AY: 2024-25

Class:	**SE**	**Semester:**	**IV**
Course Code:	**CSL404**	**Course Name:**	**Microprocessor Lab**

Name of Student:	**Bhagyashri kaleni Sutar**
Roll No. :	**75**
Experiment No.:	**2**
Title of the Experiment:	**A. Program to perform multiplication using MUL instruction** **B. Program for calculating factorial using assembly language**
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)	elow 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 : Ms. Sweety Patil

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.

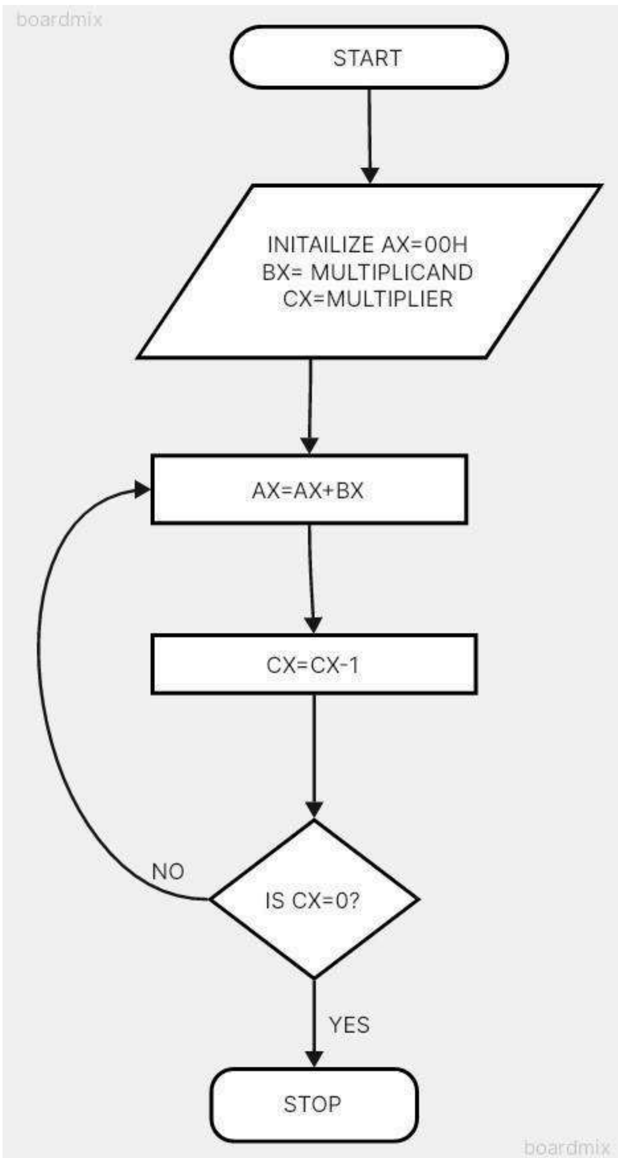
MUL BH; multiply AL with BH; result in AX.

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:



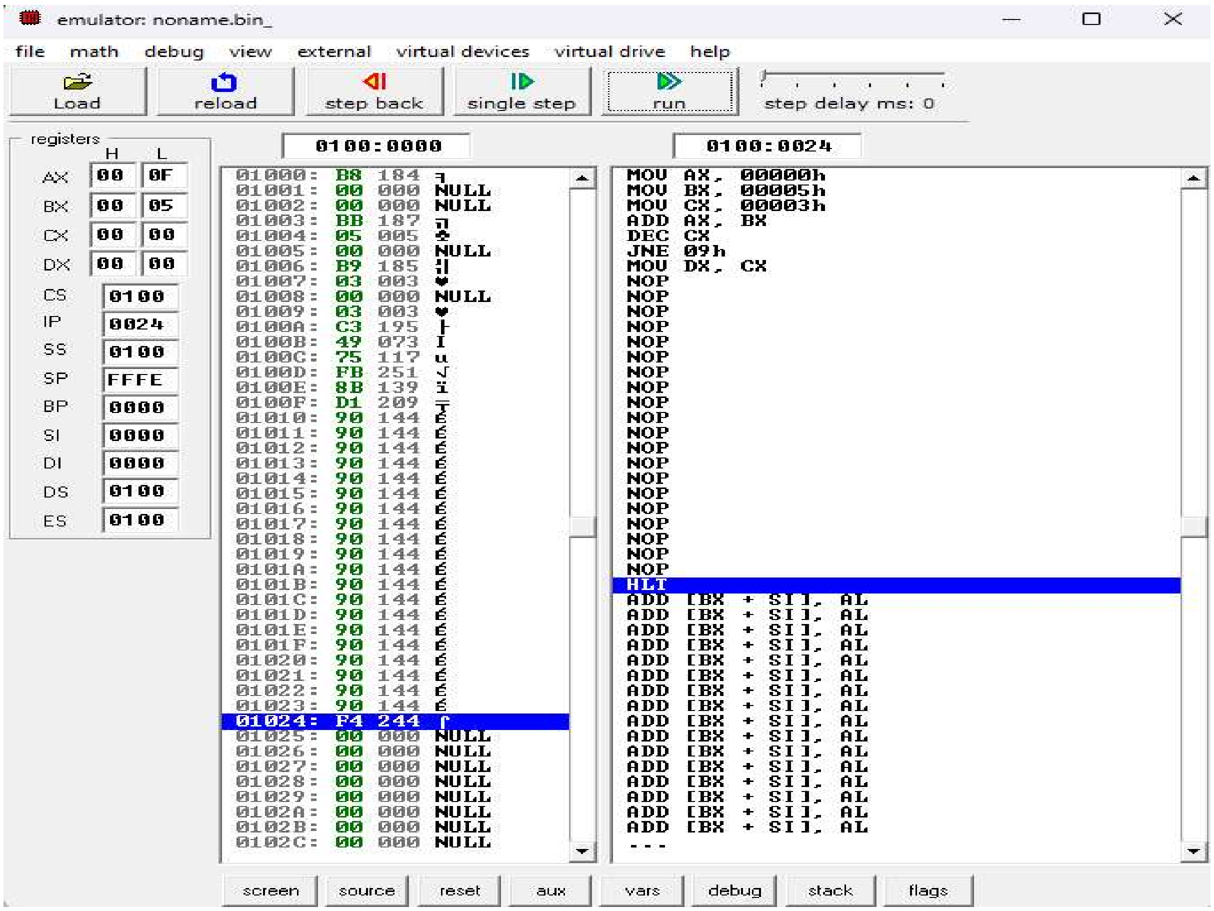
Code:

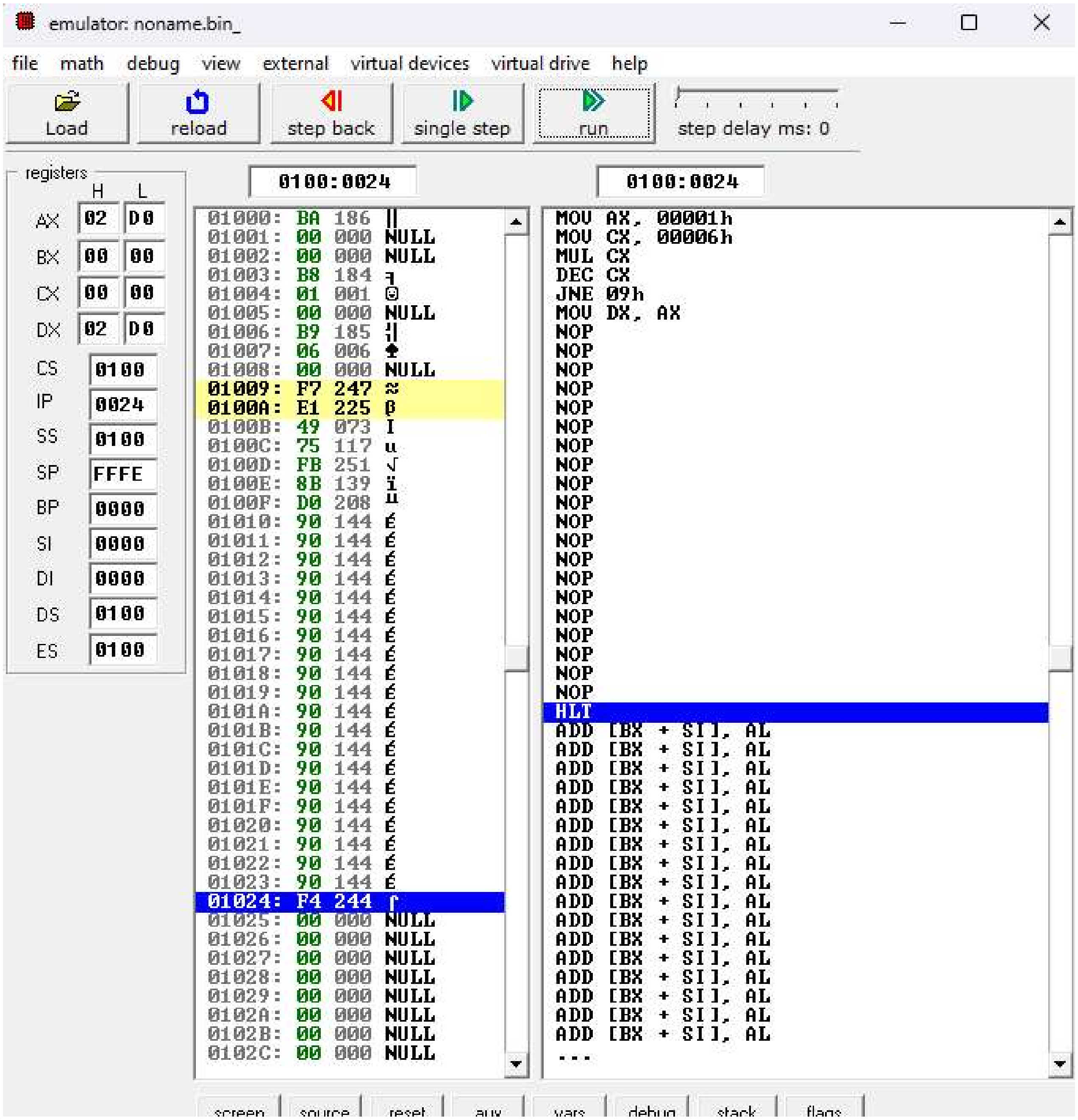
```
mov ax,0000H
mov bx,0005H
mov cx,0003H

loop:
add ax,bx
dec cx
jnz loop


mov dx,cx
```

output:-


emulator: noname.bin_
file math debug view external virtual devices virtual drive help
Load
reload
step back
single step
run
step delay ms: 0
registers
H L
AX 00 0F
BX 00 05
CX 00 00
DX 00 00
CS 0100
IP 0024
SS 0100
SP FFFE
BP 0000
SI 0000
DI 0000
DS 0100
ES 0100
0100:0000
0100:0024
01000: B8 184
01001: 00 000 NULL
01002: 00 000 NULL
01003: BB 187
01004: 05 005
01005: 00 000 NULL
01006: B9 185
01007: 03 003
01008: 00 000 NULL
01009: 03 003
0100A: C3 195
0100B: 49 073 I
0100C: 75 117 u
0100D: FB 251
0100E: 8B 139
0100F: D1 209
01024: F4 244
01025: 00 000 NULL
MOV AX, 00000h
MOV BX, 00005h
MOV CX, 00003h
ADD AX, BX
DEC CX
JNE 09h
MOV DX, CX
NOP
HLT
ADD [BX + SI], AL
screen
source
reset
aux
vars
debug
stack
flags



Conclusion:

1. Explain data transfer instructions.

Ans:-**1. Load Instructions (LD)**

- **Purpose:** Load data from memory into a register.

- **Example:** `LD R1, 1000H` – Load the value at memory address `1000H` into register `R1`.
- **Usage:** Moving data from a memory location into a register so that it can be used for further processing.

2. Store Instructions (ST)

- **Purpose:** Store data from a register into memory.
- **Example:** `ST R1, 2000H` – Store the value in register `R1` into memory address `2000H`.
- **Usage:** Saving data from a register back into memory, often done after performing operations on the data.

3. Move Instructions (MOV)

- **Purpose:** Move data from one register to another or from a register to memory, and vice versa.
- **Example:** `MOV R1, R2` – Move the contents of register `R2` into register `R1`.
- **Usage:** Typically used to copy or transfer values between registers, or between registers and memory.

4. Exchange Instructions (XCHG)

- **Purpose:** Swap the contents of two registers or a register and memory.
- **Example:** `XCHG R1, R2` – Exchange the contents of register `R1` with register `R2`.
- **Usage:** Swapping the values of two locations without needing extra temporary storage.

5. Push and Pop Instructions

- **Push:** Moves data onto the stack (usually decreases the stack pointer).
- **Pop:** Moves data from the stack back into a register (increases the stack pointer).
- **Example:**
  - `PUSH R1` – Push the contents of register `R1` onto the stack.
  - `POP R1` – Pop the top value from the stack into register `R1`.
- **Usage:** Used in managing function calls, preserving register values, and handling local variables.

6. Input/Output Instructions (IN/OUT)

- **Purpose:** Transfer data between registers and I/O devices or ports.
- **Example:**
  - `IN R1, 01H` – Read data from input port `01H` into register `R1`.
  - `OUT 01H, R1` – Send data from register `R1` to output port `01H`.

- **Usage:** For transferring data between the processor and external peripherals or devices.

7. Immediate Data Transfer (MOVI or similar)

2. **Purpose:** Load an immediate value (constant) into a register.
3. **Example:** `MOV R1, #10` – Load the immediate value `10` into register `R1`.
4. **Usage:** Used when directly specifying a value instead of loading it from memory.

2.Explain Arithmetic instructions.

Ans:-**1. Addition Instructions (ADD)**

- **Purpose:** Adds two operands (either from registers, memory, or immediate values) and stores the result.
- **Example:** `ADD R1, R2` – Add the contents of register `R2` to `R1`, and store the result in `R1`.
- **Usage:** Used to perform basic addition operations between two values.

Additional instructions related to addition:

- **ADC (Add with Carry):** Adds two operands along with the carry flag (used in multi-word or multi-byte additions).
- **ADI (Add Immediate):** Adds an immediate (constant) value to a register.

2. Subtraction Instructions (SUB)

- **Purpose:** Subtracts one operand from another and stores the result.
- **Example:** `SUB R1, R2` – Subtract the contents of register `R2` from `R1`, and store the result in `R1`.
- **Usage:** Used for basic subtraction of two values.

Additional instructions related to subtraction:

- **SBC (Subtract with Carry):** Subtracts two operands along with the borrow flag (used in multi-word or multi-byte subtractions).
- **SUI (Subtract Immediate):** Subtracts an immediate value from a register.

3. Multiplication Instructions (MUL)

- **Purpose:** Multiplies two operands and stores the result. The result could be a larger value, so it may need to be stored in multiple registers.
- **Example:** `MUL R1, R2` – Multiply the contents of registers `R1` and `R2`, and store the result in two registers (often `R1` and `R2` or a specific register pair).
- **Usage:** Used for multiplying numbers.

Additional instructions related to multiplication:

- **MULI (Multiply Immediate):** Multiplies a register value by an immediate (constant) value.

4. Division Instructions (DIV)

- **Purpose:** Divides one operand by another and stores the quotient and remainder.
- **Example:** `DIV R1, R2` – Divide the contents of register `R1` by `R2`, storing the quotient in one register and the remainder in another.
- **Usage:** Used for division operations.

Additional instructions related to division:

- **DIVI (Divide Immediate):** Divides a register value by an immediate (constant) value.

5. Increment and Decrement Instructions (INC, DEC)

- **Purpose:** Increment (add 1) or decrement (subtract 1) the value of a register or memory location.
- **Example:**
  - `INC R1` – Increment the value in register `R1` by 1.
  - `DEC R1` – Decrement the value in register `R1` by 1.
- **Usage:** Often used for loop counters, memory addressing, or simple arithmetic adjustments.

6. Negation Instructions (NEG)

- **Purpose:** Negates (changes the sign of) the value in a register.
- **Example:** `NEG R1` – Negate the value in register `R1`, changing its sign.
- **Usage:** Used when you want to perform a sign inversion (e.g., converting positive numbers to negative or vice versa).

7. Comparison Instructions (CMP)

- **Purpose:** Compares two operands by performing subtraction but does not store the result; instead, it sets the flags based on the outcome (like zero, carry, or overflow).
- **Example:** `CMP R1, R2` – Compare the contents of registers `R1` and `R2`.
- **Usage:** Often used in conjunction with conditional jump instructions, allowing branching based on the result of the comparison (e.g., `IF R1 > R2`).

Related Instructions:

- **CMPI (Compare Immediate):** Compares a register with an immediate value.

8. Set Instructions (e.g., SET, CSET)

- **Purpose:** These are used to set certain flags or conditions based on arithmetic results.
- **Example:** Some processors may have instructions like `CSET` (set condition code) that set flags based on the result of arithmetic operations.

Aim: Program to calculate the Factorial of a number.

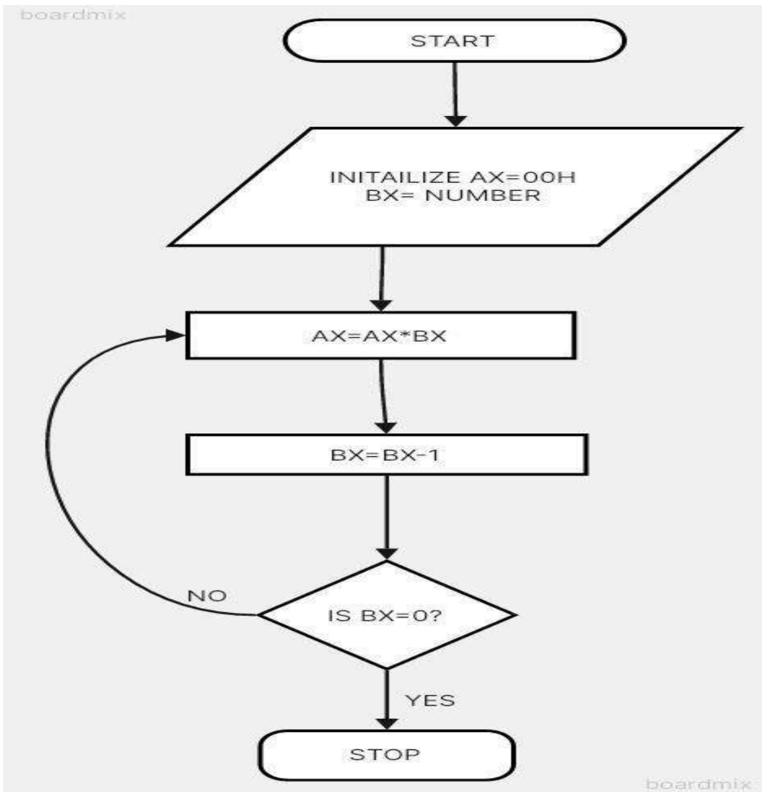
Theory:

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.

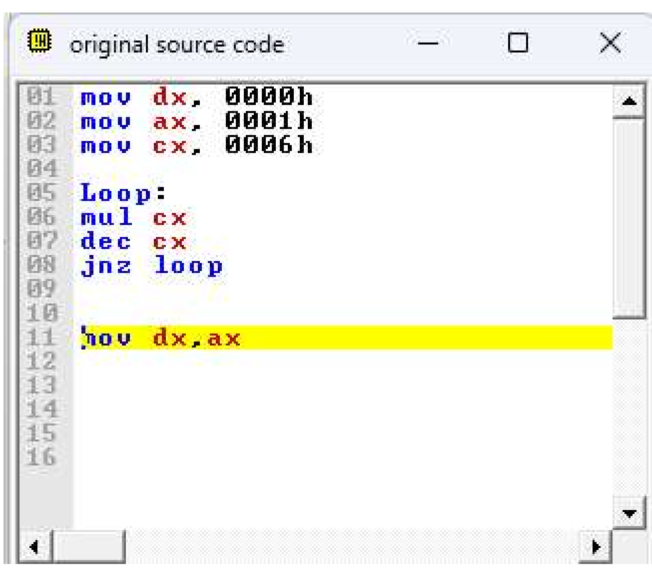
<u>Algorithm:</u>

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.

<u>Flowchart</u>:



Code:

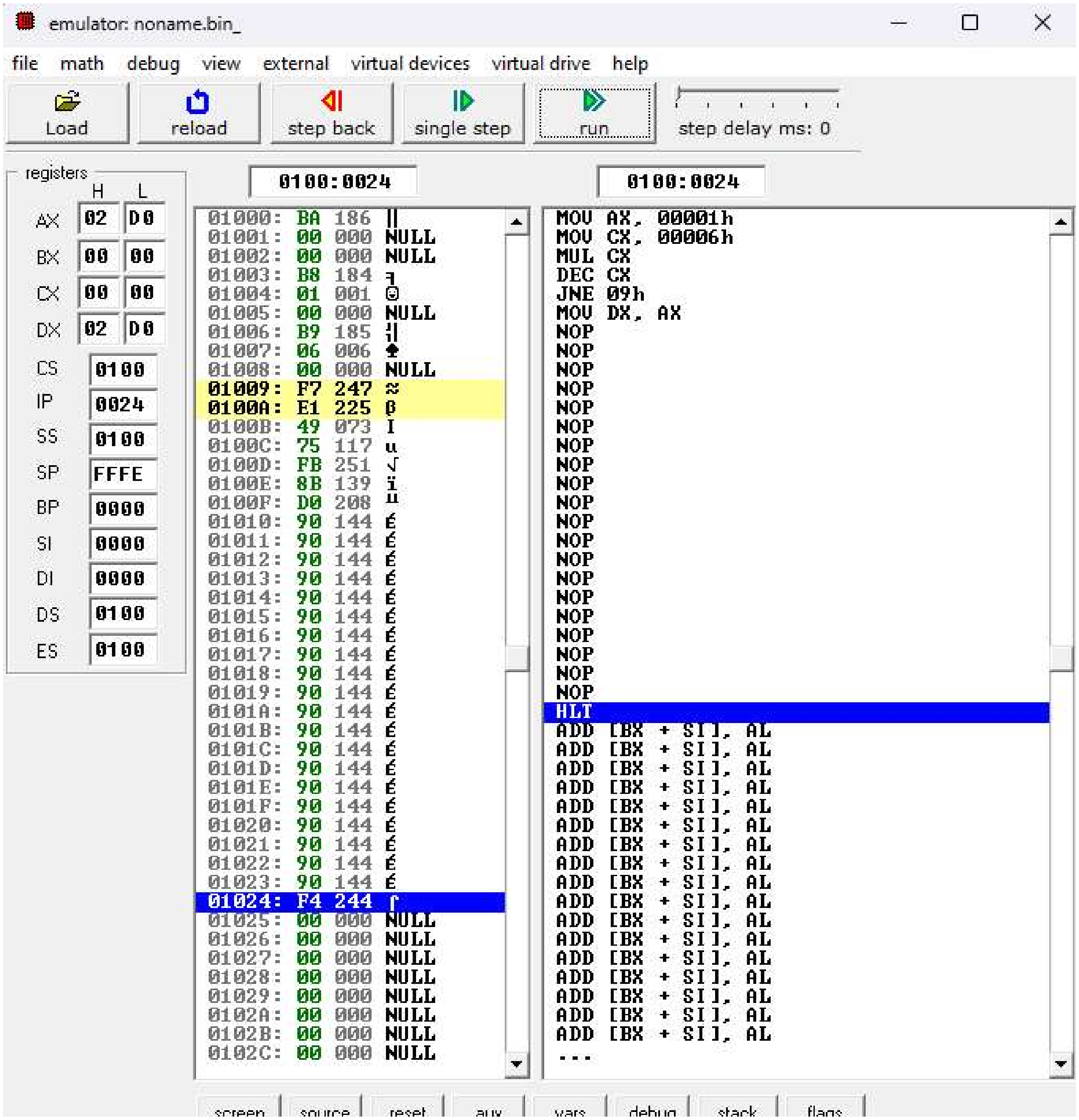


```
mov dx, 0000h
mov ax, 0001h
mov cx, 0006h

Loop:
mul cx
dec cx
jnz loop


mov dx,ax
```

Output:



Conclusion:

1. Explain shift instructions.
   Ans:-Shift instructions essentially perform bitwise shifts, which move bits to the left or right within a register or memory location. The bit positions at one end (left or right) are either discarded or filled with zeros (depending on the direction of the shift), and the bits at the other end are filled according to the specific type of shift being performed.

2. Explain rotate instructions.

Ans:-Rotate instructions are a subset of bitwise operations in assembly language and machine-level programming, used to rotate the bits of an operand (usually stored in a register) around the boundaries. Unlike shift operations, which discard the shifted-out bits, rotate operations move the bits that are shifted out back into the opposite end of the operand. This "circular" nature of rotating makes rotate instructions particularly useful in specific applications like cryptography, error detection, and bit manipulation tasks.

Types of Rotate Instructions:

1. **Rotate Left (ROL)**
   - **Purpose:** Rotates the bits of an operand to the left by a specified number of positions. The bits that are shifted out from the leftmost position are "wrapped around" to the rightmost positions.
   - **Operation:** Each bit moves to the left by the specified number of positions, and the bits shifted out from the left end are placed back at the right end.
   - **Example:**
     - `ROL R1, 1` – Rotate the contents of register `R1` to the left by 1 position.
   - **Effect:** If `R1 = 10010000` (binary), after `ROL R1, 1`, `R1` becomes `00100001` (binary).
2. **Rotate Right (ROR)**
   - **Purpose:** Rotates the bits of an operand to the right by a specified number of positions. The bits that are shifted out from the rightmost position are "wrapped around" to the leftmost positions.
   - **Operation:** Each bit moves to the right by the specified number of positions, and the bits shifted out from the right end are placed back at the left end.
   - **Example:**
     - `ROR R1, 1` – Rotate the contents of register `R1` to the right by 1 position.
   - **Effect:** If `R1 = 10010000` (binary), after `ROR R1, 1`, `R1` becomes `01001000` (binary).
3. **Rotate through Carry Left (RCL)**

- **Purpose:** Performs a rotate left operation, but instead of using zero to fill the rightmost bit, the carry flag (C) is used. The carry flag value is rotated into the leftmost bit, and the shifted-out leftmost bit is placed into the carry flag.
- **Operation:** The carry flag is included in the rotation, with the bit shifted out from the leftmost position being moved to the carry flag.
- **Example:**
  - `RCL R1, 1` – Rotate the contents of register `R1` to the left by 1 position, with the carry flag involved in the rotation.
- **Effect:** If the carry flag is `1` and `R1 = 10010000` (binary), after `RCL R1, 1`, `R1` would become `00100001` and the carry flag would become `1`.

4. **Rotate through Carry Right (RCR)**
   - **Purpose:** Performs a rotate right operation, but instead of using zero to fill the leftmost bit, the carry flag (C) is used. The carry flag value is rotated into the rightmost bit, and the shifted-out rightmost bit is placed into the carry flag.
   - **Operation:** The carry flag is included in the rotation, with the bit shifted out from the rightmost position being moved to the carry flag.
   - **Example:**
     - `RCR R1, 1` – Rotate the contents of register `R1` to the right by 1 position, with the carry flag involved in the rotation.
   - **Effect:** If the carry flag is `1` and `R1 = 10010000` (binary), after `RCR R1, 1`, `R1` would become `01001000` and the carry flag would become `1`.