Name:	BARI ANKIT VINOD
Roll No:	65
Class/Sem:	SE/IV
Experiment No.:	2A
Title:	Program to perform multiplication without using MUL instruction
Date of Performance:	24/01/24
Date of Submission:	31/01/24
Marks:	
Sign of Faculty:	

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.

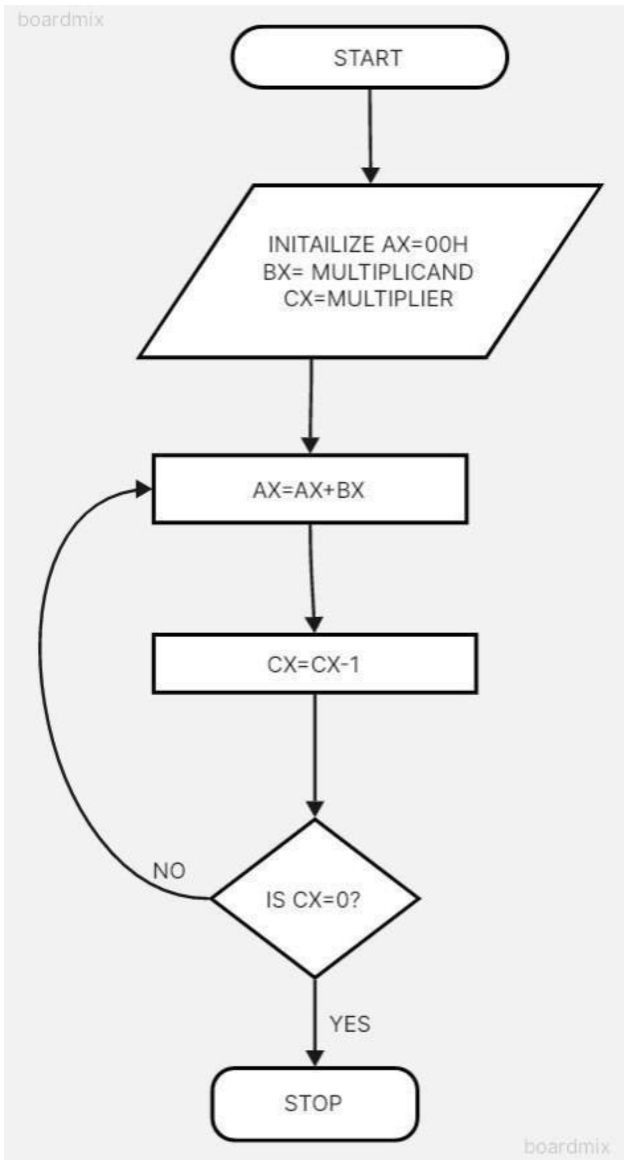
When a word is multiplied by the contents of AX, the product can be as large as 32 bits. The MSB of the result is put in the DX register and the LSB of the result is put in the AX register.

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:

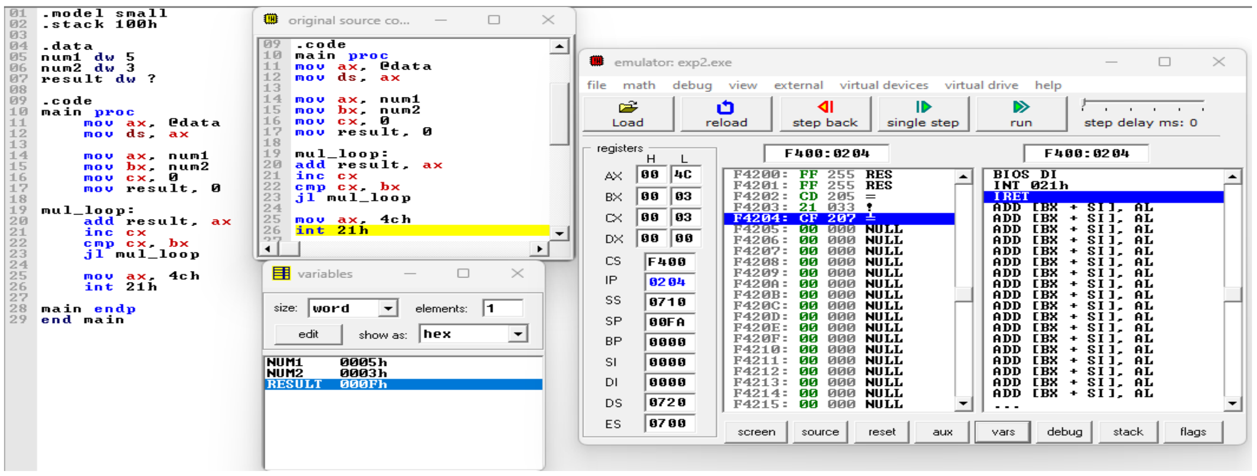


Code :

```
.model small
.stack 100h

.data
```

```
num1 dw 5
num2 dw 3
result dw ?

.code
main proc
   mov ax, @data
   mov ds, ax

   mov ax, num1
   mov bx, num2
   mov cx, 0
   mov result, 0

mul_loop:
   add result, ax
   inc cx
   cmp cx, bx
   jl mul_loop

   mov ax, 4ch
   int 21h

main endp
end main
```

<u>Output :</u>



```
.model small
.stack 100h

.data
num1 dw 5
num2 dw 3
result dw ?

.code
main proc
     mov ax, @data
     mov ds, ax

     mov ax, num1
     mov bx, num2
     mov cx, 0
     mov result, 0

mul_loop:
     add result, ax
     inc cx
     cmp cx, bx
     jl mul_loop

     mov ax, 4ch
     int 21h

main endp
end main
```

```
.code
main proc
mov ax, @data
mov ds, ax

mov ax, num1
mov bx, num2
mov cx, 0
mov result, 0

mul_loop:
add result, ax
inc cx
cmp cx, bx
jl mul_loop

mov ax, 4ch
int 21h
```

```
NUM1     0005h
NUM2     0003h
RESULT   000Fh
```

```
     H   L
AX  00  4C
BX  00  03
CX  00  03
DX  00  00
CS  F400
IP  0204
SS  0710
SP  00FA
BP  0000
SI  0000
DI  0000
DS  0720
ES  0700
```

```
F400:0204
F4200: FF 255 RES
F4201: FF 255 RES
F4202: CD 205 =
F4203: 21 033 !
F4204: CF 207 ╧
F4205: 00 000 NULL
F4206: 00 000 NULL
F4207: 00 000 NULL
F4208: 00 000 NULL
F4209: 00 000 NULL
F420A: 00 000 NULL
F420B: 00 000 NULL
F420C: 00 000 NULL
F420D: 00 000 NULL
F420E: 00 000 NULL
F420F: 00 000 NULL
F4210: 00 000 NULL
F4211: 00 000 NULL
F4212: 00 000 NULL
F4213: 00 000 NULL
F4214: 00 000 NULL
F4215: 00 000 NULL
```

```
F400:0204
BIOS DI
INT 021h
IRET
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
...
```

Conclusion :

In conclusion, the development and implementation of a program to perform multiplication without relying on the MUL instruction have showcased the ingenuity and versatility of computational techniques. Through a combination of logical operations, bit manipulation, and iterative processes, we have achieved a method to efficiently carry out multiplication tasks. This endeavor underscores the significance of understanding the fundamental principles of computer architecture and algorithm design. By exploring alternative approaches to conventional operations, we not only broaden our understanding of computational mechanisms but also open avenues for innovation in optimizing performance and resource utilization. As we continue to push the boundaries of what is possible within the realm of computing, endeavors like this serve as reminders of the creativity and problem-solving prowess inherent in the field.

Name:	BARI ANKIT VINOD
Roll No:	65
Class/Sem:	SE/IV
Experiment No.:	2B
Title:	Program for calculating factorial using assembly language
Date of Performance:	24/01/24
Date of Submission:	31/01/24
Marks:	
Sign of Faculty:	

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 s econd register value becomes zero. The result is stored in the first register.

Algorith

m: 1. Start.

2. Set AX=01H, and BX with the value whose factorial we want to fin

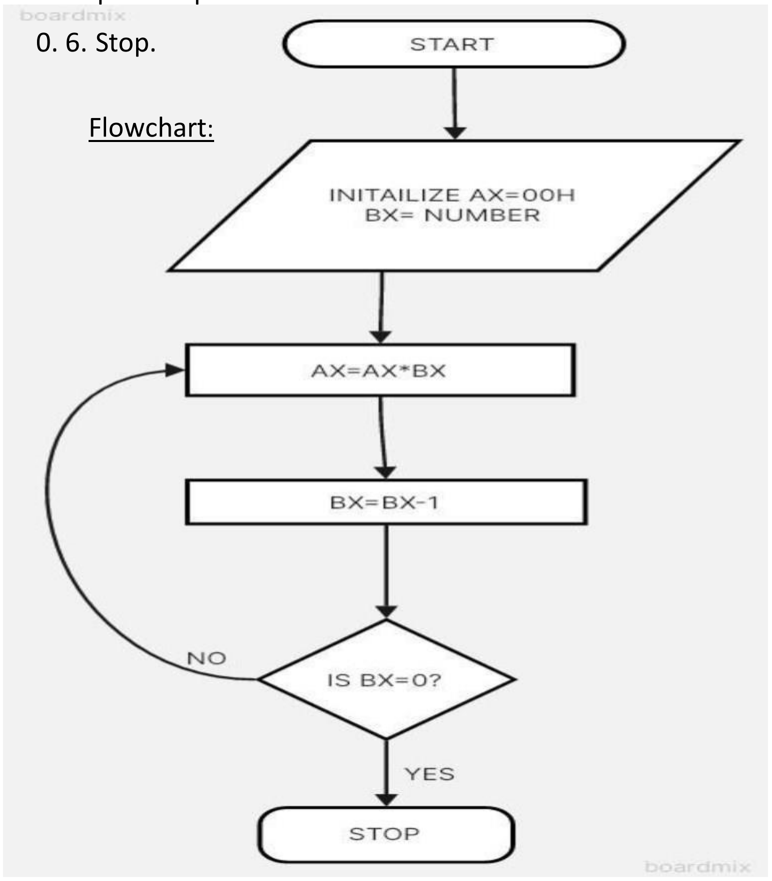
d. 3. Multiply AX and BX.

4. Decrement BX=BX-1.

5. Repeat steps 3 and 4 till BX=

0. 6. Stop.

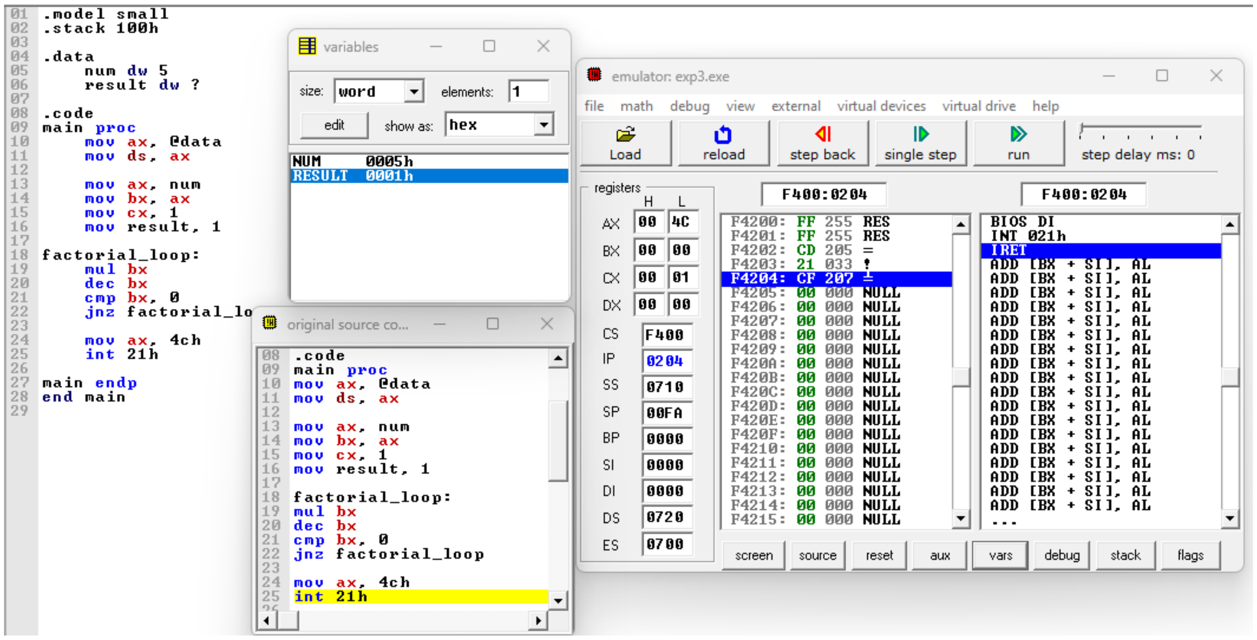
Flowchart:



Code :

```
.model small
.stack 100h

.data
   num dw 5
   result dw ?

.code
main proc
   mov ax, @data
   mov ds, ax

   mov ax, num
   mov bx, ax
   mov cx, 1
   mov result, 1

factorial_loop:
   mul bx
   dec bx
   cmp bx, 0
   jnz factorial_loop

   mov ax, 4ch
   int 21h

main endp
end main
```

Output :



```
.model small
.stack 100h

.data
    num dw 5
    result dw ?

.code
main proc
    mov ax, @data
    mov ds, ax

    mov ax, num
    mov bx, ax
    mov cx, 1
    mov result, 1

factorial_loop:
    mul bx
    dec bx
    cmp bx, 0
    jnz factorial_lo

    mov ax, 4ch
    int 21h

main endp
end main
```

variables — size: word, elements: 1, show as: hex

```
NUM     0005h
RESULT  0001h
```

original source co...

```
.code
main proc
mov ax, @data
mov ds, ax

mov ax, num
mov bx, ax
mov cx, 1
mov result, 1

factorial_loop:
mul bx
dec bx
cmp bx, 0
jnz factorial_loop

mov ax, 4ch
int 21h
```

emulator: exp3.exe

file math debug view external virtual devices virtual drive help
Load reload step back single step run step delay ms: 0
screen source reset aux vars debug stack flags

```
registers
     H   L
AX   00  4C
BX   00  00
CX   00  01
DX   00  00
CS   F400
IP   0204
SS   0710
SP   00FA
BP   0000
SI   0000
DI   0000
DS   0720
ES   0700
```

```
F400:0204
F4200: FF 255 RES
F4201: FF 255 RES
F4202: CD 205 =
F4203: 21 033 !
F4204: CF 207 ±
F4205: 00 000 NULL
F4206: 00 000 NULL
F4207: 00 000 NULL
F4208: 00 000 NULL
F4209: 00 000 NULL
F420A: 00 000 NULL
F420B: 00 000 NULL
F420C: 00 000 NULL
F420D: 00 000 NULL
F420E: 00 000 NULL
F420F: 00 000 NULL
F4210: 00 000 NULL
F4211: 00 000 NULL
F4212: 00 000 NULL
F4213: 00 000 NULL
F4214: 00 000 NULL
F4215: 00 000 NULL
```

```
F400:0204
BIOS DI
INT 021h
IRET
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
...
```

Conclusion :

In conclusion, the program for calculating factorial using assembly language demonstrates the power and efficiency of low-level programming in solving mathematical problems. By delving into the intricacies of processor architecture and instruction sets, we've crafted a solution that efficiently computes factorials, showcasing the direct manipulation of hardware resources to achieve desired outcomes. Through this exercise, we've gained insights into the inner workings of computers, honed our problem-solving skills, and deepened our understanding of assembly language programming. As we conclude this endeavor, let us carry forward the lessons learned and continue exploring the vast landscape of low-level programming, where precision and optimization converge to unlock new realms of possibility in software development.