Practical – 1

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

a) Decimal to Binary Conversion

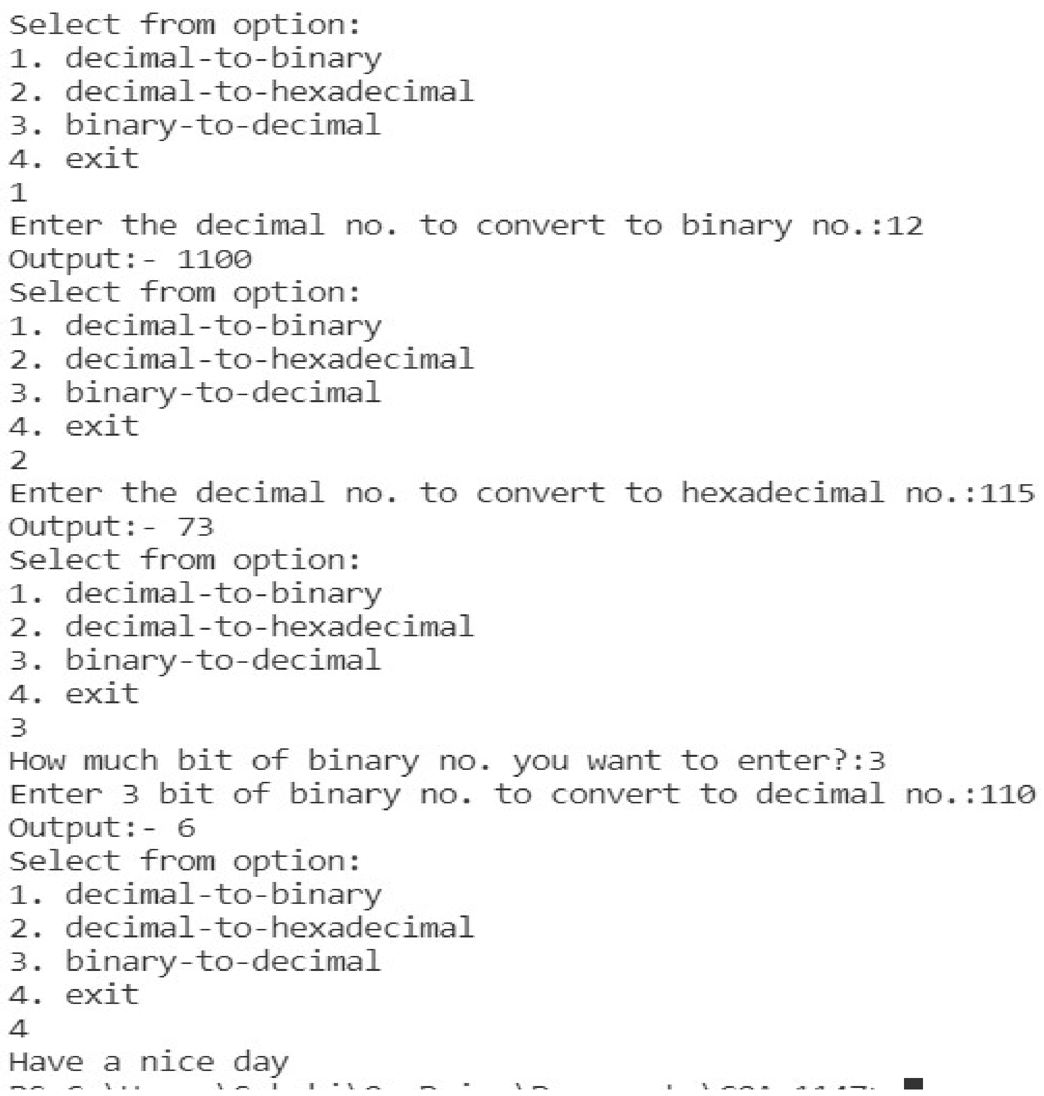
b) Decimal to Hexadecimal Conversion

c) Binary to Decimal Conversion

```
#include<iostream>
using namespace std;
#define MAX 50
class Convertor
{
   int a[MAX],temp=0,i=0,n,m,j,t,lb,sum=0,s=1,z;
   char b[MAX];
   public:
   int dectobi()
   {
     cout<<"Enter the decimal no. to convert to binary no.:";
     cin>>n;
     for(i=0;n>0;i++)
     {
        a[i]=n%2;
        n=n/2;
     }
     cout<<"Output:- ";
     for(j=i-1;j>=0;j--)
     {
        cout<<a[j];
     }
     return 0;
   }
   int dectohex()
   {
     cout<<"Enter the decimal no. to convert to hexadecimal no.:";
     cin>>m;
        while (m != 0)
        {
      temp = m % 16;
      if (temp < 10) {
            b[i] = temp + 48;
            i++;
        }
        else {
            b[i] = temp + 55;
            i++;
        }
        m = m / 16;
         }
         cout<<"Output:- ";
    for(j=i-1;j>=0;j--)
    {
        cout<<b[j];
    }
      return 0;
    }
    int bitodec()
    {
        cout<<"How much bit of binary no. you want to enter?:";
        cin>>z;
        cout<<"Enter "<<z<<" bit of binary no. to convert to decimal no.:";
        cin>>t;
        while(t>0)
        {
            lb=t%10;
            sum=sum+(lb*s);
            s=s*2;
            t=t/10;
        }
        cout<<"Output:- "<<sum;
        return 0;
    }
};
int main()
{
    Convertor c;
    int op;
    while(op!=4)
    {
```

```
    cout<<endl<<"Select from option:"<<endl;
    cout<<"1. decimal-to-binary"<<endl;
    cout<<"2. decimal-to-hexadecimal"<<endl;
    cout<<"3. binary-to-decimal"<<endl;
    cout<<"4. exit"<<endl;
    cin>>op;
    switch(op)
    {
        case 1:
        c.dectobi();
        break;
        case 2:
        c.dectohex();
        break;
        case 3:
        c.bitodec();
        break;
        case 4:
        cout<<"Have a nice day";
        break;
        default:
        cout<<"select proper option";
        break;
    }
    }
    return 0;
}
```

Output



```
Select from option:
1. decimal-to-binary
2. decimal-to-hexadecimal
3. binary-to-decimal
4. exit
1
Enter the decimal no. to convert to binary no.:12
Output:- 1100
Select from option:
1. decimal-to-binary
2. decimal-to-hexadecimal
3. binary-to-decimal
4. exit
2
Enter the decimal no. to convert to hexadecimal no.:115
Output:- 73
Select from option:
1. decimal-to-binary
2. decimal-to-hexadecimal
3. binary-to-decimal
4. exit
3
How much bit of binary no. you want to enter?:3
Enter 3 bit of binary no. to convert to decimal no.:110
Output:- 6
Select from option:
1. decimal-to-binary
2. decimal-to-hexadecimal
3. binary-to-decimal
4. exit
4
Have a nice day
```

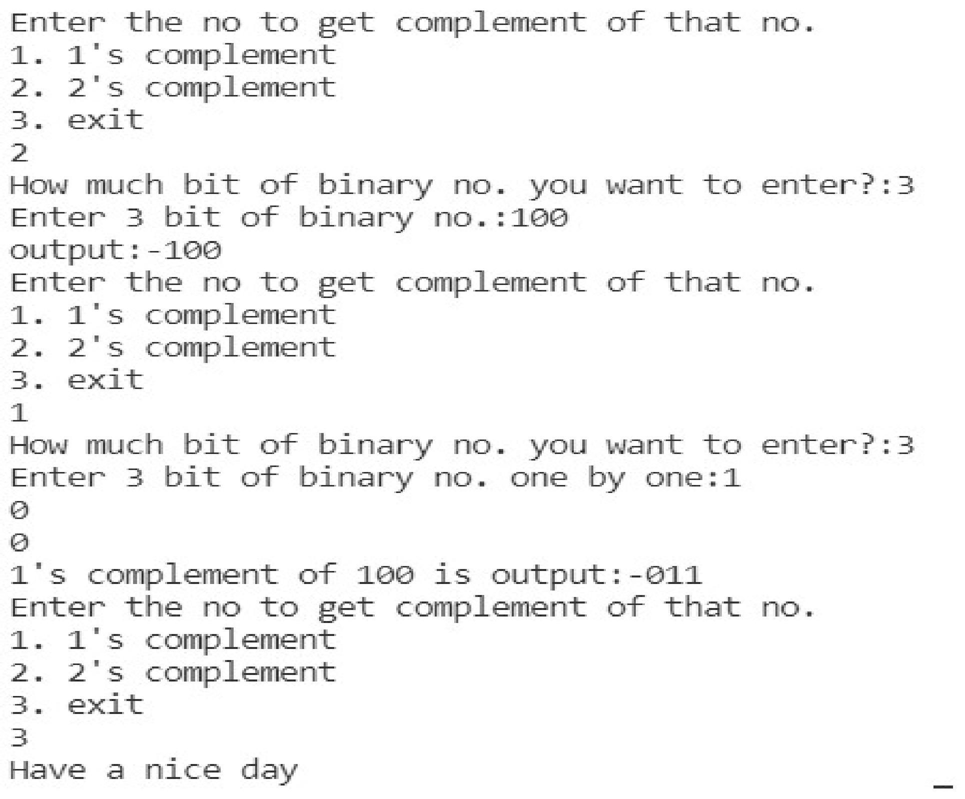
Practical – 2

Aim: Write a C program to perform the following compliment Operations.
a)1's Complement
b)2's Complement

```
#include<iostream>
using namespace std;
#define MAX 50
class Complement
{
   public:
   int a[MAX],b[MAX],i,j,n,m,t,lb,sum=0,s=1,temp=0,z=2;
   int onecomp()
   {
      cout<<"How much bit of binary no. you want to enter?:";
      cin>>n;
      cout<<"Enter "<<n<<" bit of binary no. one by one:";
      for(i=0;i<n;i++)
      {
         cin>>a[i];
      }
      cout<<"1's complement of ";
      for(i=0;i<n;i++)
      {
         cout<<a[i];
      }
      cout<<" is ";
      for(i=0;i<n;i++)
      {
         if(a[i]==0)
         a[i]=1;
         else
         a[i]=0;
      }
      cout<<"output:-";
      for(i=0;i<n;i++)
      cout<<a[i];
      return 0;
   }
   int twocomp()
```

```
    {
      cout<<"How much bit of binary no. you want to enter?:";
      cin>>m;
      cout<<"Enter "<<m<<" bit of binary no.:";
      cin>>t;
      while(t>0)
      {
         lb=t%10;
         sum=sum+(lb*s);
         s=s*2;
         t=t/10;
      }
      while(m!=0)
      {
         temp=z;
         z=z*2;
         m--;
      }
      temp=temp-sum;
      for(i=0;temp>0;i++)
     {
        b[i]=temp%2;
        temp=temp/2;
     }
     cout<<"output:-";
     for(j=i-1;j>=0;j--)
     {
        cout<<b[j];
     }
      return 0;
    }
};
int main()
{
    Complement c;
    int op;
    while(op!=3)
    {
    cout<<endl<<"Enter the no to get complement of that no.";
    cout<<endl<<"1. 1's complement"<<endl;
    cout<<"2. 2's complement"<<endl;
    cout<<"3. exit"<<endl;
    cin>>op;
    switch(op)
    {
        case 1:
        c.onecomp();
        break;
        case 2:
        c.twocomp();
        break;
        case 3:
        cout<<"Have a nice day";
        break;
        default:
        cout<<"select proper option";
        break;
    }
    }
    return 0;
}
```

Output



```
Enter the no to get complement of that no.
1. 1's complement
2. 2's complement
3. exit
2
How much bit of binary no. you want to enter?:3
Enter 3 bit of binary no.:100
output:-100
Enter the no to get complement of that no.
1. 1's complement
2. 2's complement
3. exit
1
How much bit of binary no. you want to enter?:3
Enter 3 bit of binary no. one by one:1
0
0
1's complement of 100 is output:-011
Enter the no to get complement of that no.
1. 1's complement
2. 2's complement
3. exit
3
Have a nice day
```

Practical – 3

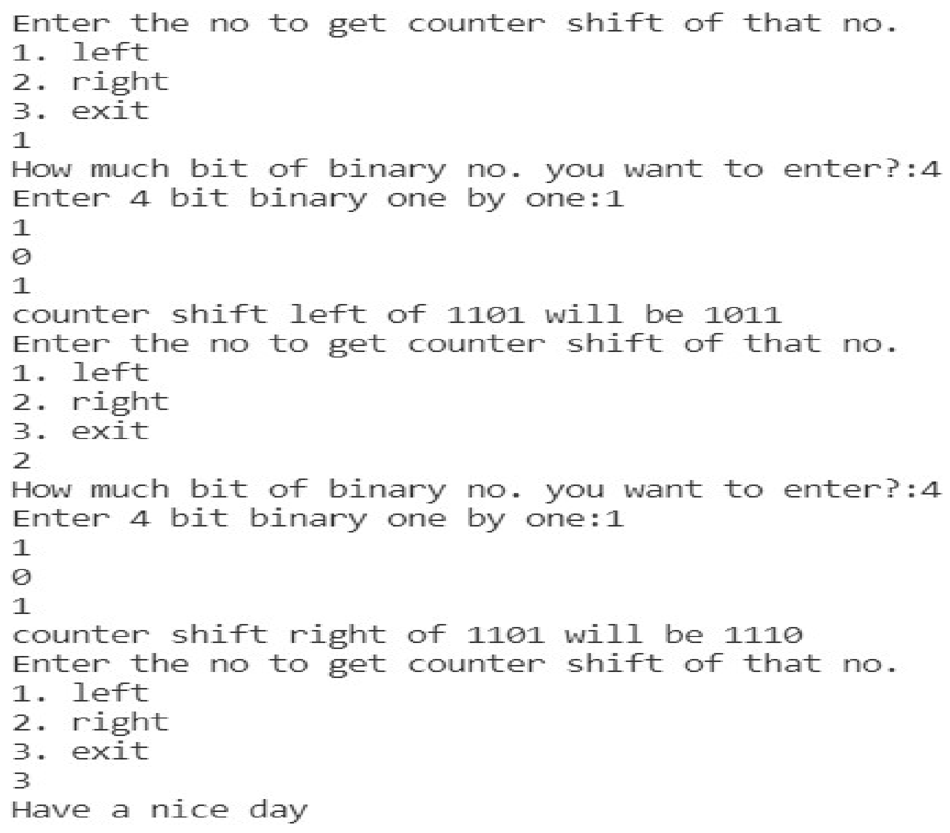
Aim: Write a C program to perform the following Microoperations.
a) Circular Shift left
b) Circular Shift Right

```
#include<iostream>
using namespace std;
#define MAX 50
class Countershift
{
   public:
   int a[MAX],b[MAX],i,j,n,m,temp,t;
   int left()
   {
      cout<<"How much bit of binary no. you want to enter?:";
      cin>>n;
      cout<<"Enter "<<n<<" bit binary one by one:";
      for(i=0;i<n;i++)
      {
         cin>>a[i];
      }
      cout<<"counter shift left of ";
      for(i=0;i<n;i++)
      {
         cout<<a[i];
      }
      cout<<" will be ";
      temp=a[0];
      for(i=0;i<n-1;i++)
      {
         a[i]=a[i+1];
      }
      a[n-1]=temp;
      for(i=0;i<n;i++)
      {
         cout<<a[i];
      }
      return 0;
   }
   int right()
```

```
    {
     cout<<"How much bit of binary no. you want to enter?:";
       cin>>m;
       cout<<"Enter "<<m<<" bit binary one by one:";
       for(j=0;j<m;j++)
       {
          cin>>b[j];
       }
       cout<<"counter shift right of ";
       for(j=0;j<m;j++)
       {
          cout<<b[j];
       }
       cout<<" will be ";
       t=b[m-1];
       for(j=m-1;j>=0;j--)
       {
          b[j]=b[j-1];
       }
       b[0]=t;
       for(j=0;j<m;j++)
       {
          cout<<b[j];
       }
       return 0;
    }
};
int main()
{
   Countershift c;
   int op;
   while(op!=3)
   {
   cout<<endl<<"Enter the no to get counter shift of that no.";
   cout<<endl<<"1. left"<<endl;
   cout<<"2. right"<<endl;
   cout<<"3. exit"<<endl;
   cin>>op;
   switch(op)
```

```
  {
    case 1:
    c.left();
    break;
    case 2:
    c.right();
    break;
    case 3:
    cout<<"Have a nice day";
    break;
    default:
    cout<<"Select proper option";
    break;
  }
  }
  return 0;
}
```

Output



```
Enter the no to get counter shift of that no.
1. left
2. right
3. exit
1
How much bit of binary no. you want to enter?:4
Enter 4 bit binary one by one:1
1
0
1
counter shift left of 1101 will be 1011
Enter the no to get counter shift of that no.
1. left
2. right
3. exit
2
How much bit of binary no. you want to enter?:4
Enter 4 bit binary one by one:1
1
0
1
counter shift right of 1101 will be 1110
Enter the no to get counter shift of that no.
1. left
2. right
3. exit
3
Have a nice day
```

Practical – 4

Aim: Introduction to GNU Simulator 8085

GNU Stimulator 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)**
    - **Pseudo operations (like preprocessor in C)**
3. Operands
4. Comments

Registers of 8085 microprocessor

A 8085 microprocessor, is a second generation 8-bit microprocessor and is the base for studying and using all the microprocessor available in the market.

Register 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).

- **Flag registers:**
- 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.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.

B_7	B_6	B_5	B_4	B_3	B_2	B_1	B_0
S	Z	—	AC	—	P	—	CY

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

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. **Auxillary 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 Auxillary Carry flag is set. If not, flag is reset. This flag is used internally for BCD(Binary-Coded decimal Number) operations.

4. **Parity Flag:** 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

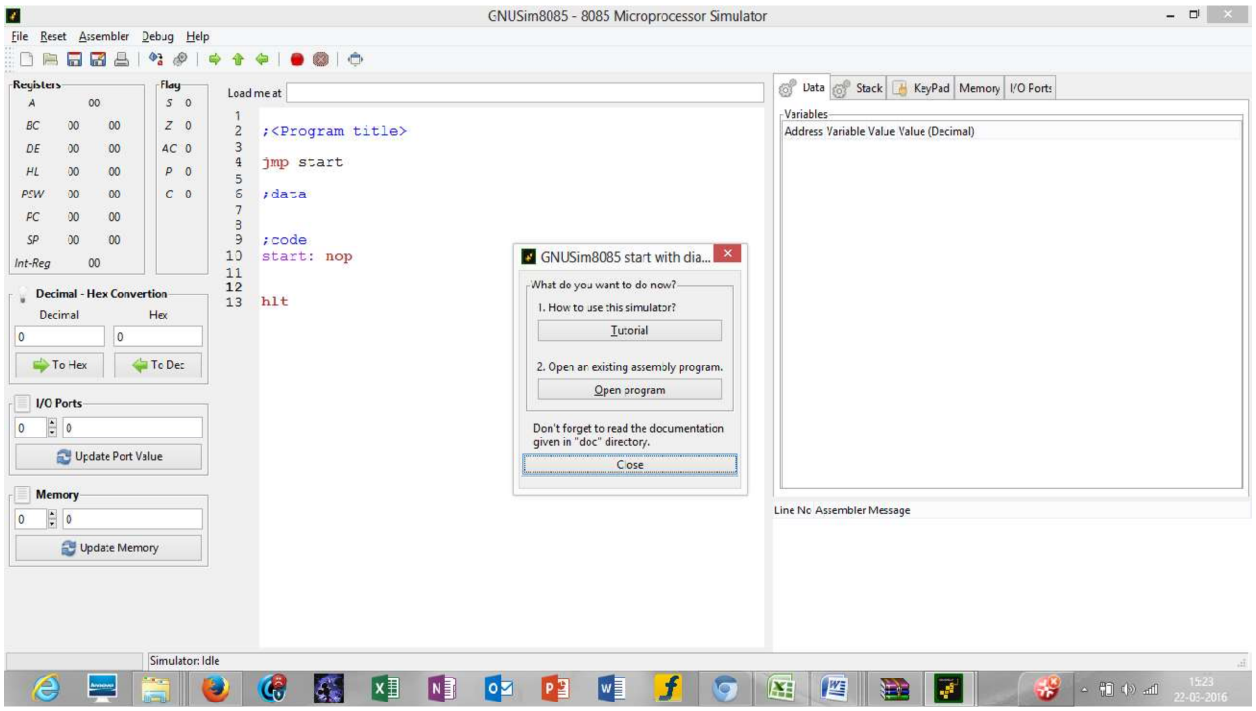
There are two 16-bit registers used to hold memory addresses. The size of these registers is 16 bits because the memory addresses are 16 bits. They are :-

- **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.
- **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 have 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 signal: 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 signal 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**.

Getting Started with GNU Simulator 8085:


GNUSim8085 - 8085 Microprocessor Simulator
File Reset Assembler Debug Help
Registers
Flag
Load me at
Data Stack KeyPad Memory I/O Ports
Variables
Address Variable Value Value (Decimal)
Decimal - Hex Convertion
Decimal Hex
To Hex To Dec
I/O Ports
Update Port Value
Memory
Update Memory
Line No Assembler Message
Simulator: Idle

```

;<Program title>

jmp start

;data


;code
start: nop


hlt
```

GNUSim8085 start with dia...

What do you want to do now?

1. How to use this simulator?

Tutorial

2. Open an existing assembly program.

Open program

Don't forget to read the documentation given in "doc" directory.

Close

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/0 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/0 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.

→ **SO, S1 (Output)**
Data Bus Status. Encoded status of the bus cycle:

S1	S0	
0	0	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 1/0 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 1/0 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 port.
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 non-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 CPU 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 l/O Tristated 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.

Instruction and Data Formats

The various techniques to specify data for instructions are:

1. 8-bit or 16-bit data may be directly given in the instruction itself.
2. The address of the memory location, I/O port or I/O device, where data resides, may be given in the instruction itself.
3. In some instructions, only one register is specified. The content of the specified register is one of the operands.
4. Some instructions specify two registers. The contents of the registers are the required data.
5. In some instructions, data is implied. The most instructions of this type operate on the content of the accumulator.

Due to different ways of specifying data for instructions, the machine codes of all instructions are not of the same length. It may 1-byte, 2-byte or 3-byte instruction.

Addressing Modes

Each instruction requires some data on which it has to operate. There are different techniques to specify data for instructions. These techniques are called **addressing modes**. Intel 8085 uses the following addressing modes:

(1) Direct Addressing: In this addressing mode, the address of the operand (data) is given in the instruction itself.

Example: **STA 2400H:** It stores the content of the accumulator in the memory location 2400H.**32, 00, 24:** The above instruction in the code form.

(2) Register Addressing: In register addressing mode, the operand is in one of the general purpose registers. The opcode specifies the address of the register(s) in addition to the operation to be performed.

Example:MOV A, B: Move the content of B register to register A.

78: The instruction in the code form.

In the above example, MOV A, B is 78H. Besides the operation to be performed the opcode also specifies source and destination registers.

(3) Register Indirect Addressing: In Register Indirect mode of addressing, the address of the operand is specified by a register pair.

Example:

- **LXI H, 2500 H** - Load H-L pair with 2500H.
- **MOV A, M** - Move the content of the memory location, whose address is in H-L pair (i.e. 2500 H) to the accumulator.
- **HLT** - Halt.

(4) Immediate Addressing: In this addressing mode, the operand is specified within the instruction itself.

Example: LXI H, 2500 is an example of immediate addressing. 2500 is 16-bit data which is given in the instruction itself. It is to be loaded into H-L pair.

(5) **Implicit Addressing:** There are certain instructions which operate on the content of the accumulator. Such instructions do not require the address of the operand.

Example: CMA, RAL, RAR, etc.

Status Flags

There is a set of five flip-flops which indicate status (condition) arising after the execution of arithmetic and logic instructions. These are:

- Carry Flag (CS)
- Parity Flag (P)
- Auxiliary Carry Flags (AC)
- Zero Flags (Z)
- Sign Flags (S)

Symbols and Abbreviations

Symbol/Abbreviations	Meaning
Addr	16-bit address of the memory location.
Data	8-bit data
data 16	16-bit data
r, r1, r2	One of the registers A, B, C, D, E, H or L
A, B, C, D, H, L	8-bit register
A	Accumulator
H-L	Register pair H-L
B-C	Register pair B-C
D-E	Register pair D-E
PSW	Program Status Word
M	Memory whose address is in H-L pair

H	Appearing at the end of the group of digits specifies hexadecimal, e.g. 2500H
Rp	One of the register pairs.
Rh	The high order register of a register pair
Rl	The low order register of a register pair
PC	16 bit program counter, PCH is high order 8 bits and PCL low order 8 bits of register PC.
CS	Carry Status
[]	The contents of the register identified within bracket
[[]]	The content of the memory location whose address is in the register pair identified within brackets
^	AND operation
∨	OR operation
⊕ or ∀	Exclusive OR
←	Move data in the direction of arrow
⇔	Exchange contents

Practical – 5

AIM: Write an assembly language program to perform the 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 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, 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 HLregister 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, 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

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-bitsor 1-Bytevalue.
→ 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

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 instructionAccumulatorwill 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 fromMemory 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

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

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,Accumulator8-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 fromA into the memory specified by 16 bit address

STA 2060H

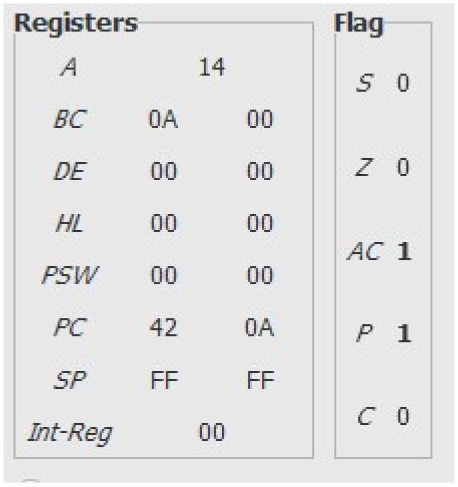
Example:

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

PROGRAMS:-

(1)To do addition:-

```
;<Program title>

jmp start

;data


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


hlt
```



Registers

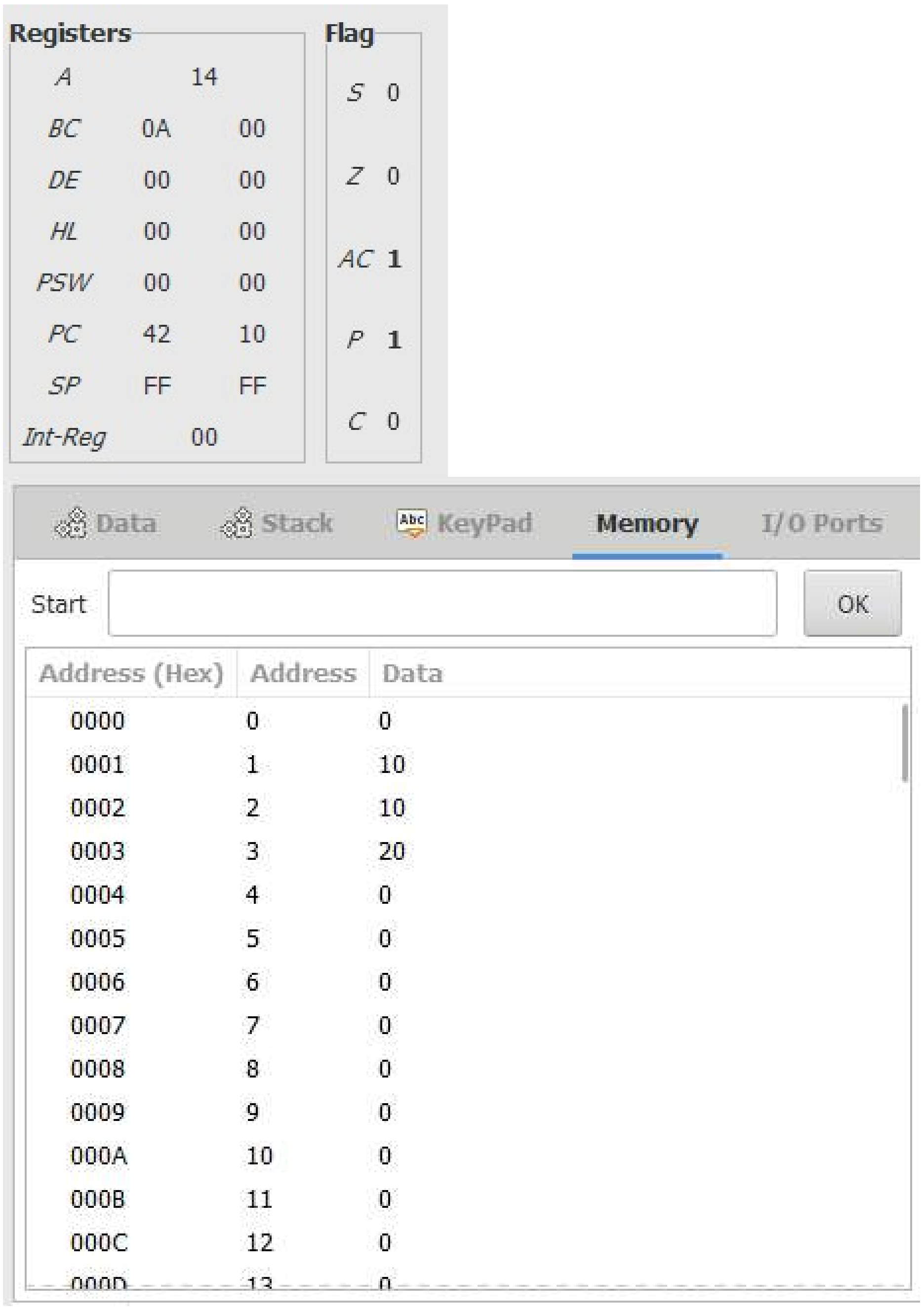
A	14	
BC	0A	00
DE	00	00
HL	00	00
PSW	00	00
PC	42	0A
SP	FF	FF
Int-Reg	00	

Flag

S	0
Z	0
AC	1
P	1
C	0

(2)To do addition:-

```
;<Program title>

jmp start

;data


;code
start: nop

LDA 0001H
MOV B,A
LDA 0002H
ADD A
STA 0003H

hlt
```



Registers

A	14	
BC	0A	00
DE	00	00
HL	00	00
PSW	00	00
PC	42	10
SP	FF	FF
Int-Reg	00	

Flag

S	0
Z	0
AC	1
P	1
C	0

Address (Hex)	Address	Data
0000	0	0
0001	1	10
0002	2	10
0003	3	20
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
000D	13	0

Add two 8-bit numbers

`Statement:` Add the contents of memory locations 4000H and 4001H and place the result in memory location 4002H

```
Sample problem
(4000H)=14H
(4001H)=89H
Result=14H+89H=9DH

Source program
LXI H 4000H:"HL points 4000H"
MOV A, M  :"Get first operand"
INX H     :"HL points 4001H"
ADD M     :"Add second operand"
INX H     :"HL points 4002H"
MOV M, A  :"Store result at 4002H"
HLT       :"Terminate program execution"
```