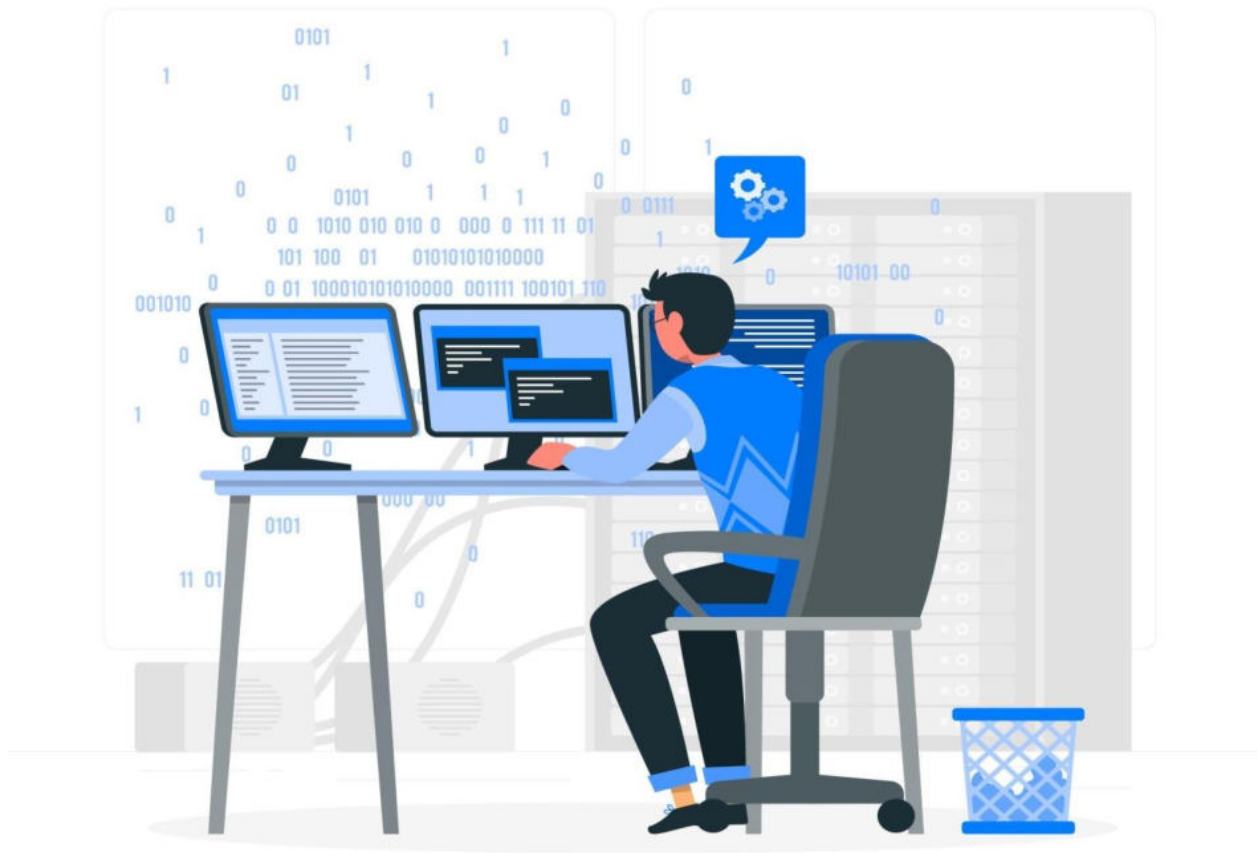


Fundamentals of 8085 Microprocessor Programming



Authors:

Dr. Manish Kashyap
Tushar Kukreti
Vivek Kumar

Published by:

Vandana Publications, Lucknow

FUNDAMENTALS OF 8085 MICROPROCESSOR PROGRAMMING

by

Dr. Manish Kashyap
Tushar Kukreti
Vivek Kumar

Published by:
Vandana Publications
78/77, New Ganesh Ganj, Opp. Kaysins Lane, Aminabad Road,
Lucknow – 226018, Uttar Pradesh, INDIA
Email: info@vandanapublications.com

Copyright © Dr. Manish Kashyap, Tushar Kukreti and Vivek Kumar

All rights reserved. No part of this publication may be reproduced or transmitted in any form or by any means, electronic or mechanical including photocopy, recording or by any information storage and retrieval system, without permission in writing from the copyright owner.

ISBN
978-93-90728-96-1

DOI
10.5281/zenodo.14885842

First Published
February 2025

All disputes are subject to Lucknow jurisdiction only.

Typesetting / Printed at:
Yellow Print, G-4, Goel Market, Lekhraj Metro Station, Indiranagar, Lucknow, India.
Ph: +91-7499403012
E-mail: yellowprints12@gmail.com

AUTHORS ARE FULLY LIABLE FOR ORIGINALITY AND WORDING

Every effort has been made to avoid errors or omissions in this publication. In spite of this, some errors might have crept in. Any mistake, error or discrepancy noted may be brought to our notice which shall be taken care of in the next edition. It is notified that neither the publisher nor the author or seller will be responsible for any damage or loss of action to anyone, of any kind, in any manner, therefrom. For binding mistakes, misprints or for missing pages etc., the publisher's liability is limited to replacement within one month of purchase by similar edition. All expenses in this condition are to be borne by the purchaser.

Fundamentals of 8085 Microprocessor Programming
by Dr. Manish Kashyap, Tushar Kukreti and Vivek Kumar

DISCLAIMER OF WARRANTY

The technical descriptions, procedures, and computer programs in this book have been developed with the greatest of care and they have been useful to the author in a broad range of applications; however, they are provided as is, without warranty of any kind.

The authors of the book titled “Fundamentals of 8085 Microprocessor Programming”, make no warranties, expressed or implied, that the equations, programs, and procedures in this book or its associated software are free of error, or are consistent with any particular standard of merchantability, or will meet your requirements for any particular application. They should not be relied upon for solving a problem whose incorrect solution could result in injury to a person or loss of property. Any use of the programs or procedures in such a manner is at the user's own risk. The editors, author, and publisher disclaim all liability for direct, incidental, or consequent damages resulting from use of the programs or procedures in this book or the associated software.

CONTENTS

Topic/Program No.	Title	Pg. No.
	Preface	5
	Acknowledgements	6
Topic/Program 1	Introduction to programming model of 8085	7
Topic/Program 2	Learning to deal with registers	8
Topic/Program 3	Learning to deal with memory.....	9
Topic/Program 4	Using HL register pair as a pointer via M register.....	10
Topic/Program 5	Learning to deal with carry flag.....	11
Topic/Program 6	Learning to deal with parity flag.....	12
Topic/Program 7	Learning to deal with auxiliary carry flag.....	13
Topic/Program 8	Learning to deal with zero flag.....	14
Topic/Program 9	Learning to deal with sign flag.....	15
Topic/Program 10	Learning how to deal with stack memory.....	16
Topic/Program 11	Learning to create out of sequence jump using labels.....	17
Topic/Program 12	Learning to create loop using labels.....	19
Topic/Program 13	Learning to deal with input and output devices connected in IO mode to 8085 microprocessor.....	21
Topic/Program 14	Learning subroutines or functions.....	22
Topic/Program 15	Learning software interrupts.....	23
Topic/Program 16	Learning Hardware Interrupts.....	25
Topic/Program 17	Learning Assembler Directives.....	26
Topic/Program 18	Instruction set of 8085 microprocessor.....	28

PREFACE

This book gets the user started with programming the 8085 based microcomputer systems. The first topic is all about getting to know the programming model of 8085 microprocessor. Once that is understood, the rest topics deal with operating various parts of the microprocessor like CPU registers, flag register, stack, memory etc. Programming concepts based on if-else conditions, loops are shown to be implemented by conditional and unconditional jump statements which are a part of the instruction set of 8085 microprocessors.

The topics are so designed so that the user can the operating procedure through illustrative programs and then can apply those concepts in finding solutions to a given problem.

ACKNOWLEDGEMENTS

The authors acknowledge the direct and indirect contribution of everybody who has inspired writing this book. We would like to acknowledge the authors of the numerous books and articles that we consulted during the writing of this book. Their work served as a valuable resource and helped shape our perspective on the subject.

Finally, we want to thank Department of Electronics and Communication Engineering, Maulana Azad National Institute of Technology for their unwavering love, patience, and understanding. Their support has been instrumental in every aspect of our professional life, and we dedicate this book to them.

Topic / Program 1. Introduction to programming model of 8085

The programming model of a microprocessors is programmer's view of the computer. That is to say - what all hardware portion is accessible to the programmer for manipulating in the program. For 8085 microprocessors, it is as follows –

A (8 bit)	S	Z	X	AC	X	P	X	C
B (8 bit)	C (8 bit)							
D (8 bit)	E (8 bit)							
H (8 bit)	L (8 bit)							
PC (16 bit)								
SP (16 bit)								
M (8 bit)								

To be able to perform any operation, microprocessor needs memory. Memory can be of many types. Shown above is the memory of CPU called CPU registers. This memory is internal to CPU and is integrated into the microprocessor chip itself. A, B, C, D, E, H, L, M, PC and SP are registers of 8085.

Register A (8 bit or 1 byte register) is called as accumulator register. As the name suggests, it accumulates the results of most of the arithmetic/ logical and many more instructions in itself after computation.

Registers B, C, D, E, H and L (8 bit or 1-byte registers) are general purpose registers which can be used to store 8-bit data. Some registers can be used in pair like BC, DE and HL (and not CB, ED and LH). Similarly, B cannot be paired with H and so on. Register pairs are useful to store 16 bit or one word data. Some register pairs have commands dedicated to them like LDAX B (discussed later). The combination of register A and F (flag) is called as program status word i.e. PSW.

HL Register pair is used as pointer register together with M (8 bit) register (not a part of hardware, but just a mirror register that mirrors the content of memory location pointed to by HL register pair). Although memory operations are dealt in relevant section but here it suffices to understand the following – 8085 is a 8 bit microprocessor because it has a bidirectional data bus is of 8 bits. It has a 16-bit unidirectional address bus. To access a memory location from externally interfaced memory (other than CPU registers shown above), a 16-bit (4 hexadecimal digit) address is put onto address bus. For pointer operation, if we put the 16-bit address in HL register pair, the data of that memory location which is 8-bit or a byte is copied into M register automatically. Also, if we edit that data at the memory location, the change is automatically reflected in M register and vice versa.

PC or program counter register holds the address of next instruction to be executed in itself. Similarly, SP or stack pointer keeps the address of top of stack in itself (More about SP in the relevant section)

The register shown to right of accumulator is called as flag register. It has 8 bits and each bit represents a special condition after the execution of a given instruction in 8085. Out of 8 bits only 5 bits are useful and 3 are don't cares (marked as 'x'). The useful bits are C-carry, P-parity, AC- Auxiliary carry, Z-Zero and S- Sign. These are discussed in relevant sections.

Topic / Program 2. Learning to deal with registers

2.1 Detailed objective and coding strategy– The detailed objective with the code defaults (i.e., where does the code start from in memory and where is the data kept etc.) is given below-

<p>Objective- Understanding register's is an important part in the journey to explore 8085 microprocessor. Registers are like small, high-speed memory locations within the processor itself.</p> <p>The 8085 microprocessor provides a set of 6 8-BIT general purpose registers, that are register B, C, D, E, H, L along with the accumulator (register A). Each of these general-purpose registers are used to store any temporary 8 bit data or be combined into 16-bit register pairs (BC, DE and HL pair only) for 16 bit data storage.</p> <p>The 8085 microprocessor offers a variety of instructions for manipulating data in its registers some of the basic instructions are-</p> <p>1.MVI :- This instruction is used to load (Move) any 8 BIT immediate data directly into a register. e.g :- MVI A, 20H The above line of code store's 20H into the accumulator. By immediate data we mean that the data is a part of instruction itself.</p> <p>2.MOV:- This is another fundamental instruction in 8085, that is used to copy the content of source register into destination register. e.g :- MOV B, C The above line of code will copy the content of register C (source) into register B (destination).</p>	Code location	5000 H		
	Subroutine 1 location	NA		
	Subroutine 2 location	NA		
	Interrupt location	NA		
	Data used in memory – Shown for the main program below			
Address	Data	Address	Data	
5000 H	3E	5008 H	00	
5001 H	A7	5009 H	00	
5002 H	47	500A H	00	
5003 H	21	500B H	00	
5004 H	10	500C H	00	
5005 H	28	500D H	00	
5006 H	76	500E H	00	
5007 H	00	500F H	00	
<p>3.LXI:- This instruction is used to load any 16 bit data to the specified register pair. This can be thought of as 16 bit equivalent of MVI command. e.g :- LXI B, 2010h The above code will load 2010H (a 16 bit data) into the register pair BC .</p> <p>NOTE:- In 8085 programming, register pairs are represented by the first register of the pair. e.g:- In above code register pair BC is represented by B. Also note that if there is 'X' in the mnemonic of the command then most probably it deals with 16 bit operation.</p>				

--	--

2.2 Code with explanation – Following is the code that fulfils the above objective with explanation –

Ins. No.	Mnemonics	Explanation
1	MVI A,0A7H	Load's A7h into register A (Accumulator). H is appended to the end of the number to tell the simulator that the number is to be treated as hexadecimal. One can use D for decimal or B for binary.
2	MOV B,A	Copies the content of register A (A7H) into register B
3	LXI H,2810H	Load's 28H to register H and 10H to register L
4	HLT	Halt/stop the computer.

2.3 Results and Further directions – The above program successfully demonstrate the basic operations on register.

Note:- Whenever we write a hexadecimal number starting with non-numeric digit (A, B, C, D, E, or F), it is necessary to add a leading zero (0) when you perform the experiment on simulator just to tell the simulator that it is actually a number that you have entered. On the actual hardware kit, it is not required.

Topic / Program 3. Learning to deal with memory

3.1 Detailed objective and coding strategy– The detailed objective with the code defaults (i.e., where does the code start from in memory and where is the data kept etc.) is given below-

<p>Objective- This program will help us learn about the externally interfaced memory of 8085. Since, the memory of internal registers is limited, in 8085 64 Kilo Byte (KB) of externally interfaced memory can be attached. Firstly this number 64 KB is not arbitrary. It comes from the size of address and data bus. Since the address bus is of 16 bits, we can have at most 2^{16} total addresses (or memory locations). $2^{16} = 2^6 \times 2^{10} = 64 \times \text{Kilo}$. Also, the data bus is of 1 byte (8 bits) and hence every memory location has a byte size storage. Hence the complete memory is of 64 Kilo Byte.</p> <p>Since addresses are of 16 bits, they can be represented by hexadecimal number of 4 digits and hence all addresses will range from</p>	Code location	2000 H		
	Subroutine 1 location	NA		
	Subroutine 2 location	NA		
	Interrupt location	NA		
	Data used in memory – (Shown for both data and code memory)			
	Address	Data	Address	Data
	1000 H	05	2003 H	3C
	1001 H	00	2004 H	32
	1002 H	00	2005 H	01
	--	--	2006 H	10
--	--	2007 H	76	
2000 H	3A	2008 H	00	
2001 H	00	2009 H	00	
2002 H	10	200A H	00	

<p>0000 to FFFF in hexadecimal. Similarly, every memory location can hold 8 bit data and hence the data range will be 00 to FF in hexadecimal.</p> <p>In this program we aim to access one memory location from externally interfaced memory. Read data from it and write data to it. To that end, we assume that at memory address 1000H a data byte of 05H is kept. We intend to read it inside CPU (Accumulator register), modify it by increasing its value by one and storing the modified result at the next location which is 1001H. The result should be 06H after execution of the program.</p>	<p>Shown above is a small section of total memory. Notice that data is stored at location 1000 H as 05 H. Then from location 2000 H the hexadecimal equivalent of the code is stored. We do not need to worry about those numbers now. They will become clear to us in relevant sections.</p> <p>At location 1001 H, currently 00 H is stored. After execution of the program, it will become 06 H. Only the locations used in the code are shown highlighted. Remember – whether it is code or data, it is stored in the same 64 KB memory at different locations</p>
--	--

3.2 Code with explanation – Following is the code that fulfils the above objective with explanation –

Ins. No.	Mnemonics	Explanation
1	LDA 1000H	LDA stands for loading into Accumulator, 1 byte data from memory location 1000 H
2	INR A	INR increments the specified register by 1
3	STA 1001H	STA stores the content of accumulator to the specified address. (In this case 1001 H)
4	HLT	Halts/Stops the computer

3.3 Results and Further directions – There are many more ways to access memory. They are discussed later in this document. Also note that if you are to access one memory location (read or write), you will also be able to read multiple such locations by doing the read and/or write operation in a loop. Loops are discussed in the coming section of this document.

Topic / Program 4. Using HL register pair as a pointer via M register

4.1 Detailed objective and coding strategy– The detailed objective with the code defaults (i.e., where does the code start from in memory and where is the data kept etc.) is given below-

Objective- In this program we aim to use the HL pair as a pointer.	Code location	5002 H
	Subroutine 1 location	NA
	Subroutine 2 location	NA
	Interrupt location	NA
H and L registers when used together can be		

<p>used to hold the 16-bit data which is treated as address if we use it with M register. The upper 8 bits are stored in H and the lower 8 bits in L. The content at this location can then be accessed using M register.</p> <p>The code explains how to copy the contents of the A(Accumulator) to the specified location by HL pair using the M(Memory) register.</p>	<p>Data used in memory – shown below for data and program both after execution of progr.am</p> <table border="1"> <thead> <tr> <th>Address</th> <th>Data</th> <th>Address</th> <th>Data</th> </tr> </thead> <tbody> <tr> <td>5000 H</td> <td>15</td> <td>5008 H</td> <td>77</td> </tr> <tr> <td>5001 H</td> <td>00</td> <td>5009 H</td> <td>76</td> </tr> <tr> <td>5002 H</td> <td>3E</td> <td>500A H</td> <td>00</td> </tr> <tr> <td>5003 H</td> <td>15</td> <td>500B H</td> <td>00</td> </tr> <tr> <td>5004 H</td> <td>26</td> <td>500C H</td> <td>00</td> </tr> <tr> <td>5005 H</td> <td>50</td> <td>500D H</td> <td>00</td> </tr> <tr> <td>5006 H</td> <td>2E</td> <td>500E H</td> <td>00</td> </tr> <tr> <td>5007 H</td> <td>00</td> <td>500F H</td> <td>00</td> </tr> </tbody> </table> <p>The content of memory location 5000H before execution will be 00H, after execution it will be 15H (as shown)</p>	Address	Data	Address	Data	5000 H	15	5008 H	77	5001 H	00	5009 H	76	5002 H	3E	500A H	00	5003 H	15	500B H	00	5004 H	26	500C H	00	5005 H	50	500D H	00	5006 H	2E	500E H	00	5007 H	00	500F H	00
Address	Data	Address	Data																																		
5000 H	15	5008 H	77																																		
5001 H	00	5009 H	76																																		
5002 H	3E	500A H	00																																		
5003 H	15	500B H	00																																		
5004 H	26	500C H	00																																		
5005 H	50	500D H	00																																		
5006 H	2E	500E H	00																																		
5007 H	00	500F H	00																																		

4.2 Code with explanation – Following is the code that fulfils the above objective with explanation –

Ins. No.	Mnemonics	Explanation
1	MVI A, 15H	Moving a dummy value (say 15h) to A(Accumulator).
2	MVI H, 50H	Loading register H with Upper 8 bits of address. (here 50h)
3	MVI L, 00H	Loading register L with Lower 8 bits of address. (here 00h). Alternatively, LXI H,5000 can be used.
4	MOV M, A	Moving the contents of Accumulator to location in HL pair using M(Memory)
5	HLT	Halt the program

4.3 Results and Further directions – The program efficiently uses the M register and updates the content at the location specified by HL pair. Note that operations like MVI M, (8 bit data) and INR M are also valid operations and might be useful in other use cases.

Topic / Program 5. Learning to deal with carry flag

5.1 Detailed objective and coding strategy– The detailed objective with the code defaults (i.e., where does the code start from in memory and where is the data kept etc.) is given below-

Objective- In this experiment, we are going to discuss the carry flag. As we have already learned about flags in experiment 5, we know	Code location	5000 H
	Subroutine 1 location	NA
	Subroutine 2 location	NA

<p>that flags serve as status indicators, providing information about the outcome of arithmetic and logical operations.</p> <p>When performing arithmetic operations such as addition or subtraction, if the result exceeds the capacity of the register, the carry flag is raised to signal a carry-out or borrow situation.</p> <p>For instance, Consider adding two numbers that surpass the register size or subtracting a larger number from a smaller one. In both cases, the resulting operation triggers the carry flag to be raised, ensuring the processor can accurately handle overflow and borrow situations.</p>	Interrupt location	NA																																				
	<p>Data used in memory – Shown for the main program below</p> <table border="1"> <thead> <tr> <th>Address</th> <th>Data</th> <th>Address</th> <th>Data</th> </tr> </thead> <tbody> <tr> <td>5000 H</td> <td>3E</td> <td>5008 H</td> <td>00</td> </tr> <tr> <td>5001 H</td> <td>FF</td> <td>5009 H</td> <td>00</td> </tr> <tr> <td>5002 H</td> <td>06</td> <td>500A H</td> <td>00</td> </tr> <tr> <td>5003 H</td> <td>01</td> <td>500B H</td> <td>00</td> </tr> <tr> <td>5004 H</td> <td>80</td> <td>500C H</td> <td>00</td> </tr> <tr> <td>5005 H</td> <td>76</td> <td>500D H</td> <td>00</td> </tr> <tr> <td>5006 H</td> <td>00</td> <td>500E H</td> <td>00</td> </tr> <tr> <td>5007 H</td> <td>00</td> <td>500F H</td> <td>00</td> </tr> </tbody> </table>			Address	Data	Address	Data	5000 H	3E	5008 H	00	5001 H	FF	5009 H	00	5002 H	06	500A H	00	5003 H	01	500B H	00	5004 H	80	500C H	00	5005 H	76	500D H	00	5006 H	00	500E H	00	5007 H	00	500F H
Address	Data	Address	Data																																			
5000 H	3E	5008 H	00																																			
5001 H	FF	5009 H	00																																			
5002 H	06	500A H	00																																			
5003 H	01	500B H	00																																			
5004 H	80	500C H	00																																			
5005 H	76	500D H	00																																			
5006 H	00	500E H	00																																			
5007 H	00	500F H	00																																			

5.2 Code with explanation – Following is the code that fulfils the above objective with explanation –

Ins. No.	Mnemonics	Explanation
1	MVI A, 0FFH	Loaded maximum possible 8-bit number in register A
2	MVI B, 01H	Load any dummy value (say 01H) to register B
3	ADD B	Add the content of register B to A, (carry flag is set to 1)
4	HLT	Halt/stop the computer

5.3 Results and Further directions – The sample code clearly illustrates the fundamental operation of the carry flag. The addition of 01 H With Register A which was preloaded with the maximum 8-bit value (FFH), sets the carry flag (i.e. make its value 1), showcasing its role in signaling arithmetic overflow. In addition to arithmetic operations like addition and subtraction, the carry flag in the 8085 microprocessor can be set in several other scenarios like logical operations, Rotation, Shifting, Increment, Decrement and many other operations, its applications are not limited and is very useful in conditional branching and jumping operations as will be demonstrated later.

Topic / Program 6. Learning to deal with parity flag

6.1 Detailed objective and coding strategy– The detailed objective with the code defaults (i.e., where does the code start from in memory and where is the data kept etc.) is given below-

<p>Objective- This experiment aims at explaining the working of the parity flag and</p>	Code location	5000 H
	Subroutine 1 location	NA

<p>how to deal with it.</p> <p>As discussed in Experiment 5, Flags are the special purpose registers whose value is set (1) or reset (0) after any arithmetic or logical operation is performed. Parity flag keeps in account if numbers of set bits are odd or even in the binary equivalent of the result in accumulator. The operation must be either arithmetic or logical.</p> <p>If the number of 1s is even in accumulator then the parity flag is set (1). If the number of 1s is odd, the parity flag is reset (0). Another way to remember this is that the number of ones in accumulator register and parity bit combined should always be odd. That is why we say that 8085 works on odd parity.</p>	Subroutine 2 location	NA		
	Interrupt location	NA		
	Data used in memory – Shown for the main program below			
	Address	Data	Address	Data
	5000 H	3E	5008 H	76
	5001 H	02	5009 H	00
	5002 H	06	500A H	00
	5003 H	02	500B H	00
	5004 H	04	500C H	00
	5005 H	80	500D H	00
5006 H	C6	500E H	00	
5007 H	02	500F H	00	

6.2 Code with explanation – Following is the code that fulfils the above objective with explanation –


Ins. No.	Mnemonics	Explanation
1	MVI A, 02H	Move a dummy value (say 02) to A(Accumulator).
2	MVI B, 02H	Move a dummy value (say 02) to B.
3	INR B	Increase the value of B by 1. (Parity is set to 1)
4	ADD B	Add the contents of B to A (Parity is set to 1)
5	ADI 02H	Add 02 to A (Parity is reset to 0)
6	HLT	Halt/Stop the computer.

6.3 Results and Further directions – The parity flag has been understood with the help of a sample program.

Topic / Program 7. Learning to deal with auxiliary carry flag

7.1 Detailed objective and coding strategy– The detailed objective with the code defaults (i.e., where does the code start from in memory and where is the data kept etc.) is given below-

<p>Objective- This experiment aims at understanding and implementing the auxiliary carry.</p> <p>Note below the numbering of bits from Least</p>	Code location	5000 H
	Subroutine 1 location	NA
	Subroutine 2 location	NA
	Interrupt location	NA
	Data used in memory – Shown for the main	

<p>significant bit or LSB (numbered 0) to most significant bit or MSB (numbered 7). A group of 4 bits is called as nibble.</p> <div style="text-align: center;">  </div> <table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td> </tr> <tr> <td colspan="4" style="text-align: center;">Upper nibble</td> <td colspan="4" style="text-align: center;">Lower nibble</td> </tr> </table> <p>During arithmetic operations involving addition or subtraction, if a carry is generated from the third to the fourth bit (or from the lower nibble to upper nibble), it sets the auxiliary carry flag.</p>	7	6	5	4	3	2	1	0	Upper nibble				Lower nibble				<p>program below</p> <table border="1"> <thead> <tr> <th>Address</th><th>Data</th><th>Address</th><th>Data</th></tr> </thead> <tbody> <tr><td>5000 H</td><td>3E</td><td>5008 H</td><td>00</td></tr> <tr><td>5001 H</td><td>09</td><td>5009 H</td><td>00</td></tr> <tr><td>5002 H</td><td>06</td><td>500A H</td><td>00</td></tr> <tr><td>5003 H</td><td>08</td><td>500B H</td><td>00</td></tr> <tr><td>5004 H</td><td>80</td><td>500C H</td><td>00</td></tr> <tr><td>5005 H</td><td>76</td><td>500D H</td><td>00</td></tr> <tr><td>5006 H</td><td>00</td><td>500E H</td><td>00</td></tr> <tr><td>5007 H</td><td>00</td><td>500F H</td><td>00</td></tr> </tbody> </table>	Address	Data	Address	Data	5000 H	3E	5008 H	00	5001 H	09	5009 H	00	5002 H	06	500A H	00	5003 H	08	500B H	00	5004 H	80	500C H	00	5005 H	76	500D H	00	5006 H	00	500E H	00	5007 H	00	500F H	00
7	6	5	4	3	2	1	0																																														
Upper nibble				Lower nibble																																																	
Address	Data	Address	Data																																																		
5000 H	3E	5008 H	00																																																		
5001 H	09	5009 H	00																																																		
5002 H	06	500A H	00																																																		
5003 H	08	500B H	00																																																		
5004 H	80	500C H	00																																																		
5005 H	76	500D H	00																																																		
5006 H	00	500E H	00																																																		
5007 H	00	500F H	00																																																		

7.2 Code with explanation – Following is the code that fulfils the above objective with explanation –

Ins. No.	Mnemonics	Explanation
1	MVI A, 09H	Load's 09h (0000 1001) into register A (Accumulator)
2	MVI B, 08H	Load's 08h (0000 1000) into register B
3	ADD B	Add the content of register B with A (A = A+B)
4	HLT	Halt/Stop the computer

7.3 Results and Further directions – In the preceding program, we gained insight into the generation of an auxiliary carry. When the accumulator, initially loaded with 09H (binary = 0000 1001), underwent addition with register B, holding the value 08H (binary = 0000 1000), a carry emerged from 3rd to 4th bit (or from lower to higher nibble). This occurrence resulted in the setting of the auxiliary carry flag. Although auxiliary carry is not used frequently during the 8085 programming, but its use cases are not limited. Some of its uses are: - Certain instructions in the 8085-assembly language, such as DAA (Decimal Adjust Accumulator), rely on the auxiliary carry flag to adjust the result of arithmetic operations. In Binary Coded Decimal (BCD) arithmetic, where numbers are represented in base-10 with each digit encoded in binary, the AC flag is particularly useful for detecting carry-out from the lower nibble to the higher nibble, ensuring accurate BCD arithmetic operations)

Topic / Program 8. Learning to deal with zero flag

8.1 Detailed objective and coding strategy– The detailed objective with the code defaults (i.e., where does the code start from in memory and where is the data kept etc.) is given below-

Objective- This experiment is all about the working of zero flag and how to deal with it.	Code location	5000 H
	Subroutine 1 location	NA

<p>As discussed earlier, Flags are the special purpose registers whose value is set (1) or reset (0) after any arithmetic or logical operation is performed.</p> <p>Zero flag checks if the result obtained after any arithmetic or logical operation is zero or not.</p> <p>If the result is zero, the zero flag is set (1).</p> <p>If the result is a non-zero number, the zero flag is reset (0).</p>	Subroutine 2 location	NA		
	Interrupt location	NA		
	Data used in memory – Shown for the main program below			
	Address	Data	Address	Data
	5000 H	06	5008 H	00
	5001 H	01	5009 H	00
	5002 H	05	500A H	00
	5003 H	04	500B H	00
	5004 H	3E	500C H	00
	5005 H	05	500D H	00
5006 H	97	500E H	00	
5007 H	76	500F H	00	

8.2 Code with explanation – Following is the code that fulfils the above objective with explanation –

Ins. No.	Mnemonics	Explanation
1	MVI B, 01H	Move a dummy value (say 01) to B.
2	DCR B	Decrease the content of B by 1. (Zero flag is set to 1)
3	INR B	Increase the value of B by 1. (Zero flag is reset to 0)
4	MVI A, 05H	Move a dummy value (say 05) to A(Accumulator).
5	SUB A	Subtract A from A. (Zero flag is set to 1)
6	HLT	Halt/stop the computer.

8.3 Results and Further directions – The experiment demonstrates the functionality of the zero flag. Zero flag finds its uses in result evaluation, conditional looping instructions (to be discussed later in the book) and even when comparing two numbers.

Topic / Program 9. Learning to deal with sign flag

9.1 Detailed objective and coding strategy– The detailed objective with the code defaults (i.e., where does the code start from in memory and where is the data kept etc.) is given below-

<p>Objective- In this Experiment we are going to discuss the sign flag. Sign Flag is again a very important flag to identify whether the result of an operation is positive or negative.</p> <p>If the Most Significant Bit (MSB) of the result is 1 then it refers to a negative integer</p>	Code location	5000 H
	Subroutine 1 location	NA
	Subroutine 2 location	NA
	Interrupt location	NA
	Data used in memory – Shown below for main code	

and the Sign Flag is set to (1). And if the MSB of the result is 0 then it refers to a positive integer and the Sign Flag is again reset to (0).

The most common use of this flag is found in branching operations, when distinct actions need to be taken based on whether the result of an operation is positive or negative. For instance, we may execute a specific task if the result is positive and proceed with a different task if it's negative.

Address	Data	Address	Data
5000 H	3E	5008 H	00
5001 H	04	5009 H	00
5002 H	06	500A H	00
5003 H	08	500B H	00
5004 H	90	500C H	00
5005 H	76	500D H	00
5006 H	00	500E H	00
5007 H	00	500F H	00

9.2 Code with explanation – Following is the code that fulfils the above objective with explanation –

Ins. No.	Mnemonics	Explanation
1	MVI A, 04H	Load's 09h (0000 1001) into register A (Accumulator)
2	MVI B, 08H	Load's 08h (0000 1000) into register B
3	SUB B	Subtract the content of register B with A (A = A-B)
4	HLT	Halt/Stop the computer

9.3 Results and Further directions – In the above program, when register A (preloaded with 04H) is subtracted by B (preloaded with 08H), then the result is a negative number which is reflected by the sign flag.

Topic / Program 10. Learning how to deal with stack memory

10.1 Detailed objective and coding strategy– The detailed objective with the code defaults (i.e., where does the code start from in memory and where is the data kept etc.) is given below-

Objective- To learn operation of stack In 8085, by default the stack is implemented from the bottom part of memory. Stack, as the name suggests, is last in first out list. It can also be initialized from anywhere in the memory by using SPHL command which works as follows – First HL pair is initialized by a desired address from where we want the stack to start. Then SPHL copies that starting	Code location	3000 H	
	Subroutine 1 location	NA	
	Subroutine 2 location	NA	
	Interrupt location	NA	
	Data used in memory – Shown for main program and stack (Status after execution of line no. 4 in code)		
Address	Data	Address	Data
3000 H	21	3008 H	E1

data into SP which is stack pointer and keeps the track of current top of stack (assumed used). PUSH command puts 16 bit data into the stack in 2 locations over the top of stack in little endian format. Suppose that the data was in register pair BC to begin with, PUSH B will put the data from higher order register B to higher address in memory out of the two bytes that are going to be filled and register C i.e., the lower order byte will be stored at lower order address out of the two locations being used.	3001 H	0F	3009 H	76
	3002 H	50	300A H	00
	3003 H	F9	--	--
	3004 H	01	500C H	00
	3005 H	13	500D H	13
	3006 H	12	500E H	12
	3007 H	C5	500F H	00
Similarly, POP command is used to take 2-byte data from stack. Hence stack pointer is decremented by 2 after every PUSH operation and incremented by 2 after every POP operation.				

10.2 Code with explanation – Following is the code that fulfils the above objective with explanation –

Ins. No.	Mnemonics	Explanation
1	LXI H, 500FH	Initialize HL pair by desired top of stack address.
2	SPHL	Top of stack initialized to 500FH (SP = 500FH)
3	LXI B, 1213H	Load B with 12 and C with 13.
4	PUSH B	The content of BC is stored on stack as follows - (SP=500DH, loc 500DH data=13, loc 500EH data =12)
5	POP H	2 bytes from the stack are stored in the HL pair in little endian format as follows- SP=500DH, H=13, L=12)
6	HLT	Halt/stop the computer.

10.3 Results and Further directions – The above code highlights the basic implementation of stack. Readers to note that the Push and Pop instructions can only be used with register pairs like BC, DE and HL. The default working of stack is the pointer is whenever the push instruction is implemented, the stack pointer is decremented by 2 and then the value gets stored using little endian format. The stack is also useful when we want to manipulate the content of flag registers and Accumulator as one entity known as PSW (Program Status Word) (will be discussed later). This is one indirect way of altering the content of flag registers.

Topic / Program 11. Learning to create out of sequence jump using labels

11.1 Detailed objective and coding strategy– The detailed objective with the code defaults (i.e., where does the code start from in memory and where is the data kept etc.) is given below-

<p>Objective- Labels are symbolic names given to memory addresses in the program. They serve as markers or placeholders to represent specific memory locations. Labels make code more readable and maintainable by providing meaningful names instead of raw memory addresses. They are commonly used with jump and call instructions to specify the location for branching.</p> <p>For the simplicity of the program, we will be using only unconditional jump (JMP).</p> <p>It is worthwhile to mention that labels do not have opcode, and therefore, when programming on the actual kit, the exact address must be given along with jump instruction.</p>	Code location	5000 H																																																	
	Subroutine 1 location	NA																																																	
	Subroutine 2 location	NA																																																	
	Interrupt location	NA																																																	
	<p>Data used in memory – (The hexadecimal equivalent of code is shown below to trace the location of L1 label in the program. Note all the memory locations and corresponding data is in hexadecimal numbers.)</p> <table border="1"> <thead> <tr> <th>Address</th> <th>Data</th> <th>Instruction</th> <th>Address</th> <th>Data</th> <th>Instruction</th> </tr> </thead> <tbody> <tr> <td>5000</td> <td>3E</td> <td rowspan="2">MVI A,02H</td> <td>5008</td> <td>80</td> <td>L1: ADD B</td> </tr> <tr> <td>5001</td> <td>02</td> <td>5009</td> <td>76</td> <td>HLT</td> </tr> <tr> <td>5002</td> <td>06</td> <td rowspan="2">MVI B,03H</td> <td>500A</td> <td>00</td> <td></td> </tr> <tr> <td>5003</td> <td>03</td> <td>500B</td> <td>00</td> <td></td> </tr> <tr> <td>5004</td> <td>C3</td> <td rowspan="3">JMP L1</td> <td>500C</td> <td>00</td> <td></td> </tr> <tr> <td>5005</td> <td>08</td> <td>500D</td> <td>00</td> <td></td> </tr> <tr> <td>5006</td> <td>50</td> <td>500E</td> <td>00</td> <td></td> </tr> <tr> <td>5007</td> <td>90</td> <td>SUB B</td> <td>500F</td> <td>00</td> <td></td> </tr> </tbody> </table>		Address	Data	Instruction	Address	Data	Instruction	5000	3E	MVI A,02H	5008	80	L1: ADD B	5001	02	5009	76	HLT	5002	06	MVI B,03H	500A	00		5003	03	500B	00		5004	C3	JMP L1	500C	00		5005	08	500D	00		5006	50	500E	00		5007	90	SUB B	500F	00
Address	Data	Instruction	Address	Data	Instruction																																														
5000	3E	MVI A,02H	5008	80	L1: ADD B																																														
5001	02		5009	76	HLT																																														
5002	06	MVI B,03H	500A	00																																															
5003	03		500B	00																																															
5004	C3	JMP L1	500C	00																																															
5005	08		500D	00																																															
5006	50		500E	00																																															
5007	90	SUB B	500F	00																																															

11.2 Code with explanation – Following is the code that fulfils the above objective with explanation –

Ins. No.	Mnemonics	Explanation
1	MVI A,02H	Dummy line of code
2	MVI B,03H	Dummy line of code
3	JMP L1	Unconditional Jump to L1 label (defined in line 5)
4	SUB B	This statement will be skipped due to jump
5	L1: ADD B	Defining L1 label
6	HLT	Halt/Stop the computer

In the above code let us first understand how to create a label and jump to it. The first two lines of code are not achieving any great purpose but they are just there to give code some length – they are dummy lines. In line number 3, unconditional jump statement JMP is used that wants to make a jump to label L1 (this L1 could be any name one desires to have). On execution of this line, the control will jump to line number 5 where this label is defined (notice the syntax of defining a label). This way line number 4 will be skipped. Any number of lines can be skipped as desired in a code.

Assuming that we started writing our code in memory from location 5000H we now need to understand what is label L1 (which could be any name the user chooses) as it is not a part of instruction set of 8085. Refer to the memory table where the code together with the number of bytes it consumes in the memory is shown. For an example, MVI A,02H consumes two bytes in memory and hence two memory locations 5000 H and 5001 H are used. We need to see what is the hexadecimal equivalent of L1 in JMP L1. It turns out that the hexadecimal equivalent of JMP is C3 and L1 is 5008 H (read in little endian format). So, a label is a 2-byte data. Interestingly at location 5008 H in memory, label L1 is defined. So, a label is just a nickname of address.

11.3 Results and Further directions – Note that even if the code were written as follows-

```
MVI A,02H
MVI B,03H
JMP 5008H
SUB B
ADD B
HLT
```

Assuming we start entering the code in memory from location 5000 H only, it would make no difference. But by using label the code was more radiable and we need not bother about the starting location of the code. We could have started from anywhere. But in the second code we were bound to start from 5000 H, only then, 5008 H would be the desired location of jump. Labels can be conditional too. They will be discussed in next program. Labels find multiple uses like loop creation, function definition etc. as discussed in coming programs.

Topic / Program 12. Learning to create loop using labels

12.1 Detailed objective and coding strategy– The detailed objective with the code defaults (i.e., where does the code start from in memory and where is the data kept etc.) is given below-

Objective- Loops are one of the most important parts of any programming language, they help the programmer to iterate through the same task with a smaller set of code. In our previous experiment (Experiment 13), We have already seen how to jump to a particular memory location using basic JMP instruction. We can also jump to other locations by using conditional Jumps which are only defined when specific conditions are	Code location		5000 H	
	Subroutine 1 location		NA	
	Subroutine 2 location		NA	
	Interrupt location		NA	
	Data used in memory – Shown for main code below			
	Address	Data	Address	Data
	5000 H	97	5008 H	C2
5001 H	06	5009 H	05	
5002 H	01	500A H	50	

met. Else they are ignored. Some commonly used conditional jump instructions are: <ol style="list-style-type: none"> 1. JC : Jump if carry flag is set (1) 2. JNC: Jump if carry flag is reset (0) 3. JZ : Jump if zero flag is set (1) 4. JNZ: Jump if zero flag is reset (0) With the help of these instructions, we can achieve looping technique by defining labels and jumping to them until a particular condition is met.	5003 H	0E	500B H	76
	5004 H	0A	500C H	00
	5005 H	80	500D H	00
	5006 H	04	500E H	00
	5007 H	0D	500F H	00
Once the condition is fulfilled, the loop terminates, allowing the program to proceed beyond the loop structure.				

12.2 Code with explanation – Following is the code that fulfils the above objective with explanation –

In the example below, we have developed a simple program to sum the first 10 positive integers from 1 to 10. To accomplish this task, we initialized our counter using register C by 0AH (decimal 10) and cleared the accumulator to 0 which will hold the result. Utilizing a label 'L1', we iterated through a loop until our counter (register C) reached 0.

Ins. No.	Mnemonics	Explanation
1	SUB A	Clearing register A by $A = A - A$
2	MVI B, 01H	Load 01H in register B (first number to be added to accumulator)
3	MVI C, 0AH	Load 0Ah value in register C (we will use this as counter for our loop – it will be decremented once every time the loop runs. The loop will terminate when C has 00H in it)
4	L1: ADD B	Added content of register B with A (successively)
5	INR B	Increment content of register B by 1
6	DCR C	Decrement content of register C by 1
7	JNZ L1	Jump to label L1, if zero flag is not set
8	HLT	Halt/Close the computer

12.3 Results and Further directions – The result of execution of above is 37H in accumulator which is equivalent to 55 in decimal. There can be loops inside loops (called nested loops). Using unconditional jump statement, infinite loop can be made if it is so desired.

This demonstration illustrates a foundational approach to implementing looping techniques in 8085 assembly programming. There are some other conditional jump instructions that can be used to create loops, such as-

JP : Jump if result of an operation is positive (Sign flag is reset (0))

JM : Jump if result of an operation is negative (Sign flag is set (1))

JPE: Jump if even parity

JPO: Jump if odd parity

Topic / Program 13. Learning to deal with input and output devices connected in IO mode to 8085 microprocessor.

13.1 **Detailed objective and coding strategy**– The detailed objective with the code defaults (i.e., where does the code start from in memory and where is the data kept etc.) is given below-

<p>Objective- In this article we will try to get an input from the input port 80H into the accumulator, increment that by one, and output it at port 81H.</p> <p>Remember that IO devices can be connected in two modes – 1) Memory mapped 2) IO mapped.</p> <p>In the first mode, IO devices are connected in the same address space where memory is connected. However in the later mode, IO devices are connected through data bus and thus a 2 hexadecimal digit address.</p>	Code location		5000 H	
	Subroutine 1 location		NA	
	Subroutine 2 location		NA	
	Interrupt location		NA	
	Data used in memory – Shown for main code below			
	Address	Data	Address	Data
	5000 H	DB	5008 H	00
	5001 H	80	5009 H	00
	5002 H	3C	500A H	00
	5003 H	D3	500B H	00
5004 H	81	500C H	00	
5005 H	76	500D H	00	
5006 H	00	500E H	00	
5007 H	00	500F H	00	

13.2 **Code with explanation** – Following is the code that fulfils the above objective with explanation –

There are only two commands to be used IN for taking input and OUT for giving output. The address of IO device will be of 2 hexadecimal digits and can range from 00H to FFH. Using these two is fairly straightforward as shown below.

Ins. No.	Mnemonics	Explanation
1	IN 80H	Input command – it inputs the data from IO device connected at port 80H into the accumulator register.
2	INR A	Increment the input data by one
3	OUT 81H	Output command – it outputs the data from accumulator register to the IO device connected at port 81H
4	HLT	Halts the system

13.3 Results and Further directions – The input device (which might be keyboard) gives 8085 microprocessor 1 byte data on port 80H. This data by the very nature of the command IN comes to accumulator register. Then it can be manipulated. In the program we have incremented it by one. The data then is outputted to port 81H. Remember 1 byte data from accumulator only will go to the designated output port (in our case 81H).

Topic / Program 14. Learning subroutines or functions

14.1 Detailed objective and coding strategy– The detailed objective with the code defaults (i.e., where does the code start from in memory and where is the data kept etc.) is given below-

<p>Objective- Learn how to create a simple subroutine.</p> <p>Subroutines in 8085 are the set of instructions that perform a particular task. They can be thought of as sub-programs (routines) other than the main program (routine). They can be ‘called’ whenever their help is necessary. They are a very powerful tool and can be called an arbitrary number of times as desired. Subroutines are generally implemented by Call (conditional and unconditional) instructions and terminated by Return (conditional and unconditional) instructions. The working of subroutine is: Whenever the call instruction is executed, the execution is transferred to the location of subroutine and the address of the next instruction to be executed in the main program moves to stack. When the return instruction is encountered the address on the stack goes to the program counter and the main routine is continued from next instruction.</p>	Code location	5000 H																																					
	Subroutine 1 location	5009 H																																					
	Subroutine 2 location	NA																																					
	Interrupt location	NA																																					
	<p>Data used in memory – (The location of main code and subroutine are both shown)</p> <table border="1"> <thead> <tr> <th>Address</th> <th>Data</th> <th>Address</th> <th>Data</th> </tr> </thead> <tbody> <tr> <td>5000 H</td> <td>3E</td> <td>5008 H</td> <td>76</td> </tr> <tr> <td>5001 H</td> <td>15</td> <td>5009 H</td> <td>80</td> </tr> <tr> <td>5002 H</td> <td>06</td> <td>500A H</td> <td>C9</td> </tr> <tr> <td>5003 H</td> <td>16</td> <td>500B H</td> <td>00</td> </tr> <tr> <td>5004 H</td> <td>CD</td> <td>500C H</td> <td>00</td> </tr> <tr> <td>5005 H</td> <td>09</td> <td>500D H</td> <td>00</td> </tr> <tr> <td>5006 H</td> <td>50</td> <td>500E H</td> <td>00</td> </tr> <tr> <td>5007 H</td> <td>27</td> <td>500F H</td> <td>00</td> </tr> </tbody> </table>				Address	Data	Address	Data	5000 H	3E	5008 H	76	5001 H	15	5009 H	80	5002 H	06	500A H	C9	5003 H	16	500B H	00	5004 H	CD	500C H	00	5005 H	09	500D H	00	5006 H	50	500E H	00	5007 H	27	500F H
Address	Data	Address	Data																																				
5000 H	3E	5008 H	76																																				
5001 H	15	5009 H	80																																				
5002 H	06	500A H	C9																																				
5003 H	16	500B H	00																																				
5004 H	CD	500C H	00																																				
5005 H	09	500D H	00																																				
5006 H	50	500E H	00																																				
5007 H	27	500F H	00																																				
<p>It is worth mentioning that labels can be used to implement subroutine in the simulators but on the actual kit, actual addresses are to be used.</p>																																							

14.2 Code with explanation – Following is the code that fulfils the above objective with explanation –

In the following program, addition of two numbers kept in registers A and B initially is done using subroutine named ADDN. ADDN is defined in line number 6 terminated by a return (RET) instruction. Also, the subroutine definition begins after HLT i.e., outside the main code.

Ins. No.	Mnemonics	Explanation	
1	MVI A, 15H	Move value 15 to A (Accumulator).	
2	MVI B, 16H	Move value 16 to B.	
3	CALL ADDN	Call subroutine ADDN	
4	DAA	Decimal adjust the result in A	
5	HLT	Halt the program.	
6	ADDN: ADD B	Function definition	Add B to A
7	RET		Return to main routine here line 4.

14.3 Results and Further directions – This program highlight the basic implementation of subroutine. There may be as many subroutines as one desires to make. Any subroutine can be called from main routine or any subroutine any number of times as desired.

Readers to note that there exists conditional call statements like CC (Call if Carry), CZ (Call if Zero), CPE (Call if Parity Even). These instructions use the status of flag register for execution. Similarly, there exists conditional return statements like RC (Return if Carry), RZ (Return if Zero), RPE (Return if Parity Even). Readers are advised to explore further about them.

Topic / Program 15. Learning software interrupts

15.1 Detailed objective and coding strategy– The detailed objective with the code defaults (i.e., where does the code start from in memory and where is the data kept etc.) is given below-

<p>Objective- In 8085 microprocessor, Interrupts are of great importance, they allow the programmer to interrupt the normal processing of the microprocessor and execute any other subroutine.</p> <p>When an interrupt occurs, the 8085 completes the current instruction and then acknowledges the interrupt. It saves the address of the next instruction to be executed on the stack and fetches the interrupt service routine (ISR) address from the predefined locations based on the type of interrupt. The ISR executes, and upon completion, the processor resumes the main program.</p> <p>There are two types of Interrupts, <i>Software Interrupts</i> and <i>Hardware Interrupts</i> (Discussed in next experiment).</p> <p>There are 8 Software interrupts, ranging from RST 0 to RST 7, here RST (Restart) prefix is used to indicate Interrupt. All these Software</p>	Code location	1000 H		
	Subroutine 1 location	NA		
	Subroutine 2 location	NA		
	Interrupt location	0028 H		
	Data used in memory – (Shown for data)			
	Address	Data	Address	Data
	5000 H	01	5008 H	00
	5001 H	09	5009 H	00
	5002 H	07	500A H	00
	5003 H	?	500B H	00
5004 H	00	500C H	00	
5005 H	00	500D H	00	
5006 H	00	500E H	00	
5007 H	00	500F H	00	
All these software Interrupts are maskable (can be ignored, when required), and can be easily masked or unmasked by using instructions:				

<p>Interrupts are <i>Vectored</i> (have fixed address) and each Interrupt is of 8 byte. The address of these interrupts ranges from 0000H to 0038H and can be calculated by multiplying the interrupt number with 8 and converting it to hexadecimal representation. For an example, RST 5 has its vectored address as $(5 \times 8 = 40)_{10} = (0028)_{16} = 0028 H$.</p>	<ol style="list-style-type: none"> 1. DI (Disable Interrupts) 2. EI (Enable Interrupts) <p><i>To better understand software interrupts, we will develop a simple program that will check the data at memory location 5000, If it is (0) then it will perform addition of next two memory bytes (i.e 5001H and 5002H) and if it is (1) then it will perform subtraction of the next two memory locations. The subtraction will be performed using RST 5 software interrupt and the result will be stored in memory location 5003H.</i></p>
---	---

15.2 Code with explanation – Following is the code that fulfils the above objective with explanation –

Ins. No.	Mnemonics	Explanation	
1	#BEGIN 1000H	(Assembler Directive – More about this in the relevant experiment), to tell where to start executing the code in memory from. HLT will automatically terminate it.	
2	#ORG 1000H	(Assembler Directive), to tell where to write the next lines of code in memory	
3	EI	Enable all interrupts	
4	LXI H, 5002H	Load HL pair with the address of second number	
5	MOV B, M	Copy this number to register B	
6	DCR L	Decrement HL pair to 5001H	
7	MOV C, M	Copy another number to register C	
8	DCR L	Decrement HL pair to 5000H	
9	MOV A, M	Copy the data of 5000H location to register A	
10	CPI 00H	Compare it with 00H	
11	JZ ADDITION	If zero flag is set (i.e 5000H contains 0), then jump to label “ADDITION”.	
12	RST 5	Call software interrupt 5 (RST 5)	
13	JMP CONCLUDE	Jump to “CONCLUDE” label	
14	ADDITION: MOV A, C	Copy First Number from register C to A	ADDITION
	ADD B	Add content of register B with A ($A = A+B$)	
16	CONCLUDE: STA 5003H	Store the result to memory location 5003H	CONCLUDE
		HALT THE PROGRAM	

	HLT		
17	INTERRUPT SERVICE ROUTING (ISR)		
18	#ORG 0028H	(Assembler Directive), Initialize to write the code from memory location 0028H	
19	MOV A, C	Copy the content of register C to A	RST 5
20	SUB B	Subtract content of register B from A (A = A - B)	
21	RET	Return to the main program	

15.3 Results and Further directions - In the above code, we learnt the basic initialization and calling of a software interrupt. We used a RET instruction at the end of our subroutine to signal our microprocessor that the subroutine is executed and now it can again go back to the main program.

Please note:- Between two consecutive vectored locations of interrupts, only 8 byte of memory locations are available. For example, between RST 5 and RST 6 whose locations are 0028H and 0030H, only 8 bytes are available. Hence ISR should be 8 bytes or less. If it is required that ISR be of more than 8 bytes, following simple trick will work – At vectored location of ISR, use the unconditional jump (JMP) instruction by choosing the address of jump as the address of ISR's definition.

Topic / Program 16. Learning Hardware Interrupts

16.1 Detailed objective and coding strategy– The detailed objective with the code defaults (i.e., where does the code start from in memory and where is the data kept etc.) is given below-

<p>Objective- We have already discussed the interrupts and software interrupts in the previous Experiment. Apart from the software interrupts discussed, 8085 also has 5 hardware interrupts. They are TRAP, RST 7.5, RST 6.5, RST 5.5, INTR. TRAP is often referred to as RST 4.5.</p> <p>All these interrupts are vectored interrupts (except INTR) and have defined addresses. The order of priority and the address is:</p>	Code location	1000 H
	Subroutine 1 location	NA
	Subroutine 2 location	NA
	Interrupt location	002C H
	<p>This program creates an infinite loop. It starts by loading the value 05H into register B, then enters LOOP2 where it continuously decrements B. Once B reaches zero, the program jumps back to LOOP, resetting B to 05H, creating an endless cycle.</p> <p>Because this is an infinite loop, the program never reaches the HLT (halt) instruction. The only way to stop it is by using a hardware</p>	
Interrupt	Address	Priority Order
TRAP	0024 H	1
RST 7.5	003C H	2
RST 6.5	0034 H	3

RST 5.5	002C H	4	interrupt. When the RST 5.5 interrupt is triggered, the interrupt subroutine is activated, redirecting execution to memory location 002CH. Here, the value of B is moved to the Accumulator, and then the program jumps to LOOP3, where the HLT instruction finally halts the program
INTR	-	5	

It is very important to write the command EI to enable interrupts. By default, all these interrupts are disabled except TRAP.

16.2 Code with explanation – Following is the code that fulfils the above objective with explanation –

Ins. No.	Mnemonics	Explanation	
1	#BEGIN 1000H	Location of the main program	
1	#ORG 1000H	Starting writing the code from memory location 1000H	
2	EI	Enable the maskable interrupts	
3	MVI A,06H	Move a value 06 to A (Accumulator).	
4	LOOP: MVI B,05H	LOOP	Move a value 05 to B
5	LOOP2: DCR B	LOOP2	Decrease the value in B by 1.
6	JNZ LOOP2	Jump to LOOP2 if zero flag reset.	
7	JZ LOOP	Jump to LOOP if zero flag set.	
8	LOOP3: HLT	LOOP3	Halt the program
9	#ORG 002CH	RST 5.5	Writing ISR from the Location of RST 5.5
10	MOV A, B		Move the contents of B to A
11	JMP LOOP3		Jump to LOOP3

16.3 Results and Further directions - Since INTR has no predefined memory location it depends on an external device for the address. Hardware interrupts are useful in error handling, safety incidents, peripheral communication etc.

Topic / Program 17. Learning Assembler Directives

17.1 Detailed objective and coding strategy– The detailed objective with the code defaults (i.e., where does the code start from in memory and where is the data kept etc.) is given below-

Objective- In this experiment, we will learn the usage of some assembler directives used in 8085 assembly programming. Assembler Directives are some special commands that make the assembly	Code location	1000 H
	Subroutine 1 location	NA
	Subroutine 2 location	NA
	Interrupt location	NA
	1. ORG- (Origin) It specifies the memory	

<p>programming more structured and understandable. These instructions typically begins with a period (.) or a hash (#) to distinguish with the normal 8085 instructions.</p> <p>Assembler Directives are not directly executed by the CPU but are interpreted by the assembler during the assembly process. This provides a way to the programmer to communicate with the assembler and specify how the code should be translated or stored before execution.</p> <p>Here are some of the basic 8085 assembler directives:-</p>	<p>address from where the code should be stored.</p> <ol style="list-style-type: none"> 2. DB- (Define Byte) It reserves a Byte of memory space for storing any variable or data. 3. DW- (Define Word) Similar to DB, it reserves two Byte or a single Word of memory space for storing any variable or data 4. EQU- (Equation) It is used to define a constant value in the memory. 5. END- (End of program) It marks the end of program. <p>We are going to develop a simple 8085 program to add two bytes of data ,and will use assembler directives to structure this program</p>
---	---

17.2 Code with explanation – Following is the code that fulfils the above objective with explanation –

Ins. No.	Mnemonics	Explanation
1	#ORG 2000H	Here, we are defining our code segment from 2000H to store the data from this location onwards.
2	#DB 28H	Store our first number (say 28H) in memory location 2000H
3	#DB 10H	Store second number (say 10H) in memory location 2001H
4	#ORG 1000H	Define the address (1000H) from where our code will be stored in the memory.
5	LXI H,2000H	Load the address 2000H in register pair HL, so that we can use register M to get the value at that memory location
6	MOV A,M	Copy our First number from register M to accumulator
7	INX H	Increment register pair HL by 1 to get address of second number
8	ADD M	Add register M (Second number) with accumulator (First number) (A = A+M)
9	#END	End the program.
10	# BEGIN 1000H	Specify where from the first line of code is to be executed.

17.3 Results and Further directions - In this program, we used ORG directive to specify the memory address from where we want to store our code, we then used DB directive to store our numbers (28H and 10H) at that particular memory location. From line number 4 onwards, we then store our code from memory location 1000H, here we sculpt a basic program to add two numbers stored at memory location 2000H and 2001H. On execution of the above program, we get the result 38H, which is correct and therefore we successfully performed our first assembly code using directives. The above program may appear to be very simple, and its execution is possible even without using assembler directives. However, incorporating these directives elevates the flexibility and readability of the program which makes this very useful in programming complex assembly codes. Line no. 10 plays extremely important role in deciding the location of 1st executable line of code. This line can be written anywhere in the code as before execution assembler directives do their work.

Topic / Program 18. Instruction set of 8085 microprocessor

8085 microprocessor has a total of 74 instructions from which 246 valid patterns can be made. These are as follows –

S.No.	Mnemonics and Operands	Opcode(HEX)	OPCODE(DEC)	Bytes
1	ACI Data	CE	206	2
2	ADC A	8F	143	1
3	ADC B	88	136	1
4	ADC C	89	137	1
5	ADC D	8A	138	1
6	ADC E	8B	139	1
7	ADC H	8C	140	1
8	ADC L	8D	141	1
9	ADC M	8E	142	1
10	ADD A	87	135	1
11	ADD B	80	128	1
12	ADD C	81	129	1
13	ADD D	82	130	1
14	ADD E	83	131	1
15	ADD H	84	132	1
16	ADD L	85	133	1
17	ADD M	86	134	1
18	ADI Data	C6	198	2
19	ANA A	A7	167	1
20	ANA B	A0	160	1

21	ANA	C	A1	161	1
22	ANA	D	A2	162	1
23	ANA	E	A3	163	1
24	ANA	H	A4	164	1
25	ANA	L	A5	165	1
26	ANA	M	A6	166	1
27	ANI	Data	E6	230	2
28	CALL	Label	CD	205	3
29	CC	Label	DC	220	3
30	CM	Label	FC	252	3
31	CMA		2F	47	1
32	CMC		3F	63	1
33	CMP	A	BF	191	1
34	CMP	B	B8	184	1
35	CMP	C	B9	185	1
36	CMP	D	BA	186	1
37	CMP	E	BB	187	1
38	CMP	H	BC	188	1
39	CMP	L	BD	189	1
40	CMP	M	BD	189	1
41	CNC	Label	D4	212	3
42	CNZ	Label	C4	196	3
43	CP	Label	F4	244	3
44	CPE	Label	EC	236	3
45	CPI	Data	FE	254	2
46	CPO	Label	E4	228	3
47	CZ	Label	CC	204	3
48	DAA		27	39	1
49	DAD	B	9	9	1
50	DAD	D	19	25	1
51	DAD	H	29	41	1
52	DAD	SP	39	57	1
53	DCR	A	3D	61	1
54	DCR	B	5	5	1
55	DCR	C	0D	13	1
56	DCR	D	15	21	1
57	DCR	E	1D	29	1
58	DCR	H	25	37	1
59	DCR	L	2D	45	1
60	DCR	M	35	53	1
61	DCX	B	0B	11	1

62	DCX	D	1B	27	1
63	DCX	H	2B	43	1
64	DCX	SP	3B	59	1
65	DI		F3	243	1
66	EI		FB	251	1
67	HLT		76	118	1
68	IN	Port-address	DB	219	2
69	INR	A	3C	60	1
70	INR	B	4	4	1
71	INR	C	0C	12	1
72	INR	D	14	20	1
73	INR	E	1C	28	1
74	INR	H	24	36	1
75	INR	L	2C	44	1
76	INR	M	34	52	1
77	INX	B	3	3	1
78	INX	D	13	19	1
79	INX	H	23	35	1
80	INX	SP	33	51	1
81	JC	Label	DA	218	3
82	JM	Label	FA	250	3
83	JMP	Label	C3	195	3
84	JNC	Label	D2	210	3
85	JNZ	Label	C2	194	3
86	JP	Label	F2	242	3
87	JPE	Label	EA	234	3
88	JPO	Label	E2	226	3
89	JZ	Label	CA	202	3
90	LDA	Address	3A	58	3
91	LDAX	B	0A	10	1
92	LDAX	D	1A	26	1
93	LHLD	Address	2A	42	3
94	LXI	B	1	1	3
95	LXI	D	11	17	3
96	LXI	H	21	33	3
97	LXI	SP	31	49	3
98	MOV	A,A	7F	127	1
99	MOV	A,B	78	120	1
100	MOV	A,C	79	121	1
101	MOV	A,D	7A	122	1
102	MOV	A,E	7B	123	1

103	MOV	A,H	7C	124	1
104	MOV	A,L	7D	125	1
105	MOV	A,M	7E	126	1
106	MOV	B,A	47	71	1
107	MOV	B,B	40	64	1
108	MOV	B,C	41	65	1
109	MOV	B,D	42	66	1
110	MOV	B,E	43	67	1
111	MOV	B,H	44	68	1
112	MOV	B,L	45	69	1
113	MOV	B,M	46	70	1
114	MOV	C,A	4F	79	1
115	MOV	C,B	48	72	1
116	MOV	C,C	49	73	1
117	MOV	C,D	4A	74	1
118	MOV	C,E	4B	75	1
119	MOV	C,H	4C	76	1
120	MOV	C,L	4D	77	1
121	MOV	C,M	4E	78	1
122	MOV	D,A	57	87	1
123	MOV	D,B	50	80	1
124	MOV	D,C	51	81	1
125	MOV	D,D	52	82	1
126	MOV	D,E	53	83	1
127	MOV	D,H	54	84	1
128	MOV	D,L	55	85	1
129	MOV	D,M	56	86	1
130	MOV	E,A	5F	95	1
131	MOV	E,B	58	88	1
132	MOV	E,C	59	89	1
133	MOV	E,D	5A	90	1
134	MOV	E,E	5B	91	1
135	MOV	E,H	5C	92	1
136	MOV	E,L	5D	93	1
137	MOV	E,M	5E	94	1
138	MOV	H,A	67	103	1
139	MOV	H,B	60	96	1
140	MOV	H,C	61	97	1
141	MOV	H,D	62	98	1
142	MOV	H,E	63	99	1
143	MOV	H,H	64	100	1

144	MOV	H,L	65	101	1
145	MOV	H,M	66	102	1
146	MOV	L,A	6F	111	1
147	MOV	L,B	68	104	1
148	MOV	L,C	69	105	1
149	MOV	L,D	6A	106	1
150	MOV	L,E	6B	107	1
151	MOV	L,H	6C	108	1
152	MOV	L,L	6D	109	1
153	MOV	L,M	6E	110	1
154	MOV	M,A	77	119	1
155	MOV	M,B	70	112	1
156	MOV	M,C	71	113	1
157	MOV	M,D	72	114	1
158	MOV	M,E	73	115	1
159	MOV	M,H	74	116	1
160	MOV	M,L	75	117	1
161	MVI	A,Data	3E	62	2
162	MVI	B,Data	6	6	2
163	MVI	C,Data	0E	14	2
164	MVI	D,Data	16	22	2
165	MVI	E,Data	1E	30	2
166	MVI	H,Data	26	38	2
167	MVI	L,Data	2E	46	2
168	MVI	M,Data	36	54	2
169	NOP		0	0	1
170	ORA	A	B7	183	1
171	ORA	B	B0	176	1
172	ORA	C	B1	177	1
173	ORA	D	B2	178	1
174	ORA	E	B3	179	1
175	ORA	H	B4	180	1
176	ORA	L	B5	181	1
177	ORA	M	B6	182	1
178	ORI	Data	F6	246	2
179	OUT	Port-Address	D3	211	2
180	PCHL		E9	233	1
181	POP	B	C1	193	1
182	POP	D	D1	209	1
183	POP	H	E1	225	1
184	POP	PSW	F1	241	1

185	PUSH	B	C5	197	1
186	PUSH	D	D5	213	1
187	PUSH	H	E5	229	1
188	PUSH	PSW	F5	245	1
189	RAL		17	23	1
190	RAR		1F	31	1
191	RC		D8	216	1
192	RET		C9	201	1
193	RIM		20	32	1
194	RLC		7	7	1
195	RM		F8	248	1
196	RNC		D0	208	1
197	RNZ		C0	192	1
198	RP		F0	240	1
199	RPE		E8	232	1
200	RPO		E0	224	1
201	RRC		0F	15	1
202	RST	0	C7	199	1
203	RST	1	CF	207	1
204	RST	2	D7	215	1
205	RST	3	DF	223	1
206	RST	4	E7	231	1
207	RST	5	EF	239	1
208	RST	6	F7	247	1
209	RST	7	FF	255	1
210	RZ		C8	200	1
211	SBB	A	9F	159	1
212	SBB	B	98	152	1
213	SBB	C	99	153	1
214	SBB	D	9A	154	1
215	SBB	E	9B	155	1
216	SBB	H	9C	156	1
217	SBB	L	9D	157	1
218	SBB	M	9E	158	1
219	SBI	Data	DE	222	2
220	SHLD	Address	22	34	3
221	SIM		30	48	1
222	SPHL		F9	249	1
223	STA	Address	32	50	3
224	STAX	B	2	2	1
225	STAX	D	12	18	1

226	STC		37	55	1
227	SUB	A	97	151	1
228	SUB	B	90	144	1
229	SUB	C	91	145	1
230	SUB	D	92	146	1
231	SUB	E	93	147	1
232	SUB	H	94	148	1
233	SUB	L	95	149	1
234	SUB	M	96	150	1
235	SUI	Data	D6	214	2
236	XCHG		EB	235	1
237	XRA	A	AF	175	1
238	XRA	B	A8	168	1
239	XRA	C	A9	169	1
240	XRA	D	AA	170	1
241	XRA	E	AB	171	1
242	XRA	H	AC	172	1
243	XRA	L	AD	173	1
244	XRA	M	AE	174	1
245	XRI	Data	EE	238	2
246	XTHL		E3	227	1

Dr. Manish Kashyap

Dr. Manish Kashyap is Assistant Professor at Department of Electronics and Communication Engineering, Maulana Azad National Institute of Technology Bhopal, India. His research areas include Digital image Processing, Embedded Systems, Artificial intelligence and Machine learning. He has authored good number of research papers in Scientific Citation Indexed and Scopus indexed journals and conferences. He is also the author of book titled – “Digital Image Processing using Python”.

His updated research profile can be accessed at –

<https://orcid.org/0000-0001-6951-8447>

Official Website –

<https://www.manit.ac.in/content/dr-manish-kashyap>

Tushar Kukreti

Tushar Kukreti is an Undergraduate Student at Department of Electronics and Communication Engineering, Maulana Azad National Institute of Technology, Bhopal, India. He has a keen interest in microprocessors and embedded systems and enjoys learning about hardware and system design.

Vivek Kumar

Vivek Kumar is an Undergraduate Student at Department of Electronics and Communication Engineering, Maulana Azad National Institute of Technology, Bhopal, India. He finds C++, assembly language and microprocessors interesting. He has a passion for understanding system design and low-level programming.

ISBN: 978-93-90728-96-1

DOI: 10.5281/zenodo.14885842



Vandana Publications

Editorial Office : UG-4, Avadh Tower, Naval Kishor Road, Opp.
Kaysons Lane, Hazratganj, Lucknow-226001,
Uttar Pradesh, INDIA.
Contact Numbers : 0522-4108552, +91-9696045327
Email ID's : info@vandanapublications.com |
mail2vandanapublications@gmail.com
Visit us : www.vandanapublications.com

