

Practical – 1

Aim: Write a C program to perform the following conversions.

a) Decimal to Binary Conversion

```
#include <iostream>
#include <cmath>
using namespace std;
int convert(long long);
int main()
{
    long long n;
    cout << "Enter a decimal number: ";
    cin >> n;
    cout << n << " in decimal = " << convert(n) << " in binary";
    return 0;
}

int convert(long long n)
{
    int dec = 0, i = 0, rem;
    while (n!=0)
    {
        rem = n % 2;
        n /= 2;
        dec += rem * pow(10, i);
        ++i;
    }
    return dec;
}
```

Output:

12002040701140

```
Enter a decimal number: 14
14 in decimal = 1110 in binary

...Program finished with exit code 0
Press ENTER to exit console. □
```

b) Decimal to Hexadecimal Conversion

```
#include <stdio.h>

int main()
{
    int decnum, rem, i = 0;      char
    hexnum[10];
    printf("Enter any decimal number: ");
    scanf("%d", &decnum);
    while (decnum != 0)
    {
        rem = decnum % 16;
        if (rem < 10)
            rem = rem + 48;
        else
            rem = rem + 55;

        hexnum[i] = rem;

        i++;

        decnum = decnum / 16;
    }

    printf("\nEquivalent Value in Hexadecimal = ");

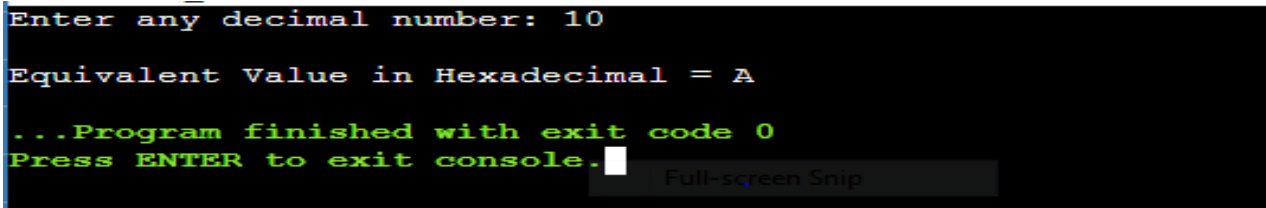
    for (i = i - 1; i >= 0; i--)
        printf("%c", hexnum[i]);

    12002040701140
```

```
return 0;
```

```
}
```

Output:



```
Enter any decimal number: 10
Equivalent Value in Hexadecimal = A
...Program finished with exit code 0
Press ENTER to exit console.
```

c) Binary to Decimal Conversion

```
#include <iostream>
#include <cmath>
using namespace std;
int convert(long long);
int main()
{
    long long n;
    cout << "Enter a binary number: ";
    cin >> n;
    cout << n << " in binary = " << convert(n) << " in decimal";
    return 0;
}
```

```
int convert(long long n)
```

```
{
    int dec = 0, i = 0, rem;
    while (n!=0) {
        rem = n % 10;
        n /= 10;
        dec += rem * pow(2, i);
        ++i;
    }
}
```

```
return dec;
```

12002040701140

}

Output:

```
Enter a binary number: 1101
1101 in binary = 13 in decimal

...Program finished with exit code 0
Press ENTER to exit console. □
```

Practical – 2

Aim: Write a C program to perform the following compliment Operations.

- a) 1's Complement
- b) 2's Complement

```
#include <stdio.h> int main()
{
    int n;

    printf("Enter the number of bits do you want to enter :");
    scanf("%d", &n);
    char binary[n + 1];
    char onescomplement[n + 1];
    char twoscomplement[n + 1];
    int carry = 1;
    printf("\nEnter the binary number : ");
    scanf("%s", binary);
    printf("%s", binary);
    printf("\nThe ones complement of the binary number is :");
    for (int i = 0; i < n; i++)
    {
        if (binary[i] == '0')
            onescomplement[i] = '1';
        else if (binary[i] == '1')
            onescomplement[i] = '0';
    }
    onescomplement[n] = '\0';
    printf("%s", onescomplement);
    printf("\nThe twos complement of a binary number is : ");
    for (int i = n - 1; i >= 0; i--)
    {
        if (onescomplement[i] == '1' && carry == 1)
        {
            twoscomplement[i] = '0';
        }
    }
}
```

```
else if (onescomplement[i] == '0' && carry == 1)
{
    twoscomplement[i]='1';
    carry = 0;
} else
{
    twoscomplement[i] = onescomplement[i];
}
}
twoscomplement[n]='\0';
printf("%s",twoscomplement); return 0;
}
```

Output:

```
Enter the number of bits do you want to enter :5
Enter the binary number : 11001
11001
The ones complement of the binary number is :00110
The twos complement of the binary number is : 00111
...Program finished with exit code 0
Press ENTER to exit console. □
```

Practical – 3

Aim: Write a C program to perform the following compliment Operations.

a) Circular shift left

```
#include<stdio.h>
```

```
int main()
```

```
{
```

```
    int num[10],i=0,temp=0,j;
```

```
    printf("Enter Binary number:");
```

```
    for(i=0;i<5;i++)
```

```
    {
```

```
        scanf("%d", &num[i]);
```

```
    }
```

```
for(i=0;i<5;i++)  
{  
    printf("%d", num[i]);  
}  
i--;  
temp=num[0];  
for(j=0;j<5;j++)  
{  
    num[j-1]=num[j];  
}  
num[i]=temp;  
for(i=0;i<5;i++)  
{  
    printf("%d", num[i]);  
}  
}
```

Output:

```
Enter Binary number:1  
0  
1  
1  
0  
1011001101  
...Program finished with exit code 0  
Press ENTER to exit console. □
```

b) Circular shift left

```
#include<stdio.h>
```

12002040701140


```
int main()
{
    int num[10],i=0,temp=0,j;
    printf("Enter Binary number:");
    for(i=0;i<5;i++)
    {
        scanf("%d", &num[i]);
    }
    for(i=0;i<5;i++)
    {
        printf("%d", num[i]);
    }
    i--;
    temp=num[i];
    for(j=(i-1);j>=0;j--)
    {
        num[j+1]=num[j];
    }
    num[0]=temp;
    for(i=0;i<5;i++)
    {
        printf("%d", num[i]);
    }
}
```

Output:

```
Enter Binary number:0
0
1
0
1
0010110010
...Program finished with exit code 0
Press ENTER to exit console. □
```

Practical – 4

Aim: Write the working of 8085 Simulator GNUsim8085 and basic architecture of 8085 along with small introduction.

GNU Simulator 8085

8085 simulator is software on which instructions are executed by writing the programs in assembly language.

GNUSim8085 is an 8085-microprocessor simulator with following features.

- A simple editor component with syntax highlighting.
- A keypad to input assembly language instructions with appropriate arguments.
- Easy view of register contents.
- Easy view of flag contents.
- Hexadecimal <--> Decimal converter.
- View of stack, memory and I/O contents.
- Support for breakpoints for programming debugging.
- Stepwise program execution.
- One clicks conversion of assembly program to opcode listing.
- Printing support (known not to work well on Windows).
- UI translated in various languages.

A basic assembly program consists of 4 parts.

1. Labels

2. Operations: - These operations can be specified as

- **Machine operations (mnemonics):** - Used to define operations in the form of opcode as mention in the instruction set of microprocessors 8085.
- **Pseudo operations (like preprocessor in C):** - These are assembly directives.

3. Operands

4. Comments

Registers of 8085 microprocessor

- A **microprocessor** is a multipurpose, programmable, clock-driven, register-based electronic device that reads binary instructions from a storage device called memory, accepts binary data as input and processes data according to those instructions and provide results as output.
- An 8085 microprocessor, is a second generation 8-bit microprocessor and is the base for studying and using all the microprocessor available in the market.

Registers in 8085:

(a) General Purpose Registers –

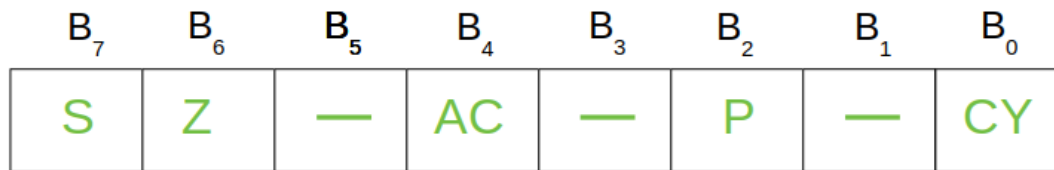
- The 8085 has six general-purpose registers to store 8-bit data; these are identified as- B, C, D, E, H, and L. These can be combined as register pairs – BC, DE, and HL, to perform some 16-bit operation. These registers are used to store or copy temporary data, by using instructions, during the execution of the program.

(b) Specific Purpose Registers –

➤ **Accumulator:**

- The accumulator is an 8-bit register (can store 8-bit data) that is the part of the arithmetic and logical unit (ALU).
- After performing arithmetical or logical operations, the result is stored in accumulator. Accumulator is also defined as register A.

➤ **Flag registers:**



fig(a)-Bit position of various flags in flag registers of 8085

- The flag register is a special purpose register and it is completely different from other registers in microprocessor.
- It consists of 8 bits and only 5 of them are useful.
- The other three are left vacant and are used in the future Intel versions.
- These 5 flags are set or reset (when value of flag is 1, then it is said to be set and when value is 0, then it is said to be reset) after an operation according to data condition of the result in the accumulator and other registers.

The 5 flag registers are:

1. **Sign Flag:** It occupies the seventh bit of the flag register, which is also known as the most significant bit. It helps the programmer to know whether the number stored in the accumulator is positive or negative. If the sign flag is set, it means that number stored in the accumulator is negative, and if reset, then the number is positive.
2. **Zero Flag:** It occupies the sixth bit of the flag register. It is set, when the operation performed in the ALU results in zero (all 8 bits are zero), otherwise it is reset. It helps in determining if two numbers are equal or not.
3. **Auxiliary Carry Flag:** It occupies the fourth bit of the flag register. In an arithmetic operation, when a carry flag is generated by the third bit and passed on to the fourth bit, then Auxiliary Carry flag is set. If not, flag is reset. This flag is used internally for BCD (Binary-Coded decimal Number) operations.

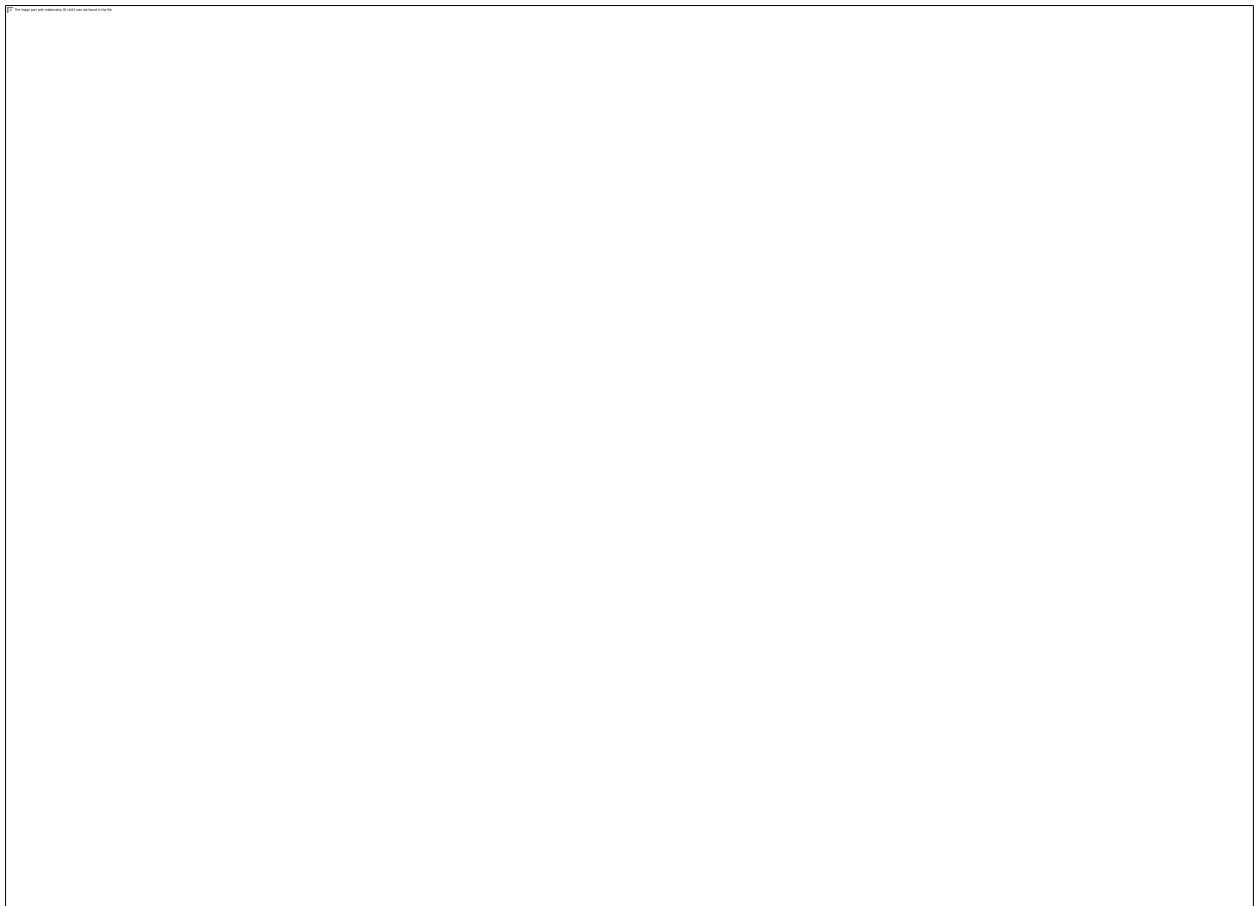
Note – This is the only flag register in 8085 which is not accessible by user.

4. **Parity FlagL:** It occupies the second bit of the flag register. This flag tests for number of 1's in the accumulator. If the accumulator holds even number of 1's, then this flag is set and it is said to even parity. On the other hand, if the number of 1's is odd, then it is reset and it is said to be odd parity.
5. **Carry Flag:** It occupies the zeroth bit of the flag register. If the arithmetic operation results in a carry (if result is more than 8 bit), then Carry Flag is set; otherwise, it is reset.

(c) Memory Registers –

- There are two 16-bit registers used to hold memory addresses. The size of these registers is 16 bits because the memory addresses are 16 bits. They are: -
- **Program Counter:** This register is used to sequence the execution of the instructions. The function of the program counter is to point to the memory address from which the next byte is to be fetched. When a byte (machine code) is being fetched, the program counter is incremented by one to point to the next memory location.
 - **Stack Pointer:** It is used as a memory pointer. It points to a memory location in read/write memory, called the stack. It is always incremented/decremented by 2 during push and pop operation.

ARCHITECTURE OF MICROPROCESSOR 8085



Instruction Decoder: - Instruction decoder identifies the instructions. It takes the information from instruction register and decodes the instruction to be performed.

Program Counter: -It is a 16-bit register used as memory pointer. It stores the memory address of the next instruction to be executed. So, we can say that this register is used to sequencing the program. Generally, the memory has 16 bit addresses so that it has 16-bit memory. The program counter is set to 0000H.

Stack Pointer: -It is also a 16-bit register used as memory pointer. It points to the memory location called stack. Generally, stack is a reserved portion of memory where information can be stores or taken back together.

Timing and Control Unit: -It provides timing and control signal to the microprocessor to perform the various operation. It has three control signals. It controls all external and internal circuits. It operates with reference to clock signal. It synchronizes all the data transfers. There are three control signals: 1.ALE-Airthmetic Latch Enable, It provides control signal to synchronize the components of microprocessor. 2.RD- This is active low used for reading operation. 3.WR-This is active low used for writing operation. There are three status signals used in microprocessor S0, S1 and IO/M. It changes its status according the provided input to these pins.

Serial Input Output Control-There are two pins in this unit. This unit is used for serial data communication.

Interrupt Unit-There are 6 interrupt pins in this unit. Generally, an external hardware is connected to these pins. These pins provide interrupt signal sent by external hardware to microprocessor and microprocessor sends acknowledgement for receiving the interrupt signal. Generally, INTA is used for acknowledgement.

Labels: - When given to any particular instruction/data in a program, takes the address of that instruction or data as its value. But it has different meaning when given to EQU directive. Then it takes the operand of EQU as its value. Labels must always be placed in the first column and must be followed by an instruction (no empty line). Labels must be followed by a : (colon), to differentiate it from other tokens.

Operations: -

As mentioned above the operations can be specified in two ways that are **mnemonics** and **pseudo operation**.

Pseudo operations can be defined by using following directives: -

There are only 3 directives currently available in our assembly language.

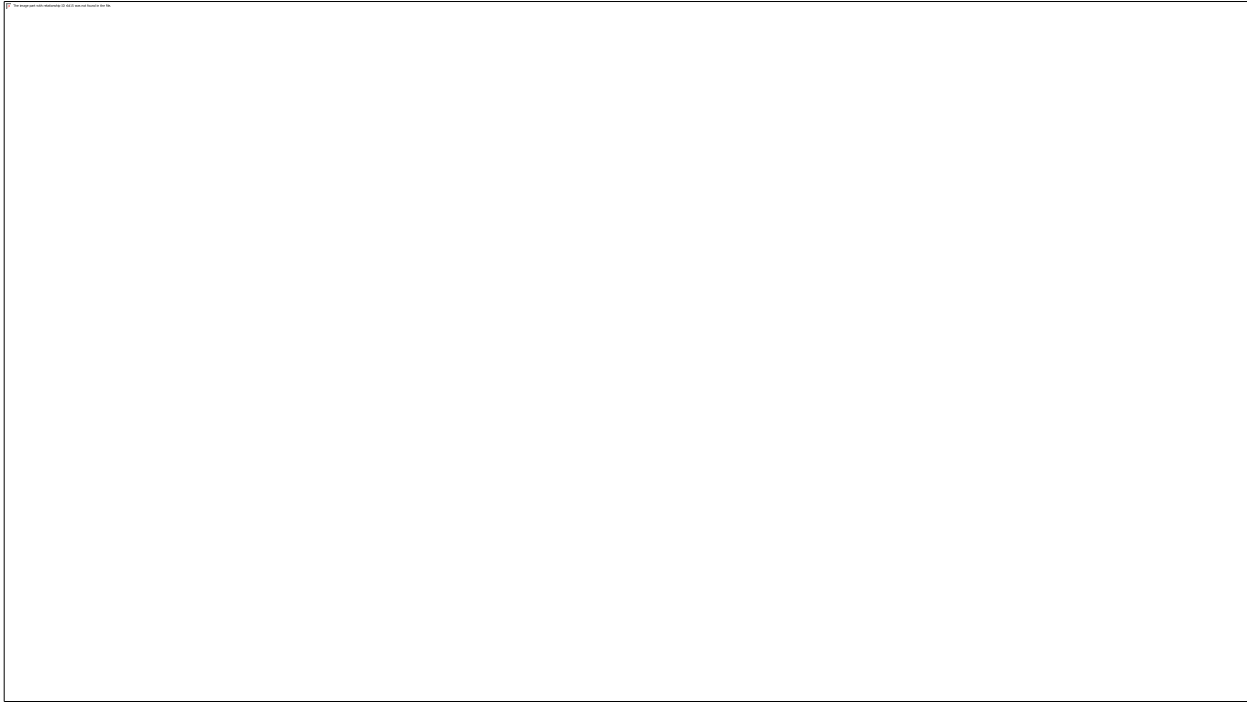
- 1. DB - define byte (8 bits)**
- 2. DS - define size (no. of bytes)**
- 3. EQU - like minimalistic #define in C**

- DB is used to define space for an array of values specified by comma separated list. And the label (If given to the beginning of DB) is assigned the address of the first data item.
- DS is used to define the specified number of bytes to be assigned and initialize them to zero. To access each byte, you can use the + or -operator along with label.
- EQU behaves similar to #define in C. But it is simple. It can be used to give names only to numeric constants. Nesting of EQU is not allowed. You can use EQU only in operands for pseudo-ops and mnemonics.

Operands: -

Operands are specified according to the user. The register set specified in the architecture of 8085 (A, B, C, D, H and L) are used to access and store data. These registers are specified as operand. In case of accessing data or storing data in the memory 'm' is specified as an operand and the address of this memory location is taken from the HL pair (data in HL pair).

Getting Started with GNU Simulator 8085:



Pin Description

The following describes the function of each pin:

- **A6 - A1s (Output 3 State)**
Address Bus; The most significant 8 bits of the memory address or the 8 bits of the I/O address, 3 stated during Hold and Halt modes.
- **AD0 - 7 (Input/output 3state)**
Multiplexed Address/Data Bus; Lower 8 bits of the memory address (or I/O addresses) appear on the bus during the first clock cycle of a machine state. It then becomes the data bus during the second and third clock cycles. 3 stated during Hold and Halt modes.
- **ALE (Output)**
Address Latch Enable: It occurs during the first clock cycle of a machine state and enables the address to get latched into the on-chip latch of peripherals. The falling edge of ALE is set to guarantee setup and hold times for the address information.
ALE can also be used to strobe the status information. ALE is never 3stated.
- **S0, S1 (Output)**
Data Bus Status. Encoded status of the bus cycle:
S1 S0
O O HALT

0 1 WRITE

1 0 READ

1 1 FETCH

S1 can be used as an advanced R/W status.

➤ **RD (Output 3state)**

READ; indicates the selected memory or I/O device is to be read and that the Data Bus is available for the data transfer.

➤ **WR (Output 3state)**

WRITE; indicates the data on the Data Bus is to be written into the selected memory or I/O location. Data is set up at the trailing edge of WR. 3stated during Hold and Halt modes.

➤ **READY (Input)**

If Ready is high during a read or write cycle, it indicates that the memory or peripheral is ready to send or receive data. If Ready is low, the CPU will wait for Ready to go high before completing the read or write cycle.

➤ **HOLD (Input)**

HOLD; indicates that another Master is requesting the use of the Address and Data Buses.

The CPU, upon receiving the Hold request, will relinquish the use of buses as soon as the completion of the current machine cycle. Internal processing can continue.

The processor can regain the buses only after the Hold is removed. When the Hold is acknowledged, the Address, Data, RD, WR, and IO/M lines are 3stated.

➤ **HLDA (Output)**

HOLD ACKNOWLEDGE; indicates that the CPU has received the Hold request and that it will relinquish the buses in the next clock cycle. HLDA goes low after the Hold request is removed. The CPU takes the buses one half clock cycle after HLDA goes low.

➤ **INTR (Input)**

INTERRUPT REQUEST; is used as a general-purpose interrupt. It is sampled only during the next to the last clock cycle of the instruction. If it is active, the Program Counter (PC) will be inhibited from incrementing and an INTA will be issued. During this cycle a **RESTART** or **CALL** instruction can be inserted to jump to the interrupt service routine. The INTR is enabled and disabled by software. It is disabled by Reset and immediately after an interrupt is accepted.

➤ **INTA (Output)**

INTERRUPT ACKNOWLEDGE; is used instead of (and has the same timing as) RD during the Instruction cycle after an INTR is accepted. It can be used to activate the 8259 Interrupt chip or

some other interrupt ports.

RST 5.5

RST 6.5 - (Inputs)

RST 7.5

RESTART INTERRUPTS; These three inputs have the same timing as I NTR except

they cause an internal RESTART to be automatically inserted.

RST 7.5 ~ Highest Priority

RST 6.5

RST 5.5 o Lowest Priority

The priority of these interrupts is ordered as shown above. These interrupts have a higher priority than the INTR.

➤ **TRAP (Input)**

Trap interrupt is a no maskable restart interrupt. It is recognized at the same time as INTR. It is unaffected by any mask or Interrupt Enable. It has the highest priority of any interrupt.

➤ **RESET IN (Input)**

Reset sets the Program Counter to zero and resets the Interrupt Enable and HLDA flipflops. None of the other flags or registers (except the instruction register) are affected. The CPU is held in the reset condition as long as Reset is applied.

➤ **RESET OUT (Output)**

Indicates CPIJ is being reset. Can be used as a system RESET. The signal is synchronized to the processor clock.

➤ **X1, X2 (Input)**

Crystal or R/C network connections to set the internal clock generator X1 can also be an external clock input instead of a crystal. The input frequency is divided by 2 to give the internal operating frequency.

➤ **CLK (Output)**

Clock Output for use as a system clock when a crystal or R/ C network is used as an input to the CPU. The period of CLK is twice the X1, X2 input period.

➤ **IO/M (Output)**

IO/M indicates whether the Read/Write is to memory or I/O Tristate during Hold and Halt modes.

➤ **SID (Input)**

Serial input data line the data on this line is loaded into accumulator bit 7 whenever a RIM instruction is executed.

➤ **SOD (output)**

Serial output data line. The output SOD is set or reset as specified by the SIM instruction.

➤ **Vcc**

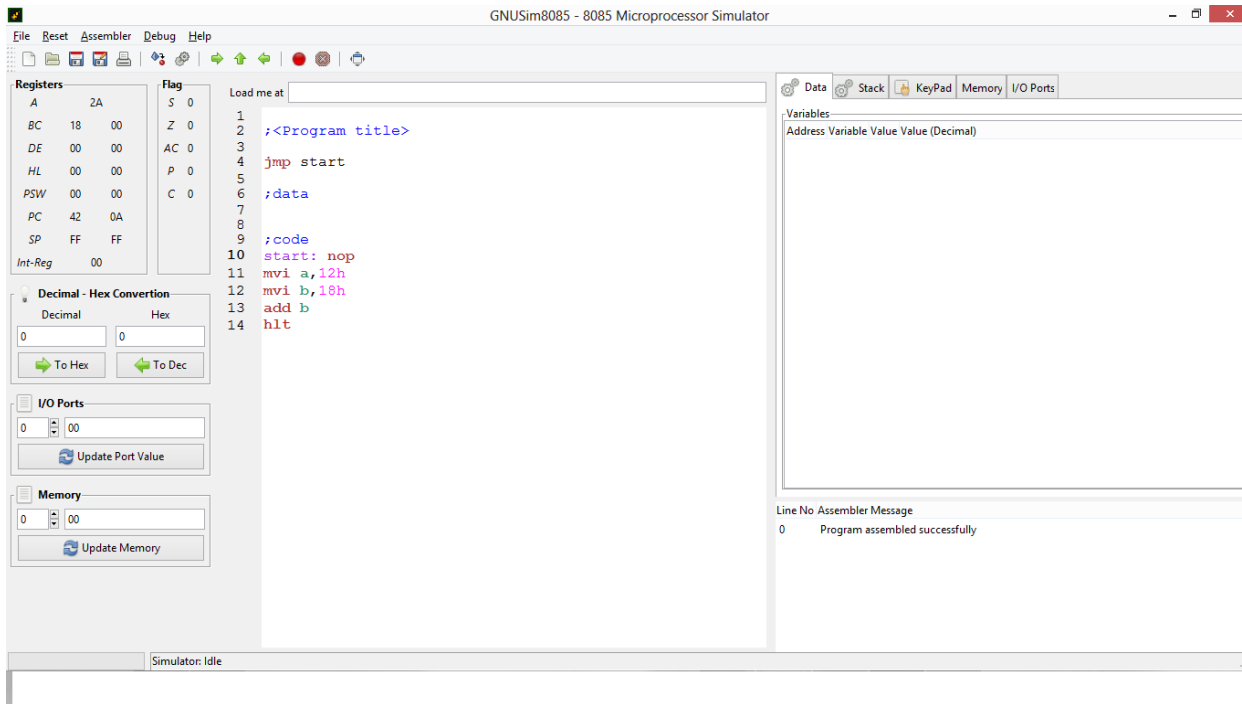
+5-volt supply.

➤ **Vss**

Ground Reference.

Program: ADDITION OF TWO NUMBERS

Source Code:



Source Code:

```
lda var1
mov b,a
lda var2
add b
sta var3
hlt
var1: db 04h
var2: db 09h
var3: db 00h
```

Practical – 5

Aim: Write an Assembly Language Program to perform the addition of two 8-bit numbers.

A. Addition of two 8-bit numbers.

MVI: Move immediate data to a register or memory location.

- **MVI** is a mnemonic, which actually means “Move Immediate”. With this instruction, we can load a register with an 8-bit or 1-Byte value.
- This instruction supports immediate addressing mode for specifying the data in the instruction.
- In the instruction “d8” stands for any 8-bit data, and ‘r’ stands for any one of the registers e.g. A, B, C, D, E, H or L. So this r can replace any one of the seven registers.
- As ‘r’ can have any of the seven register names, so there are seven opcodes for this type of instruction. It occupies 2-Bytes in the memory.

Mnemonics, Operand	Bytes
MVI A, Data	2
MVI B, Data	2
MVI C, Data	2
MVI D, Data	2
MVI E, Data	2
MVI H, Data	2
MVI L, Data	2

MVI Rd, #30H

- 30h is stored in register Rd

MVI M, #30H

- 30h is stored in memory location pointed by HL Reg
- 2 byte instruction

Examples:

MVI B, 10H [Loads the 8 bits of the 2nd byte in to the register specified]

MVI M, 30H

ADD

The content of operand are added to the content of the accumulator and the result is stored in Accumulator.

- **ADD R** is a mnemonic that stands for “Add contents of R to Accumulator”.
- As addition is a binary operation, so it requires two operands to be operated on.
- So input operands will reside on Accumulator and R registers and after addition the result will be stored back on to Accumulator.
- In this case, “R” stands for any of the following registers or memory location M pointed by HL pair.

R = A, B, C, D, E, H, L, or M

- It is 1-Byte instruction so occupies only 1-Byte in memory. As R can have any of the eight values, there are eight opcodes for this type of instruction.

Mnemonics, Operand	Bytes
ADD A	1
ADD B	1
ADD C	1
ADD D	1
ADD E	1
ADD H	1
ADD L	1
ADD M	1

ADD	R	A = A + R	ADD B
ADD	M	A = A + Mc	ADD 2050

Source code:

```
jmp start
```

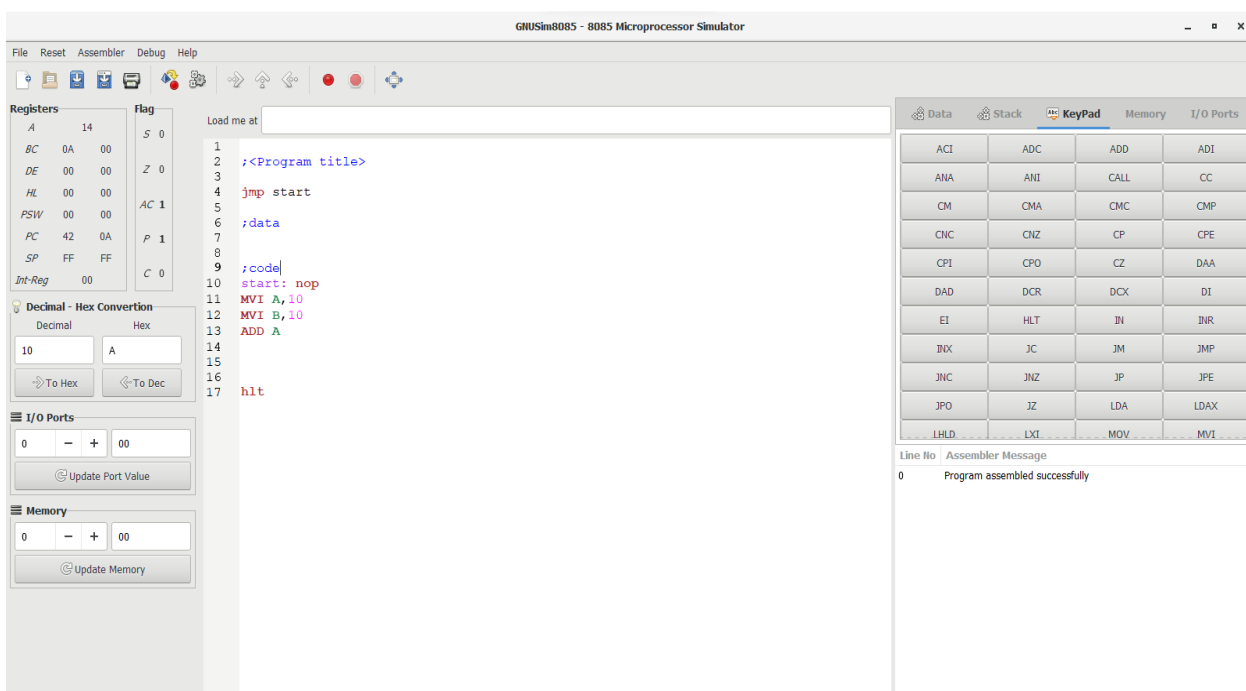
```
;data
```

12002040701140

```
;code
start: nop
MVI A,10
MVI B,10
ADD A
```

```
Hlt
```

Output:



B. Addition of two 8-bit numbers.

MOV: Copy the data from one place to another.

MOV Rd, Rs

→ Copies the content of Rs to Rd (MOV 1-byte instruction)

MOV M, Rs

→ **MOV M, r** will copy 8-bit value from the register r to the memory location as pointed by

12002040701140

HL register pair. This instruction uses register addressing for specifying the data.

→ As “r” can have any one of the seven values –

$r = A, B, C, D, E, H, \text{ or } L$

→ Copies the content of register Rs to memory location pointed by HL Register

→ 2-byte instruction

→ Copy the data byte from the register in to memory specified by the address in HL register.

→ Thus, there are seven opcodes for this type of instruction. It occupies only 1-Byte in memory.

Mnemonics, Operand	Bytes
MOV M, A	1
MOV M, B	1
MOV M, C	1
MOV M, D	1
MOV M, E	1
MOV M, H	1
MOV M, L	1

MOV Rd, M

→ Copies the content of memory location pointed by the HL register to the register Rd.

→ 2 byte instruction

→ Copy the data byte in to the register from the memory specified by the address in HL register.

→ **MOV r, M** is an instruction where the 8-bit data content of the memory location as pointed by HL register pair will be moved to the register r. Thus this is an instruction to load register r with the 8-bit value from a specified memory location whose 16-bit address is in HL register pair.

→ As r can have any of the seven values, there are seven opcodes for this type of instruction.

$r = A, B, C, D, E, H, \text{ or } L$

Mnemonics, Operand	Bytes
MOV A, M	1
MOV B, M	1
MOV C, M	1

MOV D, M	1
MOV E, M	1
MOV H, M	1
MOV L, M	1

Examples:

MVI B, 10h
MOV A, B
MOV M, B
MOV C, M

LDA: Load Accumulator

(This instruction copies the data from a given 16 bit address to the Accumulator)

- **LDA** is a mnemonic that stands for Load Accumulator with the contents from memory.
- In this instruction Accumulator will get initialized with 8-bit content from the 16-bit memory address as indicated in the instruction as a16.
- This instruction uses absolute addressing for specifying the data.
- It occupies 3-Bytes in the memory.
- First Byte specifies the opcode, and the successive 2-Bytes provide the 16-bit address, i.e. 1-Byte each for each memory location.
- 3 byte instruction

LDA 3000H

- Load the data byte from Memory into Accumulator specified by 16 bit address
- Content of memory location 3000h is copied in accumulator

Example:1

start: nop
LDA var1
hlt
var1: db 04h

Example:2 (Store 45H data to Specific Memory Location 000BH)

start: nop
lxi h, 000Bh
MVI M, 45H
LDA 000Bh
Hlt

STA

The content of accumulator is copied into the memory location.

STA 16 BIT

- **STA** is a mnemonic that stands for SToRe Accumulator contents in memory.
- In this instruction, Accumulator 8-bit content will be stored to a memory location whose 16-bit address is indicated in the instruction as a16.
- This instruction uses absolute addressing for specifying the destination.
- This instruction occupies 3-Bytes of memory.
- First Byte is required for the opcode, and next successive 2-Bytes provide the 16-bit address divided into 8-bits each consecutively.
- 3 byte instruction
- Load the data byte from A into the memory specified by 16 bit address

STA 2060H

Example:

```
start: nop
MVI A, 45h
STA 000Bh
Hlt
```

Source code:

```
jmp start
```

```
;data
```

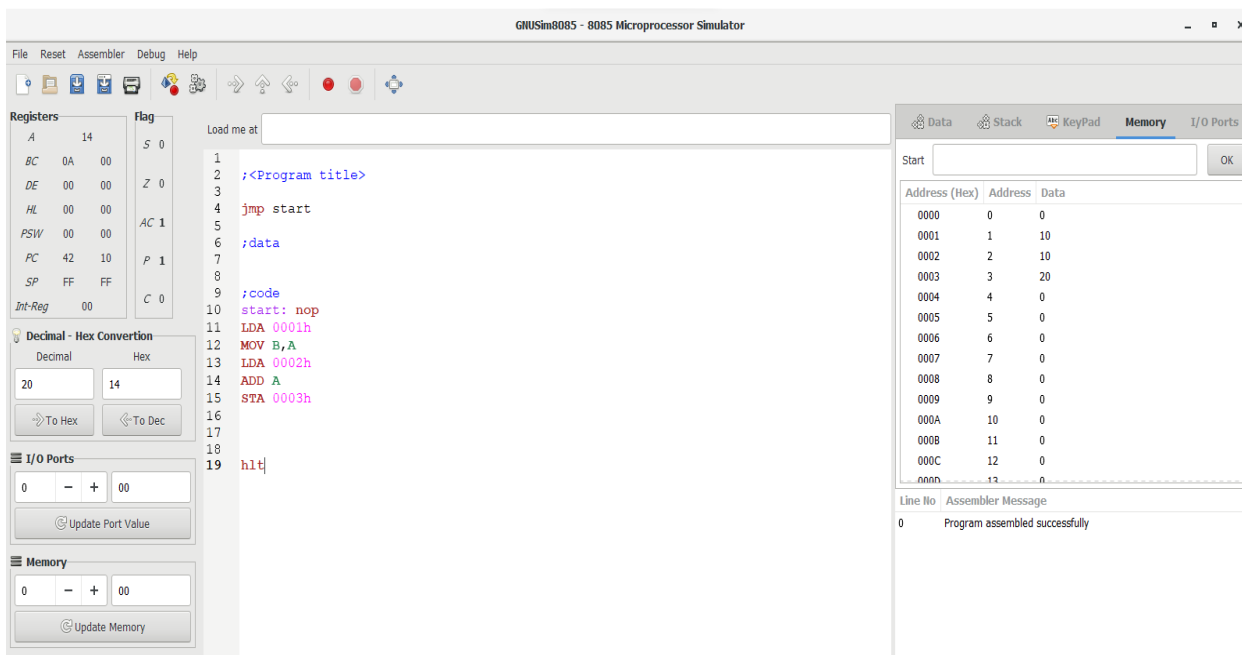
```
;code
```

```
start: nop
LDA 0001h
MOV B,A
LDA 0002h
ADD A
STA 0003h
```

```
Hlt
```

Output:

12002040701140



- C. Add the contents of memory locations 4000H and 4001H and place the result in memory location 4002H.

LXI: (Load register pair immediate)

The instruction loads 16-bit data in the register pair designated in the operand.

LXI Rp, 16 bit

- 3 byte instruction
- Load immediate 16 bit no. in a register pair
- These instructions are used to load the **16-bit** address into the register pair.
- We can use this instruction to load data from memory location using the memory address, which is stored in the register pair **rp**.
- The rp can be BC, DE, HL or SP.
- The LXI instructions and their Hex-codes are as follows.

Mnemonics, Operand	Bytes
LXI B	3
LXI D	3
LXI H	3
LXI SP	3

Example:
LXI B,2050H

INX Rp

INX is a mnemonic that stands for “Increment extended register” and **rp** stands for register pair. And it can be any one of the following register pairs.

rp = BC, DE, or HL

- 2 byte instruction
- This instruction will be used to add 1 to the present content of the rp. And thus the result of the incremented content will remain stored in rp itself.
- Though it is an arithmetic instruction, note that, flag bits are not at all affected by the execution of this instruction.
- A register pair is generally used to store 16-bit memory address.
- If flag bits got affected during increment of a memory address, then it may cause problems in many cases.
- So as per design of 8085, flag bits are not getting affected by the execution of this instruction **INXrp**.
- As rp can have any one of the three values, there are three opcodes for this type of instruction. It occupies only 1-Byte in memory.

Mnemonics, Operand	Bytes
INX B	1
INX D	1
INX H	1

- Example:
start: nop
LXI B, 4523h
INX B
Hlt

1. Sample problem
2. (4000H)=14H
3. (4001H)=89H
4. Result=14H+89H=9DH
- 5.
6. Source program
7. LXI H 4000H: "HL points 4000H"
8. MOV A, M : "Get first operand"
9. INX H : "HL points 4001H"
10. ADD M : "Add second operand"
11. INX H : "HL points 4002H"
12. MOV M, A : "Store result at 4002H"
13. HLT : "Terminate program execution"

Practical – 6

Aim: Write an Assembly Language Program to perform the subtraction of two 8-bit numbers.

Source code:

```
jmp start
```

```
;data
```

```
;code
```

```
start: nop
```

```
MVI A, 08h
```

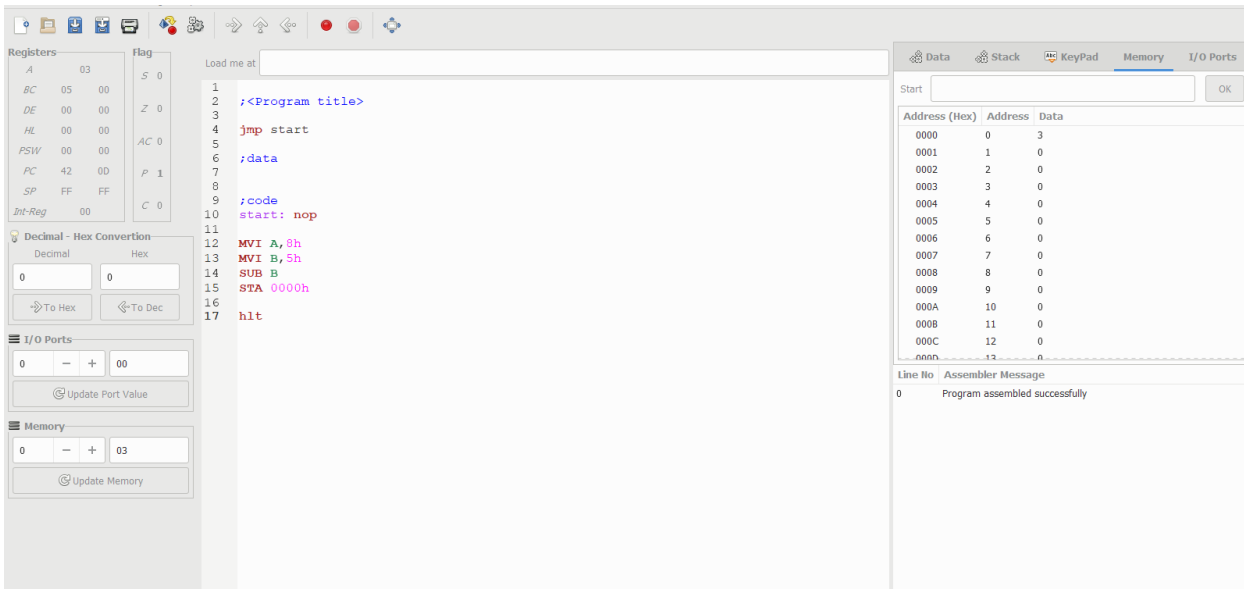
```
MVI B, 05h
```

```
SUB B
```

```
STA 0000h
```

```
Hlt
```

Output:



The screenshot shows an 8085 assembly simulator interface. The main window displays the assembly code being executed:

```
1 ;<Program title>
2
3
4 jmp start
5
6 ;data
7
8
9 ;code
10 start: nop
11
12 MVI A, 08h
13 MVI B, 05h
14 SUB B
15 STA 0000h
16
17 hlt
```

On the left, the Registers window shows the state of the 8085 registers:

Register	Value
A	03
B	05
C	00
D	00
E	00
F	00
H	00
L	00
PSW	00 00
PC	42 00
SP	FF FF
Int-Reg	00

The I/O Ports window shows the port value as 00. The Memory window shows the data stored at various addresses:

Address (Hex)	Address	Data
0000	0	3
0001	1	0
0002	2	0
0003	3	0
0004	4	0
0005	5	0
0006	6	0
0007	7	0
0008	8	0
0009	9	0
000A	10	0
000B	11	0
000C	12	0
...

The Assembler Message window at the bottom shows the message: "Program assembled successfully".

OR

12002040701140

Source code:

```
jmp start
```

```
;data
```

```
;code
```

```
start: nop
```

```
LXI H,3001H
```

```
MOV A,M
```

```
INX H;3002H
```

```
MOV C,M
```

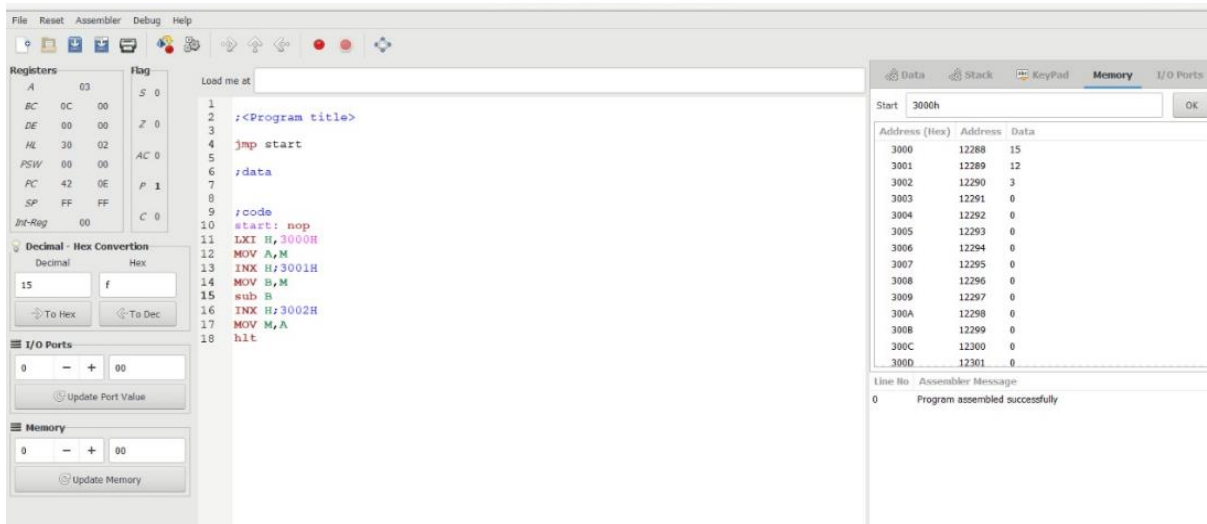
```
sub C
```

```
INX H;3003H
```

```
MOV M,A
```

```
Hlt
```

Output:



The screenshot shows an 8085 assembler simulator interface. The main window displays the assembly code being processed, with line numbers 1 through 18. The code is as follows:

```
1 ;<Program title>
2 jmp start
3
4 ;data
5
6 ;code
7 start: nop
8 LXI H,3000H
9 MOV A,M
10 INX H;3001H
11 MOV B,M
12 sub B
13 INX H;3002H
14 MOV M,A
15 Hlt
```

The left panel shows the registers (A, B, C, D, E, H, L, P, S, SP) and flags (S, Z, AC, P, C). The right panel shows the memory dump, starting at address 3000H. The memory dump shows the following data:

Address (Hex)	Address	Data
3000	12288	15
3001	12289	12
3002	12290	3
3003	12291	0
3004	12292	0
3005	12293	0
3006	12294	0
3007	12295	0
3008	12296	0
3009	12297	0
300A	12298	0
300B	12299	0
300C	12300	0
300D	12301	0

The bottom panel shows the assembler message: "Program assembled successfully".

12002040701140

Practical - 7

Aim: Write an Assembly Language Program to find 1's & 2's complement of two 8-bit numbers.

Source Code :

```
jmp start
```

```
;data
```

```
;code
```

```
start: nop
```

```
LDA 0001h
```

```
CMA
```

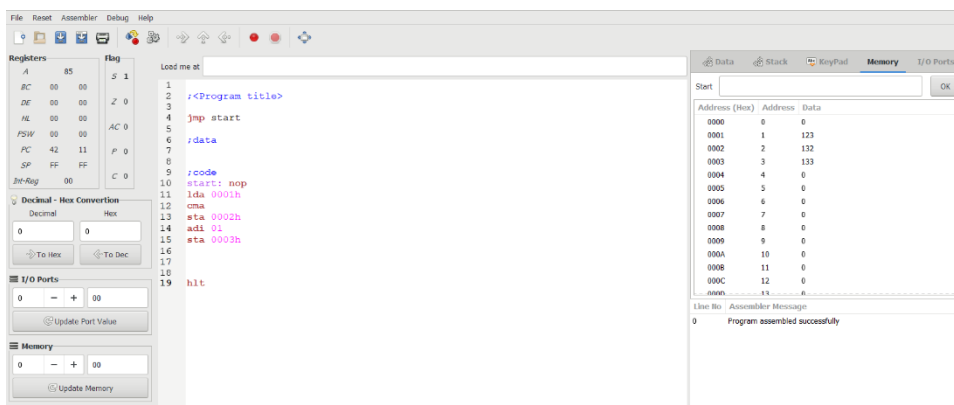
```
STA 0002h
```

```
ADI 1
```

```
STA 0003h
```

```
Hlt
```

Output :



12002040701140

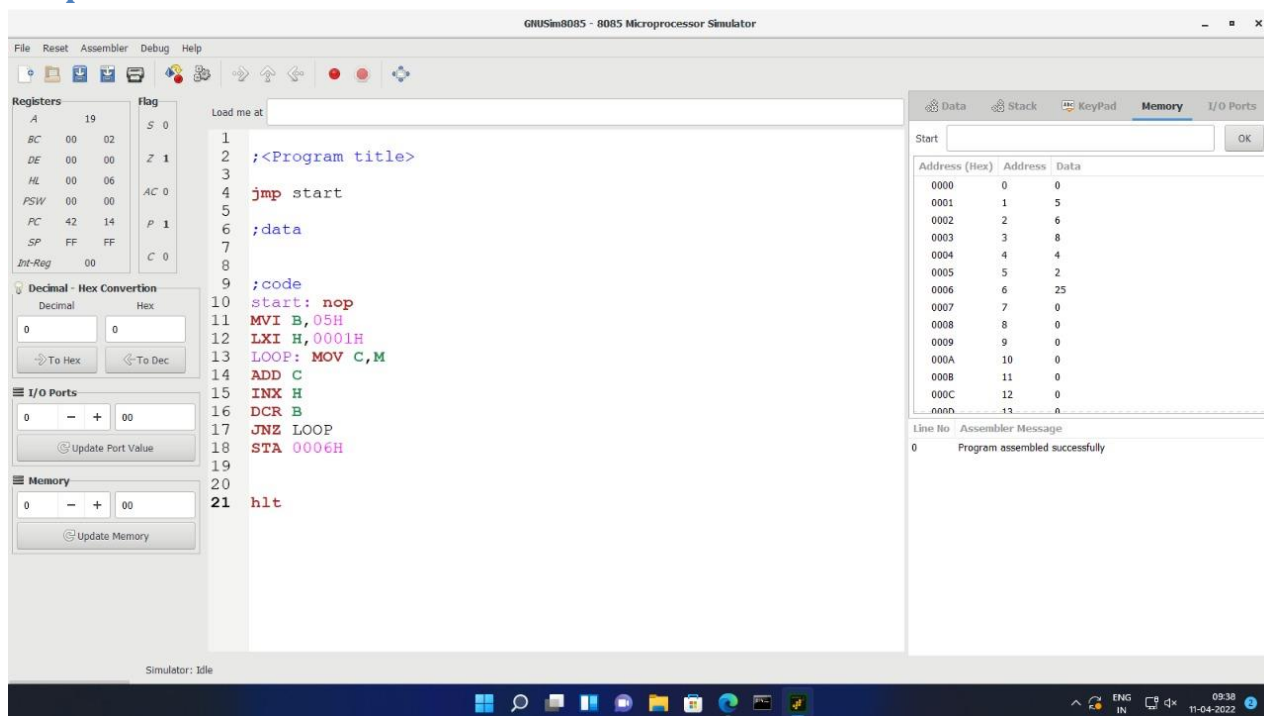
Practical-8

Aim: Write an Assembly Language Program to find the sum of 5 numbers using loop.

Source Code :

```
start: nop
MVI B, 05H
LXI H,
0001H
LOOP: MOV
C, M
ADD C
INX H
DCR B
JNZ LOOP
STA 0006H
hlt
```

Output :



The screenshot displays the GNUSim8085 - 8085 Microprocessor Simulator interface. The main window shows the assembly program being loaded and executed. The program is as follows:

```
1
2 ;<Program title>
3
4 jmp start
5
6 ;data
7
8
9 ;code
10 start: nop
11 MVI B, 05H
12 LXI H, 0001H
13 LOOP: MOV C, M
14 ADD C
15 INX H
16 DCR B
17 JNZ LOOP
18 STA 0006H
19
20
21 hlt
```

The left panel shows the registers and flags. The right panel shows the memory dump. The bottom panel shows the assembler messages.

Address (Hex)	Address	Data
0000	0	0
0001	1	5
0002	2	6
0003	3	8
0004	4	4
0005	5	2
0006	6	25
0007	7	0
0008	8	0
0009	9	0
000A	10	0
000B	11	0
000C	12	0
000D	13	0

The bottom panel shows the assembler messages:

```
Line No  Assembler Message
0        Program assembled successfully
```

