Get the multiplier in the CL register.
Add multiplicand in the AL register
perform decimal adjustment after addition
Decrement CL
Is CL ‘0’
NO
YES
Stop

Assembly Language Programming

Flowchart -

Code :
ALP to divide two 8-bit BCD numbers



Start
Load dividend into AL and Load divisor into CL
Compare dividend with divisor
dividend is less than divisor
YES
NO
Subtract divisor from dividend
Decimal adjust after subtraction
Increment quotient in AH
Stop

Assembly Language Programming

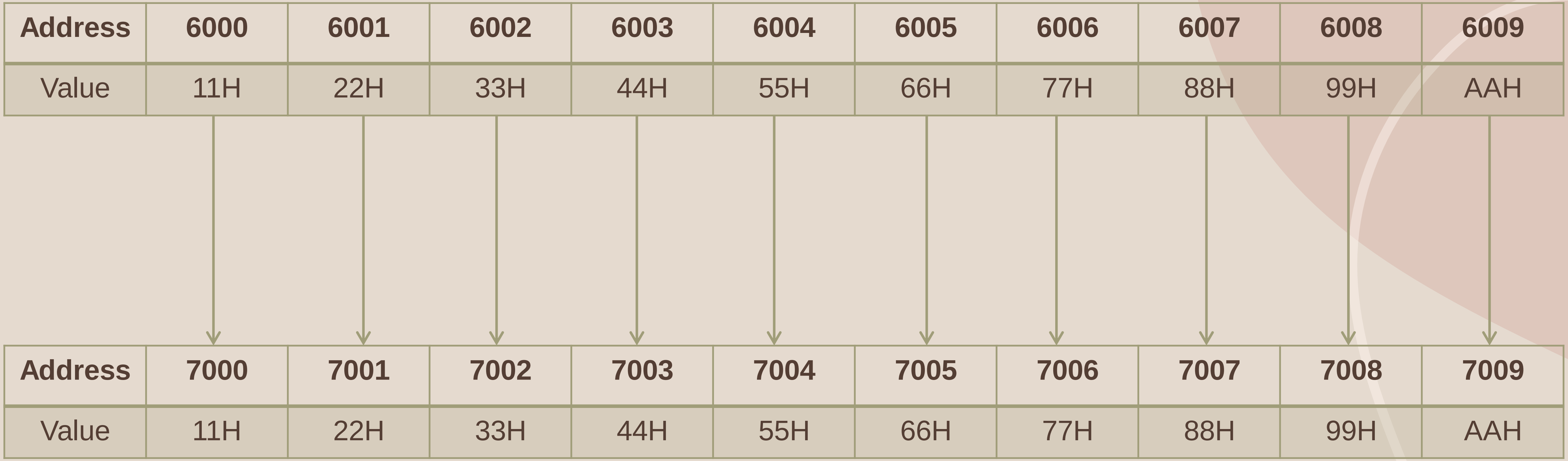
Code :
ALP to divide two 8-bit BCD numbers

```
DATA SEGMENT
   A DB 16H    ; Dividend (in BCD)
   B DB 03H    ; Divisor (in BCD)
DATA ENDS

CODE SEGMENT
   ASSUME CS:CODE, DS:DATA
START:
   MOV AX, DATA
   MOV DS, AX

   MOV AH, 00H
   MOV AL, A        ; Load dividend into AL

   MOV CL, B         ; Load divisor into CL
   MOV CH, 00H       ; Clear CH to avoid garbage data
```

```
DIV_LOOP:
   CMP AL, CL       ; Compare dividend with divisor
   JB DIV_END       ; If dividend is less than divisor, end division
   SUB AL, CL       ; Subtract divisor from dividend
   DAS              ; Decimal adjust after subtraction
   INC AH           ; Increment quotient in AH
   JMP DIV_LOOP    ; Repeat division loop

DIV_END:
   INT 03H          ; Terminate program
CODE ENDS
END START
```

Block Transfer Data

Source Block



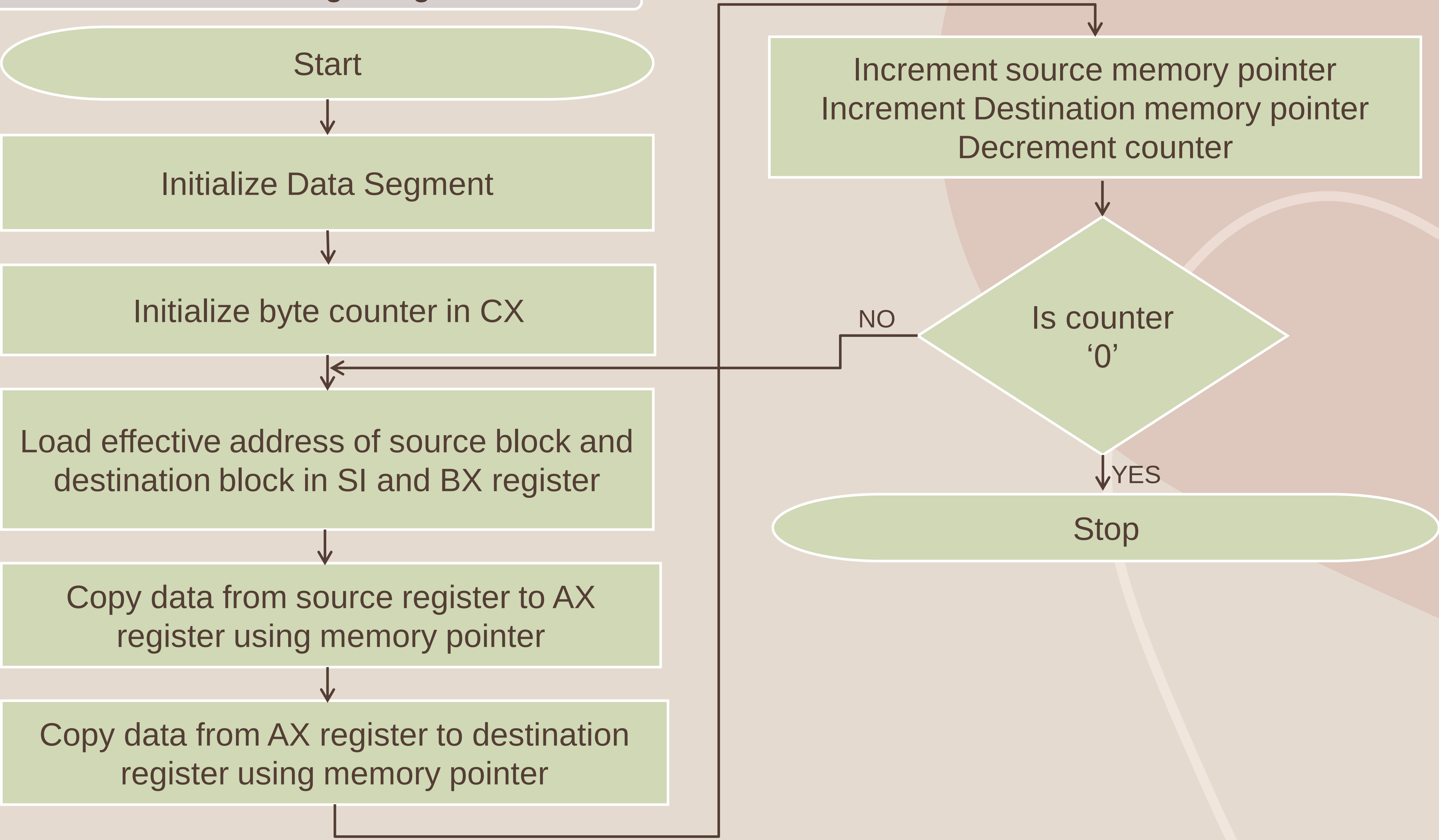
Address	6000	6001	6002	6003	6004	6005	6006	6007	6008	6009
Value	11H	22H	33H	44H	55H	66H	77H	88H	99H	AAH

Address	7000	7001	7002	7003	7004	7005	7006	7007	7008	7009
Value	11H	22H	33H	44H	55H	66H	77H	88H	99H	AAH

Destination Block

Code : Block transfer data without using string instruction

Flowchart -



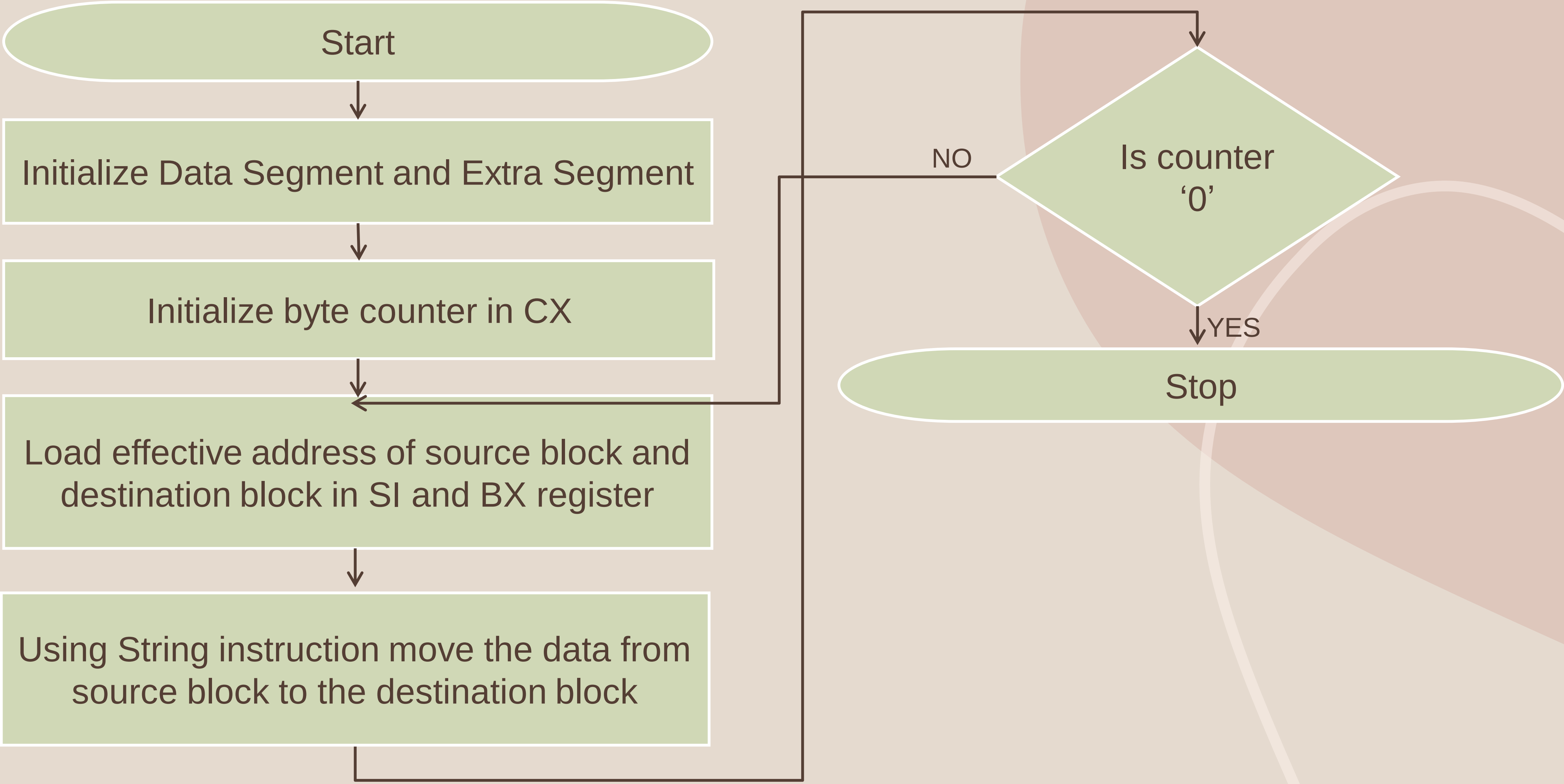
Start
Initialize Data Segment
Initialize byte counter in CX
Load effective address of source block and destination block in SI and BX register
Copy data from source register to AX register using memory pointer
Copy data from AX register to destination register using memory pointer
Increment source memory pointer
Increment Destination memory pointer
Decrement counter
Is counter ‘0’
NO
YES
Stop

Code : Block transfer data without using string instruction

```
DATA SEGMENT
    SRC_BLK DB 11H, 22H, 33H, 44H, 55H
    DST_BLK DB 5 DUP(0)
DATA ENDS

CODE SEGMENT
    ASSUME CS:CODE, DS:DATA
    START:
    MOV AX, DATA
    MOV DS, AX
    MOV CX, 05H
    LEA SI, SRC_BLK
    LEA BX, DST_BLK

    UP:
    MOV AX, [SI]
    MOV [BX], AX
    INC SI
    INC BX
    DEC CX
    JNZ UP
    INT 03H
CODE ENDS
END START
```

Code : Block transfer data using string instruction

Flowchart -



Start
Initialize Data Segment and Extra Segment
Initialize byte counter in CX
Load effective address of source block and destination block in SI and BX register
Using String instruction move the data from source block to the destination block
Is counter ‘0’
NO
YES
Stop

Code : Block transfer data using string instruction

```
DATA SEGMENT
    SRC_BLK DB 11H, 22H, 33H, 44H, 55H, 66H, 77H, 88H, 99H, 0AAH
    DST_BLK DB 0AH DUP(0)
DATA ENDS

CODE SEGMENT
    ASSUME CS:CODE, DS:DATA
    START:
    MOV AX, DATA
    MOV DS, AX
    MOV ES, AX
    MOV CX, 0AH
    LEA SI, SRC_BLK   ; MOV SI, OFFSET SRC_BLK
    LEA DI, DST_BLK

    UP:
    MOVSB             ; MOVSW TRANSFER TWO BYTES
    LOOP UP
    INT 03H
CODE ENDS
END START
```

Code : Find sum of series of Hexadecimal Numbers

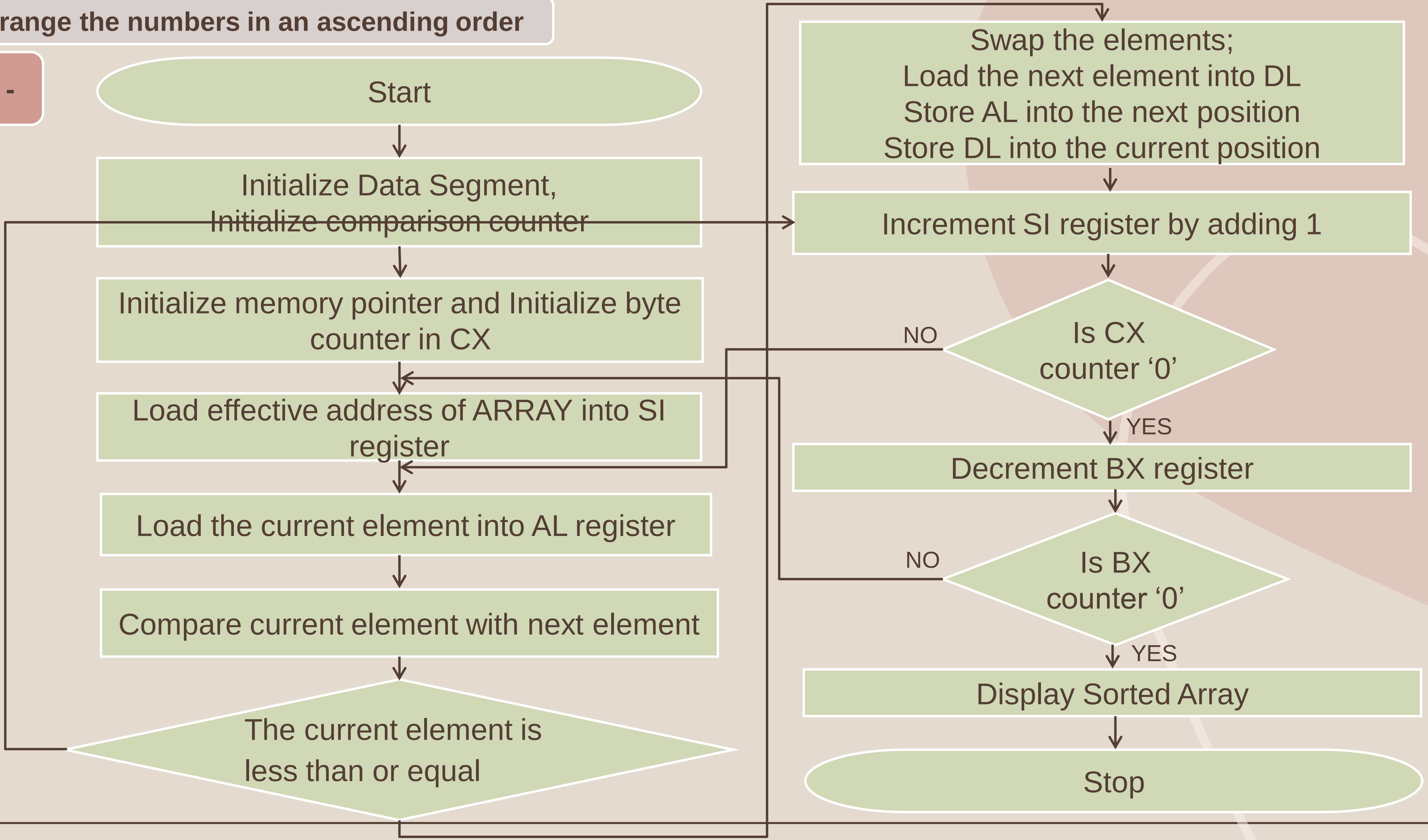
```
DATA SEGMENT
    SERIES DB 11H, 02H, 03H, 01H, 00H
    SUM DB 00H
DATA ENDS

CODE SEGMENT
    ASSUME CS:CODE, DS:DATA
    START:
    MOV AX, DATA
    MOV DS, AX
    MOV AX, 0000H
    MOV CX, 04H
    LEA BX, SERIES

    REPT:
    ADD AL, [BX]
    INC BX
    DEC CX
    JNZ REPT
    MOV SUM, AL
    MOV DL, SUM
    INT 03H
CODE ENDS
END START
```

Code : Find sum of series of BCD Numbers

```
DATA SEGMENT
    SERIES DB 11H, 22H, 22H, 11H, 55H
    SUM DB 00H
DATA ENDS

CODE SEGMENT
    ASSUME CS:CODE, DS:DATA
START:
    MOV AX, DATA
    MOV DS, AX
    MOV AX, 0000H
    MOV CX, 05H
    LEA BX, SERIES

    REPT:
    ADD AL, [BX]
    DAA
    MOV SUM, AL
    INC BX
    DEC CX
    JNZ REPT
    MOV DL, SUM
    INT 03H
CODE ENDS
END START
```

Code : Arrange the numbers in an ascending order

Flowchart -



Start

Initialize Data Segment, Initialize comparison counter

Initialize memory pointer and Initialize byte counter in CX

Load effective address of ARRAY into SI register

Load the current element into AL register

Compare current element with next element

The current element is less than or equal

Swap the elements;
Load the next element into DL
Store AL into the next position
Store DL into the current position

Increment SI register by adding 1

Is CX counter ‘0’

NO

YES

Decrement BX register

Is BX counter ‘0’

NO

YES

Display Sorted Array

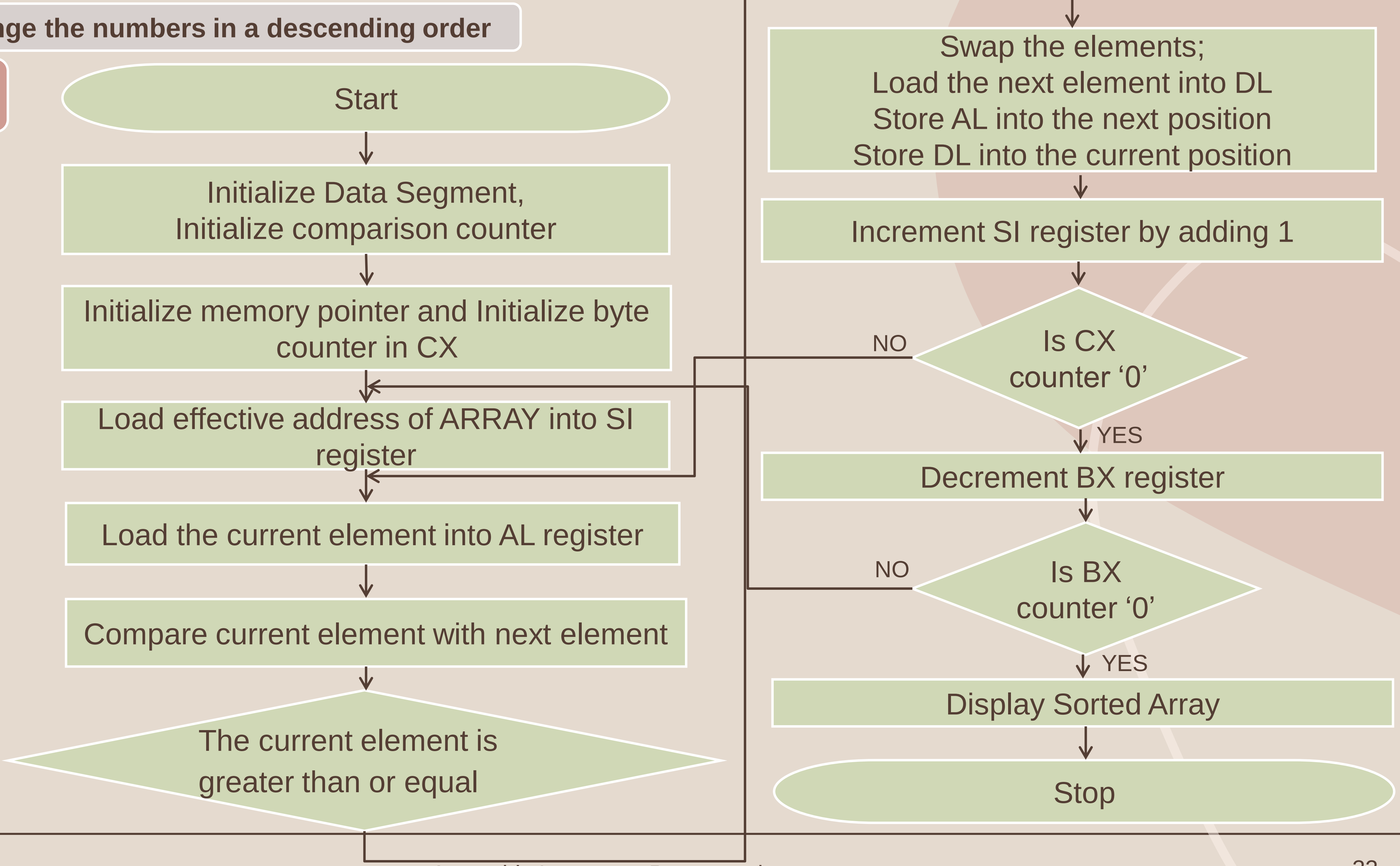
Stop

```
DATA SEGMENT
   ARRAY DB 12H, 07H, 15H, 23H, 02H
DATA ENDS

CODE SEGMENT
   ASSUME CS: CODE, DS: DATA
START:
   MOV AX, DATA
   MOV DS, AX
   MOV BX, 05H

   TOP:
   LEA SI, ARRAY
   MOV CX, 04H

   UP:
   MOV AL, [SI]      ; Load the current element into AL
   CMP AL, [SI+1]    ; Compare with the next element
   JLE DN            ; Jump if less than or equal, skip swap

   ; Swap the elements
   MOV DL, [SI+1]    ; Load the next element into DL
   MOV [SI+1], AL    ; Store AL into the next position
   MOV [SI], DL      ; Store DL into the current position
```

```
 DN:
   ADD SI, 01H
   LOOP UP
   DEC BX
   JNZ TOP

   ; Display the sorted array
   LEA SI, ARRAY
   MOV CX, 5

   DISPLAY_LOOP:
   MOV DL, [SI]    ; Load the value to be displayed
   INC SI          ; Move to the next position
   LOOP DISPLAY_LOOP ; Loop until all elements are displayed

   INT 03H         ; Display the character

CODE ENDS
END START
```

Code : Arrange the numbers in a descending order

Flowchart -



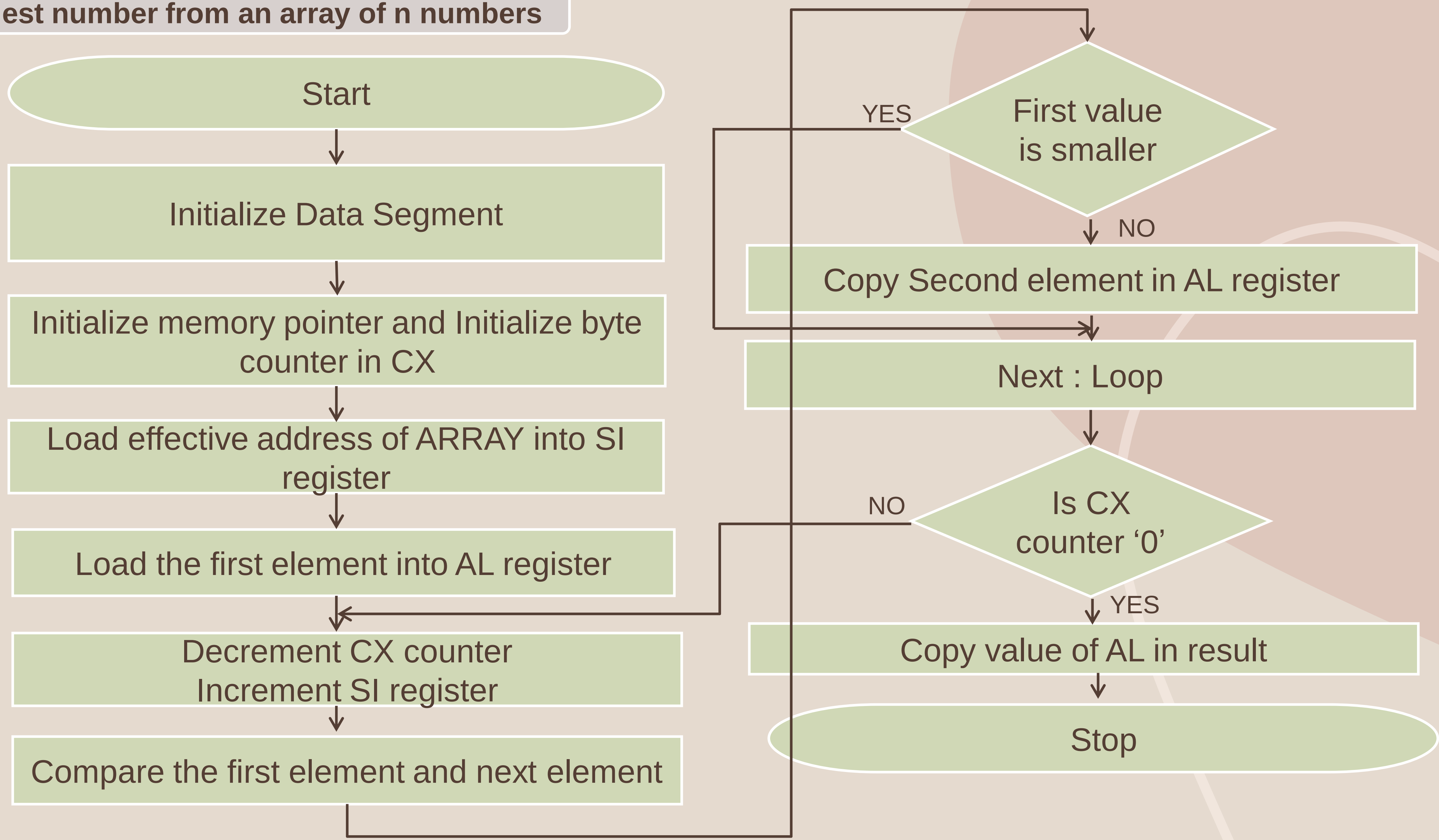
Start
Initialize Data Segment,
Initialize comparison counter
Initialize memory pointer and Initialize byte counter in CX
Load effective address of ARRAY into SI register
Load the current element into AL register
Compare current element with next element
The current element is greater than or equal
Swap the elements;
Load the next element into DL
Store AL into the next position
Store DL into the current position
Increment SI register by adding 1
Is CX counter ‘0’
NO
YES
Decrement BX register
Is BX counter ‘0’
NO
YES
Display Sorted Array
Stop

```
DATA SEGMENT
   ARRAY DB 12H, 07H, 15H, 23H, 02H
DATA ENDS

CODE SEGMENT
   ASSUME CS: CODE, DS: DATA
START:
   MOV AX, DATA
   MOV DS, AX
   MOV BX, 05H

   TOP:
   LEA SI, ARRAY
   MOV CX, 04H

   UP:
   MOV AL, [SI]      ; Load the current element into AL
   CMP AL, [SI+1]    ; Compare with the next element
   JGE DN            ; Jump if greater than or equal, skip swap

   ; Swap the elements
   MOV DL, [SI+1]    ; Load the next element into DL
   MOV [SI+1], AL    ; Store AL into the next position
   MOV [SI], DL      ; Store DL into the current position
```

```
 DN:
   ADD SI, 01H
   LOOP UP
   DEC BX
   JNZ TOP

   ; Display the sorted array
   LEA SI, ARRAY
   MOV CX, 5

   DISPLAY_LOOP:
   MOV DL, [SI]    ; Load the value to be displayed
   INC SI          ; Move to the next position
   LOOP DISPLAY_LOOP ; Loop until all elements are displayed

   INT 03H         ; Display the character

CODE ENDS
END START
```

Code : Find smallest number from an array of n numbers

Flowchart -

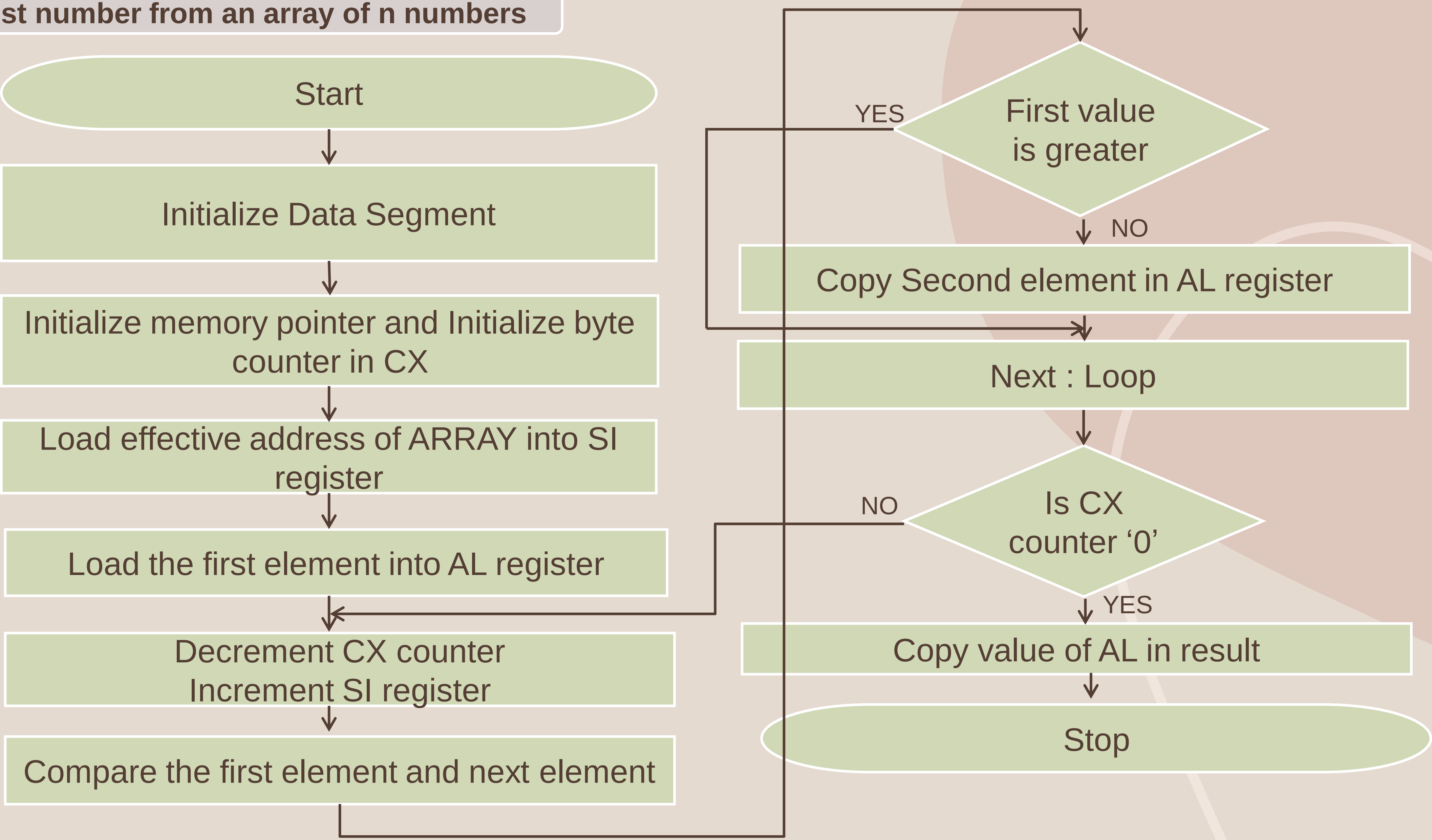

Start
Initialize Data Segment
Initialize memory pointer and Initialize byte counter in CX
Load effective address of ARRAY into SI register
Load the first element into AL register
Decrement CX counter
Increment SI register
Compare the first element and next element
First value is smaller
YES
NO
Copy Second element in AL register
Next : Loop
Is CX counter '0'
NO
YES
Copy value of AL in result
Stop

```
DATA SEGMENT
   ARRAY DB 12H, 07H, 25H, 4BH, 02H
   SMALL DB 00H
DATA ENDS

CODE SEGMENT
   ASSUME CS: CODE, DS: DATA
   START:

   MOV AX, DATA
   MOV DS, AX
   MOV AX, 0000H
   MOV CX, 05H
   LEA SI, ARRAY
   MOV AL, [SI]
   DEC CX

   UP:
   INC SI
   CMP AL, [SI]
   JLE NEXT
   MOV AL, [SI]
```

```
 NEXT:
   LOOP UP
   MOV SMALL, AL
   INT 03H
CODE ENDS
END START
```

Code : Find largest number from an array of n numbers

Flowchart -



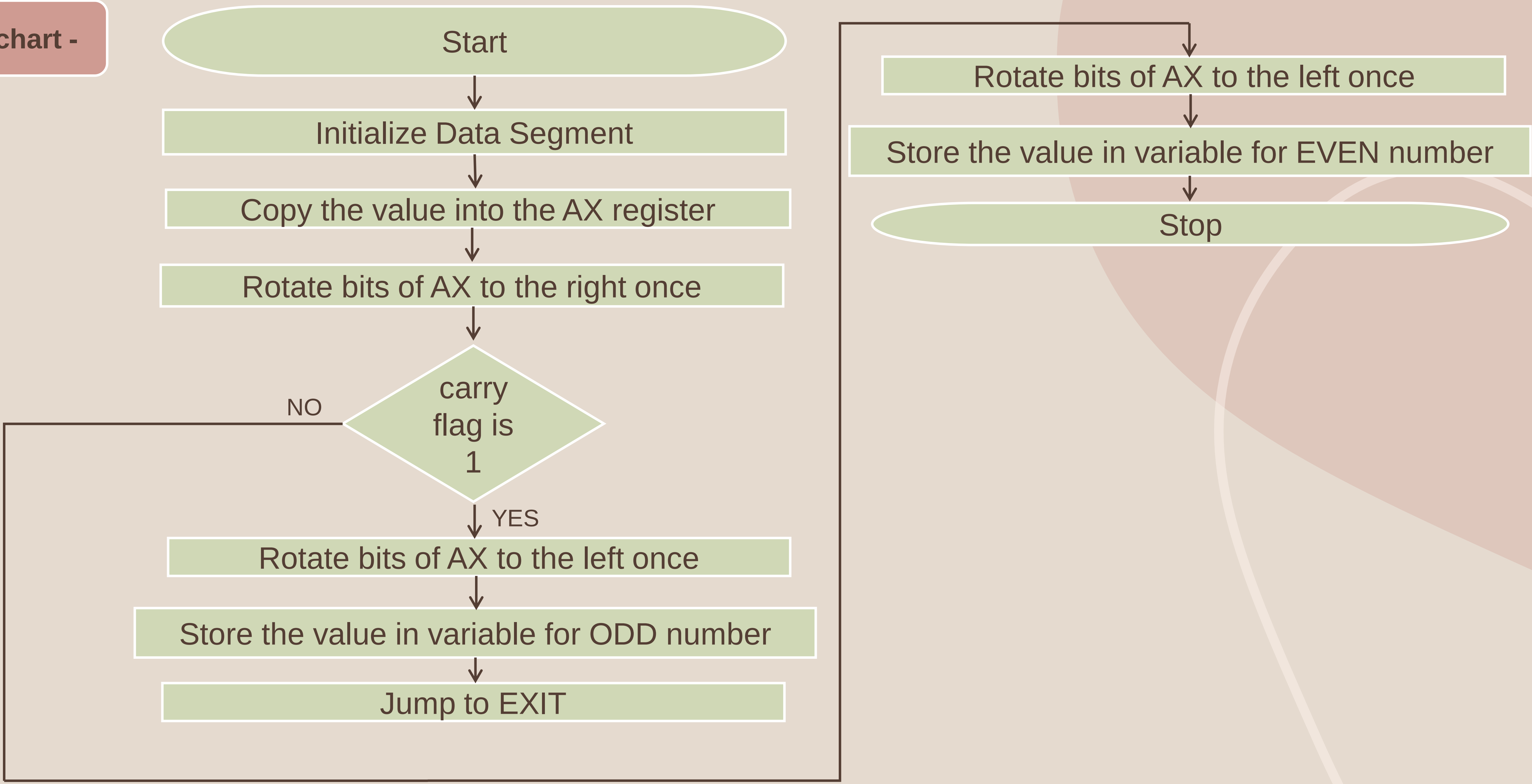
Start
Initialize Data Segment
Initialize memory pointer and Initialize byte counter in CX
Load effective address of ARRAY into SI register
Load the first element into AL register
Decrement CX counter
Increment SI register
Compare the first element and next element
First value is greater
YES
NO
Copy Second element in AL register
Next : Loop
Is CX counter ‘0’
NO
YES
Copy value of AL in result
Stop

```
DATA SEGMENT
    ARRAY DB 12H, 07H, 25H, 70H, 02H
    LARGE DB 00H
DATA ENDS

CODE SEGMENT
    ASSUME CS: CODE, DS: DATA
    START:

    MOV AX, DATA
    MOV DS, AX
    MOV AX, 0000H
    MOV CX, 05H
    LEA SI, ARRAY
    MOV AL, [SI]
    DEC CX

    UP:
    INC SI
    CMP AL, [SI]
    JNL NEXT     ; JGE NEXT
    MOV AL, [SI]
```

```
NEXT:
    LOOP UP
    MOV LARGE, AL
    INT 03H
CODE ENDS
END START
```

Code : Check if given number is odd or even

Flowchart -



Start

Initialize Data Segment

Copy the value into the AX register

Rotate bits of AX to the right once

carry flag is 1

NO

YES

Rotate bits of AX to the left once

Store the value in variable for ODD number

Jump to EXIT

Rotate bits of AX to the left once

Store the value in variable for EVEN number

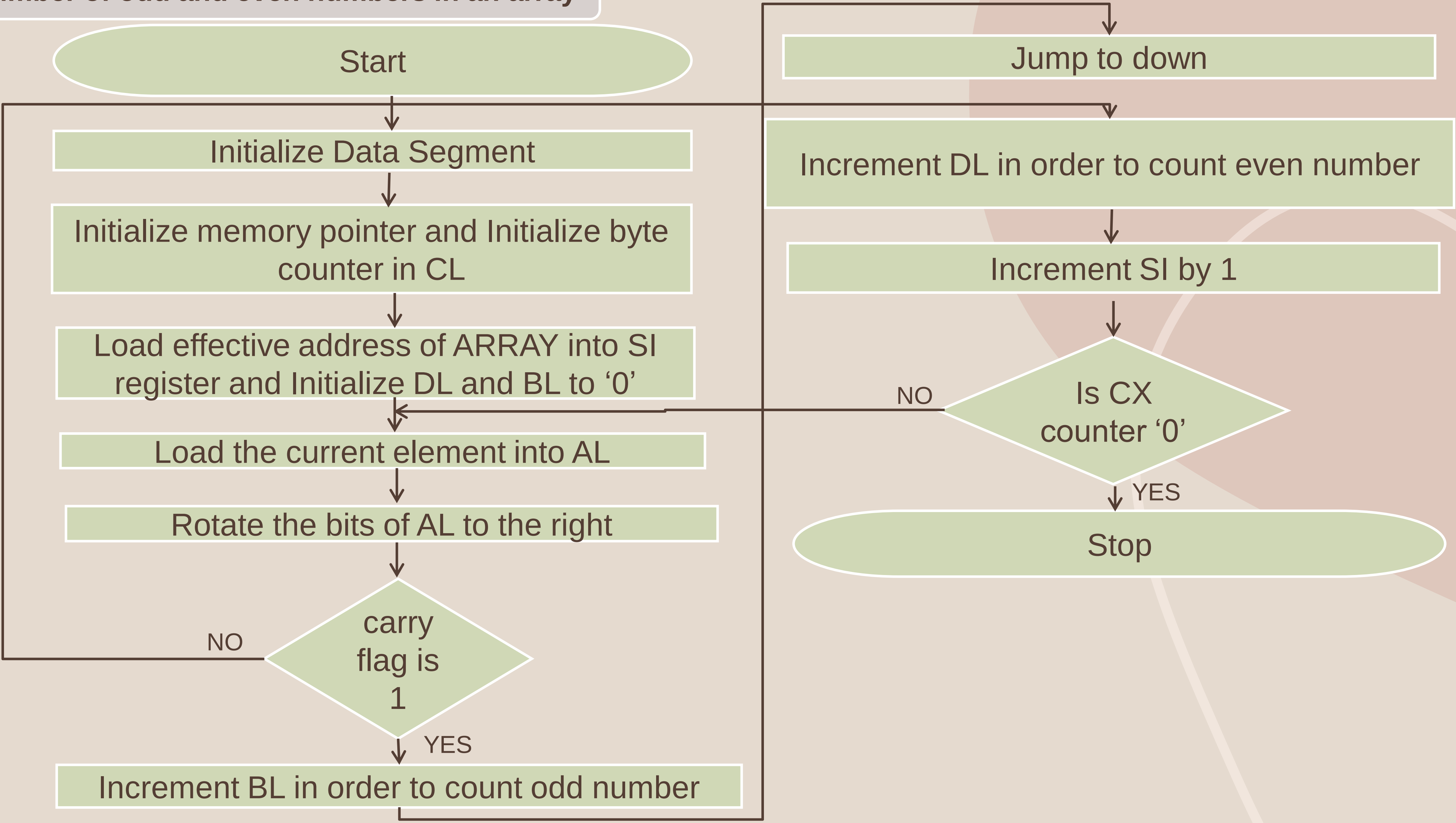
Stop

Code : Check if given number is odd or even

```
DATA SEGMENT
   NUM DW 3344H
   ODD DW 0000H
   EVEN DW 0000H
DATA ENDS

CODE SEGMENT
   ASSUME CS: CODE, DS: DATA
START:
   MOV AX, DATA
   MOV DS, AX
   MOV AX, NUM
   ROR AX, 1
   JNC DN
   ROL AX, 1
   MOV ODD, AX
   JMP EXIT

   DN:
   ROL AX, 1
   MOV EVEN, AX

   EXIT:
   INT 03H
CODE ENDS
END START
```

Code : Find number of odd and even numbers in an array

Flowchart -



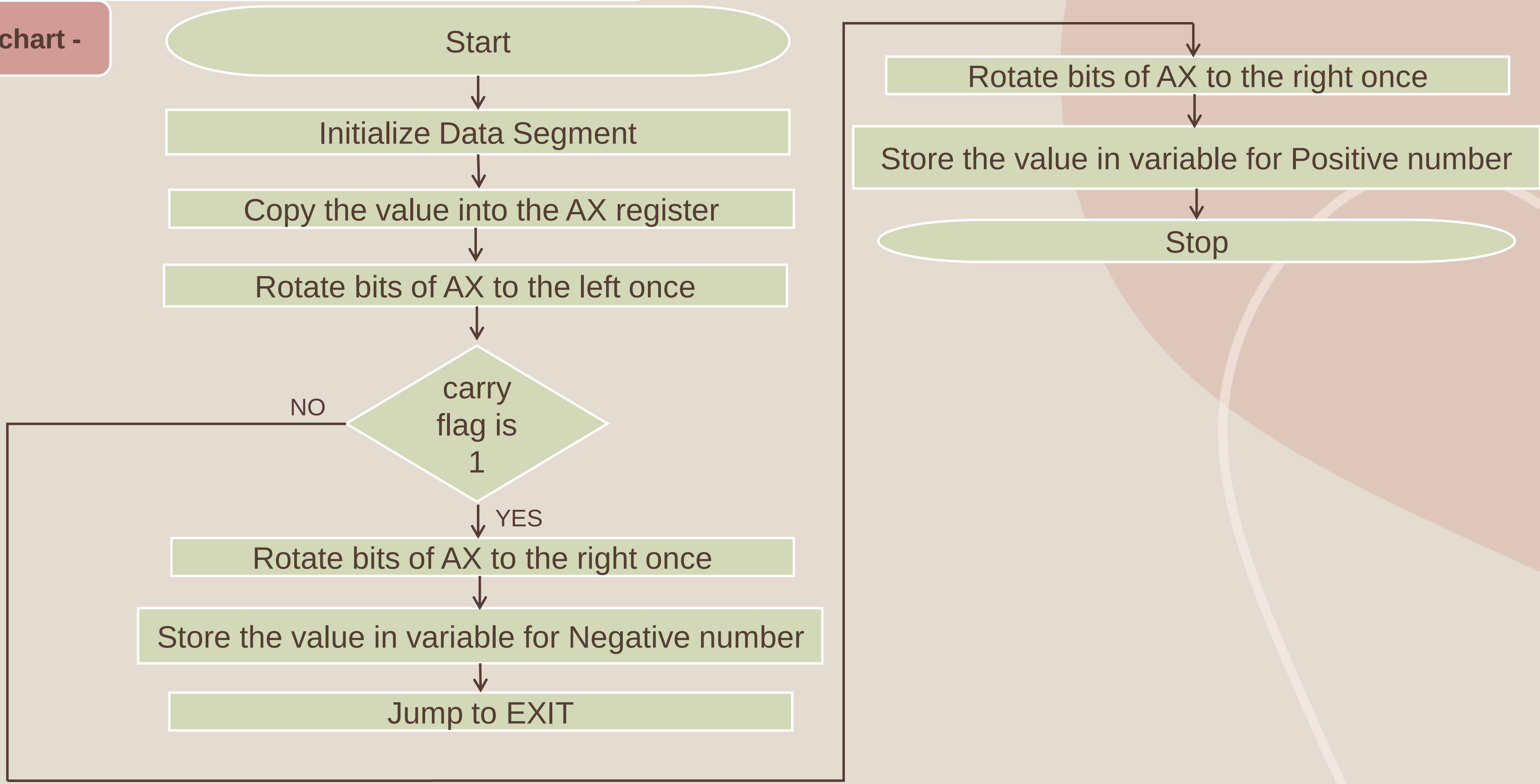
Start
Initialize Data Segment
Initialize memory pointer and Initialize byte counter in CL
Load effective address of ARRAY into SI register and Initialize DL and BL to ‘0’
Load the current element into AL
Rotate the bits of AL to the right
carry flag is 1
NO
YES
Increment BL in order to count odd number
Jump to down
Increment DL in order to count even number
Increment SI by 1
Is CX counter ‘0’
NO
YES
Stop

```
DATA SEGMENT
    ARRAY DB 11H, 12H, 15H, 29H, 45H, 89H, 99H, 22H, 42H, 72H
DATA ENDS

CODE SEGMENT
    ASSUME DS:DATA, CS:CODE
START:
    MOV AX, DATA
    MOV DS, AX

    LEA SI, ARRAY
    MOV CL, 0AH

    MOV DL, 0             ; Initialize counters for even and odd numbers
    MOV BL, 0

TOP:
    MOV AL, [SI]          ; Load the current element into AX
    ROR AL, 1             ; Check if carry is 1 to determine odd/even
    JNC EVEN              ; If zero, jump to EVEN (even number)
    INC BL                ; Increment the count of odd numbers
    JMP DN          ; Jump to CONTINUE to proceed to the next
element
```

```

EVEN:
    INC DL                ; Increment the count of even numbers

DN:
    ADD SI, 1             ; Move to the next element
    LOOP TOP              ; Loop until all elements are processed

    INT 03H

CODE ENDS
END START
```

Code : Check if given number is positive or negative

Flowchart -



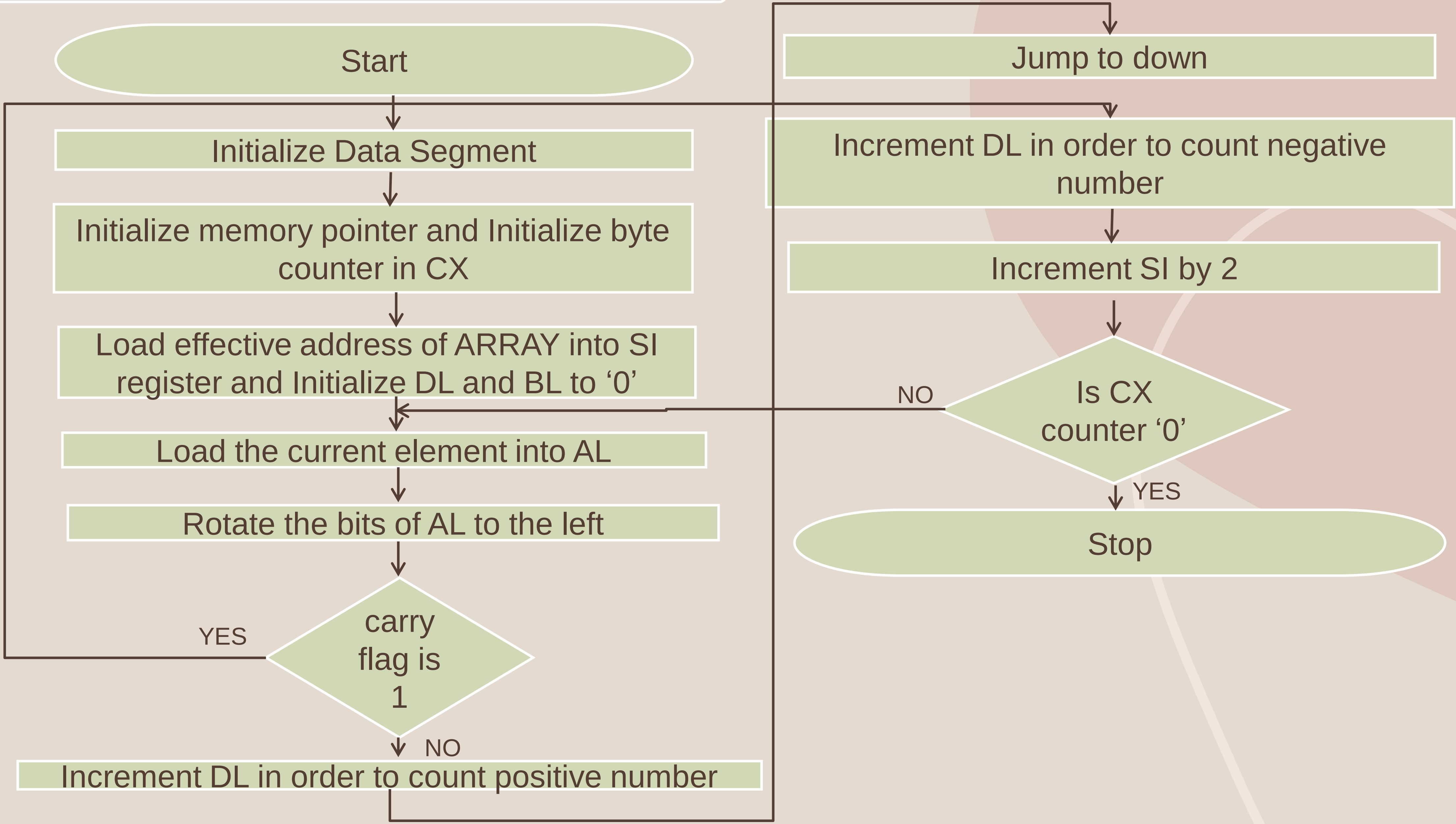
Start
Initialize Data Segment
Copy the value into the AX register
Rotate bits of AX to the left once
carry flag is 1
NO
YES
Rotate bits of AX to the right once
Store the value in variable for Negative number
Jump to EXIT
Rotate bits of AX to the right once
Store the value in variable for Positive number
Stop

Code :
Check if given number is positive or negative

```
DATA SEGMENT
    NUM DW 4002H
    POST DW 00H
    NEGT DW 00H
DATA ENDS

CODE SEGMENT
    ASSUME CS:CODE, DS:DATA
    START:
    MOV AX, DATA
    MOV DS, AX
    MOV AX, NUM
    ROL AX, 1
    JNC DN
    ROR AX, 1
    MOV NEGT, AX
    JMP EXIT

    DN:
    ROR AX,1
    MOV POST, AX

    EXIT:
    INT 03H
CODE ENDS
END START
```

Code : Find number of negative and positive numbers in an array

Flowchart -



Start

Initialize Data Segment

Initialize memory pointer and Initialize byte counter in CX

Load effective address of ARRAY into SI register and Initialize DL and BL to ‘0’

Load the current element into AL

Rotate the bits of AL to the left

carry flag is 1

YES

NO

Increment DL in order to count positive number

Jump to down

Increment DL in order to count negative number

Increment SI by 2

Is CX counter ‘0’

NO

YES

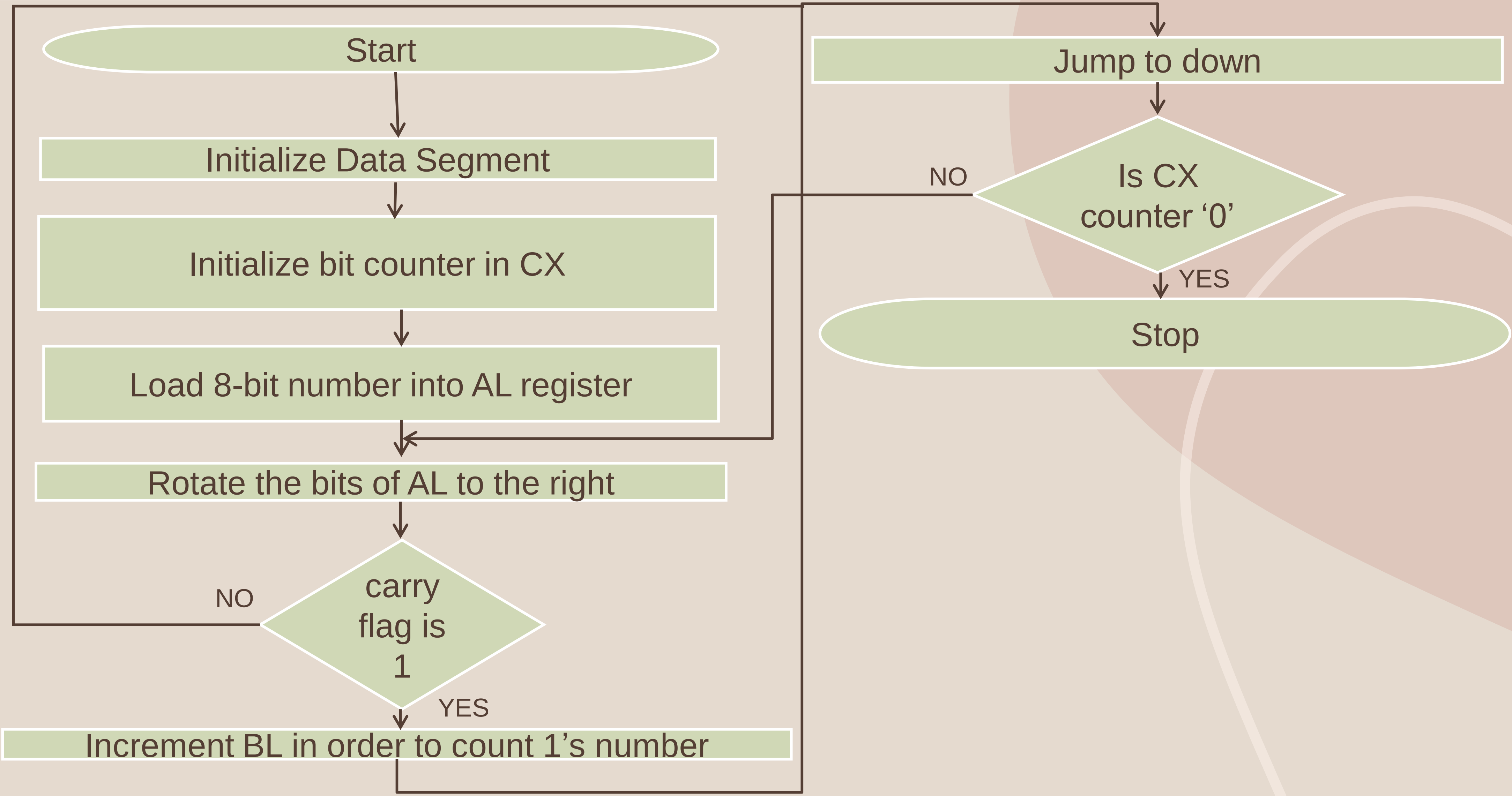
Stop

```
DATA SEGMENT
    ARRAY DW 2579H, 0004H, 5009H, 0159H, 0F900H
DATA ENDS

CODE SEGMENT
    ASSUME DS:DATA, CS:CODE
START:
    MOV AX, DATA
    MOV DS, AX

    LEA SI, ARRAY
    MOV CX, 05H

    MOV DL, 00H              ; Initialize counters for positive and
negative numbers
    MOV BL, 00H

TOP:
    MOV AX, [SI]
    ROL AX, 1
    JC NEGT                ; If zero, jump to NEGT (negative number)
    INC DL                ; Increment the count of positive numbers
    JMP DN
```

```

NEGT:
    INC BL                ; Increment the count of negative numbers

DN:
    ADD SI, 02H
    LOOP TOP

    INT 03H

CODE ENDS
END START
```

Code : Count number of 1’s in 8-bit number

Flowchart -

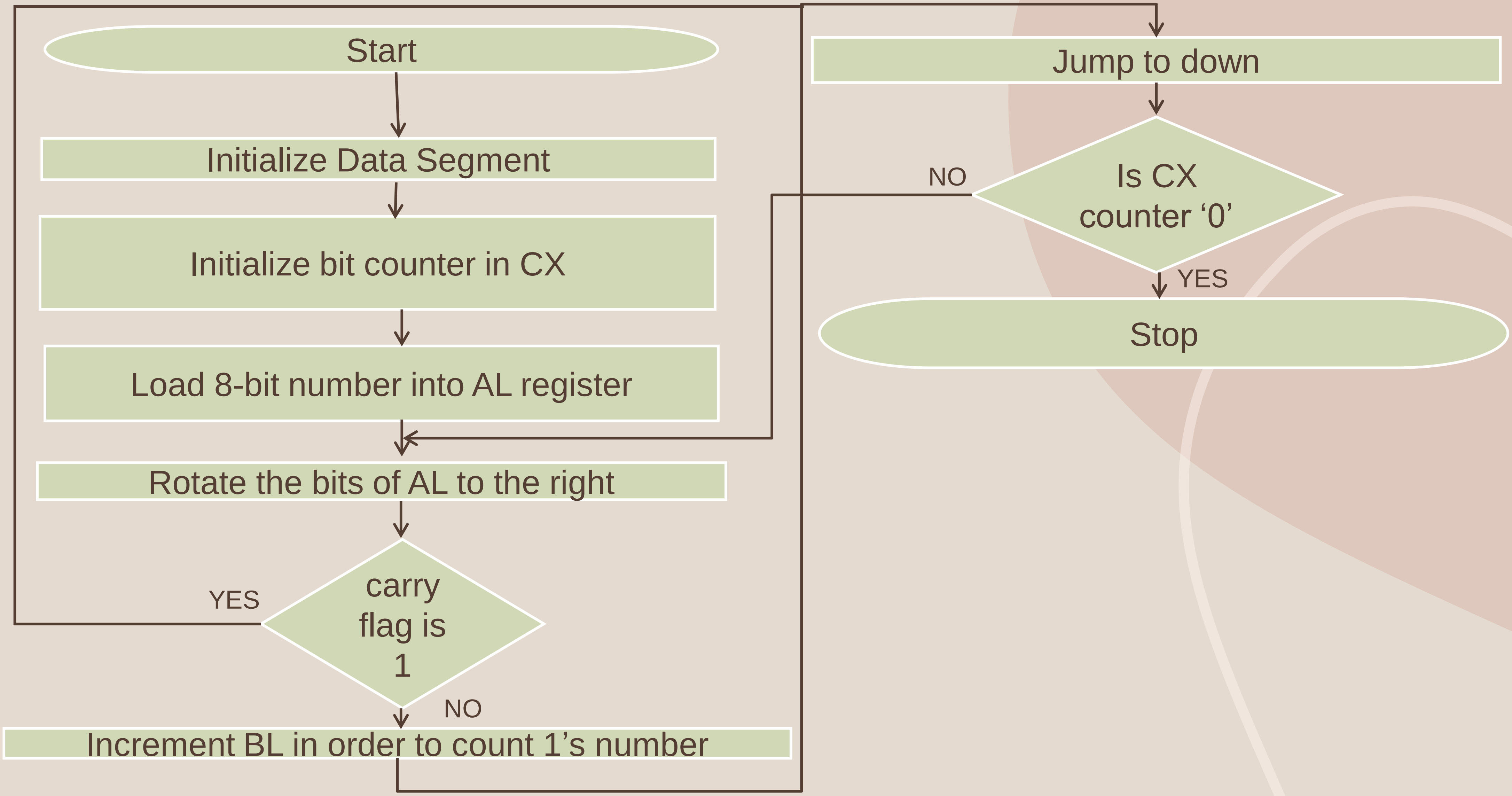

Start
Initialize Data Segment
Initialize bit counter in CX
Load 8-bit number into AL register
Rotate the bits of AL to the right
carry flag is 1
NO
YES
Increment BL in order to count 1’s number
Jump to down
Is CX counter ‘0’
NO
YES
Stop

Code :
Count number of 1’s in 8-bit number

```
DATA SEGMENT
    NUM DB 0FFH
    ONES DB 00H
DATA ENDS

CODE SEGMENT
    ASSUME CS:CODE, DS:DATA

    START:
    MOV AX, DATA
    MOV DS, AX
    MOV CL, 08H
    MOV AL, NUM

    UP:
    ROR AL, 1
    JNC DN
    INC BL      ; HOLDS THE NUMBER OF 1'S PRESENT IN THE NO.

    DN:
    LOOP UP

    INT 03H
CODE ENDS
END START
```

Code : Count number of 0’s in 8-bit number

Flowchart -

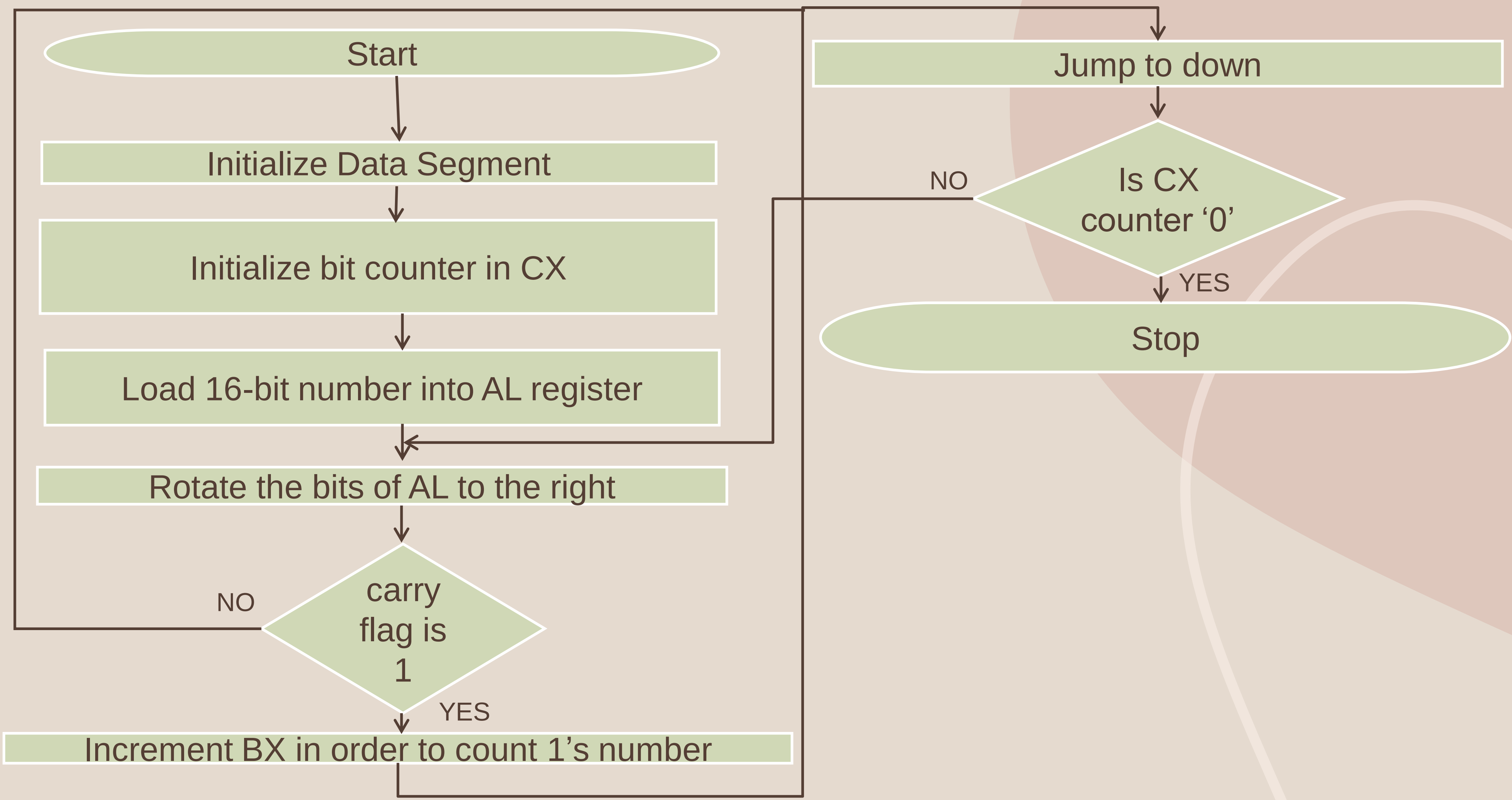

Start
Initialize Data Segment
Initialize bit counter in CX
Load 8-bit number into AL register
Rotate the bits of AL to the right
carry flag is 1
YES
NO
Increment BL in order to count 1’s number
Jump to down
Is CX counter ‘0’
NO
YES
Stop

Code :
Count number of 0’s in 8-bit number

```
DATA SEGMENT
   NUM DB 08H
   ONES DB 00H
DATA ENDS

CODE SEGMENT
   ASSUME CS:CODE, DS:DATA

   START:
   MOV AX, DATA
   MOV DS, AX
   MOV CL, 08H
   MOV AL, NUM
   UP:
   ROR AL, 1
   JC DN
   INC BL     ; HOLDS THE NUMBER OF 0'S PRESENT IN THE NUMBER

   DN:
   LOOP UP

   INT 03H
CODE ENDS
END START
```

Code : Count number of 1’s in 16-bit number

Flowchart -

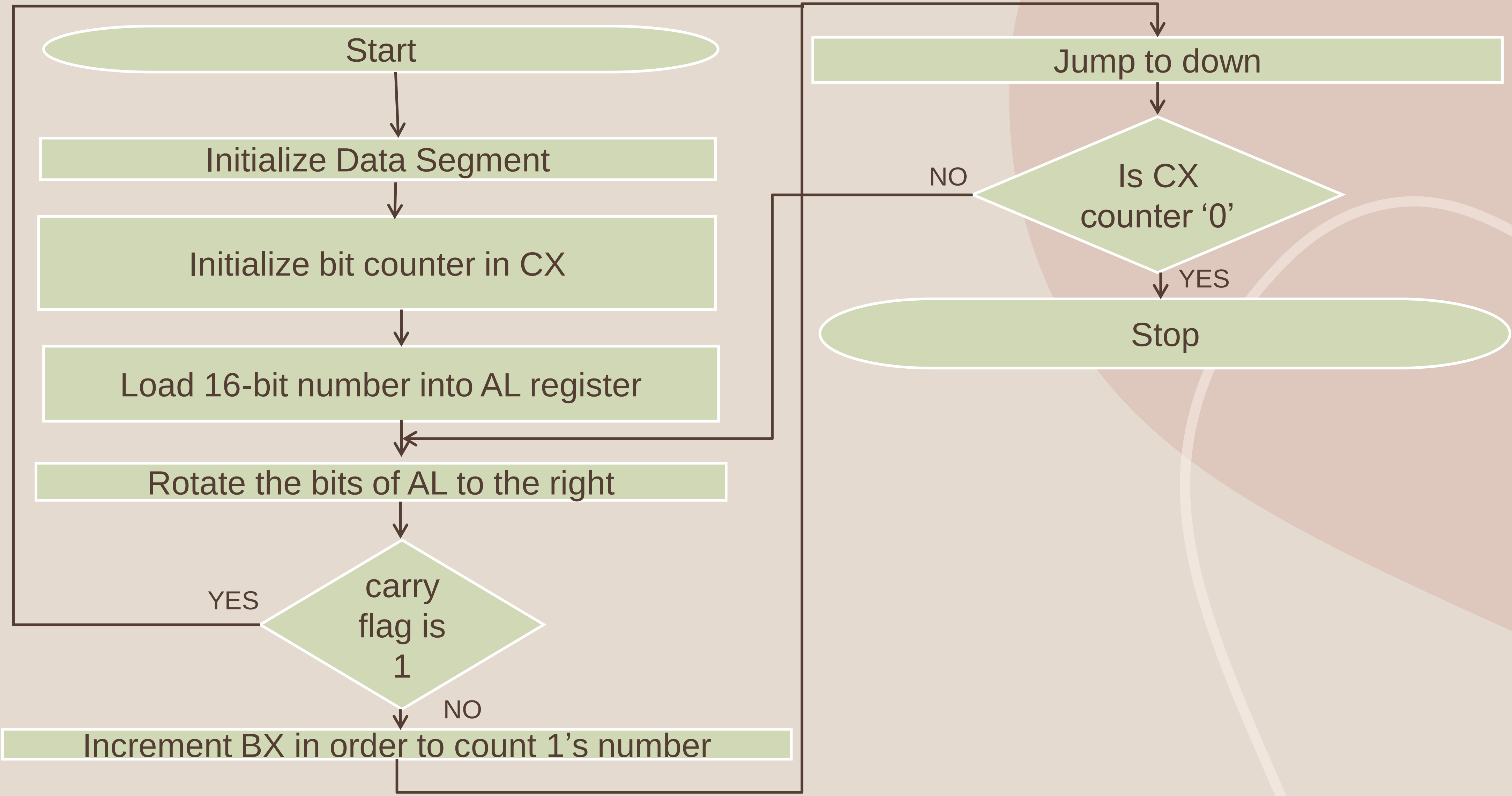

Start
Initialize Data Segment
Initialize bit counter in CX
Load 16-bit number into AL register
Rotate the bits of AL to the right
carry flag is 1
NO
YES
Increment BX in order to count 1’s number
Jump to down
Is CX counter ‘0’
NO
YES
Stop

Code :
Count number of 1’s in 16-bit number

```
DATA SEGMENT
    NUM DW  0FFFFH
    ONES DW 0000H
DATA ENDS

CODE SEGMENT
    ASSUME CS:CODE, DS:DATA

    START:
    MOV AX, DATA
    MOV DS, AX
    MOV CX, 10H
    MOV AX, NUM

    UP:
    ROR AX, 1
    JNC DN
    INC BX      ; HOLDS THE NUMBER OF 1'S PRESENT IN THE NUMBER

    DN:
    LOOP UP
    INT 03H
CODE ENDS
END START
```

Code : Count number of 0’s in 16-bit number

Flowchart -


Start

Initialize Data Segment

Initialize bit counter in CX

Load 16-bit number into AL register

Rotate the bits of AL to the right

carry flag is 1

YES

NO

Increment BX in order to count 1’s number

Jump to down

Is CX counter ‘0’

NO

YES

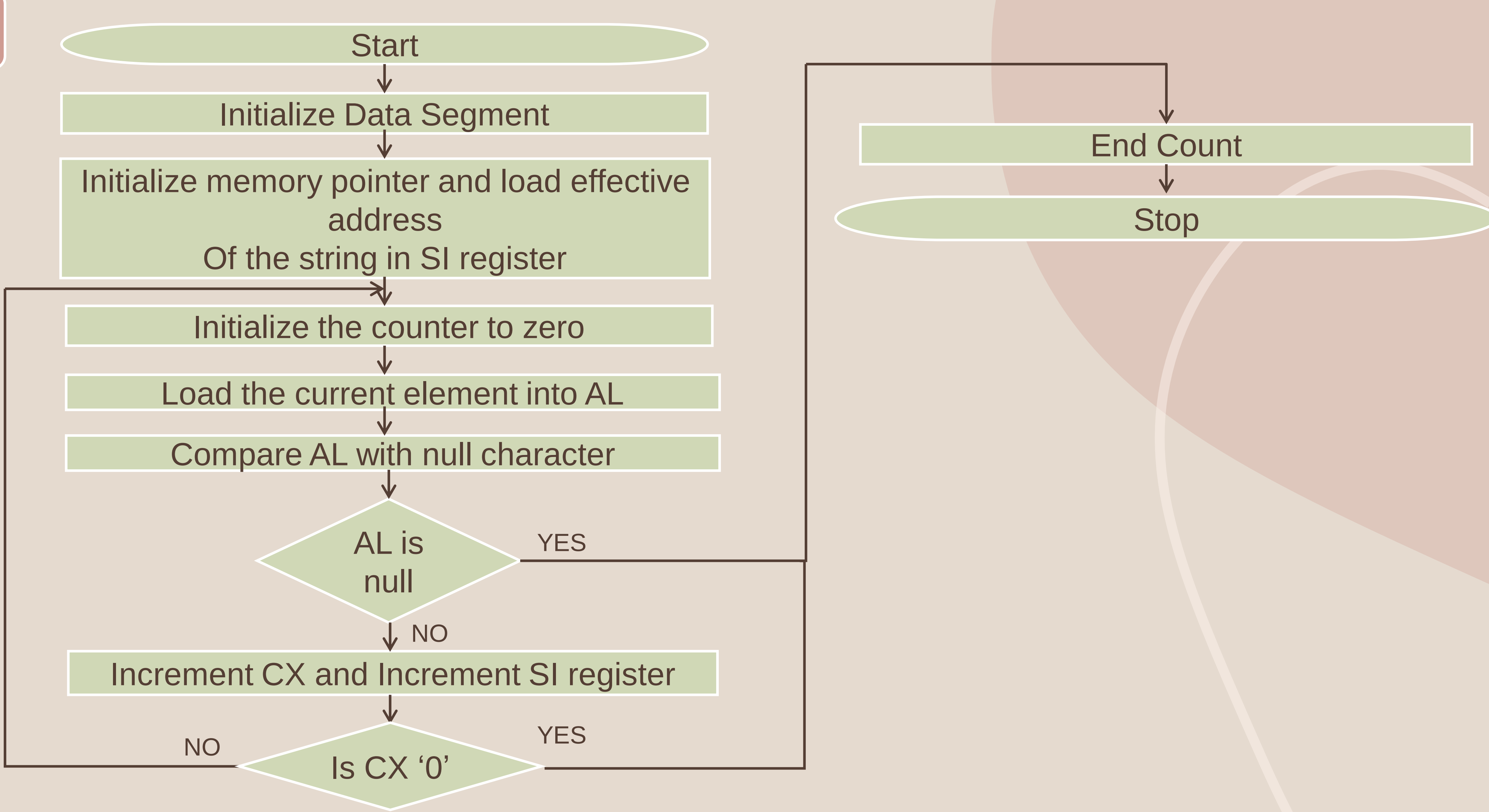
Stop

Code :
Count number of 0’s in 16-bit number

```
DATA SEGMENT
    NUM DW  0FFFFH
    ONES DW 0000H
DATA ENDS

CODE SEGMENT
    ASSUME CS:CODE, DS:DATA

    START:
    MOV AX, DATA
    MOV DS, AX
    MOV CX, 10H
    MOV AX, NUM

    UP:
    ROR AX, 1
    JC DN
    INC BX      ; HOLDS THE NUMBER OF 0'S PRESENT IN THE
NUMBER

    DN:
    LOOP UP
    INT 03H
CODE ENDS
END START
```

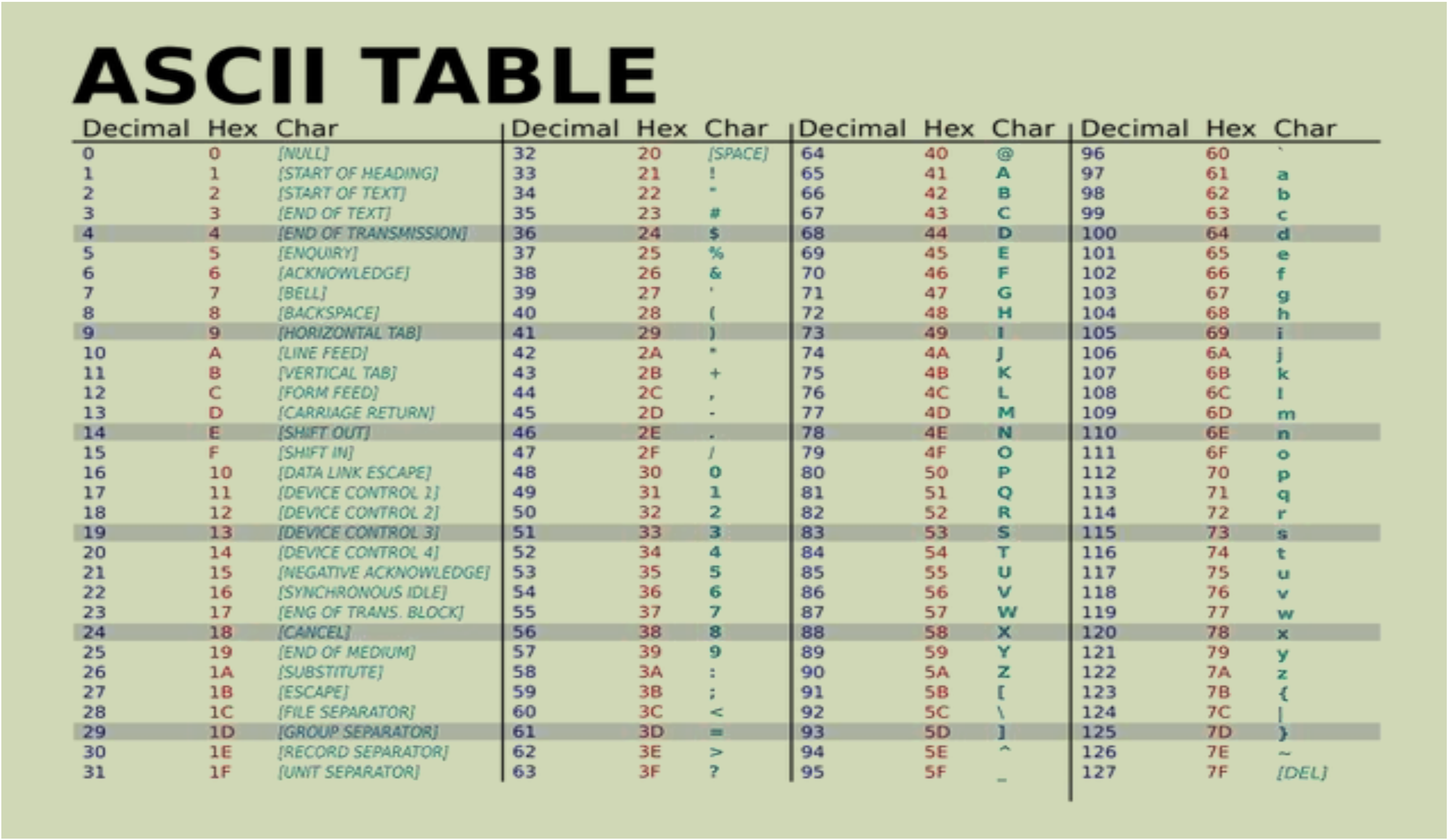
Code : Find length of the string

Flowchart -


Start
Initialize Data Segment
Initialize memory pointer and load effective address
Of the string in SI register
Initialize the counter to zero
Load the current element into AL
Compare AL with null character
AL is null
YES
NO
Increment CX and Increment SI register
Is CX '0'
NO
YES
End Count
Stop

Code : Find length of the string

Flowchart -



ASCII TABLE

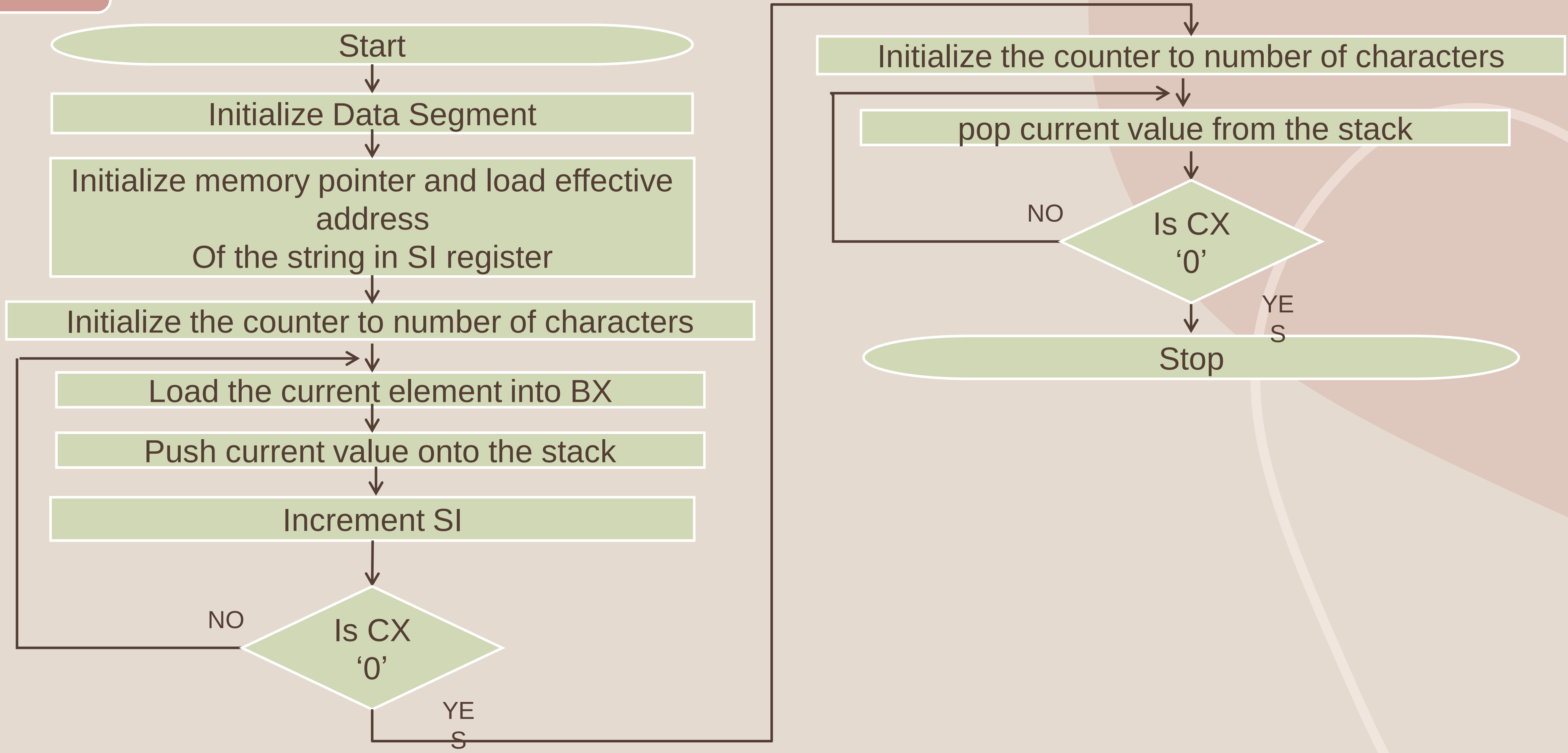
Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

Code : Find length of the string

```
DATA SEGMENT
    STRING DB 'Hello, Class', 0   ; Null-terminated string
DATA ENDS

CODE SEGMENT
    ASSUME CS:CODE, DS:DATA
START:
    MOV AX, DATA
    MOV DS, AX

    LEA SI, STRING
    MOV CX, 0           ; Initialize the counter to zero

COUNT_LOOP:
    MOV AL, [SI]
    CMP AL, 0           ; Compare AL with null character
    JE END_COUNT        ; If AL is null, end counting

    INC CX
    INC SI
    JMP COUNT_LOOP

END_COUNT:
    INT 03H
CODE ENDS
END START
```

Code : Arrange string in the reverse order

Flowchart -



Start

Initialize Data Segment

Initialize memory pointer and load effective address
Of the string in SI register

Initialize the counter to number of characters

Load the current element into BX

Push current value onto the stack

Increment SI

Is CX ‘0’

NO

YES

Initialize the counter to number of characters

pop current value from the stack

Is CX ‘0’

NO

YES

Stop

Code :
Arrange string in the reverse order

```
INCLUDE 'EMU8086.INC'
DATA SEGMENT
    STRING DB 'HEY', 0H
DATA ENDS

CODE SEGMENT
    ASSUME CS: CODE, DS: DATA
    START:
        MOV AX, DATA
        MOV DS, AX

        LEA SI, STRING
        MOV CX, 03H

        ORIGINAL:
        MOV BX, [SI]
        PUSH BX
        INC SI
        LOOP ORIGINAL

        MOV CX, 03H
```

```
        REVERSE:
        POP DX
        MOV AH, 02H
        INT 21H
        LOOP REVERSE

CODE ENDS
END START
```

Thank you

Microprocessors

Course Code
22415

Course Work Journey

		Teaching Hours	Progress Bar
1	8086 – 16 Bit Microprocessor	**8 Hours**	**100%**
2	The Art of Assembly Language Programming	**12 Hours**	**100 %**
3	Instruction Set of 8086 Microprocessor	**16 Hours**	**100 %**
4	Assembly Language Programming	**16 Hours**	**100 %**
5	Procedure and Macros	**12 Hours**	**0 %**

UNIT-5
Procedure and Macros

Objective

- Understand the role and usage of procedures and macros in assembly language programming.

Procedure

What is Procedure

A procedure is group of instructions that usually performs one task.
It is a **reusable section of a program** which is stored in memory once but **can be used as often** as necessary.

Types of Procedure

- Near Procedure
- Far Procedure

Near Procedure

A procedure is known as NEAR procedure if is written (defined) in the same code segment which is calling that procedure.

It is also called as Intra segment call.
Only Instruction Pointer(IP register) contents will be changed in NEAR procedure.

FAR Procedure

A procedure is known as FAR procedure if is written (defined) in the different code segment which is calling that procedure.

It is also called as Inter segment call.
In this case both Instruction Pointer(IP) and the Code Segment(CS) register content will be changed.

Procedure

Directives used of procedure :

PROC directive: The PROC directive is used to identify the start of a procedure. The PROC directive follows a name given to the procedure. After that, the term FAR and NEAR is used to specify the type of the procedure.

ENDP Directive: This directive is used along with the name of the procedure to indicate the end of a procedure to the assembler. The PROC and ENDP directive are used to bracket a procedure

CALL instruction : The CALL instruction is used to transfer execution to a procedure. It performs two operation. When it executes, first it stores the address of instruction after the CALL instruction on the stack.
Second it changes the content of IP register in case of Near call and changes the content of IP register and cs register in case of FAR call**.**

There are two types of calls.
1)Near Call or Intra segment call.
2) Far call or Inter Segment call

Operation for Near Call : When 8086 executes a near CALL instruction, it decrements the stack pointer by 2 and copies the IP register contents on to the stack. Then it copies address of first instruction of called procedure.
SP ← SP-2
IP → stores onto stack
IP ← starting address of a procedure.

Procedure

Operation of FAR CALL:

When 8086 executes a far call, it decrements the stack pointer by 2 and copies the contents of CS register to the stack.

It the decrements the stack pointer by 2 again and copies the content of IP register to the stack.

Finally, it loads cs register with base address of segment having procedure and IP with address of first instruction in procedure.

SP ← sp-2
cs contents → stored on stack
SP ← sp-2
IP contents → stored on stack
CS ← Base address of segment having procedure
IP ← address of first instruction in procedure.

Procedure

RET instruction :
The RET instruction will return execution from a procedure to the next instruction after call in the main program. At the end of every procedure RET instruction must be executed.

Operation for Near Procedure : For NEAR procedure ,the return is done by replacing the IP register with an address popped off from stack and then SP will be incremented by 2.

IP ← Address from top of stack
SP ← SP+2

Operation for FAR procedure : IP register is replaced by address popped off from top of stack, then SP will be incremented by 2.The CS register is replaced with a address popped off from top of stack. Again, SP will be incremented by 2.

IP ← Address from top of stack
SP ← SP+2
CS ← Address from top of stack
SP ← SP+2

Procedure

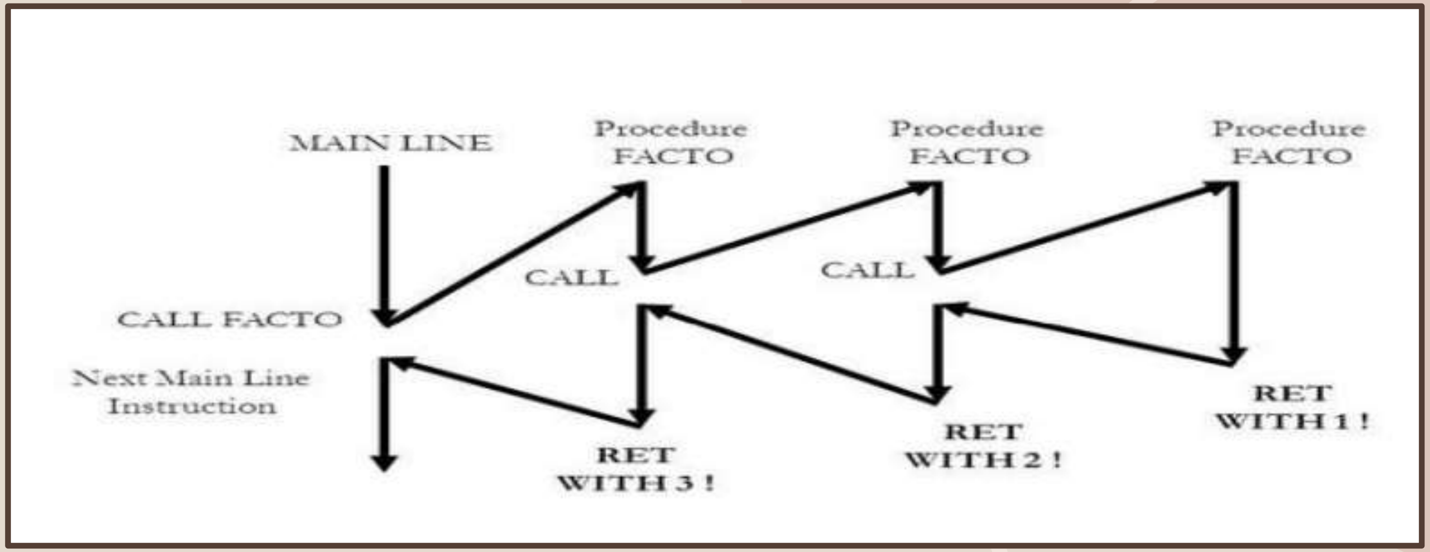
Aspect	Near Procedure	Far Procedure
Location in Code Segment	Same code segment	Different code segment
Call Type	Intra-segment call	Inter-segment call
Replaces Registers	Old IP with new IP	CS & IP with new CS & IP
Call Keyword	NEAR	FAR
Stack Locations	Requires fewer locations	Requires more locations

Types of Procedure

Recursive Procedure:
A recursive procedure is one that calls itself. This is often used when working with complex data structures like trees.

Each time the procedure is called, a counter (usually denoted as N) is decremented by one. The procedure continues to call itself until N reaches zero, at which point it stops.

Recursive procedures are efficient for certain tasks but require a termination condition to avoid infinite loops.

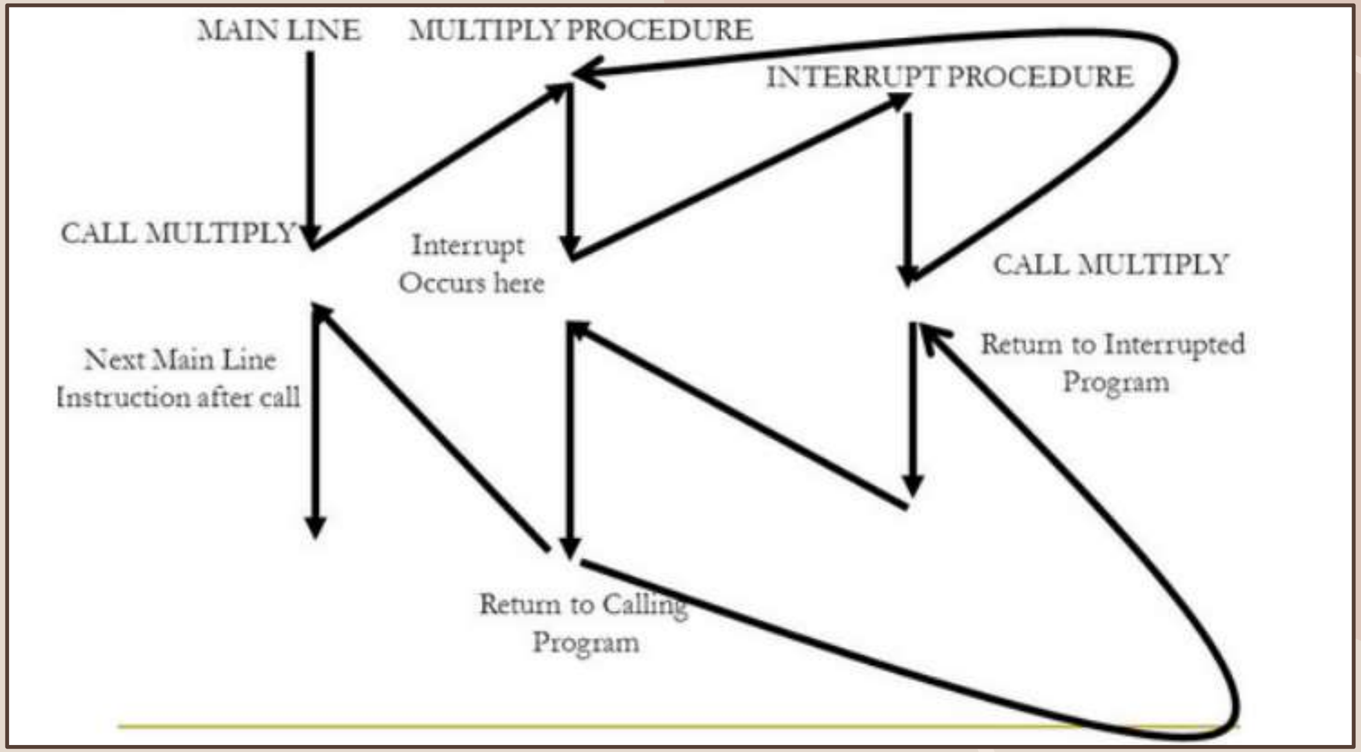


Types of Procedure

Re-entrant Procedure:
A Re-entrant procedure is one that allows program execution flow to re-enter the procedure multiple times, even if it's already being executed.

This can happen in situations where Procedure 1 is called from the main program, then Procedure 2 is called from Procedure 1, and finally Procedure 1 is called again from Procedure 2.

In this case, the program execution flow re-enters Procedure 1, both the first time when it was called from the main program and the second time when it was called from Procedure 2.
This ability to re-enter a procedure without interference is what makes it reentrant.



Advantages and Disadvantages of Procedure

Advantages:

1) Allows to save memory space.
2) Program development becomes easier.
3) Debugging of errors in program become easy.
4) Reduced size of program
5) Reusability of procedure.

Disadvantages:

1) CALL and RET instructions are always required to integrate with procedures.
2) Requires the extra time to link procedure and return from it.
3) For small group of instructions, linking and returning time more than the execution time, hence for small group of instructions procedures cannot be preferred

Macro

A MACRO is group of small instructions that usually performs one task. It is a **reusable section of a program**.

A macro can be defined anywhere in a program using directive MACRO & ENDM.

Advantages:

1) Program written with macro is more readable.
2) Macro can be called just writing by its name along with parameters, hence no extra code is required like CALL & RET.
3) Execution time is less because of no linking and returning
4) Finding errors during debugging is easier.

Disadvantages:

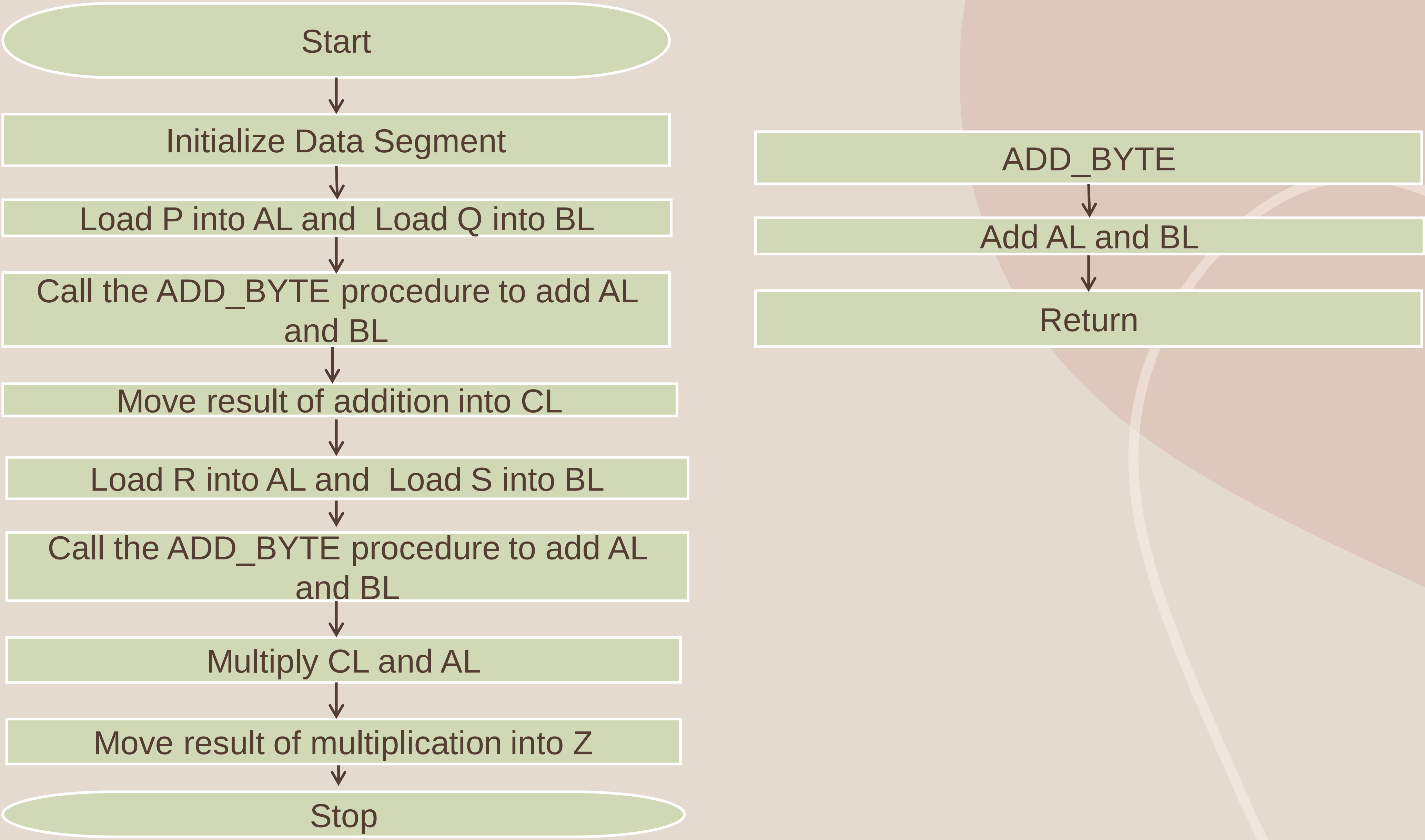
1) object code is generated every time a macro is called hence object file becomes lengthy.
2) For large group of instructions macro cannot be preferred

Macro

Aspect	Procedure	Macro
Purpose	Used for large groups of instructions	Used for small groups of instructions
Object Code Generation	Generated only once in memory	Generated every time the macro is called
Call and Return	CALL & RET instructions are used	Macro can be called by writing its name
Length of Object File	Object file size is less	Object file size can become lengthy
Execution Time	More time is required for execution	Less time is required for execution
Definition Directives	PROC & ENDP are used for defining	MACRO and ENDM are used for defining
Syntax	Procedure_name PROC -------------------- Procedure_name ENDP	Macro_name MACRO[ARGUMENT ,………. ARGUMENT N] -------------------- Macro_name ENDM

Code : Write an ALP for Z = (P + Q) * (R + S) using PROCEDURE.

Flowchart -



Start

Initialize Data Segment

Load P into AL and Load Q into BL

Call the ADD_BYTE procedure to add AL and BL

Move result of addition into CL

Load R into AL and Load S into BL

Call the ADD_BYTE procedure to add AL and BL

Multiply CL and AL

Move result of multiplication into Z

Stop

ADD_BYTE

Add AL and BL

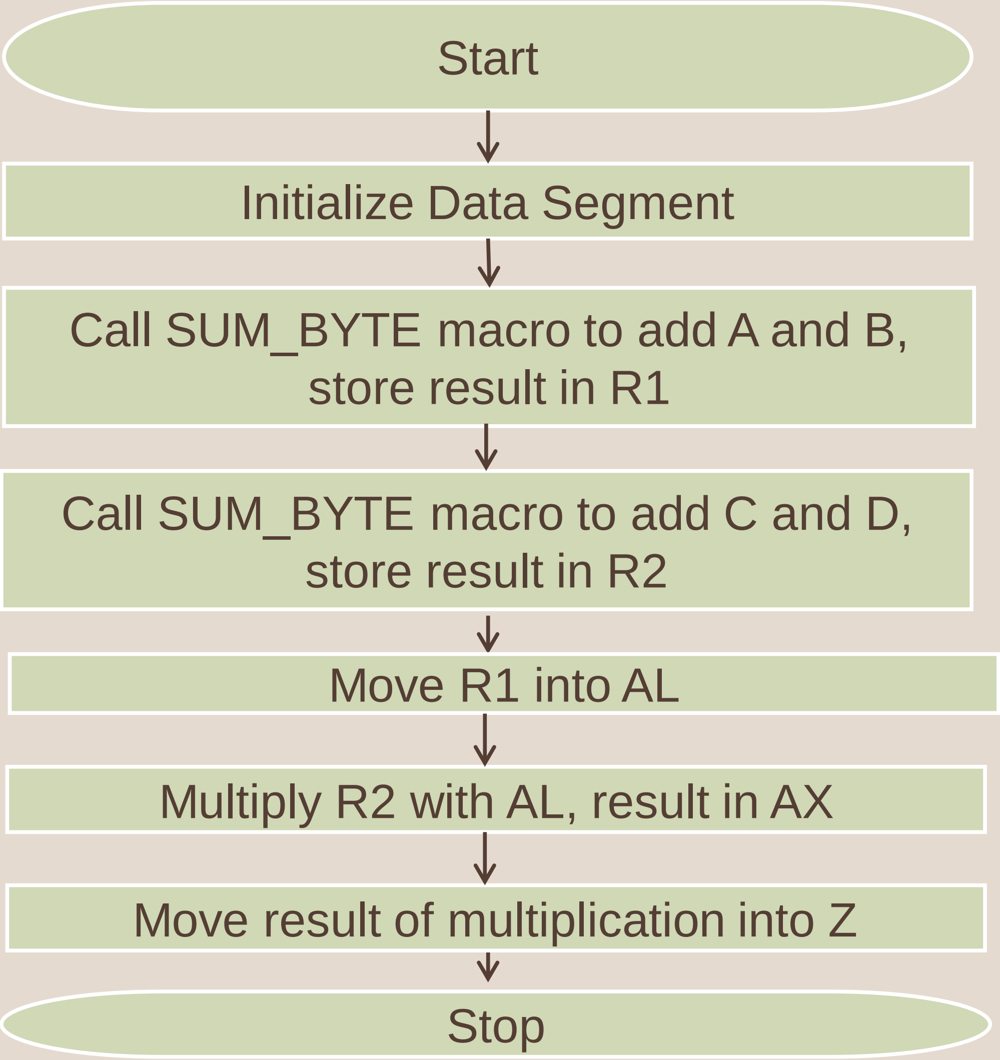
Return

```
DATA SEGMENT
   P DB 04H
   Q DB 02H
   R DB 01H
   S DB 02H
   Z DW 00H
DATA ENDS

CODE SEGMENT
   ASSUME CS: CODE, DS: DATA
START:
   MOV AX, DATA
   MOV DS, AX

   MOV AL, P     ; Load P into AL
   MOV BL, Q     ; Load Q into BL
   CALL ADD_BYTE ; Call the ADD_BYTE procedure to add AL and BL

   MOV CL, AL    ; Move result of addition into CL
   MOV AL, R     ; Load R into AL
   MOV BL, S     ; Load S into BL
   CALL ADD_BYTE ; Call ADD_BYTE to add AL and BL
```

```
 MUL CL        ; Multiply CL and AL

   MOV Z, AX     ; Move result of multiplication into Z
   INT 3H        ; Terminate program

ADD_BYTE PROC
   ADD AL, BL    ; Add AL and BL
   RET           ; Return from procedure
ENDP


CODE ENDS
END START
```

Code : Write an ALP for Z = (A + B) * (C + D) using MACRO.

Flowchart -



Start

Initialize Data Segment

Call SUM_BYTE macro to add A and B, store result in R1

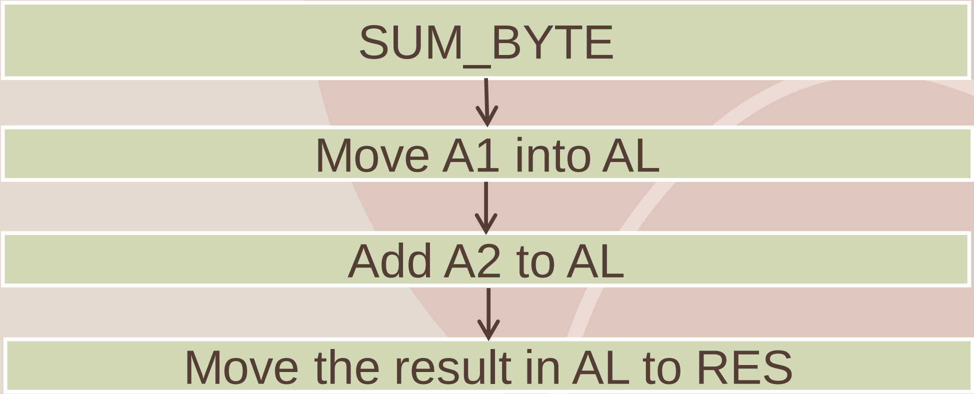
Call SUM_BYTE macro to add C and D, store result in R2

Move R1 into AL

Multiply R2 with AL, result in AX

Move result of multiplication into Z

Stop



SUM_BYTE

Move A1 into AL

Add A2 to AL

Move the result in AL to RES

```
DATA SEGMENT
    A DB 04H
    B DB 04H
    C DB 01H
    D DB 02H
    R1 DB 00H
    R2 DB 00H
    Z DW 00H
DATA ENDS

SUM_BYTE MACRO A1, A2, RES
    MOV AL, A1      ; Move A1 into AL
    ADD AL, A2      ; Add A2 to AL
    MOV RES, AL     ; Move the result in AL to RES
ENDM

CODE SEGMENT
    ASSUME CS: CODE, DS: DATA

START:
    MOV AX, DATA
    MOV DS, AX
```

```
  SUM_BYTE A, B, R1    ; Call SUM_BYTE macro to add A and B, store result in R1

SUM_BYTE C, D, R2    ; Call SUM_BYTE macro to add C and D, store result in R2

    MOV AL, R1      ; Move R1 into AL
    MUL R2          ; Multiply R2 with AL, result in AX

    MOV Z, AX       ; Move result of multiplication into Z
    INT 3H

CODE ENDS
END START
```

Thank you