



# Vidyavardhini's College of Engineering and Technology

Department of Artificial Intelligence & Data Science

AY: 2024-25

|                     |        |                     |                    |
|---------------------|--------|---------------------|--------------------|
| <b>Class:</b>       | SE     | <b>Semester:</b>    | IV                 |
| <b>Course Code:</b> | CSL404 | <b>Course Name:</b> | Microprocessor Lab |

|                                 |  |
|---------------------------------|--|
| <b>Name of Student:</b>         | Bhagyashri kaleni Sutar  |
| <b>Roll No. :</b>               | 75   |
| <b>Experiment No.:</b>          | 2  |
| <b>Title of the Experiment:</b> | A. Program to perform multiplication using MUL instruction<br>B. Program for calculating factorial using assembly language |
| <b>Date of Performance:</b>     |  |
| <b>Date of Submission:</b>      |  |

## 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 : Ms. Sweety Patil

Signature :

Date:



# Vidyavardhini's College of Engineering and Technology

## Department of Artificial Intelligence & Data Science

---

**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:**

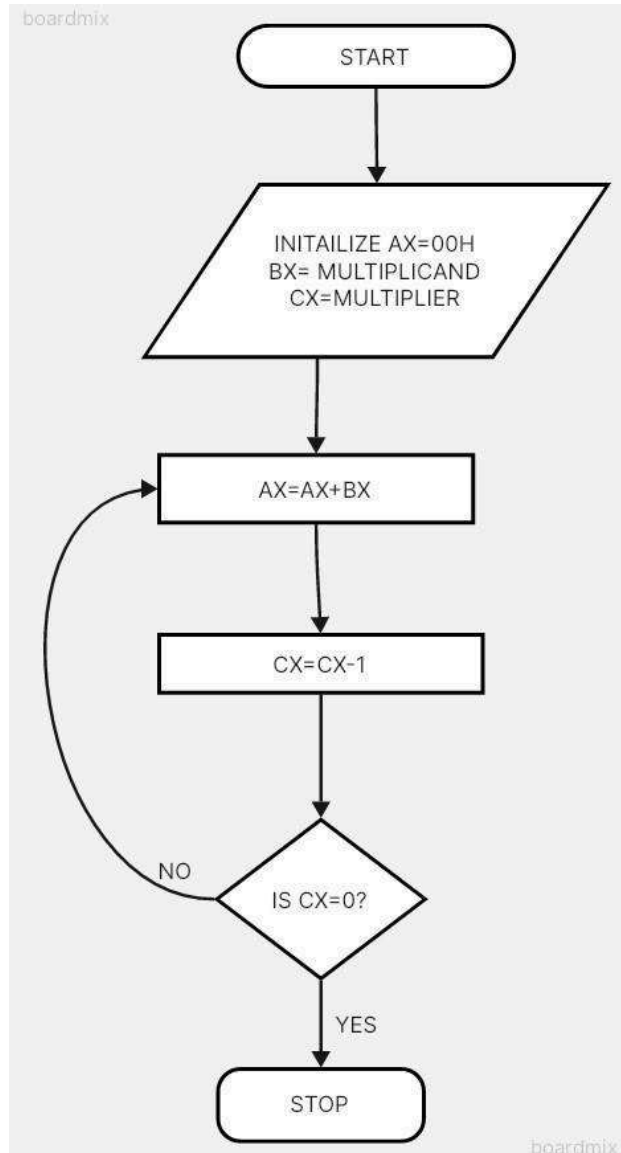
1. Start.
2. Set AX=00H, BX= Multiplicand, CX=Multiplier 3 Add the content of AX and BX.
4. Decrement content of CX.
5. Repeat steps 3 and 4 till CX=0.
6. Stop.

**Flowchart:**



# Vidyavardhini's College of Engineering and Technology

Department of Artificial Intelligence & Data Science



## Code:

```
01  
02 mov ax,0000H  
03 mov bx,0005H  
04 mov cx,0003H  
05  
06 loop:  
07 add ax,bx  
08 dec cx  
09 jnz loop  
10  
11  
12 mov dx,cx  
13  
14
```



# Vidyavardhini's College of Engineering and Technology

Department of Artificial Intelligence & Data Science

output:-

The screenshot shows an 8086 emulator window titled "emulator: noname.bin\_". The interface includes a menu bar (file, math, debug, view, external, virtual devices, virtual drive, help) and a toolbar with buttons for Load, reload, step back, single step, run, and a step delay slider set to 0 ms.

On the left, the "registers" panel displays the following values:

|    | 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 |    |

The main window displays assembly code in two columns. The left column shows memory addresses from 0100:0000 to 0102:C, with the instruction at 01024 (F4 244) highlighted in blue. The right column shows the corresponding assembly instructions, with the instruction at 0100:0024 (HLT) highlighted in blue.

```
0100:0000  B8 184 3 NULL
0100:0001  00 000 NULL
0100:0002  00 000 NULL
0100:0003  B8 187 3
0100:0004  05 005 5
0100:0005  00 000 NULL
0100:0006  B9 185 3
0100:0007  03 003 3
0100:0008  00 000 NULL
0100:0009  03 003 3
0100:000A  C3 195 3
0100:000B  49 073 3
0100:000C  75 117 3
0100:000D  FB 251 3
0100:000E  8B 139 3
0100:000F  D1 209 3
0100:0010  90 144 3
0100:0011  90 144 3
0100:0012  90 144 3
0100:0013  90 144 3
0100:0014  90 144 3
0100:0015  90 144 3
0100:0016  90 144 3
0100:0017  90 144 3
0100:0018  90 144 3
0100:0019  90 144 3
0100:001A  90 144 3
0100:001B  90 144 3
0100:001C  90 144 3
0100:001D  90 144 3
0100:001E  90 144 3
0100:001F  90 144 3
0100:0020  90 144 3
0100:0021  90 144 3
0100:0022  90 144 3
0100:0023  90 144 3
01024: F4 244 3
01025: 00 000 NULL
01026: 00 000 NULL
01027: 00 000 NULL
01028: 00 000 NULL
01029: 00 000 NULL
0102A: 00 000 NULL
0102B: 00 000 NULL
0102C: 00 000 NULL

0100:0024  MOV AX, 00000h
0100:0025  MOV BX, 00005h
0100:0026  MOV CX, 00003h
0100:0027  ADD AX, BX
0100:0028  DEC CX
0100:0029  JNE 09h
0100:002A  MOV DX, CX
0100:002B  NOP
0100:002C  NOP
0100:002D  NOP
0100:002E  NOP
0100:002F  NOP
0100:0030  NOP
0100:0031  NOP
0100:0032  NOP
0100:0033  NOP
0100:0034  NOP
0100:0035  NOP
0100:0036  NOP
0100:0037  NOP
0100:0038  NOP
0100:0039  NOP
0100:003A  NOP
0100:003B  NOP
0100:003C  NOP
0100:003D  NOP
0100:003E  NOP
0100:003F  NOP
0100:0040  HLT
0100:0041  ADD [BX + SI], AL
0100:0042  ADD [BX + SI], AL
0100:0043  ADD [BX + SI], AL
0100:0044  ADD [BX + SI], AL
0100:0045  ADD [BX + SI], AL
0100:0046  ADD [BX + SI], AL
0100:0047  ADD [BX + SI], AL
0100:0048  ADD [BX + SI], AL
0100:0049  ADD [BX + SI], AL
0100:004A  ADD [BX + SI], AL
0100:004B  ADD [BX + SI], AL
0100:004C  ADD [BX + SI], AL
0100:004D  ADD [BX + SI], AL
0100:004E  ADD [BX + SI], AL
0100:004F  ADD [BX + SI], AL
0100:0050  ...
```

[illegible]

### Conclusion:

1. Explain data transfer instructions.

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

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



# Vidyavardhini's College of Engineering and Technology

## Department of Artificial Intelligence & Data Science

---

- **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)



# Vidyavardhini's College of Engineering and Technology

## Department of Artificial Intelligence & Data Science

---

- **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)





# Vidyavardhini's College of Engineering and Technology

## Department of Artificial Intelligence & Data Science

---

- **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.



# Vidyavardhini's College of Engineering and Technology

## Department of Artificial Intelligence & Data Science

---

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

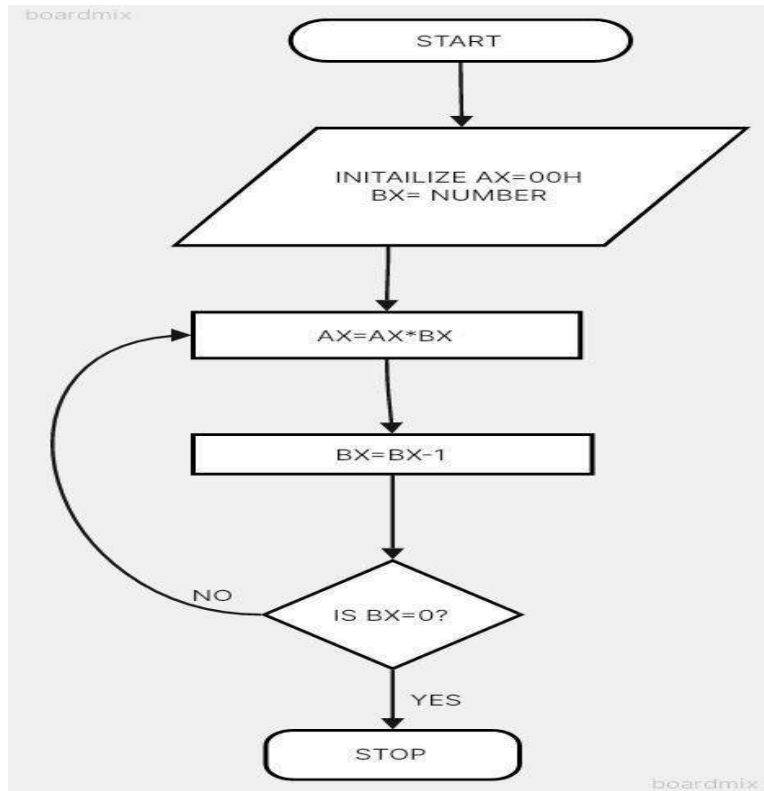
**Theory:**

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:**



### Code:

```
original source code
01 mov dx, 0000h
02 mov ax, 0001h
03 mov cx, 0006h
04
05 Loop:
06 mul cx
07 dec cx
08 jnz loop
09
10
11 mov dx, ax
12
13
14
15
16
```

**Output:**

The screenshot displays the x86-64 emulator interface. At the top, the title bar reads "emulator: noname.bin\_". Below it is a menu bar with options: file, math, debug, view, external, virtual devices, virtual drive, and help. A toolbar contains icons for Load, reload, step back, single step, run, and a step delay slider set to 0 ms.

The "registers" panel on the left shows the following values:

|    | H    | L  |
|----|------|----|
| AX | 02   | D0 |
| BX | 00   | 00 |
| CX | 00   | 00 |
| DX | 02   | D0 |
| CS | 0100 |    |
| IP | 0024 |    |
| SS | 0100 |    |
| SP | FFFE |    |
| BP | 0000 |    |
| SI | 0000 |    |
| DI | 0000 |    |
| DS | 0100 |    |
| ES | 0100 |    |

The main window is split into two panes. The left pane shows memory addresses from 0100:0024 to 0102C:0000. The right pane shows the corresponding assembly instructions. The instruction at address 01024:0000 is highlighted in blue.

| Address | Hex | Dec | Symbol | Instruction       |
|---------|-----|-----|--------|-------------------|
| 01000:  | BA  | 186 |        | MOV AX, 00001h    |
| 01001:  | 00  | 000 | NULL   | MOV CX, 00006h    |
| 01002:  | 00  | 000 | NULL   | MUL CX            |
| 01003:  | B8  | 184 | 7      | DEC CX            |
| 01004:  | 01  | 001 | 0      | JNE 09h           |
| 01005:  | 00  | 000 | NULL   | MOV DX, AX        |
| 01006:  | B9  | 185 |        | NOP               |
| 01007:  | 06  | 006 | 4      | NOP               |
| 01008:  | 00  | 000 | NULL   | NOP               |
| 01009:  | F7  | 247 | ~      | NOP               |
| 0100A:  | E1  | 225 | 0      | NOP               |
| 0100B:  | 49  | 073 | I      | NOP               |
| 0100C:  | 75  | 117 | u      | NOP               |
| 0100D:  | FB  | 251 | √      | NOP               |
| 0100E:  | 8B  | 139 | i      | NOP               |
| 0100F:  | D0  | 208 | u      | NOP               |
| 01010:  | 90  | 144 | É      | NOP               |
| 01011:  | 90  | 144 | É      | NOP               |
| 01012:  | 90  | 144 | É      | NOP               |
| 01013:  | 90  | 144 | É      | NOP               |
| 01014:  | 90  | 144 | É      | NOP               |
| 01015:  | 90  | 144 | É      | NOP               |
| 01016:  | 90  | 144 | É      | NOP               |
| 01017:  | 90  | 144 | É      | NOP               |
| 01018:  | 90  | 144 | É      | NOP               |
| 01019:  | 90  | 144 | É      | NOP               |
| 0101A:  | 90  | 144 | É      | HLT               |
| 0101B:  | 90  | 144 | É      | ADD [BX + SI], AL |
| 0101C:  | 90  | 144 | É      | ADD [BX + SI], AL |
| 0101D:  | 90  | 144 | É      | ADD [BX + SI], AL |
| 0101E:  | 90  | 144 | É      | ADD [BX + SI], AL |
| 0101F:  | 90  | 144 | É      | ADD [BX + SI], AL |
| 01020:  | 90  | 144 | É      | ADD [BX + SI], AL |
| 01021:  | 90  | 144 | É      | ADD [BX + SI], AL |
| 01022:  | 90  | 144 | É      | ADD [BX + SI], AL |
| 01023:  | 90  | 144 | É      | ADD [BX + SI], AL |
| 01024:  | F4  | 244 | ↑      | ADD [BX + SI], AL |
| 01025:  | 00  | 000 | NULL   | ADD [BX + SI], AL |
| 01026:  | 00  | 000 | NULL   | ADD [BX + SI], AL |
| 01027:  | 00  | 000 | NULL   | ADD [BX + SI], AL |
| 01028:  | 00  | 000 | NULL   | ADD [BX + SI], AL |
| 01029:  | 00  | 000 | NULL   | ADD [BX + SI], AL |
| 0102A:  | 00  | 000 | NULL   | ADD [BX + SI], AL |
| 0102B:  | 00  | 000 | NULL   | ADD [BX + SI], AL |
| 0102C:  | 00  | 000 | NULL   | ...               |

### Conclusion:



# Vidyavardhini's College of Engineering and Technology

## Department of Artificial Intelligence & Data Science

---

### 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)



# Vidyavardhini's College of Engineering and Technology

## Department of Artificial Intelligence & Data Science

---

- **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.



