Name:	Rahul Yadav
Roll No:	76
Class/Sem:	SE/IV
Experiment No.:	6
Title:	To perform program to reverse the word in string
Date of Performance:	14/2/24
Date of Submission:	24/2/24
Marks:	
Sign of Faculty:	

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

Theory:

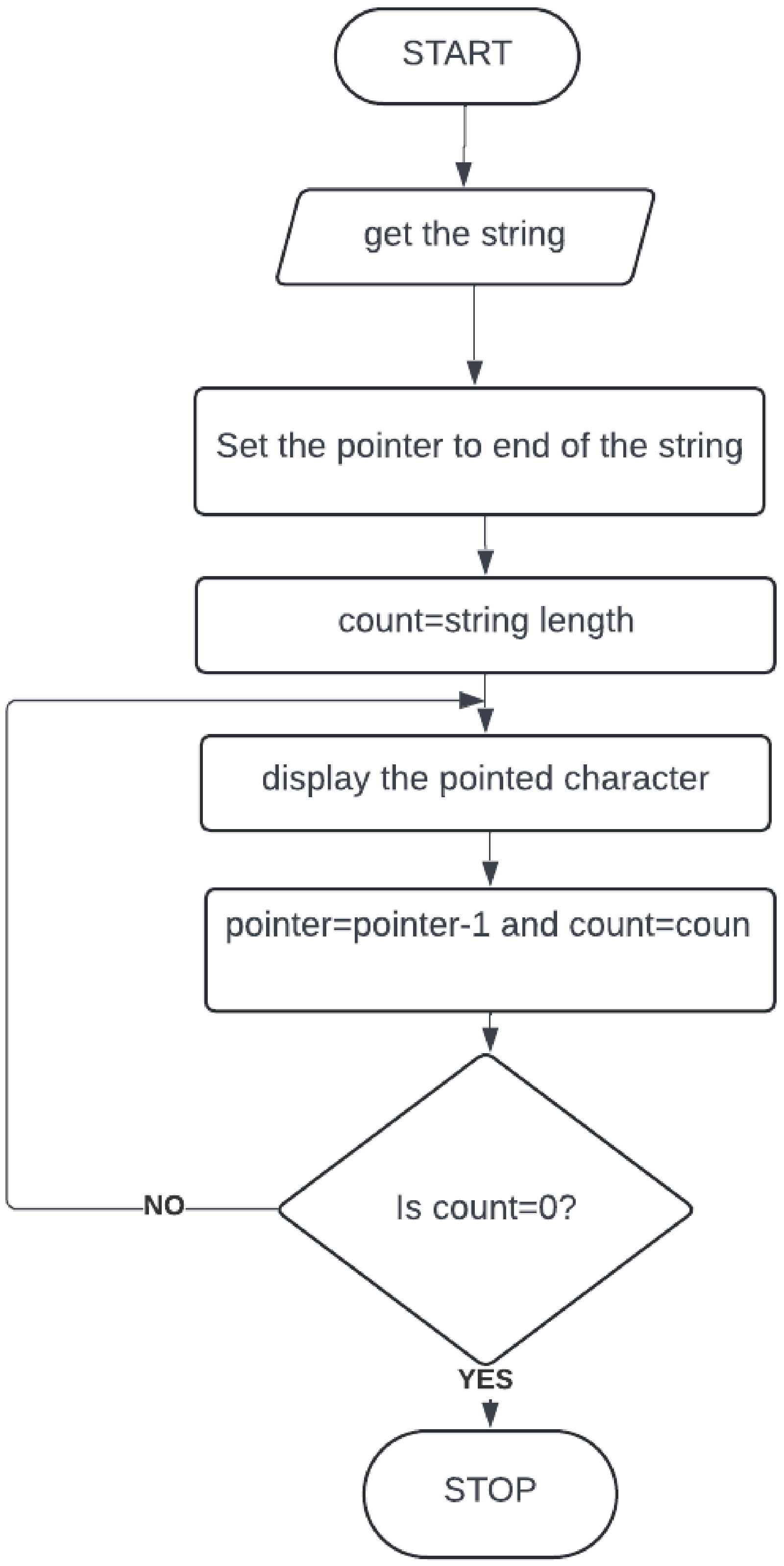
This program will read the string entered by the user and then reverse it. Reverse a string is the technique that reverses or changes the order of a given string so that the last character of the string becomes the first character of the string and so on.

Algorithm:

- Start
- Initialize the data segment
- Display the message -1
- Input the string
- Display the message 2
- Take characters count in DI

- Point to the end character and read it
- Display the character
- Decrement the count
- Repeat until the count is zero
- To terminate the program using DOS interrupt
    - Initialize AH with 4ch
    - Call interrupt INT 21h
- Stop

Flowchart:

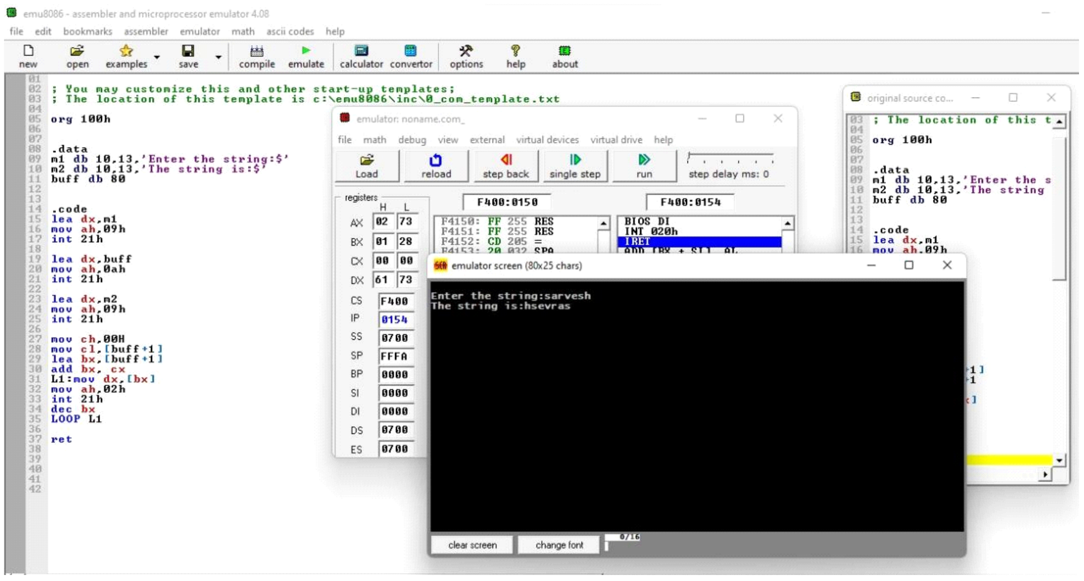

START
get the string
Set the pointer to end of the string
count=string length
display the pointed character
pointer=pointer-1 and count=coun
Is count=0?
NO
YES
STOP

Implementation:

Code:

```
org 100h

.data
m1 db 10,13,'Enter the string:$'
m2 db 10,13,'The string is:$'
buff db 80


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

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

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

mov cl,[buff+1]
lea bx, buff+2
L1:mov dx,[bx]
mov ah,02h
int 21h
inc bx
LOOP L1
```

Output



```
; You may customize this and other start-up templates;
; The location of this template is c:\emu8086\inc\0_com_template.txt

org 100h


.data
m1 db 10,13,'Enter the string:$'
m2 db 10,13,'The string is:$'
buff db 80


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

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

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

mov ch,00H
mov cl,[buff+1]
lea bx,[buff+1]
add bx, cx
L1:mov dx,[bx]
mov ah,02h
int 21h
dec bx
LOOP L1

ret
```

registers	H	L
AX	02	73
BX	01	28
CX	00	00
DX	61	73
CS	F400	
IP	0154	
SS	0700	
SP	FFFA	
BP	0000	
SI	0000	
DI	0000	
DS	0700	
ES	0700	

```
F400:0150
F4150: FF 255 RES
F4151: FF 255 RES
F4152: CD 205 =
F4153: 20 032 SPA
```

```
F400:0154
BIOS DI
INT 020h
IRET
ADD [BX + SI], AL
```

```
Enter the string:sarvesh
The string is:hsevras
```

emu8086 - assembler and microprocessor emulator 4.08
file edit bookmarks assembler emulator math ascii codes help
new open examples save compile emulate calculator convertor options help about
emulator: noname.com_
file math debug view external virtual devices virtual drive help
Load reload step back single step run step delay ms: 0
emulator screen (80x25 chars)
clear screen change font
0/16
original source co...

:

- **Explain the difference between XLAT and XLATB.**

XLAT (Translate) and XLATB (Translate Byte) are both x86 assembly language instructions used for data manipulation, particularly in converting data from one form to another. Here's the difference:

XLAT:

Operation: The XLAT instruction takes the value in the AL register as an index into a lookup table (often referred to as a translation table) located in memory. It replaces the value in AL with the byte located at the address formed by adding the value in the BX register to the value in the AL register.

Syntax: XLAT

Example:

Let's say AL contains the value 0x02 and BX contains the memory offset where the lookup table starts. If the byte at the memory location BX + AL is 0x67, after executing XLAT, AL would contain 0x67.

XLATB:

Operation: XLATB is similar to XLAT but operates specifically on the AL register. It takes the value in the AL register as an index into a lookup table located in memory and replaces the value in AL with the byte located at the address formed by adding the value in the BX register to the value in the AL register.

Syntax: XLATB

Example:

Similar to XLAT, if AL contains the value 0x02 and BX contains the memory offset where the lookup table starts, after executing XLATB, AL would contain the byte at the memory location BX + 0x02.

In summary, both XLAT and XLATB perform a similar operation of looking up a byte in memory using an index in the AL register, but XLAT operates with BX as an offset while XLATB operates specifically on AL.

- **Explain the instruction LAHF.**

The LAHF instruction in x86 assembly language is used to load the lower byte of the flags register (FLAGS) into the AH register. Here's a breakdown:

Operation: LAHF stands for "Load AH from Flags". It copies the lower byte of the flags register (FLAGS) into the AH register. The FLAGS register contains various status flags that reflect the outcome of arithmetic and logical operations performed by the processor.

Usage: After executing arithmetic or logical operations, you might need to inspect the status of the flags to determine the outcome of the operation. LAHF allows you to access these flags indirectly by loading them into a general-purpose register (AH) where they can be manipulated or inspected further.

Syntax : LAHF

The LAHF instruction in x86 assembly language is used to load the lower byte of the flags register (FLAGS) into the AH register. Here's a breakdown:

Operation: LAHF stands for "Load AH from Flags". It copies the lower byte of the flags register (FLAGS) into the AH register. The FLAGS register contains various status flags that reflect the outcome of arithmetic and logical operations performed by the processor.

Usage: After executing arithmetic or logical operations, you might need to inspect the status of the flags to determine the outcome of the operation. LAHF allows you to access these flags indirectly by loading them into a general-purpose register (AH) where they can be manipulated or inspected further.

Syntax:

Copy code
LAHF
Affected Flags: The lower byte of the FLAGS register includes the following flags (in order from least significant bit to most significant bit):

Carry flag (CF)
Parity flag (PF)
Auxiliary Carry flag (AF)
Zero flag (ZF)
Sign flag (SF)
Trap flag (TF)
Interrupt flag (IF)

Direction flag (DF)
Overflow flag (OF)
Example:
After executing LAHF, the AH register would contain a copy of the lower byte of the FLAGS register, with each bit corresponding to the state of a particular flag.

Conclusion:

Conclusively, the experiment to perform a program to reverse the word in a string on a microprocessor offers valuable insights into low-level programming and string manipulation techniques. Through this experiment, participants gain a deeper understanding of assembly language instructions, memory management, and algorithmic thinking.

By implementing the reverse word algorithm in assembly language, participants develop skills in:

Instruction Set Understanding: They become familiar with the specific instructions available in the microprocessor's architecture and learn how to leverage them effectively for string manipulation tasks.

Memory Management: Participants grasp the concept of memory addressing and learn how to access and modify data stored in memory locations.

Algorithm Design: Crafting an efficient algorithm to reverse words within a string fosters algorithmic thinking, encouraging participants to consider factors such as time complexity and space complexity.

Debugging and Optimization: As they encounter errors and inefficiencies in their code, participants refine their debugging skills and explore optimization techniques to enhance the performance of their programs.