# Vidyavardhini's College of Engineering and Technology

## Department of Artificial Intelligence & Data Science

**AY: 2023-24**

| | | | |
|---|---|---|---|
| **Class:** | | **Semester:** | |
| **Course Code:** | | **Course Name:** | |

| | |
|---|---|
| **Name of Student:** | |
| **Roll No. :** | |
| **Experiment No.:** | |
| **Title of the Experiment:** | |
| **Date of Performance:** | |
| **Date of Submission:** | |

## Evaluation

| Performance Indicator | Max. Marks | Marks Obtained |
|---|---|---|
| Performance | 5 | |
| Understanding | 5 | |
| Journal work and timely submission | 10 | |
| Total | 20 | |

| Performance Indicator | Exceed Expectations (EE) | Meet Expectations (ME) | Below Expectations (BE) |
|---|---|---|---|
| Performance | 4-5 | 2-3 | 1 |
| Understanding | 4-5 | 2-3 | 1 |
| Journal work and timely submission | 8-10 | 5-8 | 1-4 |

**Checked by**

**Name of Faculty :**

**Signature :**

**Date :**

**Aim:** Assembly Language Program to perform basic arithmetic operations (addition, subtraction, multiplication, and division) on 16-bit data.

**Theory:**

**MOV:** MOV Destination, Source.
The MOV instruction copies data from a specified destination. word or byte of data from a specified destination.
Source: Register, Memory Location, Immediate Number
Destination: Register, Memory Location
MOV CX, 037AH; Put immediate number 037AH to CX.

**ADD:** ADD Destination, Source.
These instructions add a number source to a number from some destination and put the result in the specified destination.
Source: Register, Memory Location, Immediate Number
Destination: Register, Memory Location
The source and the destination in an instruction cannot both be memory locations.
ADD AL, 74H; add the immediate number to 74H to the content of AL. Result in AL.
**SUB:** SUB Destination, Source.
These instructions subtract the number in some source from the number in some destination and put the result in the destination.
Source: Immediate Number, Register, or Memory Location.
Destination: Register or a Memory Location.
The source and the destination in an instruction cannot both be memory locations.
SUB AX, 3427H; Subtract immediate number 3427H from AX.
**MUL:** MUL Source.
This instruction multiplies an unsigned byte from some source times an unsigned byte in the AL register or an unsigned word from some source times an unsigned word in the AX register.
Source: Register, Memory Location.
MUL CX; Multiply AX with CX; result in high word in DX, low word in AX.
**DIV:** DIV Source.
This instruction is used to divide an unsigned word by a byte or to divide an unsigned double word (32 bits) by a word.
Source: Register, Memory Location.

If the divisor is 8-bit, then the dividend is in AX register. After division, the quotient is in AL and the remainder in AH.
If the divisor is 16-bit, then the dividend is in DX-AX register. After division, the quotient is in AX and the remainder in DX.
DIV CX; divide double word in DX and AX by word in CX; Quotient in AX; and remainder in DX.

Algorithm to add two 16-bit numbers

1. Load the first number in AX
2. Load the second number in BX
3 Add the second number to AX
4. Store the result in AX.

Algorithm to subtract two 16-bit numbers

1. Load the first number in AX.
2. Load the second number. in BX 3. Subtract the second number AX

4. Store the result in AX.

Algorithm to multiply a 16-bit number by an 8-bit number

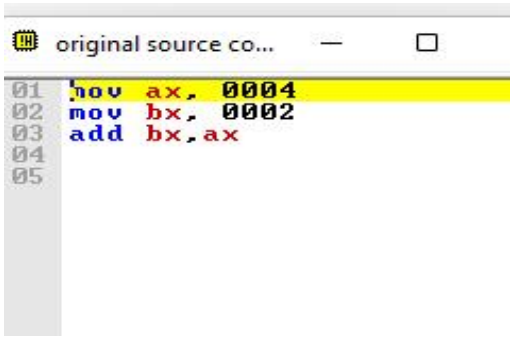
1. Load the first number in AX.
2. Load the second number. in BL
3. Multiply DX and AX.
4. The result is in DX and AX.

Algorithm to divide a 16-bit number by an 8-bit number

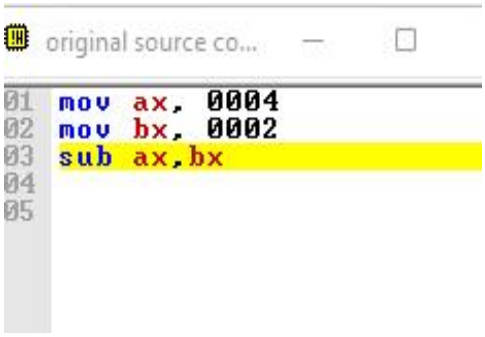
1. Load the first number in AX.
2. Load the second number. in BL
3. Divide AX by BL.
4. After division, the quotient is in AL and the remainder is in AH.

**Assembly code:**

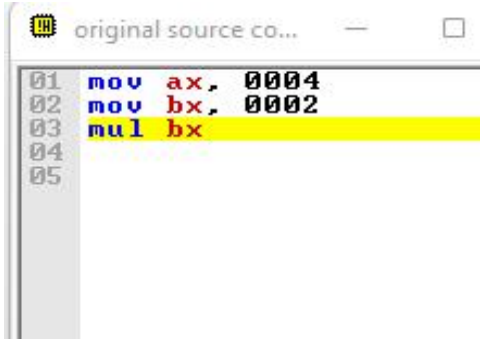
1. Add



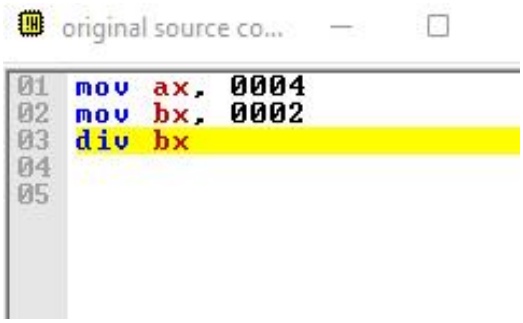

2. Subtract




3. Multiply

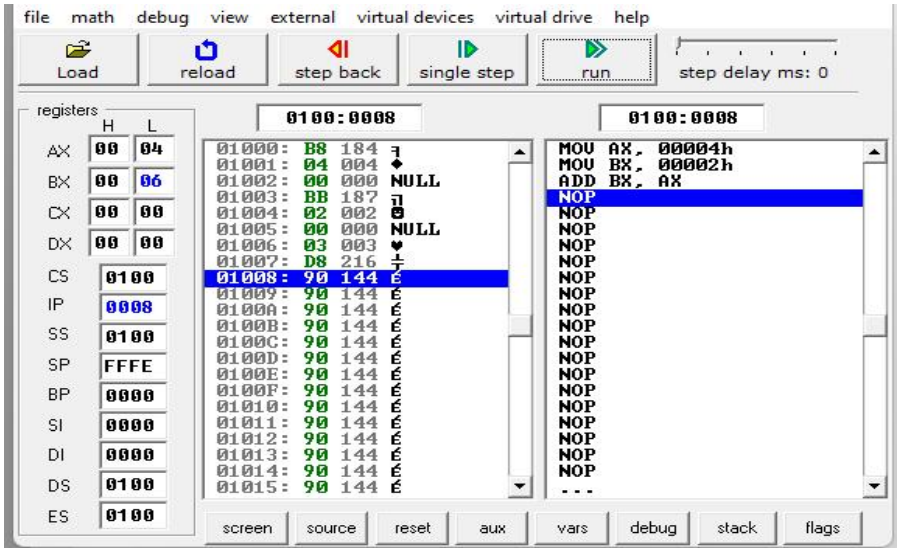


4. Division

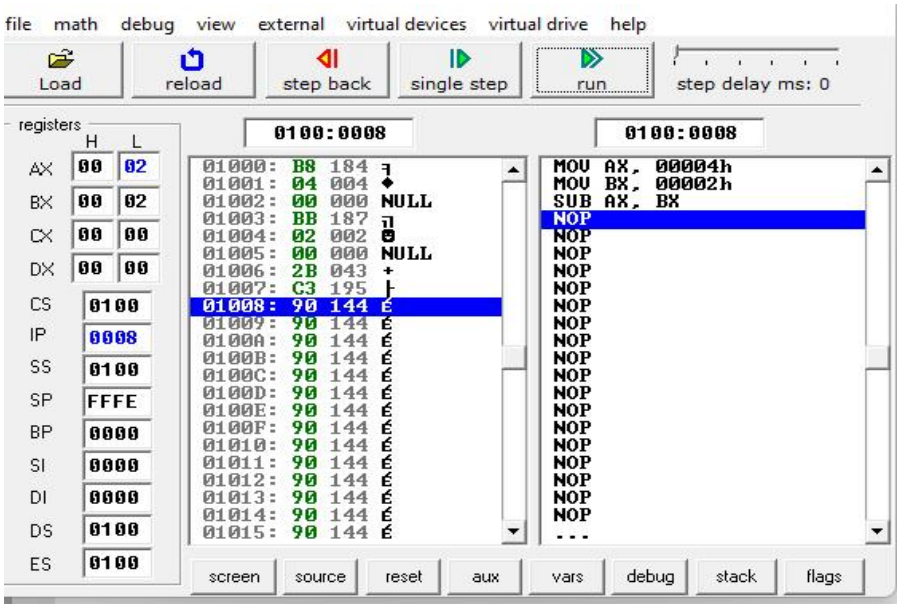



**Output:**

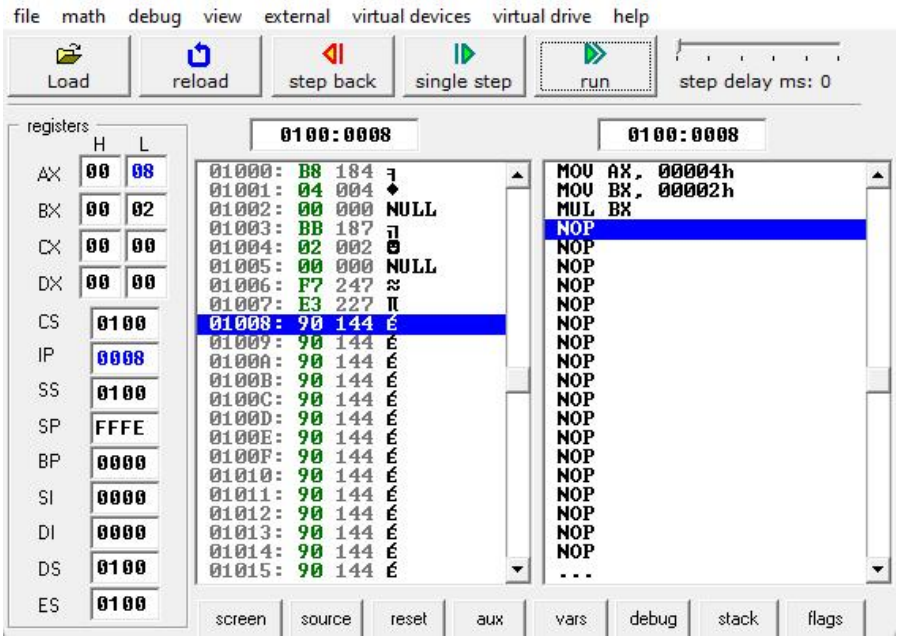
1. Add



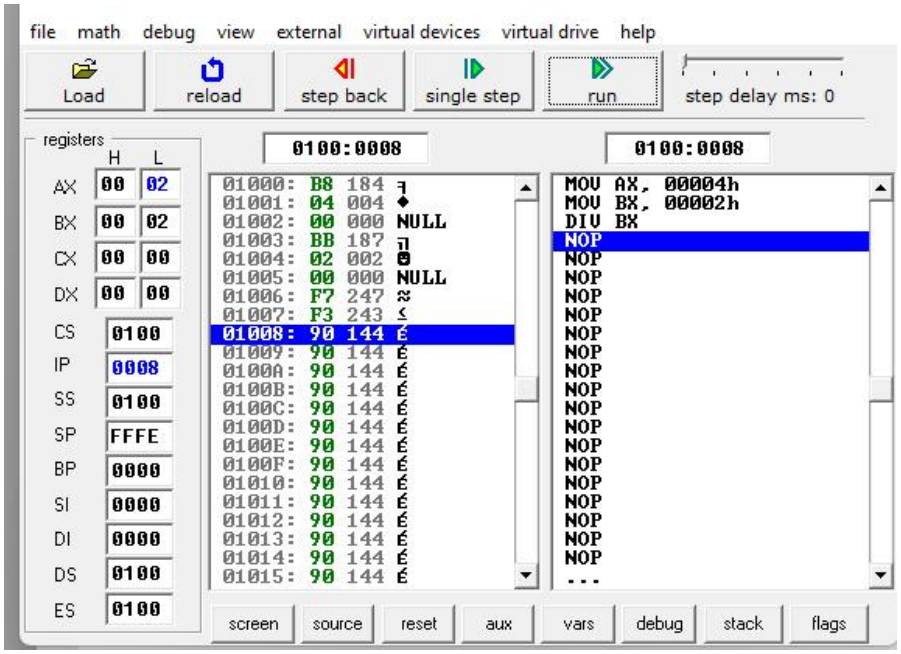

2. Subtract



3. Multiply




4. Division



**Conclusion:**

1. Explain the features of 8086.

Ans. The Intel 8086, a 16-bit microprocessor, revolutionized computing with its features. It boasts a 16-bit architecture, allowing it to process data in 16-bit chunks for enhanced performance. With a clock speed up to 5 MHz, it executes instructions swiftly. Its 20-bit address bus can access up to 1 MB of memory, facilitating large-scale applications. The 8086 supports a rich instruction set, including arithmetic, logical, and control operations, enabling versatile computing tasks. It incorporates powerful segmentation and addressing modes for efficient memory management. Additionally, its compatibility with a wide range of peripherals and software makes it a cornerstone in the evolution of personal computing.

2. Explain general purpose and special purpose registers.

Ans. General-purpose registers (GPRs) are registers within a computer processor that can be used for a wide variety of purposes, hence the name "general-purpose." These registers are typically used to store data temporarily during program execution. They are versatile and can be employed by the programmer for tasks such as arithmetic operations, data manipulation, memory addressing, and holding intermediate results.

Special-purpose registers, on the other hand, are registers that are designed for specific functions within the processor. These registers serve dedicated purposes and often have predefined roles in the execution of instructions or in managing the processor's operation. Examples of special-purpose registers include the program counter (PC), which holds the memory address of the next instruction to be executed; the instruction register (IR), which holds the currently fetched instruction; and the status register (flags register), which stores information about the outcome of arithmetic or logical operations, such as whether the result is zero or negative.

In summary, while general-purpose registers offer flexibility and can be used for various tasks, special-purpose registers serve specific functions critical to the operation of the processor and execution of instructions.

# Vidyavardhini's College of Engineering and Technology

## Department of Artificial Intelligence & Data Science

**AY: 2023-24**

| | | | |
|---|---|---|---|
| **Class:** | | **Semester:** | |
| **Course Code:** | | **Course Name:** | |

| | |
|---|---|
| **Name of Student:** | |
| **Roll No. :** | |
| **Experiment No.:** | |
| **Title of the Experiment:** | |
| **Date of Performance:** | |
| **Date of Submission:** | |

## Evaluation

| Performance Indicator | Max. Marks | Marks Obtained |
|---|---|---|
| Performance | 5 | |
| Understanding | 5 | |
| Journal work and timely submission | 10 | |
| Total | 20 | |

| Performance Indicator | Exceed Expectations (EE) | Meet Expectations (ME) | Below Expectations (BE) |
|---|---|---|---|
| Performance | 4-5 | 2-3 | 1 |
| Understanding | 4-5 | 2-3 | 1 |
| Journal work and timely submission | 8-10 | 5-8 | 1-4 |

**Checked by**

**Name of Faculty :**

**Signature :**

**Date :**

**Aim:** Program for multiplication without using the multiplication instruction.

**Theory:**

In the multiplication program, we multiply the two numbers without using the direct instructions MUL. Here we can successive addition methods to get the product of two numbers. For that, in one register we will take multiplicand so that we can add multiplicand itself till the multiplier stored in another register becomes zero.

**ORG 100H:**

It is a compiler directive. It tells the compiler how to handle source code. It tells the compiler that the executable file will be loaded at the offset of 100H (256 bytes.)

**INT 21H:**

The instruction INT 21H transfers control to the operating system, to a subprogram that handles I/O operations.

**MUL:** MUL Source.
This instruction multiplies an unsigned byte from some source times an unsigned byte in the AL register or an unsigned word from some source times an unsigned word in the AX register.
Source: Register, Memory Location.
When a byte is multiplied by the content of AL, the result (product) is put in AX. A 16-bit destination is required because the result of multiplying an 8-bit number by an 8-bit number can be as large as 16-bits. The MSB of the result is put in AH and the LSB of the result is put in AL.

When a word is multiplied by the contents of AX, the product can be as large as 32 bits. The MSB of the result is put in the DX register and the LSB of the result is put in the AX register.
MUL BH; multiply AL with BH; result in AX.

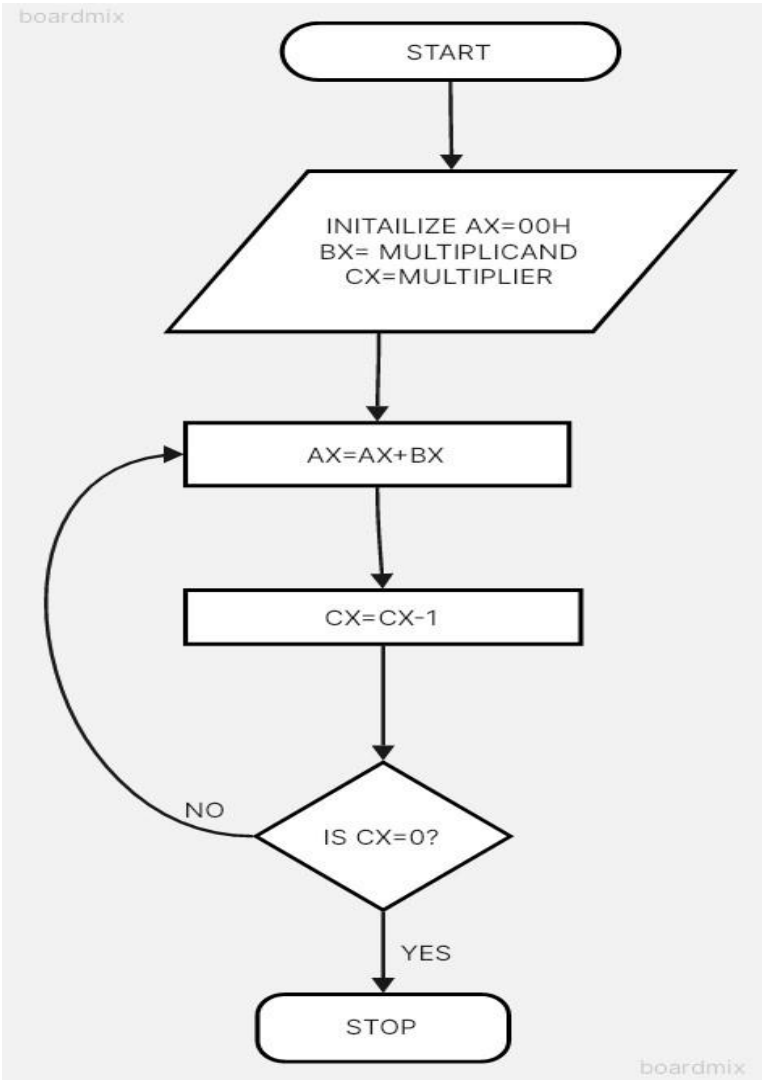
**Algorithm:**

1. Start.
2. Set AX=00H, BX= Multiplicand, CX=Multiplier 3 Add the content of AX and

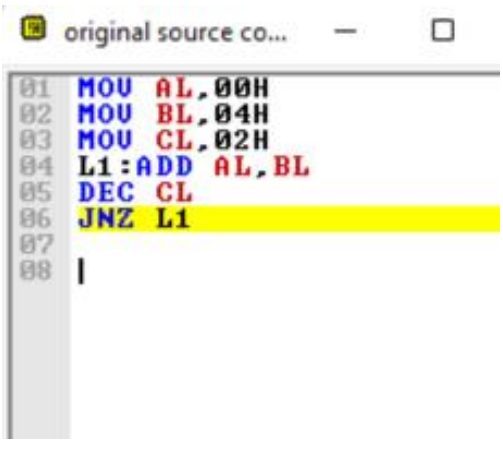
   BX.

4. Decrement content of CX.
5. Repeat steps 3 and 4 till CX=0.
6. Stop.

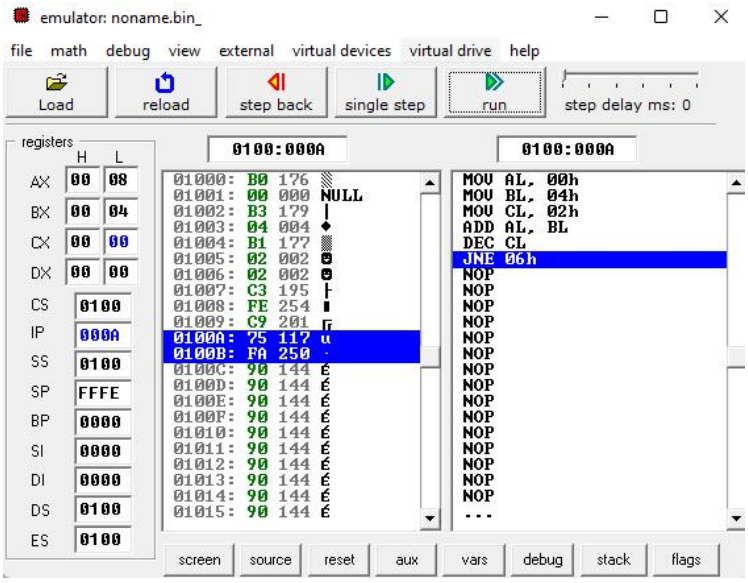
**Flowchart:**




**Assembly Code:**



**Output:**




**Conclusion:**

1. Explain data transfer instructions.

Ans. Data transfer instructions are essential in computer architecture for moving data between memory, registers, and I/O devices. Here's an overview of these instructions:

a) Load (MOV): Copies data from memory or an I/O device into a register, enabling further processing.
b) Store (MOV): Transfers data from a register to memory or an I/O device, preserving it for later use.
c) Exchange (XCHG): Swaps the contents of two registers, aiding in data manipulation or register management.
d) Input and Output (IN, OUT): Reads data from an input port or sends data to an output port, facilitating communication with peripherals.
e) Move String (MOVS): Transfers a block of data from one memory location to another, commonly used in string manipulation operations.

These instructions enable efficient data movement and manipulation, fundamental to program execution in a computer system.

2. Explain Arithmetic instructions.

Ans. Arithmetic instructions are fundamental operations in computer architecture for manipulating numerical data:

a) Addition (ADD): Adds the content of a specified memory location or register to the content of the accumulator, storing the result in the accumulator.
b) Subtraction (SUB): Subtracts the content of a specified memory location or register from the content of the accumulator, storing the result in the accumulator.
c) Multiply (MUL): Multiplies the content of the accumulator by a specified memory location or register, storing the result in the accumulator.
d) Divide (DIV): Divides the content of the accumulator by a specified memory location or register, storing the quotient in the accumulator and the remainder in another specified register.
e) Increment and Decrement (INR, DER): Instructions to increment or decrement the content of a specified register or memory location by one, respectively.

These instructions are essential for performing mathematical operations in programs, making them indispensable for virtually all computing tasks.

# Vidyavardhini's College of Engineering and Technology

## Department of Artificial Intelligence & Data Science

**AY: 2023-24**

| | | | |
|---|---|---|---|
| **Class:** | | **Semester:** | |
| **Course Code:** | | **Course Name:** | |

| | |
|---|---|
| **Name of Student:** | |
| **Roll No. :** | |
| **Experiment No.:** | |
| **Title of the Experiment:** | |
| **Date of Performance:** | |
| **Date of Submission:** | |

## Evaluation

| Performance Indicator | Max. Marks | Marks Obtained |
|---|---|---|
| Performance | 5 | |
| Understanding | 5 | |
| Journal work and timely submission | 10 | |
| Total | 20 | |

| Performance Indicator | Exceed Expectations (EE) | Meet Expectations (ME) | Below Expectations (BE) |
|---|---|---|---|
| Performance | 4-5 | 2-3 | 1 |
| Understanding | 4-5 | 2-3 | 1 |
| Journal work and timely submission | 8-10 | 5-8 | 1-4 |

**Checked by**

**Name of Faculty :**

**Signature :**

**Date :**

**Aim:** Program to calculate the Factorial of a number.

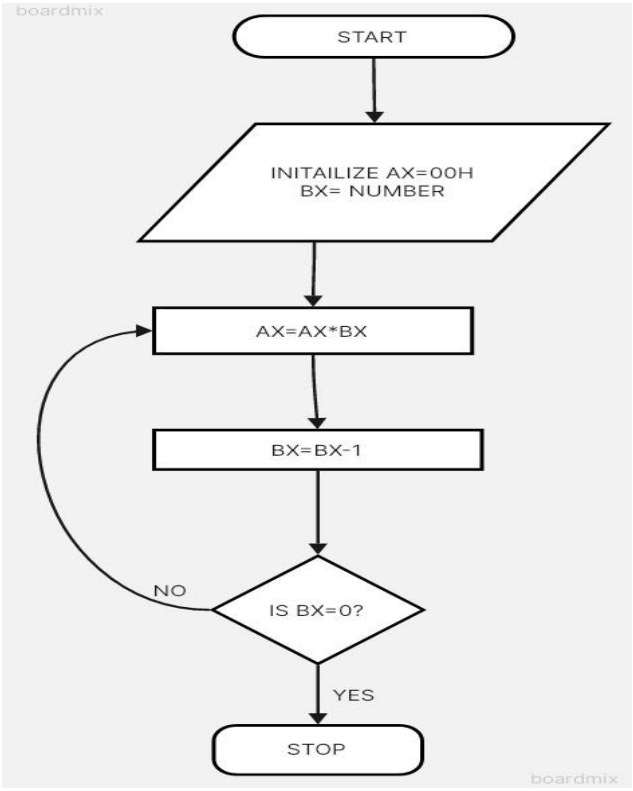
**Theory:**

To calculate the factorial of any number, we use MUL instruction. Here, initially, we initialize the first register by value 1. The second register is initialized by the value of the second register. After multiplication, decrement the value of the second register and repeat the multiplying step till the second register value becomes zero. The result is stored in the first register.

**Algorithm:**

1. Start.
2. Set AX=01H, and BX with the value whose factorial we want to find.
3. Multiply AX and BX.
4. Decrement BX=BX-1.
5. Repeat steps 3 and 4 till BX=0.
6. Stop.

**Flowchart:**

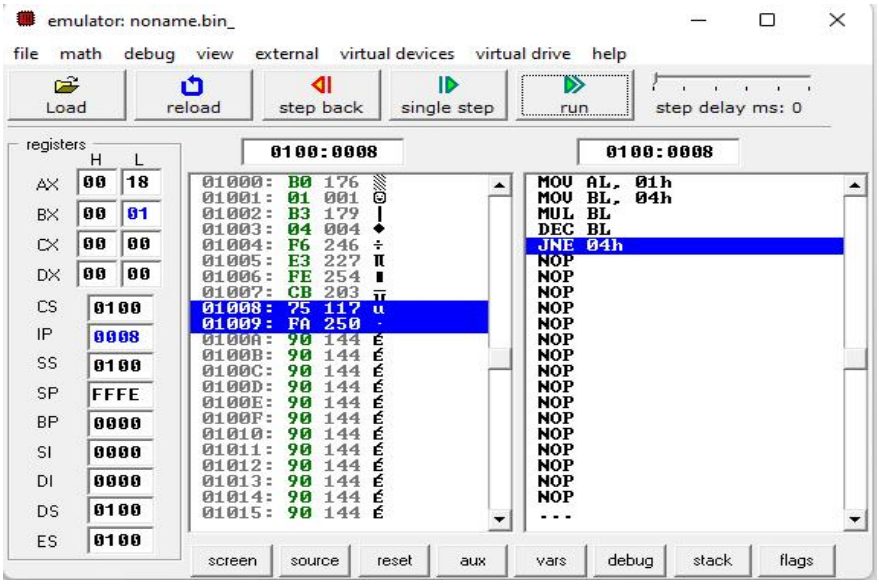



**Assembly Code:**

original source co...

```
01 MOV AL,01H
02 MOV BL,04H
03 L1:MUL BL
04 DEC BL
05 JNZ L1
06
07
```

**Output:**




**Conclusion:**

1. Explain shift instructions.

Ans. Shift instructions are vital operations in computer architecture used to manipulate the binary representation of data by shifting its bits left or right. Here's an explanation of these instructions:

a) Logical Shift Left (SHL): Shifts the bits of a binary number to the left by a specified number of positions, filling the vacant positions with zeros. This operation effectively multiplies the number by powers of 2.

b) Logical Shift Right (SHR): Shifts the bits of a binary number to the right by a specified number of positions, filling the vacant positions with zeros. This operation effectively divides the number by powers of 2.

c) Arithmetic Shift Left (SAL): Similar to logical shift left, but in arithmetic shift left, the most significant bit (MSB) is shifted into the carry flag, and the least significant bit (LSB) is set to zero. It preserves the sign of signed numbers.

d) Arithmetic Shift Right (SAR): Similar to logical shift right, but in arithmetic shift right, the MSB is shifted into the carry flag, and the vacant positions are filled with copies of the original MSB, preserving the sign of signed numbers.

2. Explain rotate instructions.

Ans. Rotate instructions are essential operations in computer architecture that manipulate the binary representation of data by circularly shifting its bits left or right. Here's an explanation of these instructions:

a) Rotate Left (ROL): Circularly shifts the bits of a binary number to the left by a specified number of positions. The shifted-out bits are rotated back to the rightmost positions. This operation is useful for creating circular patterns in binary data.

b) Rotate Right (ROR): Circularly shifts the bits of a binary number to the right by a specified number of positions. The shifted-out bits are rotated back to the leftmost positions. This operation is helpful for circular buffer implementations and encryption algorithms.

c) Rotate Through Carry Left (RCL): Similar to ROL, but includes the carry flag in the rotation. The carry flag is shifted into the least significant bit position, and the most significant bit is shifted into the carry flag.

# Vidyavardhini's College of Engineering and Technology

## Department of Artificial Intelligence & Data Science

**AY: 2023-24**

| | | | |
|---|---|---|---|
| **Class:** | | **Semester:** | |
| **Course Code:** | | **Course Name:** | |

| | |
|---|---|
| **Name of Student:** | |
| **Roll No. :** | |
| **Experiment No.:** | |
| **Title of the Experiment:** | |
| **Date of Performance:** | |
| **Date of Submission:** | |

## Evaluation

| Performance Indicator | Max. Marks | Marks Obtained |
|---|---|---|
| Performance | 5 | |
| Understanding | 5 | |
| Journal work and timely submission | 10 | |
| Total | 20 | |

| Performance Indicator | Exceed Expectations (EE) | Meet Expectations (ME) | Below Expectations (BE) |
|---|---|---|---|
| Performance | 4-5 | 2-3 | 1 |
| Understanding | 4-5 | 2-3 | 1 |
| Journal work and timely submission | 8-10 | 5-8 | 1-4 |

**Checked by**

**Name of Faculty :**

**Signature :**

**Date :**

**Aim:** Program for drawing square using Assembly Language.

**Theory:** INT 10h is a video service bios interrupt. It includes services like setting the video mode, character and string output and reading and writing pixels in graphics mode. To use the BIOS interrupt load ah with the desired sub-function. Load other required parameters in other registers and make a call to INT 10h.

INT 10h/AH = 0ch -Write graphics pixel.

**Input:**
AL = pixel colour
CX = column
DX = row

**Algorithm:**

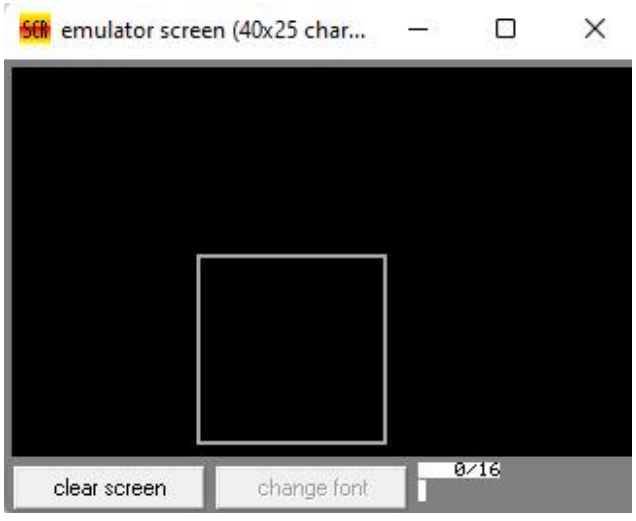
1. Start
2. Initialize ax to 0013h for graphics mode.
3. Set the Counter bx to 60 h.
4. Initialize the co-ordinates cx and dx to 60h.
5. Set the Color.
6. Set Display Mode function by making ah = 0ch.
7. Increment cx and Decrement bx.
8. Repeat step 7 until bx = 0.
9. Initialize the counter by making bx = 60h.
10. Set the color.
11. Set Display Mode function by making ah = 0ch.
12. Increment dx & Decrement bx.
13. Repeat step 12 until bx = 0.
14. Initialize the counter by making bx = 60h.
15. Set the Color.
16. Set Display Mode function by making ah = 0ch.
17. Decrement cx and Decrement bx.
18. Repeat step 17 until bx = 0.
19. Initialize the counter by making bx = 60h.
20. Set the color.

21. Set Display Mede function by making ah = 0ch.
22. Decrement dx & Decrement bx.
23. Repeat step 22 until bx = 0.
24. To end the program use DOS interrupt:
    1) Load ah = 4ch.
    2) Call int 21h.
25. Stop.

**Assembly Code:**

```
INC DX
DEC BX
INT 10H
JNZ L2
MOV BX,60H
L3: MOV AH,0ch
DEC CX
DEC BX
INT 10H
JNZ L3
MOV BX,60H
L4: MOV AH,0ch
DEC DX
DEC BX
INT 10H
JNZ L4
```

**Output:**



**Conclusion:**

1. Explain the use of int 10.

Ans. The `INT 10h` instruction is a software interrupt in x86 assembly language that serves as a gateway to BIOS video services. It's commonly used for controlling video display output on IBM-compatible PCs. Here's how it's used:

a) Video Mode Setting: One primary use of `INT 10h` is to set the video mode, which determines the resolution and color depth of the display. Different modes offer various configurations suitable for text mode, graphics mode, or specific display resolutions.
b) Cursor Control: `INT 10h` allows for manipulation of the cursor position and its appearance on the screen. This includes setting the cursor's position, hiding or showing the cursor, and changing its shape.
c) Character and String Output: `INT 10h` facilitates the output of characters and strings to the screen. This includes writing characters directly to specific locations on the screen or printing strings of characters.
d) Color and Attribute Control: It enables setting the foreground and background colors of characters displayed on the screen, as well as controlling attributes like blinking and brightness.
e) Scrolling: `INT 10h` allows for scrolling the contents of the screen both vertically and horizontally.
f) Palette Control (in some cases): In certain video modes, `INT 10h` provides functionality for manipulating the color palette used for graphics display.

In summary, `INT 10h` is a versatile instruction that provides access to a wide range of video-related functions, making it essential for controlling display output in DOS and BIOS-based systems.

2. Explain hardware interrupts.

Ans. Hardware interrupts are signals generated by hardware components to request attention from the CPU. Here's an explanation of hardware interrupts:

a) Purpose: Hardware interrupts are used by peripherals like keyboards, mice, disks, and network adapters to notify the CPU that they require servicing.
b) Types: There are several types of hardware interrupts, including:
- Maskable Interrupts: Can be disabled or enabled by the CPU.
- Non-Maskable Interrupts (NMI): Cannot be disabled and usually signify critical system errors.
- External Interrupts: Generated by external devices connected to the CPU.
- Internal Interrupts: Generated by internal CPU events like divide-by-zero errors or page faults.

a) Handling: When a hardware interrupt occurs, the CPU interrupts its current execution, saves the context, and jumps to an interrupt service routine (ISR) associated with the interrupt. The ISR then services the interrupt, which may involve reading data from or writing data to the device, acknowledging the interrupt, and restoring the CPU's previous context to resume execution of the interrupted program.
b) Priority: Interrupts may have different priority levels, and the CPU typically services higher priority interrupts before lower priority ones to ensure timely response to critical events.
c) IRQs: In PC architecture, hardware interrupts are often managed through interrupt request lines (IRQs), with each device connected to a specific IRQ line. The CPU communicates with devices through these lines to determine which device is requesting service.

Hardware interrupts are essential for real-time processing and efficient utilization of system resources in modern computer systems.

# Vidyavardhini's College of Engineering and Technology

## Department of Artificial Intelligence & Data Science

**AY: 2023-24**

| | | | |
|---|---|---|---|
| **Class:** | | **Semester:** | |
| **Course Code:** | | **Course Name:** | |

| | |
|---|---|
| **Name of Student:** | |
| **Roll No. :** | |
| **Experiment No.:** | |
| **Title of the Experiment:** | |
| **Date of Performance:** | |
| **Date of Submission:** | |

## Evaluation

| Performance Indicator | Max. Marks | Marks Obtained |
|---|---|---|
| Performance | 5 | |
| Understanding | 5 | |
| Journal work and timely submission | 10 | |
| Total | 20 | |

| Performance Indicator | Exceed Expectations (EE) | Meet Expectations (ME) | Below Expectations (BE) |
|---|---|---|---|
| Performance | 4-5 | 2-3 | 1 |
| Understanding | 4-5 | 2-3 | 1 |
| Journal work and timely submission | 8-10 | 5-8 | 1-4 |

**Checked by**

**Name of Faculty :**

**Signature :**

**Date :**

**Aim**: Assembly Language Program to display character A to z in both uppercase and lowercase

**Theory**:

DOS provide various interrupt services that are used by the system programmer. The most commonly used interrupt is INT 21H. It invokes inbuilt DOS functions which can be used to perform various tasks. The most common tasks are reading a user input character from the screen, displaying result on the exiding program etc.

In this program, we display the characters A to Z on the DOS prompt. DOS interrupt function 02 displays the contents of DL (ASCII code) on the screen. By loading the ASCII code of 'A' in the DL register, loading AH register with 02h and calling INT 21h it is possible to display character from A to Z on the screen.

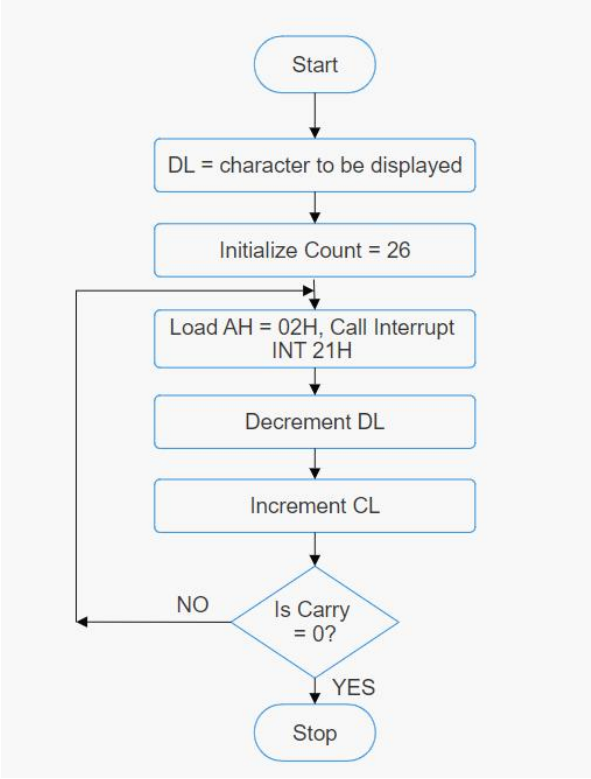
INT 21h/AH = 2 - write character to standard output.

Entry: DL = character to write, after execution AL = DL.

Example :-

```
mov ah , 2

mov dl , 'a'

int 21h
```

**Flowchart**:



**Algorithm:**

1. Start.
2. Initialize DL with 'A'.
3. Load CL with count = 26.
4. Load AH = 02H and call INT 21H.
5. Increment DL, to next character.
6. Decrement the count.
7. Repeat steps 4,5,6 till CL is not zero.
8. To end the program use DOS interrupt:
   1) Load AH = 41H.
   2) Call INT 21 H.
9. Stop.

**Assembly Code:**

```
mov cx,26
mov dl,'a'
L1: mov ah,02h
int 21h
inc dl
dec cx
JNZ L1

mov dl,0ah
int 21h
mov dl,0dh
int 21h

mov cx,26
mov dl,'A'
L2: mov ah,02h
int 21h
inc dl
dec cx
JNZ L2

mov dl,0ah
int 21h
mov dl,0dh
int 21h

mov cx,9
mov dl,'1'
L3: mov ah,02h
int 21h
inc dl
dec cx
JNZ L3
```

**Output:**

```
abcdefghijklm
ABCDEFGHIJKLM
123456789
```

**Conclusion:**

1. Explain INT 21H.

Ans. INT 21h is a software interrupt in x86 assembly language, primarily used in DOS (Disk Operating System) environments. It provides access to various DOS services, such as file operations, input/output, and system functions. Here's an explanation of INT 21h:

a) Purpose: INT 21h serves as a gateway to DOS services, allowing programmers to perform various tasks related to file management, input/output operations, and system control.
b) Functionality: The specific functionality accessed through INT 21h depends on the value passed in the AH (accumulator high) register before invoking the interrupt. Different values in AH correspond to different DOS services.
c) Common Services:
   - File Operations: Creating, opening, closing, reading from, and writing to files.
   - Directory Operations: Creating, removing, and navigating directories.
   - Input/Output: Reading characters from the keyboard, displaying characters on the screen, and handling input/output redirection.
d) Example Usage: For instance, to print a character to the screen, the programmer would load the character into the DL register and set AH to the appropriate value for the print function (e.g., AH = 02h for printing a character). Then, the programmer would invoke the INT 21h interrupt, which would cause DOS to perform the requested operation.
e) Limitations: While INT 21h provides convenient access to DOS services, it is specific to DOS environments and may not be available in other operating systems or modern computing environments. INT 21h remains a crucial mechanism for interacting with DOS services and is commonly used in legacy software and DOS emulators.

2. Explain working of increment and decrement instructions.

Ans. Increment and decrement instructions are fundamental operations in computer architecture used to increase or decrease the value stored in a register or memory location by one. Here's how they work:

a) Increment (INC):
   a. Register Mode: The content of the specified register is incremented by one.
   b. Memory Mode: The content of the memory location addressed by the specified memory address is incremented by one.
   c. After the increment operation, the zero flag (ZF) is set if the result is zero, and the sign flag (SF) is set based on the result's sign.
b) Decrement (DEC):
   a. Register Mode: The content of the specified register is decremented by one.
   b. Memory Mode: The content of the memory location addressed by the specified memory address is decremented by one.
   c. After the decrement operation, the zero flag (ZF) is set if the result is zero, and the sign flag (SF) is set based on the result's sign.
c) Overflow and Carry Flags:
   a. These instructions do not affect the overflow (OF) or carry (CF) flags, which are used to indicate arithmetic overflow or carry conditions.
   b. Overflow occurs when the result of an arithmetic operation cannot be represented using the available number of bits.
   c. Carry occurs when an addition operation generates a carry-out or a subtraction operation requires borrowing.

# Vidyavardhini's College of Engineering and Technology

## Department of Artificial Intelligence & Data Science

**AY: 2023-24**

| | | | |
|---|---|---|---|
| **Class:** | | **Semester:** | |
| **Course Code:** | | **Course Name:** | |

| | |
|---|---|
| **Name of Student:** | |
| **Roll No. :** | |
| **Experiment No.:** | |
| **Title of the Experiment:** | |
| **Date of Performance:** | |
| **Date of Submission:** | |

## Evaluation

| Performance Indicator | Max. Marks | Marks Obtained |
|---|---|---|
| Performance | 5 | |
| Understanding | 5 | |
| Journal work and timely submission | 10 | |
| Total | 20 | |

| Performance Indicator | Exceed Expectations (EE) | Meet Expectations (ME) | Below Expectations (BE) |
|---|---|---|---|
| Performance | 4-5 | 2-3 | 1 |
| Understanding | 4-5 | 2-3 | 1 |
| Journal work and timely submission | 8-10 | 5-8 | 1-4 |

**Checked by**

**Name of Faculty :**

**Signature :**

**Date :**

**Aim:** Program to display string in Lowercase.

**Theory:**

The program will take Uppercase string as input and convert it to lowercase string. Int 21h is a DOS interrupt. To use the DOS interrupt 21h load with the desired sub-function. Load other required parameters in other registers and make a call to INT 21h.

INT 21h/AH = 9

output of string at DS: ・ String mest be terminated by "'$"

example :

```
org 100h

mov dx, offset msg

mov ah, 9

int 21h
ret

msg db "hello world $"
```

INT 21h/AH = 0AH – input of string to DS:DX, first byte is buffer size, second byte is number of chars actually read this function does not add '$' in the end of string to print using INT 21h/AH = 9 you must set dollar character at the end of it and start printing from address DS : DX + 2. The function does not allow to enter more characters than the specified buffer size.

**Algorithm:**

1. Start.
2. Initialize the Data Segment.
3. Display message -1.
4. Input the string.
5. Display message-2.
6 Take the character count in CX.
7. Point to the first character.
8. Convert it to Lowercase.
9. Display the character.
10. Decrement the character coun.

11. If not Zero, repeat from step 6.
12. To terminate the program, using the DOS interrupt:
    1) Initialize AH with 4CH
    2) Call interrupt INT 21H.
13. Stop.

**Assembly Code:**

original source code

```
12
13 .code
14 lea dx,m1
15 mov ah,09h
16 int 21h
17
18 lea dx,BUFF
19 mov ah,0ah
20 int 21h
21
22 lea dx,m2
23 mov ah,09h
24 int 21h
25
26 mov cl,[BUFF+1]
27 lea bx,BUFF+2
28
29 L1: mov dx,[bx]
30 add dx,20h
31 mov ah,02h
32 int 21h
33 inc bx
34 LOOP L1
35
```

**Output:**

emulator screen (80x25 chars)

```
Enter String:HELLO
The Enter String is:hello
```

**Conclusion:**

1. Explain instruction AAA.

Ans. The AAA instruction, standing for "ASCII Adjust After Addition," is a processor instruction used in x86 assembly language to adjust the result of a binary-coded decimal (BCD) addition operation. Here's an explanation of the AAA instruction:

a) Purpose: AAA is specifically designed to adjust the result in the AL register after adding two unpacked BCD digits together. It adjusts the AL register to represent a valid BCD result in the range of 0 to 9, and it also adjusts the flags accordingly
b) Usage: AAA typically follows an ADD instruction that adds two BCD digits together. After the addition operation, AAA examines the lower 4 bits (nibble) of the AL register. If this nibble contains a value between 0 and 9, indicating a valid BCD result, no adjustment is needed. However, if the nibble contains a value between 10 and 15, indicating an invalid BCD result, AAA adjusts the AL register, adds 6 to the result, and sets the carry flag (CF) and auxiliary carry flag (AF).
c) Example: For example, if AL contains the value 0x13 after an addition operation (which is invalid in BCD), AAA will adjust AL to 0x19 and set the carry and auxiliary carry flags.
d) Flags: AAA affects the carry flag (CF) and auxiliary carry flag (AF) to indicate the result's validity and adjustment.

In summary, AAA is used to adjust the result of a BCD addition operation in the AL register, ensuring that it represents a valid BCD digit.

2. Explain instruction AAS.

Ans. The AAS instruction, short for "ASCII Adjust After Subtraction," is an x86 assembly language instruction used to adjust the result of a binary-coded decimal (BCD) subtraction operation. Here's an explanation of the AAS instruction:

a) Purpose: AAS is specifically designed to adjust the result in the AL register after subtracting one BCD digit from another. It ensures that the AL register represents a valid BCD result in the range of 0 to 9 and adjusts the flags accordingly.
b) Usage: AAS is typically used after a subtraction operation involving two BCD digits. After the subtraction, AAS examines the lower 4 bits (nibble) of the AL register. If this nibble contains a value between 0 and 9, indicating a valid BCD result, no adjustment is needed. However, if the nibble contains a value between 10 and 15, indicating an invalid BCD result, AAS adjusts the AL register, subtracts 6 from the result, and sets the carry flag (CF) and auxiliary carry flag (AF).
c) Example: For instance, if AL contains the value 0xFA after a subtraction operation (which is invalid in BCD), AAS will adjust AL to 0x00 and set the carry and auxiliary carry flags.
d) Flags: AAS affects the carry flag (CF) and auxiliary carry flag (AF) to indicate the result's validity and adjustment.

In summary, AAS is used to adjust the result of a BCD subtraction operation in the AL register, ensuring that it represents a valid BCD digit.

# Vidyavardhini's College of Engineering and Technology

## Department of Artificial Intelligence & Data Science

**AY: 2023-24**

| | | | |
|---|---|---|---|
| **Class:** | | **Semester:** | |
| **Course Code:** | | **Course Name:** | |

| | |
|---|---|
| **Name of Student:** | |
| **Roll No. :** | |
| **Experiment No.:** | |
| **Title of the Experiment:** | |
| **Date of Performance:** | |
| **Date of Submission:** | |

## Evaluation

| Performance Indicator | Max. Marks | Marks Obtained |
|---|---|---|
| Performance | 5 | |
| Understanding | 5 | |
| Journal work and timely submission | 10 | |
| Total | 20 | |

| Performance Indicator | Exceed Expectations (EE) | Meet Expectations (ME) | Below Expectations (BE) |
|---|---|---|---|
| Performance | 4-5 | 2-3 | 1 |
| Understanding | 4-5 | 2-3 | 1 |
| Journal work and timely submission | 8-10 | 5-8 | 1-4 |

**Checked by**

**Name of Faculty :**

**Signature :**

**Date :**

**Aim:** Assembly Language Program to reverse the word in string.

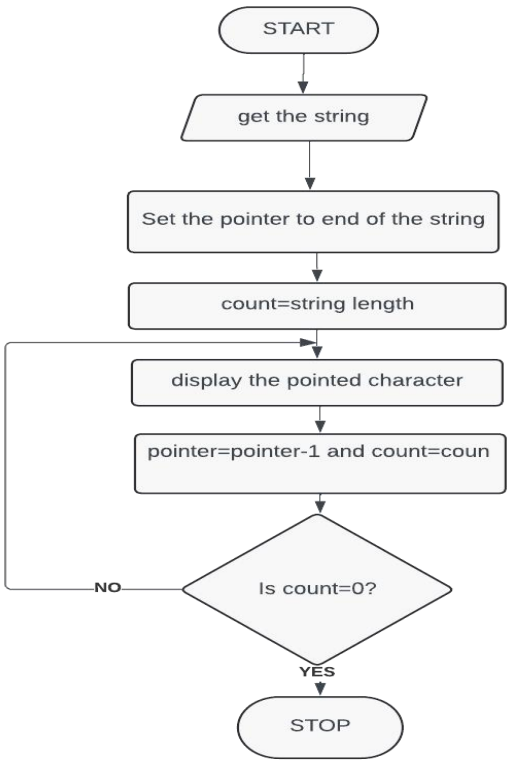
**Theory:**

This program will read the string entered by the user and then reverse it. Reverse a string is the technique that reverses or changes the order of a given string so that the last character of the string becomes the first character of the string and so on.

**Algorithm:**

1. Start
2. Initialize the data segment
3. Display the message -1
4. Input the string
5. Display the message 2
6. Take characters count in DI
7. Point to the end character and read it
8. Display the character
9. Decrement the count
10. Repeat until the count is zero
11. To terminate the program using DOS interrupt
    a. Initialize AH with 4ch
    b. Call interrupt INT 21h
12. Stop

**Flowchart:**



## Assembly Code:

original source code

```
01
02 ; You may customize this and ot
03 ; The location of this template
04
05 org 100h
06
07 .data
08 m1 db 10,13,'enter string: $'
09 m2 db 10,13,'your string is: $'
10 buff db 80
11 .code
12 lea dx,m1
13 mov ah,09h
14 int 21h
15
16 lea dx,buff
17 mov ah,0ah
18 int 21h
19
20 lea dx,m2
21 mov ah,09h
22 int 21h
23
24 mov ch,00h
25 mov cl,[buff+1]
26 lea bx,buff+1
27 add bx,cx
28
29 L1:
30 mov dx,[bx]
31 mov ah,02h
32 int 21h
33 dec bx
34 Loop L1
35
```

## Output:

emulator screen (191x63 chars)

```
enter string: yo
your string is: oy
```

## Conclusion:

1. Explain the difference between XLAT and XLATB

Ans. XLAT and XLATB are x86 assembly language instructions used for table lookups, but they have some differences:

a) XLAT:
   a. XLAT is an instruction that performs a lookup in a translation table.
   b. It uses the AL register as an index to access a byte-sized entry in a translation table, typically located in memory.
   c. After the lookup, XLAT replaces the content of AL with the byte value found at the memory address calculated using the sum of the contents of AL and the base address of the translation table.

b) XLATB:
   a. XLATB is essentially the same as XLAT, but it's a legacy alternative mnemonic for the same instruction. The "B" suffix in XLATB stands for "byte."
   b. In terms of functionality, XLATB operates exactly like XLAT and performs the same table lookup using the AL register as an index.

In summary, both XLAT and XLATB instructions perform byte-sized table lookups using the content of the AL register as an index. The difference lies in their naming convention, with XLAT being the more commonly used mnemonic, while XLATB is a legacy alternative. The choice between them is mostly a matter of programmer preference.

2. Explain the instruction LAHF.

Ans. The LAHF instruction, standing for "Load AH from Flags," is an x86 assembly language instruction used to load the contents of the flags register into the AH register. Here's an explanation of the LAHF instruction:

a) Purpose:
   a. LAHF is used to transfer the status of the lower byte of the flags register (EFLAGS) into the AH register.
   b. The flags register contains various status flags that reflect the outcome of arithmetic and logical operations, such as the zero flag (ZF), sign flag (SF), carry flag (CF), and others.

b) Usage:
   a. LAHF is typically used in conjunction with the SAHF instruction (Store AH into Flags), which performs the opposite operation.
   b. Before using LAHF, the programmer often saves the contents of the AH register if it holds important data, as LAHF will overwrite its contents.
   c. After executing LAHF, the AH register contains the lower byte of the flags register, where each bit corresponds to a specific flag's status.

c) Example:
   a. For instance, if the carry flag (CF) and the zero flag (ZF) are set in the flags register, the value loaded into AH by LAHF would be 00000011 in binary, corresponding to a hexadecimal value of 0x03.

d) Flags Represented:
   a. The bits in the AH register represent the status of the following flags (from bit 0 to bit 7): CF, reserved, PF, reserved, AF, reserved, ZF, SF.

In summary, the LAHF instruction is used to load the lower byte of the flags register into the AH register, providing access to the status of various processor flags for further manipulation or decision-making in assembly language programs.

# Vidyavardhini's College of Engineering and Technology

## Department of Artificial Intelligence & Data Science

**AY: 2023-24**

| | | | |
|---|---|---|---|
| **Class:** | | **Semester:** | |
| **Course Code:** | | **Course Name:** | |

| | |
|---|---|
| **Name of Student:** | |
| **Roll No. :** | |
| **Experiment No.:** | |
| **Title of the Experiment:** | |
| **Date of Performance:** | |
| **Date of Submission:** | |

## Evaluation

| Performance Indicator | Max. Marks | Marks Obtained |
|---|---|---|
| Performance | 5 | |
| Understanding | 5 | |
| Journal work and timely submission | 10 | |
| Total | 20 | |

| Performance Indicator | Exceed Expectations (EE) | Meet Expectations (ME) | Below Expectations (BE) |
|---|---|---|---|
| Performance | 4-5 | 2-3 | 1 |
| Understanding | 4-5 | 2-3 | 1 |
| Journal work and timely submission | 8-10 | 5-8 | 1-4 |

**Checked by**

**Name of Faculty :**

**Signature :**

**Date :**

**Aim:** Assembly Language Program to find given string is Palindrome or not**.**

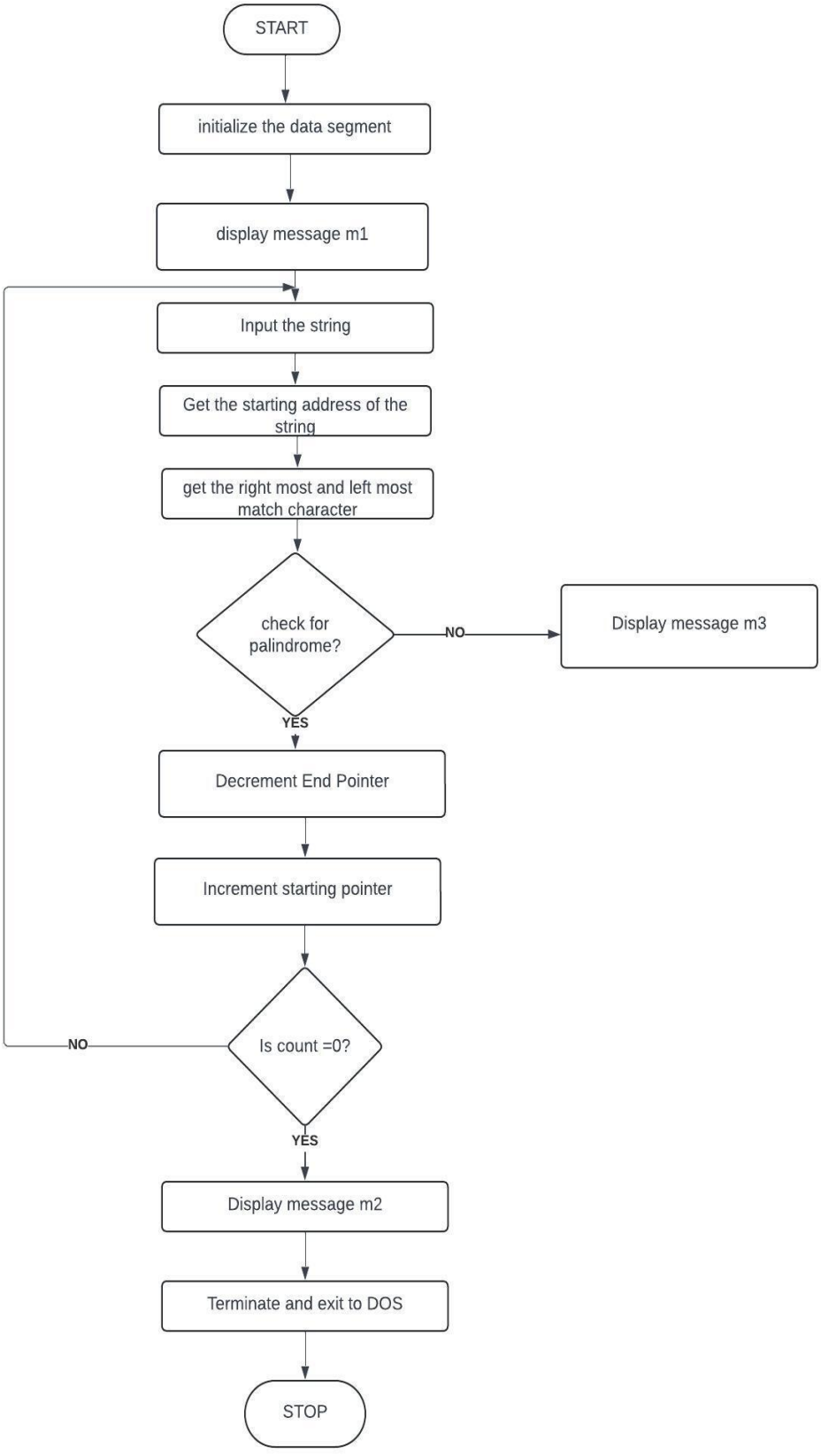
**Theory:**

A palindrome string is a string when read in a forward or backward direction remains the same. One of the approach to check this is iterate through the string till middle of the string and compare the character from back and forth.

**Algorithm:**

13. Initialize the data segment.
14. Display the message M1
15. Input the string
16. Get the string address of the string
17. Get the right most character
18. Get the left most character
19. Check for palindrome.
20. If not Goto step 14
21. Decrement the end pointer
22. Increment the starting pointer.
23. Decrement the counter
24. If count not equal to zero go to step 5
25. Display the message m2
26. Display the message m3
27. To terminate the program using DOS interrupt
    c. Initialize AH with 4ch
    d. Call interrupt INT 21h
28. Stop

**Flowchart :**


START
initialize the data segment
display message m1
Input the string
Get the starting address of the string
get the right most and left most match character
check for palindrome?
NO
Display message m3
YES
Decrement End Pointer
Increment starting pointer
Is count =0?
NO
YES
Display message m2
Terminate and exit to DOS
STOP

**Assembly Code:**

```
; You may customize thi
; The location of this
org 100h

.data
m1 db 10,13,'enter stri
m2 db 10,13,'your strin
m3 db 10,13,'your strin

buff db 80
.code
lea dx,m1
mov ah,09h
int 21h

lea dx,buff
mov ah,0ah
int 21h

lea bx,buff+1
mov ch,00h
mov cl,[buff+1]
mov di,cx
mov si,01h
sar cl,01h


pal:
mov al,[bx+di]
mov ah,[bx+si]
cmp ah,al
JC L1
inc si
dec di
loop pal


lea dx,m2
mov ah,09h
int 21h
jmp L2

L1:
lea dx,m3
mov ah,09h
int 21h
jmp L2

L2:
mov ah,4ch
int 21h

ret
```

**Output:**

```
enter string: NAMAN
your string is a palindrome
```

## Conclusion:

1. Explain SAR INSTRUCTION

Ans. The SAR (Shift Arithmetic Right) instruction is a fundamental operation in x86 assembly language used to perform a signed right shift operation on binary data. Here's an explanation of the SAR instruction:

a) Purpose: SAR is used for arithmetic right shifts on binary numbers, shifting all bits to the right by a specified number of positions while preserving the sign of the original number. It's commonly used for division by powers of 2 and signed number manipulation.
b) Usage: SAR typically takes two operands: the data to be shifted and the count specifying the number of bit positions to shift by. The content of the specified register or memory location is then shifted to the right by the specified count.
c) Sign Preservation: SAR ensures that the sign bit, the leftmost bit or the most significant bit, is preserved during the shift operation. This preserves the sign of the number, ensuring that positive numbers remain positive and negative numbers remain negative after the shift.
d) Flags: SAR affects various flags such as overflow (OF), zero (ZF), sign (SF), and parity (PF) flags based on the result of the shift operation.
e) Example: SAR applied to the binary number 10101100 (representing -84 in two's complement notation) by one position results in 11010110 (representing -42), preserving the sign bit during the shift.

2. Explain DAA instruction.

Ans. The DAA (Decimal Adjust for Addition) instruction is a crucial operation in x86 assembly language, primarily used to correct the result of addition operations involving unpacked decimal numbers stored in the AL register. Here's a breakdown of its functionality:

a) Purpose: DAA ensures that the result of an addition operation on BCD (Binary Coded Decimal) numbers in the AL register remains within the valid range of 0 to 9 for each decimal digit.
b) Usage: Following an addition operation involving BCD digits, DAA is applied to adjust the result in the AL register, making it suitable for further BCD operations or output.
c) Algorithm: DAA examines the lower nibble of AL and the auxiliary carry flag (AF) to determine if adjustments are necessary. It adds 6 to AL if the lower nibble is greater than 9 or if AF is set. Additionally, if the upper nibble is greater than 9 or if the carry flag (CF) is set, DAA adds 96 to AL.
d) Flags: DAA may modify the carry flag (CF) and auxiliary carry flag (AF) based on the adjustments performed during the operation.
e) Example: If AL contains 0x15 after adding two BCD digits, DAA adjusts it to 0x21, correcting the result to the proper BCD representation of 21. Overall, DAA ensures accurate BCD arithmetic by correcting the result of addition operations to comply with BCD formatting requirements.

# Vidyavardhini's College of Engineering and Technology

## Department of Artificial Intelligence & Data Science

**AY: 2023-24**

| | | | |
|---|---|---|---|
| **Class:** | | **Semester:** | |
| **Course Code:** | | **Course Name:** | |

| | |
|---|---|
| **Name of Student:** | |
| **Roll No. :** | |
| **Experiment No.:** | |
| **Title of the Experiment:** | |
| **Date of Performance:** | |
| **Date of Submission:** | |

## Evaluation

| Performance Indicator | Max. Marks | Marks Obtained |
|---|---|---|
| Performance | 5 | |
| Understanding | 5 | |
| Journal work and timely submission | 10 | |
| Total | 20 | |

| Performance Indicator | Exceed Expectations (EE) | Meet Expectations (ME) | Below Expectations (BE) |
|---|---|---|---|
| Performance | 4-5 | 2-3 | 1 |
| Understanding | 4-5 | 2-3 | 1 |
| Journal work and timely submission | 8-10 | 5-8 | 1-4 |

**Checked by**

**Name of Faculty :**

**Signature :**

**Date :**

**Aim:** Mixed language program for adding two numbers.

**Theory:**

C generates an object code that is extremely fast and compact but it is not as fast as the object code generated by a good programmer using assembly language. The time needed to write a program in assembly language is much more than the time taken in higher level languages like C.

However, there are special cases wher a function is coded in assembly language to reduce the execution time.

Eg: The floating point math package must be loaded assembly language as it is used frequently and its execution speed will have great effect on the overall speed of the program that uses it.

There are also situations in which special hardware devices need exact timing and it is must to write a program in assembly language to meet this strict timing requirement. Certain instructions cannot be executed by a C program

Eg: There is no built in bit wise rotate operation in C. To efficiently perform this it is necessary to use assembly language routine.

Inspite of C being very powerful , routines must be written in assembly language to:

1.Increase the speed and efficiency of the routine

2.Perform machine specific function not available in Microsoft C or Turbo C.

3. Use third party routines

Combining C and assembly:

Built-In-Inline assembles is used to include assembly language routines in C program without any need for a specific assembler.

Such assembly language routines are called in-line assembly.

They are compiled right along with C routines rather than being assembled separately and then linked together using linker modules provided by the C compiler.

Turbo C has inline assembles.

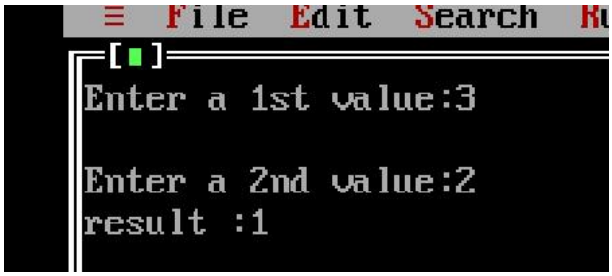
In mixed language program, prefix the keyword asm for a function and write Assembly instruction in the curly braces in a C program

**Assembly Code:**

```
#include <stdio.h>

#include <conio.h>

int main(){

int a,b,c;

clrscr();

printf("Enter a 1st value:");

scanf("%d",&a);

printf("\nEnter a 2nd value:");

scanf("%d",&b);
```

```
asm{

mov ax,a

mov bx,b

SUB ax,bx

mov c,ax

}

printf("result :%d",c);

return 0;

}
```

**Output:**




**Conclusion:**

1. Explain any 2 branch instructions.

Ans. Branch instructions are crucial in assembly language programming as they enable conditional and unconditional branching, altering the program flow based on certain conditions or criteria. Here are explanations of two commonly used branch instructions:

a) JMP (Jump):
   a. JMP is an unconditional branch instruction that directs the CPU to jump to a specified memory location without any condition evaluation.
   b. It allows altering the program flow unconditionally, transferring control to the target instruction specified by the operand.
   c. Syntax: JMP destination
   d. Example: JMP label directs the CPU to jump to the instruction labeled "label" without any condition evaluation.

b) JE (Jump if Equal):
   a. JE is a conditional branch instruction that performs a jump to a specified memory location if the Zero Flag (ZF) is set, indicating that the previous comparison resulted in equality.
   b. It's commonly used in conjunction with comparison operations to implement conditional branches based on equality.
   c. Syntax: JE destination
   d. Example: CMP AX, BX followed by JE label would jump to the instruction labeled "label" if the contents of AX and BX registers are equal.

These instructions are fundamental for implementing decision-making and loop constructs in assembly language programs, enabling efficient control flow management.

2. Explain the syntax of loop.

Ans. The LOOP instruction is a handy construct in assembly language for implementing loops. Here's the syntax of the LOOP instruction:

a) Syntax:

LOOP destination

b) Explanation:

- LOOP: This is the mnemonic for the loop instruction. It's followed by the destination, which is the label or memory address where the CPU should jump if the loop counter (CX register) is not zero.
- Destination: This specifies the target location in the code where the CPU should jump if the loop counter (CX register) is not zero. It can be specified as a label or a memory address.

c) Usage:

- The LOOP instruction decrements the loop counter (CX register) by one and checks if it's zero.
- If the loop counter is not zero after decrementing, the CPU jumps to the destination specified by the label or memory address.
- This process repeats until the loop counter becomes zero.

Example:

```assembly
MOV CX, 10              ; Initialize loop counter to 10
L1:
; Loop body instructions
DEC CX                  ; Decrement loop counter
LOOP L1                 ; Jump back to L1 if CX is not zero
```

In this example, the LOOP instruction jumps back to label L1 if the loop counter CX is not zero, effectively creating a loop that iterates 10 times.

# Vidyavardhini's College of Engineering and Technology

## Department of Artificial Intelligence & Data Science

**AY: 2023-24**

| | | | |
|---|---|---|---|
| **Class:** | | **Semester:** | |
| **Course Code:** | | **Course Name:** | |

| | |
|---|---|
| **Name of Student:** | |
| **Roll No. :** | |
| **Experiment No.:** | |
| **Title of the Experiment:** | |
| **Date of Performance:** | |
| **Date of Submission:** | |

## Evaluation

| Performance Indicator | Max. Marks | Marks Obtained |
|---|---|---|
| Performance | 5 | |
| Understanding | 5 | |
| Journal work and timely submission | 10 | |
| Total | 20 | |

| Performance Indicator | Exceed Expectations (EE) | Meet Expectations (ME) | Below Expectations (BE) |
|---|---|---|---|
| Performance | 4-5 | 2-3 | 1 |
| Understanding | 4-5 | 2-3 | 1 |
| Journal work and timely submission | 8-10 | 5-8 | 1-4 |

**Checked by**

**Name of Faculty :**

**Signature :**

**Date :**

**Aim:** 8255 is configured in mode O is simple Inuput / Output Mode. Ports A,B,C are in mode 0. All the posts are in output mode and data is transmitted to the respective ports.

**Apparatus :** Microprocessor 8086 and 8255 PPI experimental setup kit

**Theory:**

The programmable Peripheral Interface chip 8255 has three 8-bit Input / Output ports i.e. Port A, Port B, Port C upper (PCU) and Port C lower (PCL). Direct bit set/reset capability is available for port C. 8255 is a very powerful tool for interfacing peripheral equipment to the microprocessor. It is flexible enough to interface with any I/o device without the need of external logic.

**Procedure :**

1. Connect 8086 kit to 8255 PPI kit using 50 pin FRU cable.
2. Default I/O address ranges are :

| SELECTION | ADDRESS |
|---|---|
| Port A | 30 H |
| Port B | 31 H |
| Port C | 32 H |
| Command Port | 33 H |

3. 80 H is the control word for 8255. It is set in simple I/O mode and all the ports are in output mode 0

| D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

| Always 1 for I/O | Group A mode 0 | Port A (output) | Port C1 (output) | Group B (output) | Port B (output) | Port C2 (output) |
|---|---|---|---|---|---|---|

4. The LED's connected to the pins at Port A glow according to the data transmitted on port A.
5. The LED's connected to the pins of port B glow according to the data transmitted on Port B.
6. The LED's connected to the pins of port C glow according to the data transmitted on Port C.

**Program :**

Segment : C000
Offset : C000

| Memory | Opcode | Instructions | Comments |
|---|---|---|---|
| C000 | B0 | MOV AL,80H | Mode 0, All ports in output mode |
| C001 | 80 | | |
| C002 | E6 | OUT CWR, AL | |
| C003 | 33 | | |
| C004 | B0 | MOV AL, 55H | Data for Port A |
| C005 | 55 | | |
| C006 | E6 | OUT PORT A,AL | |
| C007 | 30 | | |
| C008 | B0 | MOV AL,AAH | Data for port B |
| C009 | AA | | |
| C00A | E6 | OUT PORT B,AL | |
| C00B | 31 | | |
| C00C | B0 | MOV AL,0FH | Data for port C |
| C00D | 0F | | |
| C00E | E6 | OUT PORTC,AL | |
| C00F | 32 | | |
| C010 | CC | INT 3 | Stop |

**Conclusion :**

1. Explain the modes of 8255.

Ans. The 8255 Programmable Peripheral Interface (PPI) offers three operational modes, each serving distinct input/output configurations:

a) Mode 0 (Basic Input/Output Mode):
   a. Port A acts as an 8-bit input or output port.
   b. Port B serves as a bidirectional 8-bit port.
   c. Port C is divided into two 4-bit ports: Port C upper (PC7-PC4) and Port C lower (PC3-PC0), independently configurable as inputs or outputs.

b) Mode 1 (Strobed Input/Output Mode):
   a. Similar to Mode 0 but with additional handshaking features.
   b. Control logic enables Port A and Port B only when the CPU sends a specific signal (STB A and STB B).
   c. Useful for interfacing devices that require synchronization with the CPU.

c) Mode 2 (Bi-directional Bus Configuration):
   a. All three ports function as bi-directional 8-bit ports.
   b. Supports interconnection between multiple processors or systems via shared buses.
   c. Offers versatility for various data transfer configurations.

Each mode offers flexibility in configuring input/output ports, catering to diverse interfacing requirements in embedded systems and peripheral device control.

2. Explain the format of control word of 8255 PIC

Ans. The control word of the 8255 Programmable Peripheral Interface (PPI) is a configuration command used to set the operational mode and various parameters of the device. Here's the format of the control word:

a) Bit 7 (D7): Mode Selection Bit
   a. Determines the operating mode of the 8255.
   b. D7 = 0: Mode 0 or Mode 1 selected.
   c. D7 = 1: Mode 2 selected.

b) Bit 6 and Bit 5 (D6 and D5): Group Selection Bits
   a. Used to select the group of I/O ports for operation in Mode 1.
   b. D6 and D5 select the group as follows:
   c. 00: Group A selected.
   d. 01: Group B selected.
   e. 10: Group C selected.
   f. 11: Not used.

c) Bit 4 (D4): Port Selection Bit
   a. Used to select between Port A and Port C in Mode 1.
   b. D4 = 0: Port A selected.
   c. D4 = 1: Port C selected.

d) Bit 3 to Bit 0 (D3-D0): I/O Mode Selection Bits
   a. Used to configure the direction of each individual port.
   b. In Mode 0 or Mode 1, each pair of bits corresponds to the direction of the respective port:
   c. 00: Input mode.
   d. 01: Output mode.
   e. 10: Bidirectional mode.
   f. 11: Mode dependent.

This format allows precise configuration of the 8255 device, enabling the selection of operating modes, I/O port directions, and group settings as per the specific requirements of the system.

# Vidyavardhini's College of Engineering and Technology

## Department of Artificial Intelligence & Data Science

**AY: 2023-24**

| | | | |
|---|---|---|---|
| **Class:** | | **Semester:** | |
| **Course Code:** | | **Course Name:** | |

| | |
|---|---|
| **Name of Student:** | |
| **Roll No. :** | |
| **Experiment No.:** | |
| **Title of the Experiment:** | |
| **Date of Performance:** | |
| **Date of Submission:** | |

## Evaluation

| Performance Indicator | Max. Marks | Marks Obtained |
|---|---|---|
| Performance | 5 | |
| Understanding | 5 | |
| Journal work and timely submission | 10 | |
| Total | 20 | |

| Performance Indicator | Exceed Expectations (EE) | Meet Expectations (ME) | Below Expectations (BE) |
|---|---|---|---|
| Performance | 4-5 | 2-3 | 1 |
| Understanding | 4-5 | 2-3 | 1 |
| Journal work and timely submission | 8-10 | 5-8 | 1-4 |

**Checked by**

**Name of Faculty :**

**Signature :**

**Date :**

**Aim:** Program for printing the string using procedure and macro.

**Theory:**

**Procedures:-**

- Procedures are used for large group of instructions to be repeated.
- Object code generated only once. Length of the object file is less the memory
- CALL and RET instructions are used to call procedure and return from procedure.
- More time required for its execution.
- Procedure Can be defined as:

```
Procedure_name PROC
……
……
Procedure_name ENDP
```

Example:

```
Addition PROC near
……
……
Addition ENDP
```

**Macro:-**

- Macro is used for small group of instructions to be repeated.
- Object code is generated every time the macro is called.
- Object file becomes very lengthy.
- Macro can be called just by writing.
- Directives MACRO and ENDM are used for defining macro.
- Less time required for its execution.
- Macro can be defined as:

```
Macro_name MACRO [Argument, .... , Argument N]
......
......
ENDM
```

Example:-

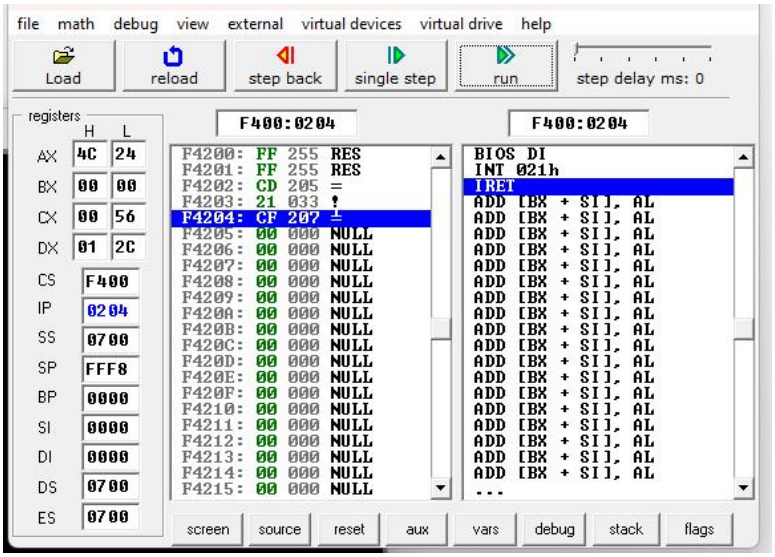
```
Display MACRO msg
.....
.....
ENDM
```

Thus, the program for printing the string is successfully implemented using procedure and macro.

**Assembly Code:**

```
org 100h

.data

msg1 db 10,13,'Learning Procedure$'

msg2 db 10,13,'Procedure are funs$'

msg3 db 10,13,'Hello world$'

.code

lea dx, msg1

call print

lea dx, msg2

call print

lea dx, msg3

call print

mov ah, 4CH

int 21h

print PROC

mov ah,09h

int 21h

ret

print ENDP

Ret
```

**Output :**




**Conclusion:**

1. Differentiate between procedure and macros.

Ans. Procedures and macros are both used in assembly language programming to encapsulate and reuse code segments, but they have distinct differences:

a) Procedures:
   a. A procedure is a named block of code that performs a specific task or operation.
   b. Procedures are defined using labels and typically terminated by a return instruction.
   c. They can accept parameters passed through registers, stack, or memory.
   d. Procedures support local variables and utilize stack space for parameter passing and local variable storage.
   e. They offer flexibility in code organization and enable modular programming.
   f. Procedures are usually more flexible and powerful but may have overhead due to parameter passing and stack manipulation.

b) Macros:
   a. Macros are code templates that are expanded inline at compile-time or assembly-time.
   b. They are defined using macros directives, which specify the macro name and its replacement text.
   c. Macros are invoked using their name and can accept arguments, similar to functions in higher-level languages.
   d. They are expanded by the assembler wherever they are invoked, effectively replicating the macro's code at each invocation.
   e. Macros are useful for repetitive tasks, code generation, and providing syntactic sugar, but they do not support local variables or parameter passing like procedures.
   f. Macros offer efficiency in code size and execution speed as they eliminate the overhead of procedure calls, but they may result in larger source files.

In summary, procedures are reusable code segments that execute at runtime and support parameters and local variables, while macros are expanded inline during assembly and are suitable for code generation and repetition but lack the capabilities of procedures in terms of parameter passing and local variable storage.

2. Explain CALL and RET instructions.

Ans. CALL and RET are essential instructions in assembly language programming for implementing subroutine calls and returns. Here's an explanation of each:

a) CALL (Call):
   a. CALL is used to transfer control to a subroutine or procedure.
   b. It saves the return address (the address of the instruction immediately following the CALL instruction) onto the stack before transferring control to the subroutine.
   c. CALL instructions are usually followed by the address of the subroutine to be called.
   d. After executing the CALL instruction, the CPU begins executing the first instruction of the subroutine.

b) RET (Return):
   a. RET is used to return control from a subroutine back to the main program.
   b. It retrieves the return address from the top of the stack and transfers control to that address.
   c. RET instructions do not require any operands.
   d. After executing the RET instruction, the CPU resumes execution at the instruction following the original CALL instruction.

Together, CALL and RET instructions allow for the implementation of subroutines or procedures, enabling modular programming and code reuse. Subroutines encapsulate specific tasks or operations, and CALL/RET facilitate their invocation and return, providing a structured way to organize code and manage program flow.