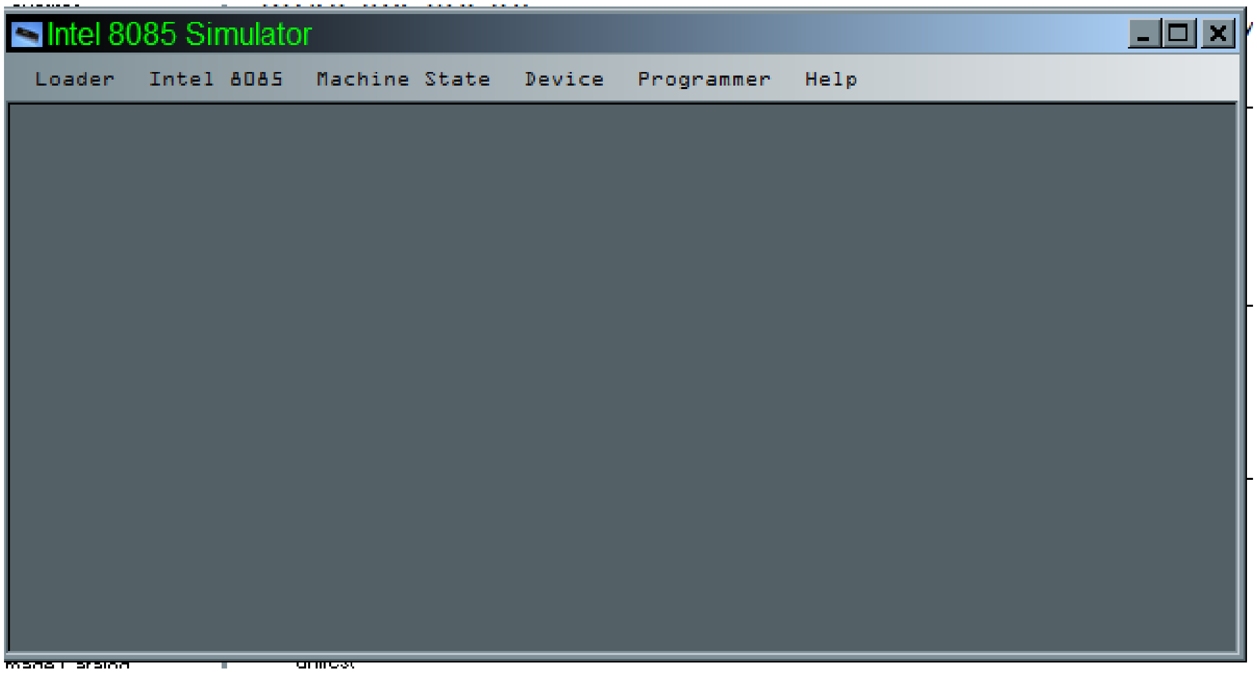
INTEL 8085 SIMULATOR

MANUAL

♦ Running the Program

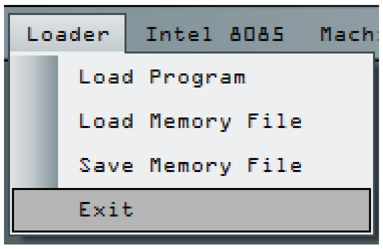
In the simulator directory, locate the "Intel8085Sim.exe", under "/bin" folder. Double click on it to run it. The main IDE opens, where you can start your development. The main IDE screen looks like the image below:



The IDE design is described below:

- Title bar: The title bar is similar to other Windows GUI application. It has the System Menu at the right side, which contains Minimize, Maximize and Close button. In the left side, it shows the name of the application "Intel 8085 Simulator". When a program had been executed, it will execute the machine state within square bracket. It can be [STOPPED] or [RUNNING]. By default, the status is Stopped. While the execution continues, the status indicates [RUNNING].
- Menu Bar: The menu bar will show the items available during program execution and development. It has the following items by default: Loader, Intel 8085, Machine State, Device, Programmer, Help. All these items will be separately discussed in this document.
- MDI Screen: The rest of the screen is an MDI interface. It will hold all the child windows, that will open up during the development and execution point.

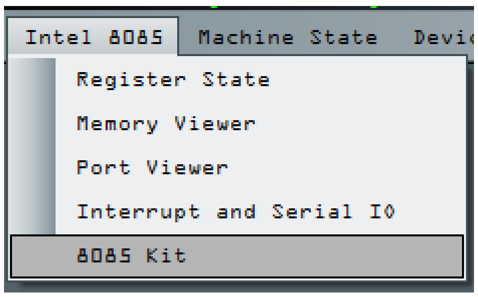
♦ Loader Menu



The loader menu had 4 items-

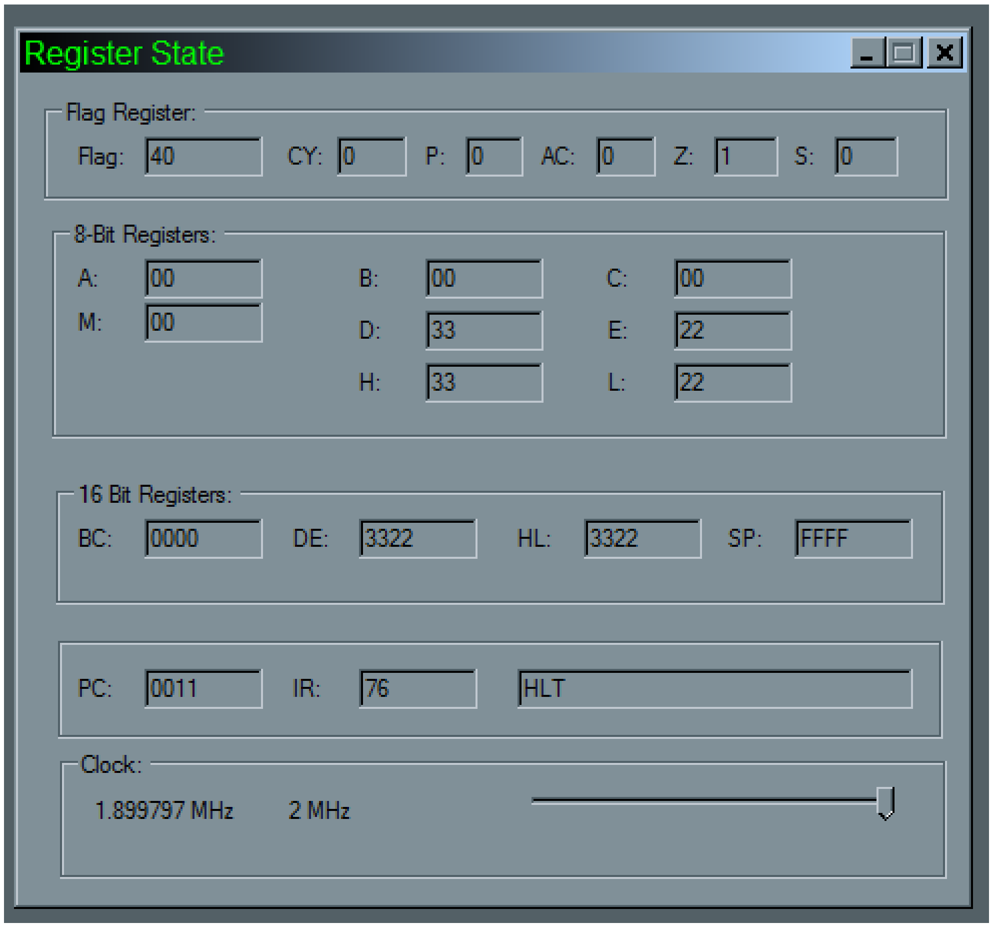
- Load Program - This item is used to load the compiled binary in 2 formats-Intel Hex Format and Memory Dump File. The Intel Hex File is a popular binary format, and is generally used by most of the available simulators and SDK kits. The memory dump file is, however, a binary file that contains 64 KB of 1 byte data. The memory dump file contains a snapshot of the 8085 memory, so it will contain the data as well as instructions along with the register contents.
- Load Memory File - Through this menu item, you can load a Memory dump file.
- Save Memory File - This menu item will create a snapshot of the 8085 memory and dump it into a file on disk.
- Exit - This option will stop the present executing program, save the dirty updates to disk. And will finally close the program.

♦ **Intel 8085 Menu**



The Intel 8085 menu item has 5 sub-items:

- Register State - It will show a new dialog. The dialog will show the current state of 8085 registers A, B, C, D, E, H, L, M[HL], pair registers BC, DE, HL, special registers SP, IR, PC, and flag register. It will also show the present clock speed. This dialog also shows a Horizontal Slider, which can be used to adjust the clock frequency. If the clock speed is reduced the program will execute slowly, and is sometimes essential for debugging. A screenshot of the Register dialog is shown below:


Register State

Flag Register:

Flag: 40 CY: 0 P: 0 AC: 0 Z: 1 S: 0

8-Bit Registers:

A:	00	B:	00	C:	00
M:	00	D:	33	E:	22
		H:	33	L:	22

16 Bit Registers:

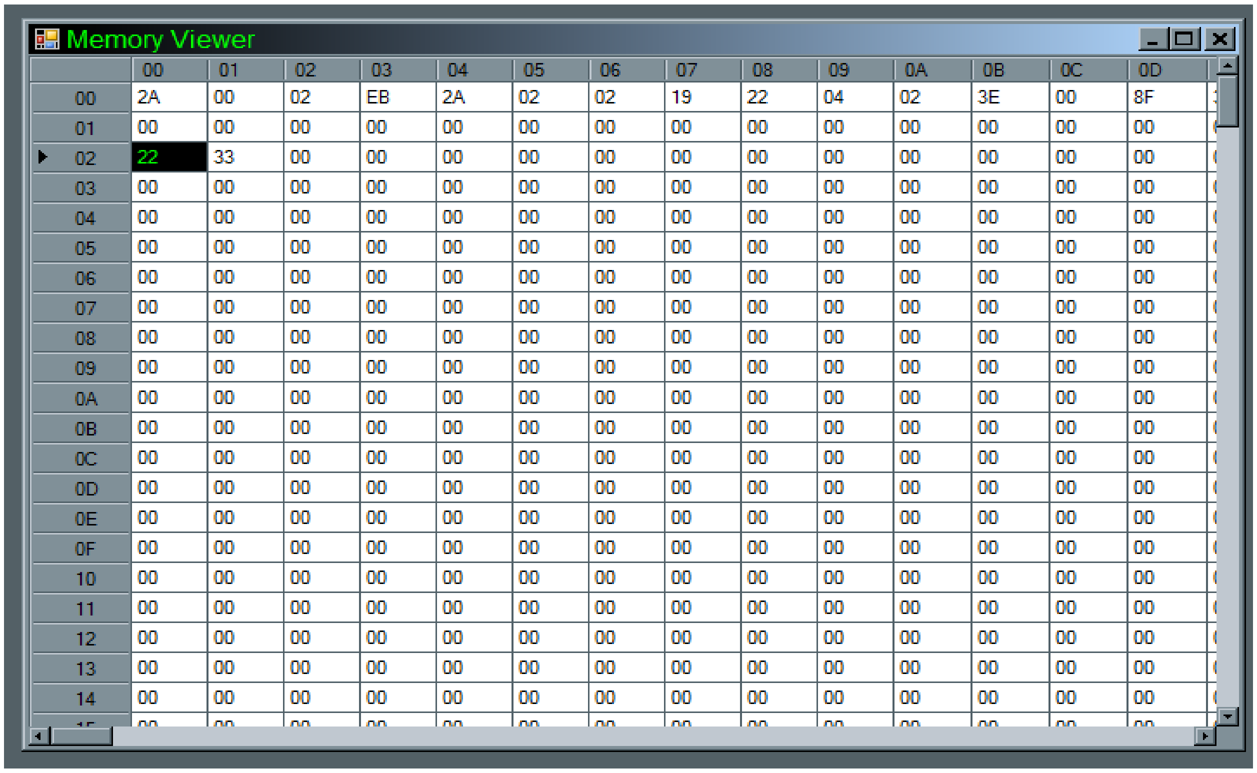
BC: 0000 DE: 3322 HL: 3322 SP: FFFF

PC: 0011 IR: 76 HLT

Clock:

1.899797 MHz 2 MHz

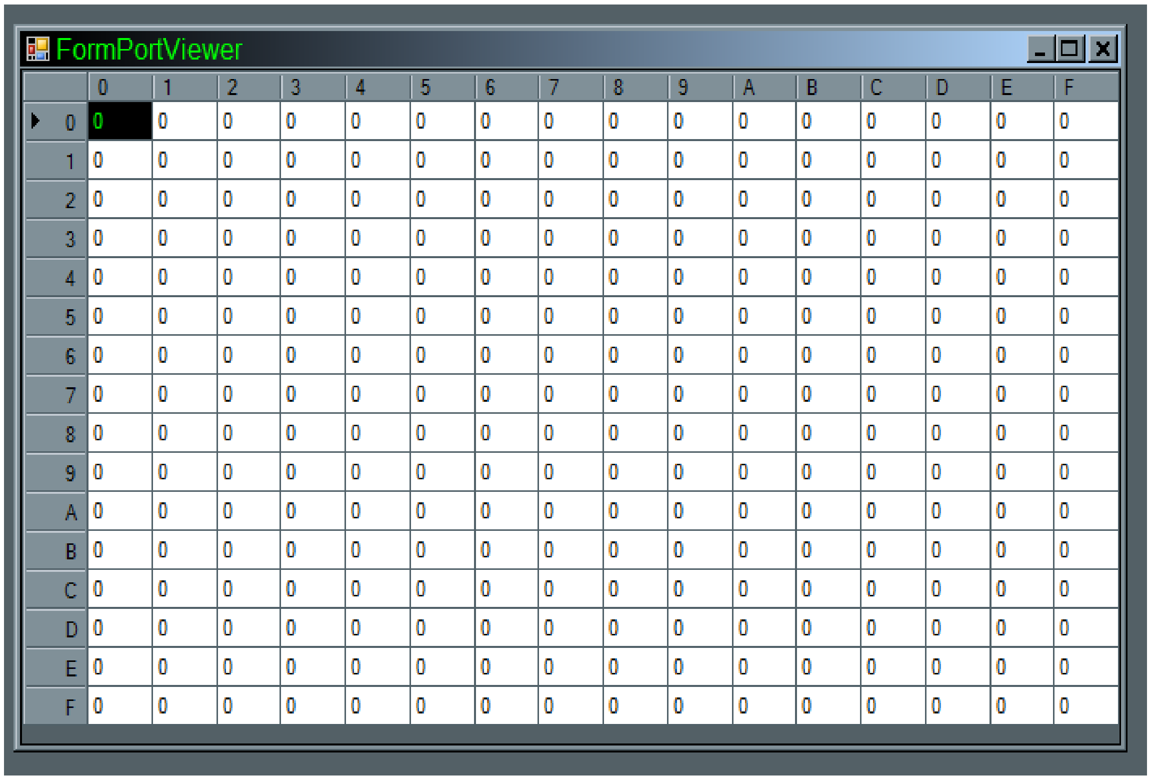
- Memory Viewer - The memory viewer will show a new dialog. this dialog has a Worksheet view. It will show the content of the 64 KB of memory that is available for the 8085 processor. The memory content view shown there is editable, but the value specified must be in hexadecimal format and must positive. Also the data range of each cell must be in range of 00 to FF. The memory locations of 8085 are in range from 0000 to FFFF hexadecimal. The column values 00 to FF are the lower byte of the memory address. But the row values in range of 00 to FF specify higher byte of memory. If we select the cell whose row value is AB and column value is C1, then the result cell address in memory is ABC1. To edit the value, just double click on it and change it. Then you must click on any other cell, for the value to get updated in the memory. Otherwise, the new value will not be reflected. All the cells of 8085 are editable, that is, they have read and write permissions. A screen show of the Memory Viewer is shown below:



	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D
00	2A	00	02	EB	2A	02	02	19	22	04	02	3E	00	8F
01	00	00	00	00	00	00	00	00	00	00	00	00	00	00
▶ 02	22	33	00	00	00	00	00	00	00	00	00	00	00	00
03	00	00	00	00	00	00	00	00	00	00	00	00	00	00
04	00	00	00	00	00	00	00	00	00	00	00	00	00	00
05	00	00	00	00	00	00	00	00	00	00	00	00	00	00
06	00	00	00	00	00	00	00	00	00	00	00	00	00	00
07	00	00	00	00	00	00	00	00	00	00	00	00	00	00
08	00	00	00	00	00	00	00	00	00	00	00	00	00	00
09	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0A	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0B	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0C	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0D	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0E	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0F	00	00	00	00	00	00	00	00	00	00	00	00	00	00
10	00	00	00	00	00	00	00	00	00	00	00	00	00	00
11	00	00	00	00	00	00	00	00	00	00	00	00	00	00
12	00	00	00	00	00	00	00	00	00	00	00	00	00	00
13	00	00	00	00	00	00	00	00	00	00	00	00	00	00
14	00	00	00	00	00	00	00	00	00	00	00	00	00	00

Note: The Memory viewer can take a little time to show, during which time the application may be in a non-responsive. Please bear with, the dialog will however show up eventually.

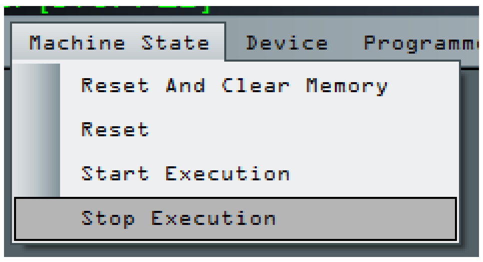
- Port Viewer - Intel 8085 supports external communication through ports. There are 256 ports supported by the 8085 processor. These are addressed in the range of 00 to FF, occupying 1 byte address. The Port viewer is a dialog that displays the data currently being transmitted out or to the port. It has a Worksheet type display similar to the Memory Viewer. But, in here, the column specify the lower nibble of the IO port and the rows represents the higher nibble of the port. The cells are editable, so change the port input data, double click on the required cell and replace the value. Then click on any other cell, so that the port receive the data. The snapshot of the Port Viewer is shown below:


FormPortViewer

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
A	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
B	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
C	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
D	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
E	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
F	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

- Interrupt and Serial IO- It opens up a new dialog that will show the Interrupt level details of 8085. It will be elaborately discussed later.
- 8085 Kit - It will show up a kit that is frontend similar to the original 8085 SDK-85 kit. It can be used to program, execute and debug a 8085 program, independently.

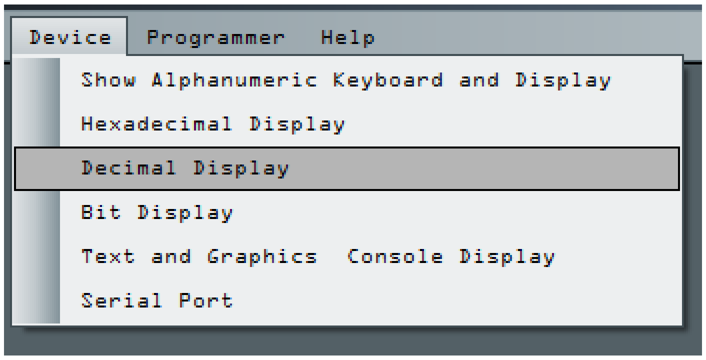
♦ Machine State



The machine state menu is used to execute a program ,reset the machine, clear the memory. The options available in sub-menu are given below:

- Reset and Clear Memory - This menu item, if clicked, will stop the executing program. Then it will reset program counter PC to memory location 0000. Then it will clear the entire 64 KB of memory and set the value to 00.
- Reset - This menu item, if clicked, will stop the running program and reset the machine. It will only change the PC value to 0000, but the contents of memory locations, flag register, and processor registers are preserved.
- Start Execution - This menu item, if clicked, will start the execution of the program. The execution will actually start from the memory location pointed currently by the program counter PC. The machine must be Reset every time you like to start a new execution block.
- Stop Execution - Stops the currently executing program. It will not reset the memory or affect the machine state by itself. After the program is stopped, the value of PC currently result after the instruction execute is preserved.

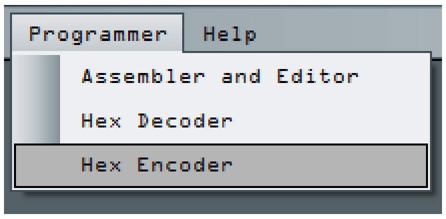
♦ **Device Menu**



The Device menu contains a list of available virtual devices that can be interfaced through their respective IO ports. The menu options are shown below:

- Show Alphanumeric keyboard and Display - It is a virtual device connected to the IO port 00H. It is a bidirectional device. Any byte sent out through port 00h will be displayed as output through this virtual device as ASCII characters. And there is also a input text box, where the text string typed will be streamed in though the input port 00h as byte stream from left to right. Each characters fed are fed through as an equivalent ASCII code.
- Hexadecimal display: This virtual device is connected to the port 01H and is a bidirectional device. The data byte sent out from the out port 01h will be displayed in the hexadecimal format at the output. But any 1 byte hexadecimal value entered at the input will be sent through the in port 01h.
- Decimal Display : This virtual device is connected to the port 02H and is a bidirectional device. The data byte sent out from the out port 02h will be displayed in the decimal format at the output. But any 1 byte decimal value in range of -128 to 127, entered at the input will be sent through the in port 02h as a hexadecimal value.
- Bit Display: This virtual device is connected to the port 03H and is a bidirectional device. There are 8 check boxes marked Bit 0 through Bit 7. When a value is sent out through port 03h, the check boxes corresponding to the bit which is 1 will be checked and also highlighted. Again, in the case when the bit display is used an input mode, the checked bit will be treated as 1 when received from the input port.
- Text and Graphics Console Display - This is a Output device and will be discussed later in details, as it is an advanced display device.
- Serial Port - This device is actually connected to the system COM ports, which can themselves be real or virtual. This device is discussed later in details.

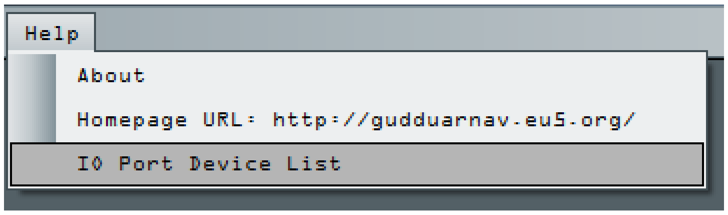
♦ **Programmer**



This menu is used to program in assembly language-

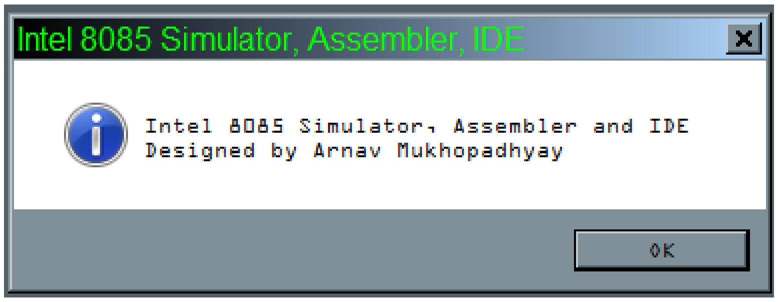
- Assembler and Editor - On clicking this menu item, the internal editor will start. This editor can be used to write the source code in assembly language. The editor when opened will provide its own options, to save, load a source ASM code, and assemble the written code.
- Hex Decoder - The Hex decoder is an inbuilt tool to view the content of the Intel Hex File and show the Hexadecimal machine code.
- Hex Encoder - The Hex encoder is an inbuilt tool to convert the given Hex machine code to Intel Hex format code which can be written down to disk using Notepad or any other text editor.

♦ Help



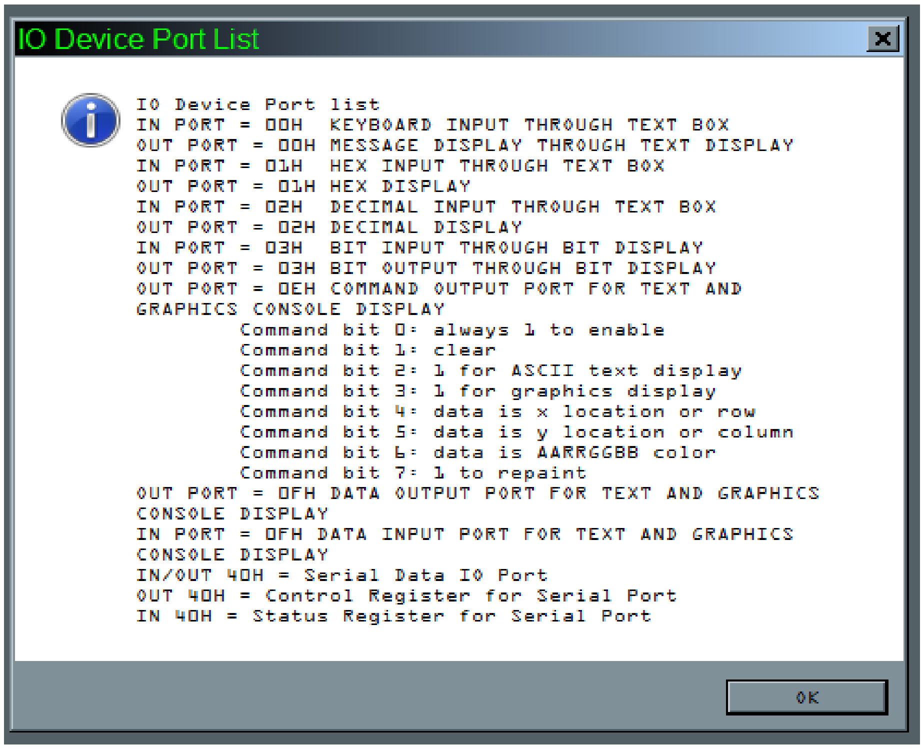
The help menu will provide some information about the software-

- About - The about menu will show the information about the software.



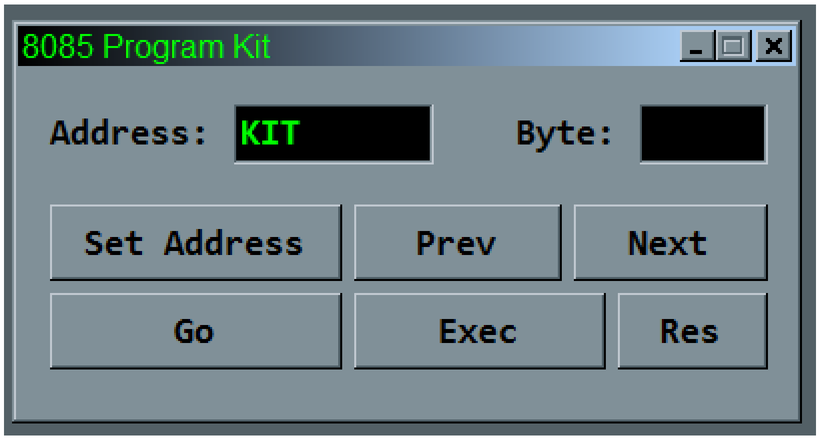
```
Intel 8085 Simulator, Assembler and IDE
Designed by Arnav Mukhopadhyay
```

- Homepage URL - This item when clicked will take you to the homepage of the Software, in the browser.
- IO Port Device List - This item when clicked will show up a dialog, which shows the IO device port details.



```
IO Device Port list
IN PORT = 00H  KEYBOARD INPUT THROUGH TEXT BOX
OUT PORT = 00H MESSAGE DISPLAY THROUGH TEXT DISPLAY
IN PORT = 01H  HEX INPUT THROUGH TEXT BOX
OUT PORT = 01H HEX DISPLAY
IN PORT = 02H  DECIMAL INPUT THROUGH TEXT BOX
OUT PORT = 02H DECIMAL DISPLAY
IN PORT = 03H  BIT INPUT THROUGH BIT DISPLAY
OUT PORT = 03H BIT OUTPUT THROUGH BIT DISPLAY
OUT PORT = 0EH COMMAND OUTPUT PORT FOR TEXT AND
GRAPHICS CONSOLE DISPLAY
        Command bit 0: always 1 to enable
        Command bit 1: clear
        Command bit 2: 1 for ASCII text display
        Command bit 3: 1 for graphics display
        Command bit 4: data is x location or row
        Command bit 5: data is y location or column
        Command bit 6: data is AARRGGBB color
        Command bit 7: 1 to repaint
OUT PORT = 0FH DATA OUTPUT PORT FOR TEXT AND GRAPHICS
CONSOLE DISPLAY
IN PORT = 0FH DATA INPUT PORT FOR TEXT AND GRAPHICS
CONSOLE DISPLAY
IN/OUT 40H = Serial Data IO Port
OUT 40H = Control Register for Serial Port
IN 40H = Status Register for Serial Port
```

Writing Assembly Program using Inbuilt SDK Kit of 8085



The screen shot of the SDK dialog is shown above. In order to understand the operation of the SDK, we will consider a working program, that will add 2 numbers at memory locations C100H and C101H and save the result in C102H, the carry into the next memory location C103H. The starting address of the program being C000H. The 8085 source will be

```
LXI H, C100H
MOV B, M
INX H
MOV C, M
INX H
MOV A, B
ADD C
MOV M ,A
XRA A
ADC A
INX H
MOV M, A
HLT
```

The above source code cannot be directly understood by the machine. So you need to hand assemble the source into binary code, then hand edit the memory locations of the 8085 memory. These hand assembled hexadecimal codes are entered as data. Now we will proceed to hand assembling the above code:

Memory Address	Machine Code	Mnemonics
C000	21	LXI H
C001	00	00
C002	C1	C1
C003	46	MOV B, M

C004	23	INX H
C005	4E	MOV C, M
C006	23	INX H
C007	78	MOV A, B
C008	81	ADD C
C009	77	MOV M, A
C00A	AF	XRA A
C00B	8F	ADC A
C00C	23	INX H
C00D	77	MOV M, A
C00D	76	HLT

So now we have to first enter the machine code to the machine. In order to do that we proceed as follows:

1. Start the 8085 Simulator.
2. From "Intel 8085" menu, select "8085 Kit" to start the kit dialog.
3. Now, click on the "Set Address" button. Then in "Address" text box, enter the starting address of the program, to be C000H.
4. Press "Next" button, will bring up the data in C000H memory address. And display the data of that memory location in the "Byte" text box. There enter the vale 2A.
5. Press "Next" button, will now bring up the C001H memory location. There enter the value 00H.
6. Continue this step continuously until the data in C00D H location is written, then press "Next" button.
7. Now as we are done, press the "Res" button to reset the system.
8. Next step is to enter the data, to do so we proceed as above again, but this time for data.
9. Our data is stored in locations C100H, C101H.
10. Click on "Set Address" button, and enter the address value C100H.
11. Press "Next" button and enter the first value, say 11H.
12. Press "Next" button and enter the second value, sat 56H.
13. Press "Next" again.
14. Now, as we are done. Press the "Res" button to reset the system.
15. Now, we are ready to run the program, so press the "Go" button. Then in the address box, enter the starting address of the program. In our case, we will enter C000H.
16. Now, press the "Exec" button, and the program starts executing.
17. After the program is done, press "Reset" button.
18. Now we have to view the result, so press "Set Address" and enter C100H in "address" text box.
19. Click "Next" and see the first input data in C100H.
20. Then click "Next" again, to see second input in C101H.
21. Then click "Next" again, to see the sum value at C102H.
22. Then click "Next" again, to see the carry in C103H.

The following table will show the result for our set:

Memory Address	**Data**
C100	11
C101	56
C102	67
C103	00

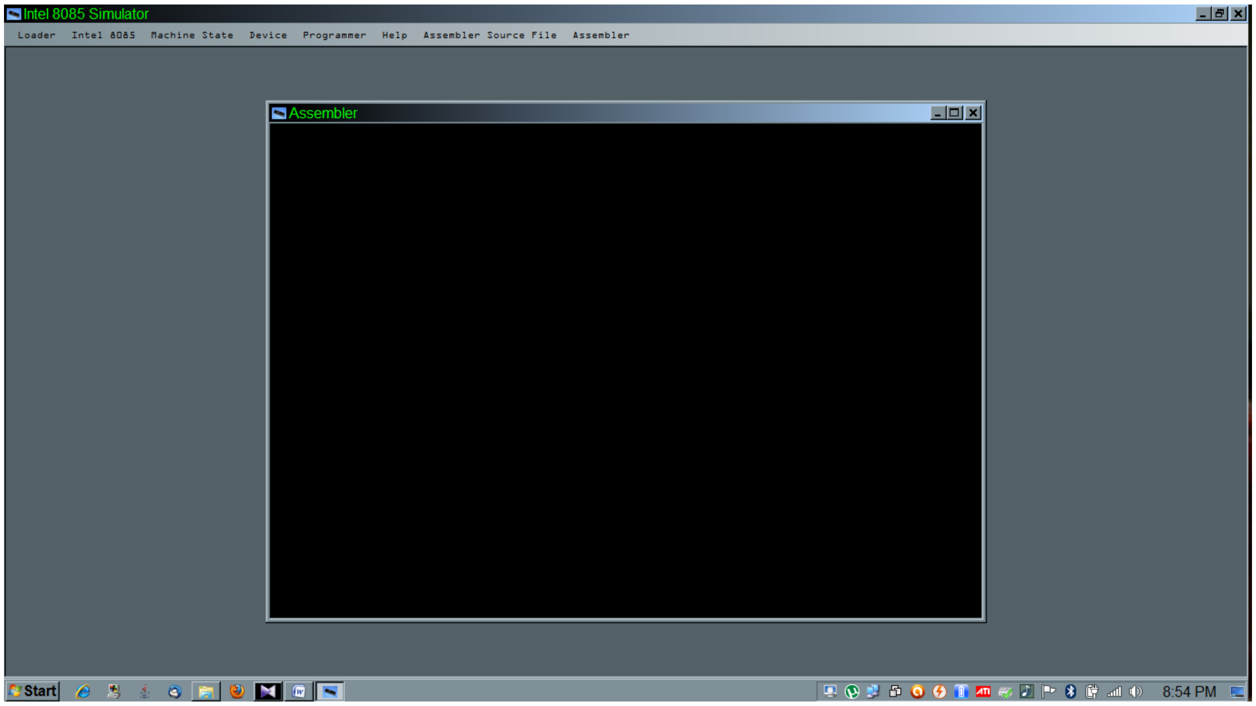
** The above steps are listed according to the guidance specified for submission of ASM 8085 Lab Assignments of various Universities in India and abroad.*

Writing and Executing Assembly language program in the Simulator

There is an internal IDE present within the Simulator that has an attached assembler for converting the assembly language code into their equivalent machine language code, and loading the assembled code into the Simulator executable memory. To bring up the IDE, proceed as follows:

1. From Simulator menu, click on "Programmer" menu.
2. From the sub-menu, click on "Assembler and Editor" option.
3. It will open up the programming IDE.

The screenshot of Assembler IDE opened within the Simulator is shown below:



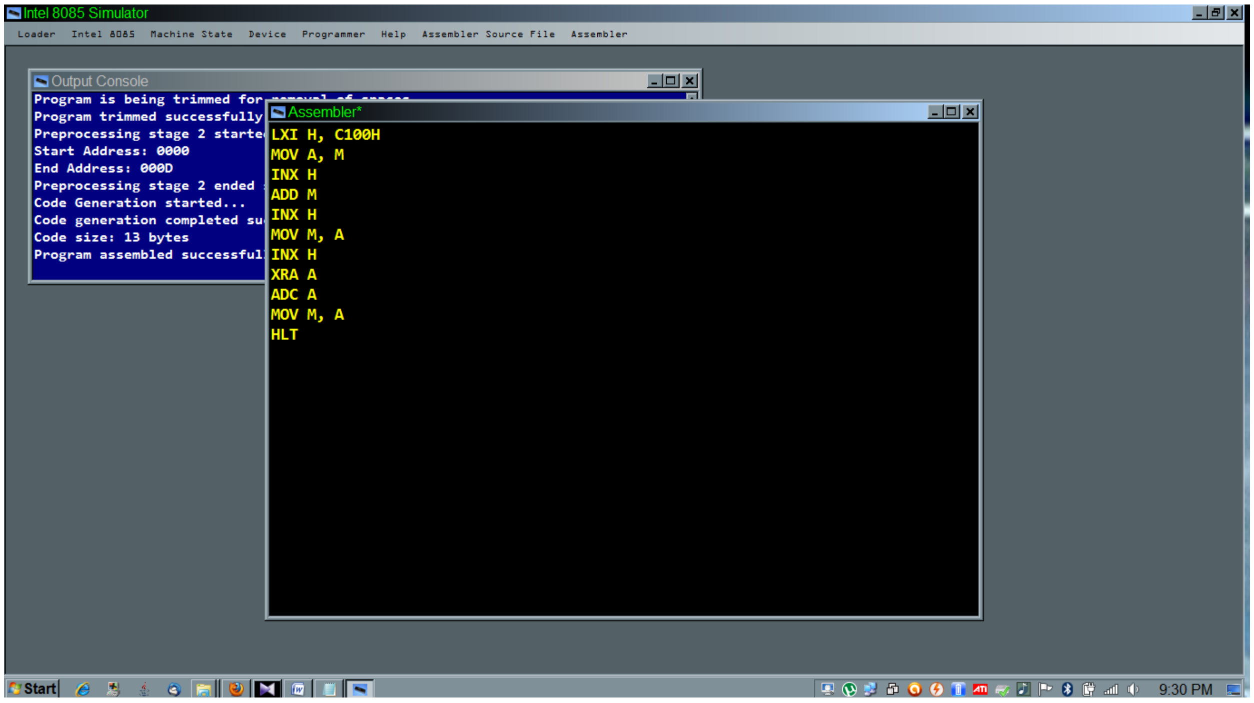
It can be seen from the above figure that, 3 new menu items are added to the main menu, when the IDE is in focus. These menu items are Assembler Source File, Assembler. The "Assembler Source File" is used to clear the editor, save or load the source file, and close the assembler. The "Assembler" file menu can be used to assemble the source file, save or load the binary code.

Consider the previous example, that will add 2 numbers at locations C100 and C101, save the sum in C102 and carry bit in C103. The source code is:

```
LXI H, C100H
MOV A, M
INX H
```

```
ADD M
INX H
MOV M, A
INX H
XRA A
ADC A
MOV M, A
HLT
```

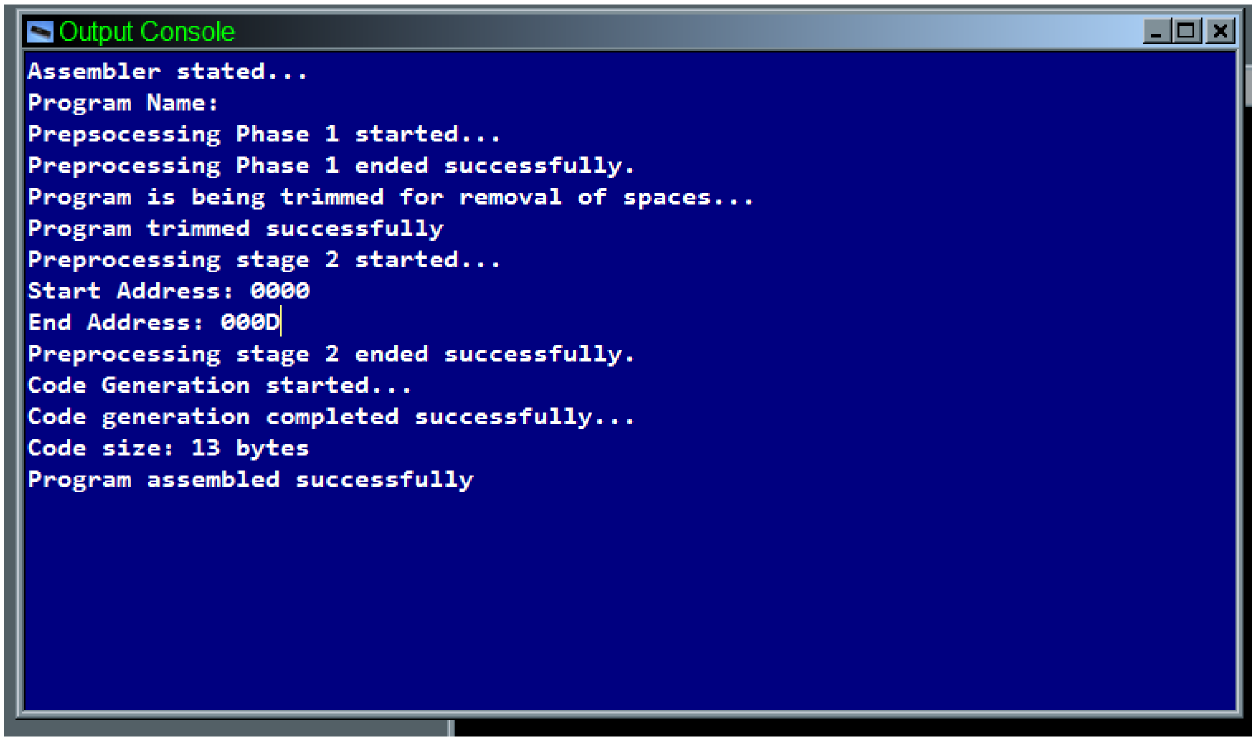
So, in order to write the program above, we first open the IDE using the method discussed above. Then open a new workspace, by clicking "Assembler Source File" and then select "New Source File". The IDE will be cleared, now write the code within the IDE. A snapshot is shown below:



```
Program is being trimmed for
Program trimmed successfully
Preprocessing stage 2 starte
Start Address: 0000
End Address: 000D
Preprocessing stage 2 ended
Code Generation started...
Code generation completed su
Code size: 13 bytes
Program assembled successful
```

```
LXI H, C100H
MOV A, M
INX H
ADD M
INX H
MOV M, A
INX H
XRA A
ADC A
MOV M, A
HLT
```

Once the program is written down, as above; the next step is to assemble it and load it into the simulator execute memory. For this, click within the Assembler IDE to bring it into focus, then click on the "Assemble" menu, the from the submenu select "Assemble and Load Program" option. This will invoke the internal assembler which will assemble the program and generate the machine code directly into the simulator memory. The compilation report with errors are shown in the debug dialog that will now pop up.



```
Assembler stated...
Program Name:
Prepsocessing Phase 1 started...
Preprocessing Phase 1 ended successfully.
Program is being trimmed for removal of spaces...
Program trimmed successfully
Preprocessing stage 2 started...
Start Address: 0000
End Address: 000D
Preprocessing stage 2 ended successfully.
Code Generation started...
Code generation completed successfully...
Code size: 13 bytes
Program assembled successfully
```

If some error is reported, then correct the source and remove the errors, then retry the above process. In the present case, there are no error. So the program had been loaded into the simulator memory starting at location 0000H.

Next load the memory viewer, by clicking "Intel 8085" menu and click on "Memory Viewer" submenu option. This will bring up the memory viewer showing the contents of the memory locations. Locate the row C1 now and enter 2 input values at the columns 00H, 01H, which will load up the values at memory locations C100H, C101H.

Next step, is to execute the program, so go to "Machine State" menu and click on "Reset" option to reset and make the machine IP jump to location 0000H. Again from "Machine State" menu, click on "Start Execution". On doing this, the machine state will be "RUNNING" for very short time, and will again be "STOPPED", as indicated in the title bar.

Now we will have to view the result. To do so, go to the memory viewer and locate row C1 and column 02 for sum located at C102H. And the carry bit in column 03H. This completes the whole process of assembling and executing a 8085 program.

Further advanced programming, like looping, port communications and subroutines are discussed in the further chapters that proceeds.

CONDITION EVALUATION AND LOOPING IN 8085

Every programming language must implement statements for taking decisions based on certain conditions and also instructions to repeatedly perform certain tasks a required number of times. But closely looking at looping statements of every programming language you can see that the looping statements can be broken down to 3 basic statements:

- Initialize the iteration variable, to be used to keep track of the loop.
- Check the state of the iteration variable for entry or exit condition.
- Iterate the looping variable.

In this session, we will first discuss the simulation of decision statements. In common HLL, the decision statements are blocked by IF-THEN-ELSE with operators and instruction blocks for condition and execution terms. But in assembly language, there are mnemonics that uses the flag register and jump statements that are used to break a HLL IF-THEN-ELSE statement to basic components.

Let us first consider a small program to fetch a value from C100H memory location and check if it is negative, positive or zero. If it is zero, the store 00 in next memory location. Otherwise, store 01 for positive and 02 for negative. The code to be entered in the IDE editor is shown below:

```
      LXI H, C100H
      XRA A
      ADD M
      INX H
      JP POS
      JM NEG
      XRA A
      JMP SVE
POS:  MVI A, 01H
      JMP SVE
NEG:  MVI A, 02H
      MOV M, A
      HLT
```

Thus, after entering the program into the assembler IDE and the assembling a nd loading the program into the simulator memory, open the Memory Viewer. Then locate the memory cell C100H, enter any hexadecimal number. Now Reset and Execute the Simulator. After the execution completes, then re-open the memory viewer and locate the memory cell C101H, which will contain the state representation of the number at C100H.

A closer analysis of the program reveals that, there are jump labels. The jump labels are actually converted to memory locations during assembling. Thus, programmers can use Jump levels to simplify the programming task and alternatively, can also hard code the 16 bit memory location values in their program. Thus, using flag register, where we actually have used the Sign flag, for testing of the positivity or negativity condition and consequently jump to our target location. But for Zero, no comparison is done, because, if all testing results in false, then no jump redirection takes place, and the control falls through, to the zero detect-execute part, where we have actually assigned 0 to accumulator, which is later saved to C101H.

Next, we will examine the loop execution. Here, if we closely analyze a looping program, we will see that it has a test and jump instruction. The only difference is that, the jump will not be a forward jump to an exit point but to a point within the program from where the test and jump can be executed again. So a looping program has exit-condition or entry-condition when written in assembly language.

Example- Here we will write a program, that will load 2 values from C100H and C101H and loop through an empty NOP instruction to just cause some delay.

```
        LXI H, C100H
        XRA A
        ADD M
        INX H
LOOP:   CMP M
        JZ EXT
        NOP
        NOP
        NOP
        NOP
        NOP
        INR A
        JMP LOOP
EXT:    HLT
```

In looping program above, we see that, the condition is checked at entry and not at exit. If the value of A is the same as that of the content of the memory location C101H, then the JZ EXT is executed, as now the CMP M will cause the Z flag to be set. his JZ EXT cause the program control to be redirected to the HLT instruction out of the body of the loop. Thus, on exit from the loop the program ends.

Otherwise, the NOP sequences are executed, which cause empty delays, then the value of A is incremented by 1. Finally, the JMP LOOP causes a unconditional jump to the beginning of the loop, where again the loop condition tests the value of A against content if memory location C101H.

SUBROUTINES IN ASSEMBLER

Subroutines can be defined as an individual segment of program, which can be called by the parent program several times. The subroutines can be described as a way to modular development of programming. The advantage of using subroutine is that, the individual subroutines can be individually tested and debugged, then ultimately the subroutine can be integrated within the main program. As this subroutine can be utilized by multiple program, any number of times, the subroutine introduces the concept of reusability.

In order to use Subroutine, we are required to use stack. When a subroutine is called, the following action takes place-

- Program counter (PC) is loaded with the address of the next instruction, that will be executed on return from the subroutine.
- The new PC is saved to the address as indicated by the SP register. The SP is decremented by 2.
- The PC is now loaded with the start address of the subroutine.
- The execution of the subroutine begins.

When the execution of the subroutine is done, the following action takes place-

- The end of subroutine is indicated by one of ***ret***urn instruction.
- When RET is invoked, the stack top vales are popped to PC and SP is now incremented by 2.
- And execution continues with this new PC, whose address lies within the address range of the calling parent program.

But, when working with subroutines and stacks, the SP (stack pointer) must be explicitly initialized with the starting address of the stack top. Also, the programmer must ensure that the SP address must be such that, the stack must not enter the code space or the data space. If the SP is not initialized, the simulator will by default use the starting address of stack as FFFF. But in real simulator, this can cause unexpected behavior. Hence, it is a proper programming technique, to always initialize the SP with a custom but well-defined address value.

In order to demonstrate the coding of subroutine we will consider a program to sort 2 values in memory locations C100H and C101H. Here we will design a subroutine which will just sort two values in register B and C.

```
 LXI SP, F100H
LXI H, C100H
MOV B, M
INX H
MOV C, M
CALL SORT
DCX H
```

```
       MOV M, B
       INX H
       MOV M, C
       HLT
SORT:  MOV A, B
       CMP C
       JM SKIP
       MOV D, B
       MOV B, C
       MOV C, D
SKIP:  RET
```

In this program ,we have first initialized the stack pointer to F100H and then assigned the starting address for our data in HL register to C100H. Then, we fetched the 2 input data into registers B and C. When the data are loaded into B, C registers, we invoke the subroutine named SORT. The SORT subroutine actually does the sorting in ascending order by comparing the values in B and C. If the value in B is greater than the value in C, the content of B and C registers are swapped. Else the swapping section is skipped. Then finally the RET instruction is executed and the parent program beings execution. When the execution of main program resumes, the content of the registers B and C are in sorted order, so we save the content of the registers B and C to the data locations C100H and C101H, respectively.

SIMPLE IO PORT INTERFACING AND COMMUNICATIONS

In 8085, there is no way to visually obtain the results and processed in information, unless we attach a visual device to the kit. But attaching any device will be useless, unless we establish a way of communicating with the device. This communication, in its simplest form, takes place with the IO port. Although alternate communication through Memory mapped IO is available for actual kits, the Simulator presently supports only IO mapped IO through ports. This communication requires only 2 available 8085 instructions - IN and OUT.

The IN instruction is used to receive a data byte from an external port to the accumulator A of the microprocessor. The target port number is specified as an operand to the instruction. Similarly, the OUT instruction is used to send a data from the microprocessor accumulator A to an external, through port number specified.

This simulator supports multiple virtual devices that can be used through their respective port addresses. We will discuss the communication with these devices with examples.

Consider the example of communicating with text input and output device. In this example, we will receive 5 characters from the text input and display the output in LIFO style, on the output text box. These devices have both input and output port address of 00H. The code is shown below-

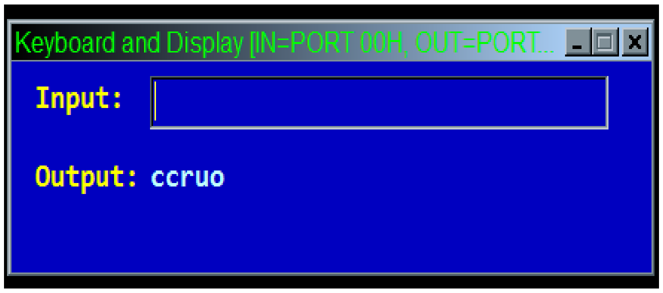
```
LXI SP, F100H

IN 00H
PUSH PSW
IN 00H
PUSH PSW
IN 00H
PUSH PSW
IN 00H
PUSH PSW
IN 00H
PUSH PSW

POP PSW
OUT 00H
POP PSW
OUT 00H
POP PSW
OUT 00H
POP PSW
OUT 00H
POP PSW
OUT 00H
```

```
HLT
```

A sample screen shown below shows the result of execution:



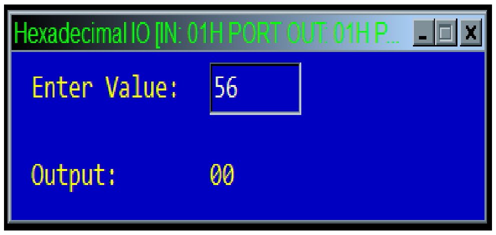
Input:

Output: ccruo

Next example, we will consider the Hex device through port 01H. It is also similar to the decimal device at port 02h. So we will only see the example of the device at port 01h, and the operational analogue of port 02h can be established by the user. Here we will consider an example, where we will get 2 numbers as input and produce the sum on the output.

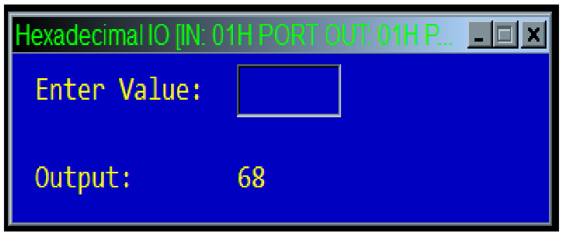
```
 XRA A
IN 01H
MOV B, A
IN 01H
ADD B
OUT 01H
HLT
```

The sample screen shots are shown below:



Enter Value: 56

Output: 00



Enter Value:

Output: 68

Next example, we will show the code to use the Bit Set Reset Display. But, you must remember that, that the bit set reset display should not be confused with the BSR mode. This display is useful when designing a ring counter. The code of which is shown below:

```
     MVI A, 80H
MAIN: OUT 03H
     RRC
     STA 0202H
     CALL DELAY
     LDA 0202H
     JMP MAIN
     HLT
DELAY: MVI A, FFH
DELAY1: NOP
     NOP
     NOP
     DCR A
     JNZ DELAY1
     RET
```

As a final example, we will write a code to use the communication port device using the simulator.