LABORATORY MANUAL

Microprocessor and Microcontroller

ELECTRONICS & COMMUNICATION ENGINEERING

COLLEGE OF ENGINEERING AND MANAGEMENT, KOLAGHAT

Revised December 2024

Vision

Pursuing Excellence in Teaching-Learning Process to Produce High-Quality Electronics and Communication Engineering Professionals.

Mission

To enhance the employability of our students by strengthening their creativity with different innovative ideas by imparting high-quality technical and professional education with continuous performance improvement monitoring systems.

To carry out research through constant interaction with research organizations and industry.

Program Outcomes (POs)

1	**Engineering knowledge:** Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
2	**Problem analysis:** Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using the first principles of mathematics, natural sciences, and engineering sciences
3	**Design/development of solutions:** Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for public health and safety, and cultural, societal, and environmental considerations.
4	**Conduct investigations of complex problems:** Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
5	**Modern tool usage:** Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
6	**The engineer and society:** Apply reason informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
7	**Environment and sustainability:** Understand the impact of professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
8	**Ethics:** Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice
9	**Individual and team work:** Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
10	**Communication:** Communicate effectively on complex engineering activities with the engineering community and with society at large, such as being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
11	**Project management and finance:** Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
12	**Life-long learning:** Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

Program Specific Outcomes (PSOs)

PSO-1	An ability to design and conduct the experiments, analyse and interpret the data using modern software or hardware tools with proper understanding (basic conceptions) of Electronics and Communication Engineering.
PSO-2	Ability to identify, formulate & solve problems and apply the knowledge of electronics and communication to develop techno-commercial applications

Course Outcomes (COs)

CO-1	The students will be able to acquire the knowledge of internal architecture of 8085 microprocessor and the skills in Assembly Language Programming of 8085 microprocessor.
CO-2	The students will acquire the knowledge of internal architecture of 8051 microcontroller and also develop their skills in Assembly Language Programming of 8051 microcontroller.
CO-3	The student will be able to gather knowledge of interfacing 8085 microprocessor and 8051 microcontroller with various hardware devices along with the software interaction and integration.
CO-4	The students will be able to apply the concepts in the design of microprocessor/ microcontroller based systems in real time applications.

CO – PO Mapping

	PO-1	**PO-2**	**PO-3**	**PO-4**	**PO-5**	**PO-6**	**PO-7**	**PO-8**	**PO-9**	**PO-10**	**PO-11**	**PO-12**
CO-1	3	3	3	3	3	0	0	0	0	2	2	2
CO-2	3	2	3	3	2	0	0	0	0	2	2	2
CO-3	3	3	3	3	3	1	0	0	3	2	3	3
CO-4	3	3	3	3	3	1	0	0	3	2	3	3

CO – PSO Mapping

	PSO-1	**PSO-2**
CO-1	3	3
CO-2	3	3
CO-3	3	3
CO-4	3	3

LIST OF CONTENTS

Contents of 8085 Microprocessor

Contents of 8085 Microprocessor

Contents of 8051 Microcontroller

Contents of 8051 Microcontroller

Appendix		
A.	Index Sheet	**A1**
B.	Coding Sheet	**B1**
C.	Sample Lab Assessment Sheet	**C1**

SYLLABUS

1) Familiarization with 8085 & 8051simulator on PC.
2) Study of prewritten programs using basic instruction set (data transfer, Load/ Store, Arithmetic, Logical) on the KIT. Assignment based on above.
3) Programming using kit and simulator for:
   i. Table look up
   ii. Copying a block of memory
   iii. Shifting a block of memory
   iv. Packing and unpacking of BCD numbers
   v. Addition of BCD numbers
   vi. Binary to ASCII conversion
   vii. String Matching, Multiplication using shift and add method and Booth's Algorithm.
4) Program using subroutine calls and IN/OUT instructions using 8255 PPI on the trainer kit e.g. subroutine for delay, reading switch state and glowing LEDs accordingly.
5) Study of timing diagram of an instruction on oscilloscope.
6) Interfacing of 8255: Keyboard and Multi-digit Display with multiplexing using 8255
7) Study of 8051 Micro controller kit and writing programs as mentioned in S/L3. Write programs to interface of Keyboard, DAC and ADC using the kit.
8) Serial communication between two trainer kits.

1. Familiarization with 8085 simulator

A simulator is an application that mimics the environment and the operation of a practical system, providing the result without any test on that system. The advantages and disadvantages of a simulator are given below.

Less Financial Risk: Simulation is less expensive than real life experimentation. The potential costs of testing theories of real world systems can include expenditure of wastage of materials, cost of replacement of non-functioning parts etc. Due to this reason simulation allows you to test theories and avoid costly mistakes in real life.

Exact Repeated Testing: A simulation allows you to test different theories and innovations time after time against the exact same circumstances. This means you can thoroughly test and compare different ideas without deviation.

Examine Long-Term Impacts: A simulation can be created to let you see into the future by accurately modeling the impact of years of use in just a few seconds. This lets you see both short and long-term impacts so you can confidently make informed investment decisions which can provide benefits in the future.

Assess Random Events: A simulation can also be used to assess random events such as an unexpected events.

Test Non-Standard Distributions: A simulation can take account of changing and non-standard distributions, rather than having to repeat only set parameters. By taking such changing parameters into account, a simulation can more accurately mimic the real world.

Security and safety: Simulation also provides security and safety to a novice user in a complex system.

Limitations: Most of the time simulation can not provide accurate result of a large and complicated system. After testing on simulator when it is experimented on the actual physical system, there are some deviation or error compared to the real time results of the system. Due to this reason, simulations have limitations when it is realized in real-world situations.

8085 simulator is used to simulate mainly the assembly language programs on a PC and also makes it easy to test the same program already executed in simulator on the 8085 Kit. In this laboratory ***Jubin's 8085 Simulator*** is used to test the programs of 8085 microprocessor. It is a Java based application which comes with .jar file extension. This software is compatible for all operating systems like Windows, Linux and Mac. For any operating system Java Run-time Environment (JRE) must be installed to run this simulator. After installing JRE into the system, the application can be executed only by double-clicking on it. Therefore this simulator is not required to be installed in the system.

1.1 Installation of Jubin's 8085 simulator

Installation on Windows:
Step1 – Download Java greater than version 6.0 for Windows.
Step2 – Install Java in Windows.
Step3 – Copy the Jubin's 8085 Simulator with .jar file extension.
Step4 – Double click on .jar file to open and run the simulator.
Installation on Ubuntu:
Step1 – Install Java using the following commands:

$ sudo apt install default-jre
$ sudo apt install default-jdk

Step2 – Copy the Jubin's 8085 Simulator with .jar file extension into a directory.
Step3 – Double click on .jar file to open and run the simulator.

1.2 How to Use: The screenshots of Jubin's 8085 Simulator are shown in Fig-1.1, Fig-1.2 and Fig-1.3 respectively.

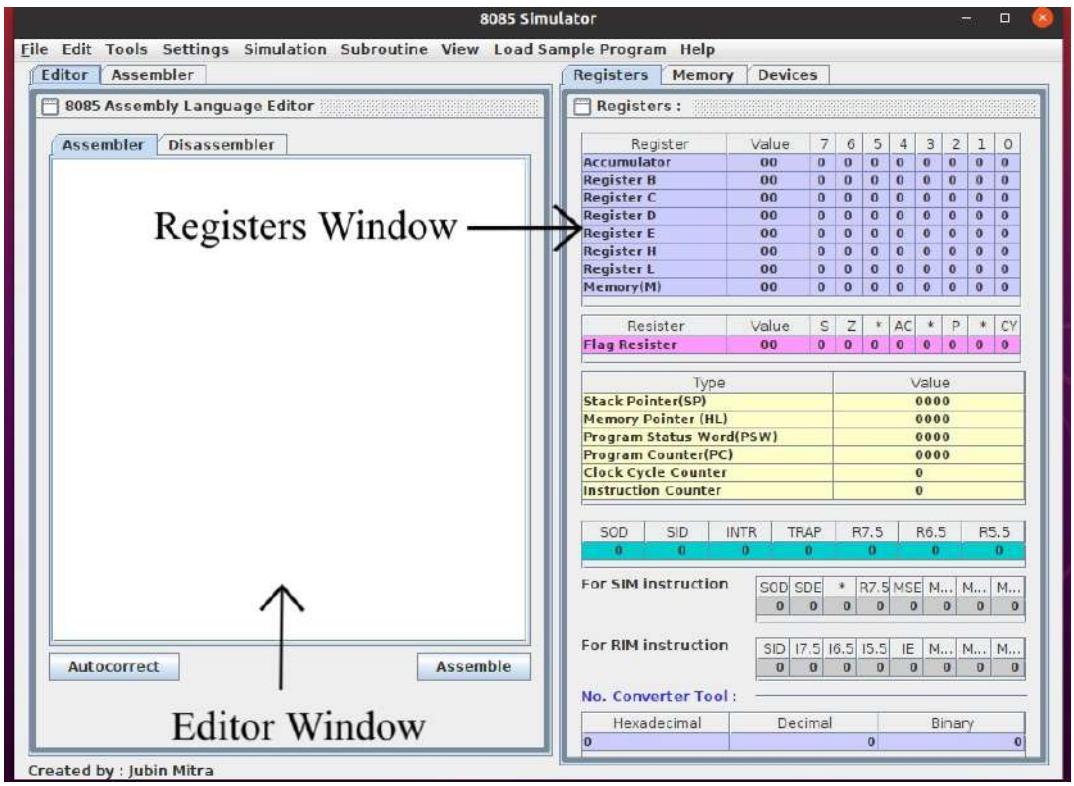


Fig-1.1: Editor and Registers Windows of Jubin' 8085 Simulator

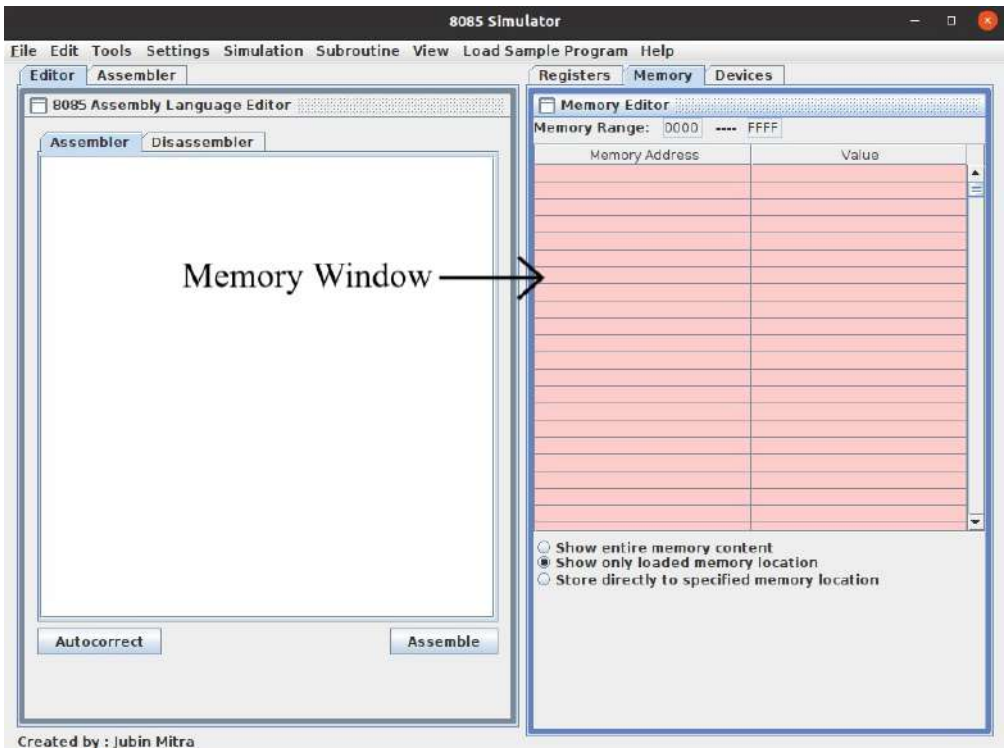


Fig-1.2: Memory Window of Jubin' 8085 Simulator

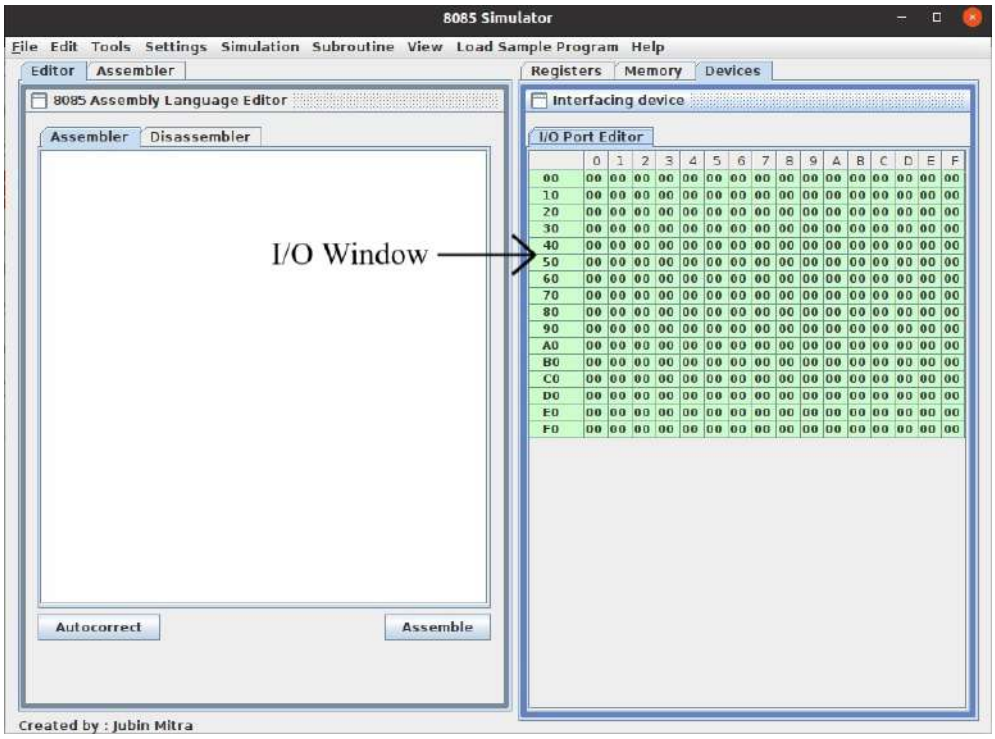


Fig-1.3: I/O Window of Jubin' 8085 Simulator

In the white space of Editor window the assembly language program of 8085 microprocessor is written. The status of various Registers is displayed in Registers Window at the right side of the Editor Window. The contents of the memory locations with their addresses are shown in the Memory Window and the status of I/O devices with their 8-bit addresses (IO mapped IO) is reflected on the I/O Window of the simulator.

1.3 Procedure to write a program:

- Every program in Jubin's 8085 simulator should be started with the assembler directive "#ORG" followed by the starting address of the program. In our laboratory it is better to start the program from any address beyond 8000H, because in 8085 Kit does not support to load a program below 8000H as this portion of memory is reserved for BIOS programming in the 8085 Kit. For example - #ORG 8000H will load the 8085 program starting from memory location 8000H.

  #ORG Starting Address of Program

- The next line after "#ORG" should include "#BEGIN" followed by the same starting address mentioned after "#ORG". This directive compiles the program from the address mentioned after "#BEGIN". Normally we should compile the programming code from the starting address, since the addresses after "#ORG" and "#BEGIN" should be same in case of this simulator.

  #BEGIN Starting Address of Program

- Programming code should be placed after the #BEGIN and should be terminated by the instruction HLT.

- ***Comment inside program:*** To give a comment inside the program, the comment line must be preceded by '//'. The simulator excludes these comment lines during debug. Giving comment in the program is a good practice to specify explanation of the program. This practice helps the programmer to recapitulate the logic of a big complicated program in future.

- ***Storing Data inside Memory:*** To store the data inside the memory prior to the execution of the program the following assembler directives should be used.

  #ORG Address of the Memory from where
  data can be stored consecutively
  #DB Data1, Data2, Data3, Data4, ……..

  DB (Data Byte) is a directive which stores all the 8-bit data (Data1, Data2, Data3,…..) consecutively starting from the address specified by "#ORG". This simulator can not store 16-bit data at a time in the two successive memory addresses by using any assembler directive.

  #ORG 8050
  #DB 02H, FFH, CDH, 32H, DDH

In the above example the simulator will load five 8-bit data (02H, FFH, CDH, 32H, DDH) consecutively starting from memory location 8050H.

The above mentioned task can be done also directly using the Memory Window of the simulator as shown in Fig-1.2.
Step1 - Open the Memory Window by clicking the Memory Tab.
Step2 - Select the option "Store directly to specified memory location".
Step3 – Click on the cell under *Memory Address* and enter the 16-bit address to store data.
Step4 – Press the Tab button in the Keyboard which will select the cell under *Value.*
Step5 – Enter 8-bit data and press Tab button once again. This will select the next memory address.
Step6 – Repeat from Step3 to store multiple data in consecutive addresses.

- As the option "Show only loaded memory location" are selected by default, this simulator only shows the content of the memory addresses which are used to store program code and data of the program input and output. It does not show entire memory locations. To show this select the option "Show entire memory content".

- ***Storing Data in I/O Address:*** The range of I/O addresses under the Tab "Devices" is 00H – F0H for this software as shown in Fig-1.3. Hence any I/O location can be accessed either Input or Output data. When a particular location is used as input, the 8-bit data should be placed into that location by left-clicking on it and the data sent as output will be reflected on the particular output location automatically.

- The content of any register can not be altered directly in this simulator. The change of the contents of registered can be viewed only in this simulator.

- ***Syntax Error Checking:*** After completion of the program writing "Autocorrect" button below the Editor Window should be clicked to align the program properly and to check any syntax error in the program code.

- ***Assemble Program:*** Now click the "Assemble" button which shows a green window where memory addresses of the program code, Label name, mnemonics, Hex codes (Opcodes and Operands), Instruction length, no. of Machine cycles and no. of T-states are presented in a tabular format. This view gives every detail of the program code.

Note: *Jubin's 8085 Simulator is not case-sensitive i.e. the program code may be written either in block letters or in small letters.*

1.4 Procedure to save/ load a program:

- After writing an assembly language program it should be saved by clicking the option "Save Assembly Language Code" under File menu. During saving the file must be saved with a file extension .asm.

- An existing program saved in an asm file can be loaded in the simulator using the option "Load Assembly Language Code" under File menu.

1.5 Procedure to execute a program:

- ***Run Entire Program:*** Click the button "Run all at a time" at the bottom of the Assembler window to execute the whole program at a time. After the execution of the program the desired output can be checked in the Registers under Registers Window or in the memory locations under the Memory Window or in the I/O locations under the Devices window as per the program.

- ***Run Step by Step:*** If the button "Step By Step" is clicked, only one instruction with every click will be executed. This mode of operation is called Single Line Execution. After execution of an instruction the register or memory location or I/O location will be updated as per the operation of that particular instruction. This option is very useful to detect any logical error inside the program.

The screenshots of Jubin's 8085 Simulator with a program written in Editor Window is shown in Fig-1.4 followed by the same program displayed on Assembler Window in Fig-1.5.

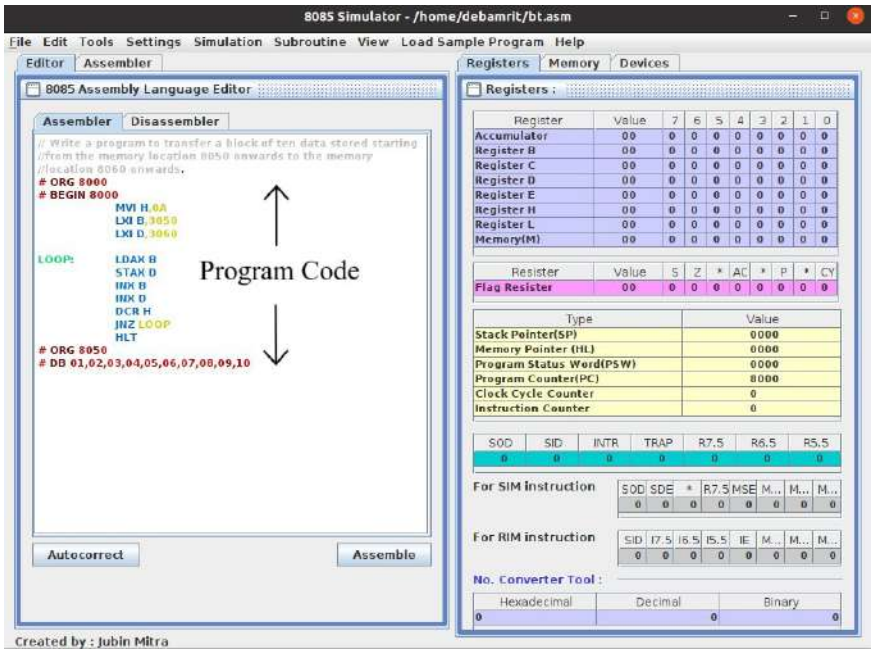


Fig-1.4: Editor window with program code

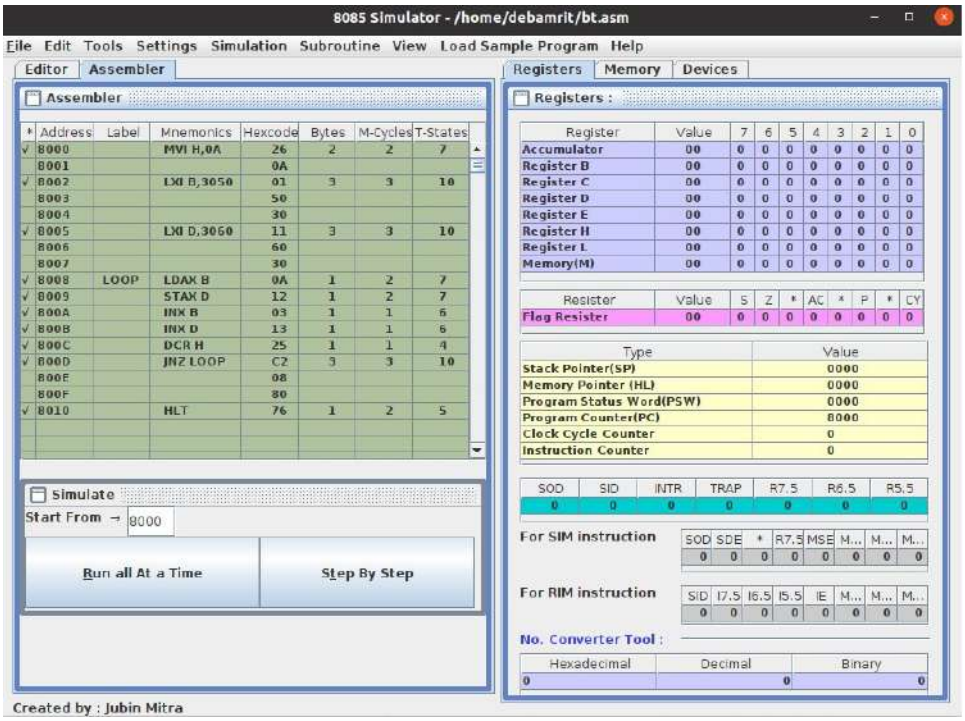


Fig-1.5: Assembler window

2. Familiarization with 8085 Trainer Kit

ALS SDA 85 is a 8085 microprocessor Trainer Kit to dump the hex codes of 8085 program in the memory, execute the dumped code and show the result on 7 segment display board connected to this kit. The specification of this Kit is given below.

1) Operating frequency of the microprocessor 8085 inside this kit is 3.072 MHz.
2) Maximum size of memory 64 KB (32 KB EEPROM and 32 KB RAM). This kit is supplied with 16 KB EEPROM and 8 KB RAM with battery backup.
3) *I/O Parallel:* 48 IO lines using two 8255.
4) *IO Serial:* 0ne RS232 compatible interface
5) *Timer:* Three 16 bit counter timer using 8253.
6) *Keyboard:* Consisting of 28 numbers of computer grade keys.
7) *Display:* Six numbers of seven segment displays.
8) *BUS Signals:* All Address, Data and Control signals are terminated in 50 pin berg stick.
9) *Monitor Software:* 16KB of powerful monitor software with keyboard and serial modes.
10) *Interrupt Controller:* 8 interrupts of 8259 interrupt controller IC are terminated in berg stick.

There are two models of 8085 Trainer Kit in the laboratory – 1) SDA85H and 2) SDA85M. The top views of both the trainer kits are given in Fig-2.1.

Fig-2.1(a): View of SDA 85H Trainer Kit

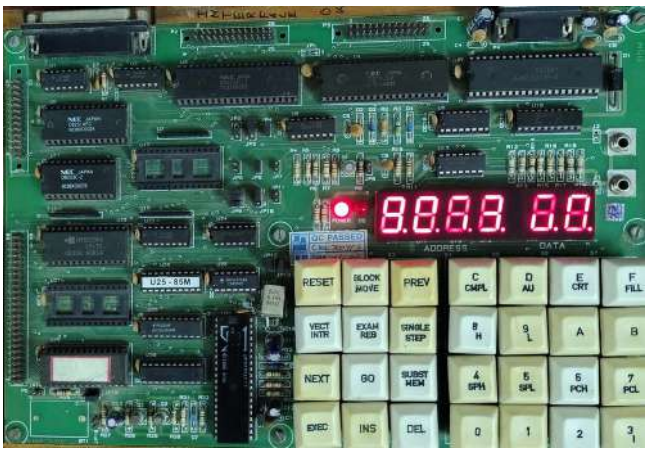


Fig-2.1(b): View of SDA85M Trainer Kit

2.1 Functions of different keys of 8085 Trainer Kit

Key	Function
RESET	It resets the system
SUBST MEM	It is used to display the content of memory location and modify the content of that memory location
EXAM REG	It is used to display the content of particular register
NEXT	It is used to show the content of a particular memory location for SUBST MEM and the content of any register for EXAM REG. It is also used to show the content of the next memory location or the content of the next register of 8085.
PREV	It is also used to show the content of the previous memory address or the content of the previous register of 8085.
GO	It is used to provide the starting address of the program for its execution.
EXEC	It is used to run the entire program at a time after specifying the starting address of the program using GO key.
SINGLE STEP	It is used to execute the program in single step mode.
BLOCK MOVE	It allows user to move a block of memory to another memory space.
VECT INTR	It provides hardware interrupt (RST 7.5) via keyboard
INS	It inserts the part of the program or data with relocation, by one or more bytes
DEL	It deletes the part of program or data, with relocation by one or more bytes.

After the successful execution of a program in Jubin's 8085 Simulator, the same program may be tested on this Kit. For this purpose all the Hex codes are required to dump them into the memory of the 8085 Kit. Jubin's 8085 Simulator provides these Hex codes. The Hex codes are dumped into the memory of the Kit and executed them using the following procedure.

2.2 **Procedure to load program code:**

- Press "RESET" key to reset the system. After resetting "Sda 85" text will appear on the display.
- Press the key "SUBST MEM" to load the program code.
- Enter the starting address of the program and press "NEXT" key.

- Now the content of the memory address specified will be displayed. Enter the correct Hex code with the help of keyboard and press "NEXT" button. This will replace the previous 8-bit data by the present 8-bit data. It is important to mention that until the "NEXT" key is press the new data will not be saved into the specified memory location.
- After pressing "NEXT" it will show the content of the next memory location. Again replace it with new data and press "NEXT".
- This process will continue until the end of the program. At the end of the program enter the hex code "EF" which is the opcode of RST5 software interrupt. This interrupt returns the program control again to its monitor program.
- Sometimes if it is required to change the data of the previous memory location, press "PREV" key in the kit.
- To check whether the program code is loaded or not, press "SUBST MEM" to give the starting address of the program and continue to press "NEXT" button until the end of the program to verify every hex code.
- If any input data are available for the program, insert the input data to desired memory location following the above mentioned process.

2.3 Procedure to execute program:

- To run the program loaded into the memory, first press "RESET" to reset the system.
- Now press "GO" button and provide the starting address of the program.
- Lastly press "EXEC" key to run the program. After the successful execution of the program the text "Sda 85" appears on the display of the Kit once again.

2.4 Procedure to show the result:

- After the successful execution of the program, if the result is stored in memory, press the "SUBST MEM" button, enter the memory address where the result is stored and press "NEXT" to display the result.
- If the result is stored in any register, press "EXAM REG", enter the name of the register by pressing the designated alphabet from the keyboard and press "NEXT" to show the content of that register which is nothing but a result of the program. For example – to get the content of register A or accumulator press "A" key, for register B, press "B", for register C, press "C" etc.

3. Programs on Arithmetic and Logical Operations

3.1: Write a program to add two 8-bit binary numbers which are stored at the memory locations 8050 and 8051 and also store the result of addition into DE register pair.

Method 1: In case of addition of two 8-bit binary numbers, the maximum result will be 1FE when both of the numbers are maximum i.e. FF (FF + FF = 01FE). Hence it is clear that we need an extra bit to store the result. That means a single 8-bit general purpose register (A, B, C, D, E, H, L) of 8085 microprocessor is not sufficient to store the result of two 8-bit binary numbers addition. It needs atleast two 8-bit registers to store the result. That's why register pair DE has been used in the above program to store the result. The flowchart of the above program is given below in Fig-3.1.

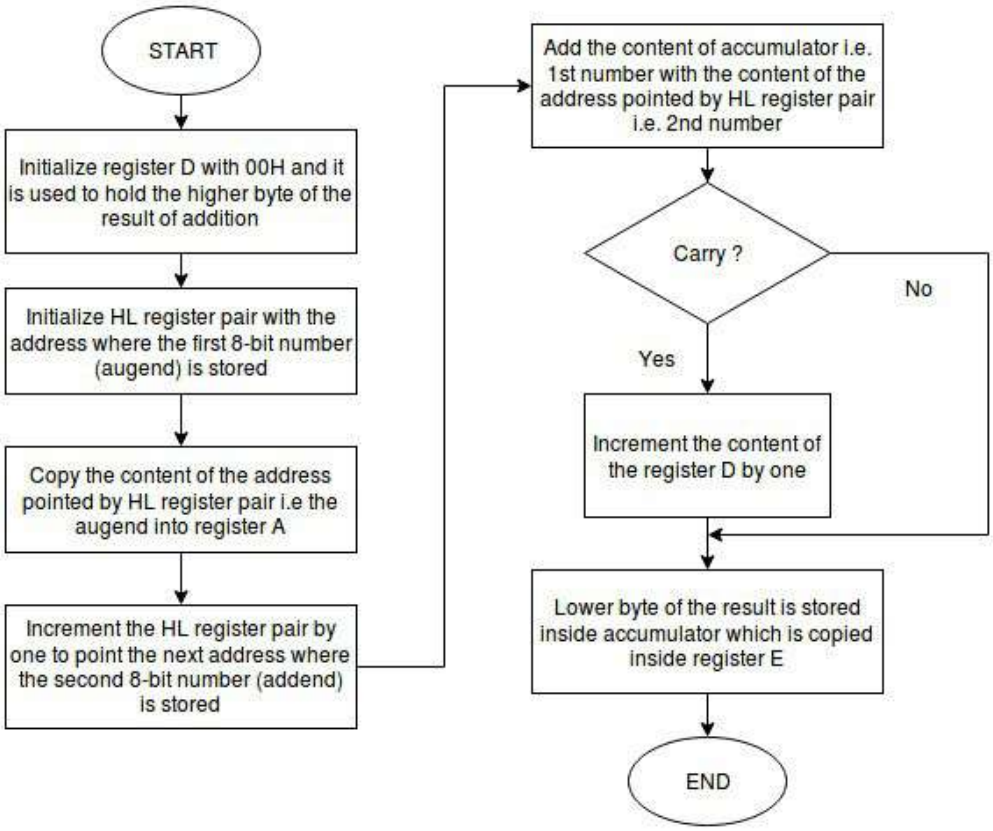


Fig-3.1: Flowchart of the program to add two 8-bit numbers and store the result in DE register pair

Assembly Language Program 3.1 (Method 1):

SL.	Addresses	Label	Mnemonics	Hex Codes			No. of Bytes	No. of T-States
1	8000		MVI D, 00	16	00		2	7
2	8002		LXI H, 8050	21	50	80	3	10
3	8005		MOV A, M	7E			1	7
4	8006		INX H	23			1	6
5	8007		ADD M	86			1	7
6	8008		JNC No_carry	D2	0C	80	3	10 (True) / 7 (False)
7	800B		INR D	14			1	4
8	800C	No_carry	MOV E, A	5F			1	4
9	800D		HLT	76			1	5
							TOTAL = 14	

Method 2: In this technique, the conditional jump instruction "JNC XXXX" is not used. Instead of that, the instruction "ADC R" is used. If the content of Accumulator is made zero and the instruction "ADC A" is used, the carry flag will be stored inside the Accumulator i.e. the higher byte of the result of two 8-bit numbers addition will be stored inside the Accumulator. The flowchart in this technique is shown in Fig-3.2.

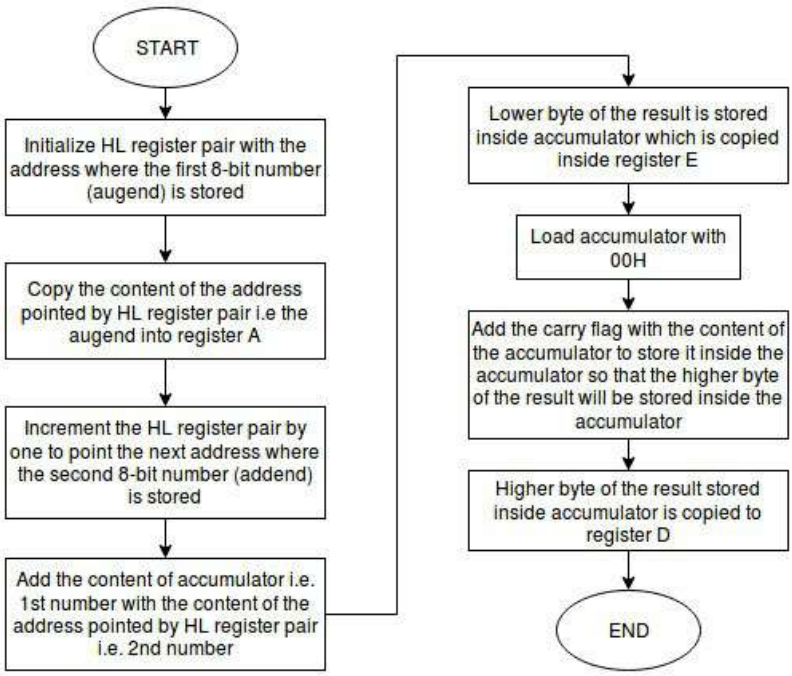


Fig-3.2: Flowchart of the program to add two 8-bit numbers and store the result in DE register pair

Assembly Language Program 3.1 (Method 2):

SL.	Addresses	Label	Mnemonics	Hex Codes			No. of Bytes	No. of T-States
1	8000		LXI H, 8050	21	50	80	3	10
2	8003		MOV A, M	7E			1	7
3	8004		INX H	23			1	6
4	8005		ADD M	86			1	7
5	8006		MOV E, A	5F			1	4
6	8007		MVI A, 00	3E	00		2	7
7	8009		ADC A	8F			1	4
8	800A		MOV D, A	57			1	4
9	800B		HLT	76			1	5
							TOTAL = 12	

Result of Program3.1:

SET1 ►

Input

Mem. Address	Content	Remarks
8050	0A	No1
8051	DD	No2

Output

D → 00 → Higher Byte of Result
E → E7 → Lower Byte of Result

SET2 ►

Input

Mem. Address	Content	Remarks
8050	FF	No1
8051	FE	No2

Output

D → 01 → Higher Byte of Result
E → FD → Lower Byte of Result

Comparisons between Method 1 and Method2

- Now if we compare two methods described above to add two 8-bit numbers, we can observe that the total size of the program in method 2 (12 Bytes) is less than the total size of the first program (14 Bytes). That means in case of the second method, the program will occupy less memory space compared to the first method.

Moreover, execution time in the second method is less than that of the first method. Therefore we can conclude lastly that the second method is better than the first method.

3.2: Write a program to add ten 8-bit binary numbers which are stored at the memory locations starting from 8050 to 8059 and also store the result of addition starting from 805A onward.

It is to determine first, what will be the maximum value of the result of ten 8-bit numbers Addition so that it can be determined that how many bytes is required to store the result. Naturally the result of addition will be maximum, if all the ten 8-bit numbers having their maximum value i.e FF.
Hence FF + FF + FF + FF + FF + FF + FF + FF + FF + FF = 9F6 i.e 09F6

Therefore it is clear that atleast 2 bytes are required to store the result of ten 8-bit numbers addition. We have to use two consecutive memory locations – one 805A and another 805B for storing the lower byte and higher byte of the result respectively.

The concept of this program is that addition should be performed repeatedly for n times for addition of n no. of 8-bit numbers and a register is to be taken for counting the no of carries occurred for these multiple no. of addition. In this case register D has been taken to hold how many times the carry occurred during 9 times addition of ten 8-bit numbers. Each time if a carry occurs the content of register D is to be incremented by one. The ten 8-bit numbers are stored in consecutive memory locations starting from 8050 to 8059 and the lower byte and the higher byte of the result will be stored at address 805A and 805B respectively, which is shown pictorially in the following Fig-3.3.

Addresses	Contents
8050	No 1
8051	No 2
8052	No 3
8053	No 4
8054	No 5
8055	No 6
8056	No7
8057	No 8
8058	No 9
8059	No 10
805A	Lower Byte of Result
805B	Higher Byte of Result

Fig-3.3: Ten 8-bit numbers and the result of addition are stored consecutively from 8050

Method 1: The flowchart of the above mentioned program is given below in Fig-3.4.

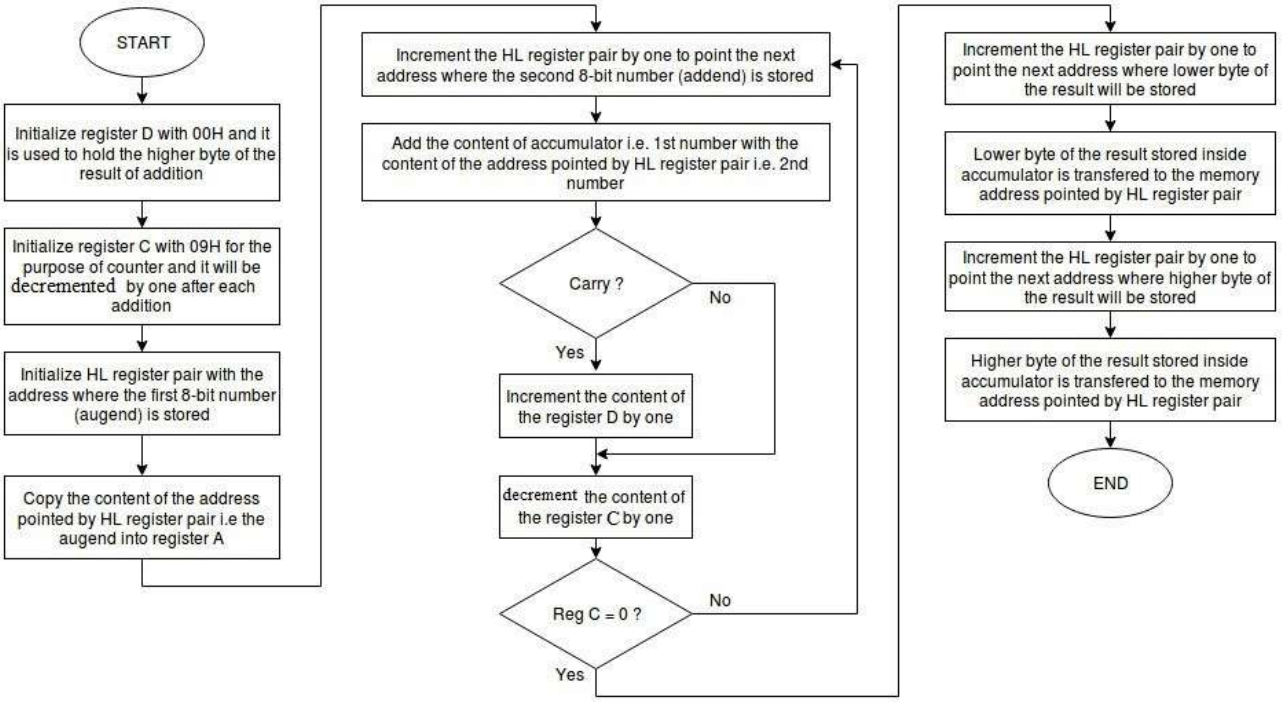


Fig-3.4: Flowchart of the program to add ten 8-bit numbers stored consecutively

Assembly Language Program 3.2 (Method 1):

SL.	Addresses	Label	Mnemonics	Hex Codes			No. of Bytes	No. of T-States
1	8000		MVI D, 00	16	00		2	7
2	8002		MVI C, 09	0E	09		2	7
3	8004		LXI H, 8050	21	50	80	3	10
4	8007		MOV A, M	7E			1	7
5	8008	Repeat	INX H	23			1	6
6	8009		ADD M	86			1	7
7	800A		JNC No_carry	D2	0E	80	3	10 (True) / 7 (False)
8	800D		INR D	14			1	4
9	800E	No_carry	DCR C	0D			1	4
10	800F		JNZ Repeat	C2	08	80	3	10 (true) / 7

SL.	Addresses	Label	Mnemonics	Hex Codes			No. of Bytes	No. of T-States
								(False)
11	8012		INX H	23			1	6
12	8013		MOV M, A	77			1	7
13	8014		INX H	23			1	6
14	8015		MOV M, D	72			1	7
15	8016		HLT	76			1	5
							TOTAL = 23	

In the above program, it is being seen that the counter register C is initialized to 09, beacuse in this case ten 8-bit numbers are being added. This implies that in case of the addition of N no. of 8-bit numbers, the counter register should be initialized to a value (N – 1), if the above procedure is followed. But if we follow the following process, then we have to initialize the counter register with a value which is equal to the no. of 8-bit numbers which are being added. This procedure is given using the assembly language program below.

Assembly Language Program 3.2 (Method 1):

SL.	Addresses	Label	Mnemonics	Hex Codes			No. of Bytes	No. of T-States
1	8000		MVI D, 00	16	00		2	7
2	8002		MVI C, 0A	0E	0A		2	7
3	8004		LXI H, 8050	21	50	80	3	10
4	8007		XRA A	AF			1	4
5	8008	Repeat	ADD M	86			1	7
6	8009		JNC No_carry	D2	0D	80	3	10 (True) / 7 (False)
7	800C		INR D	14			1	4
8	800D	No_carry	INX H	23			1	6
9	800E		DCR C	0D			1	4
10	800F		JNZ Repeat	C2	08	80	3	10 (true) / 7 (False)
11	8012		MOV M, A	77			1	7
12	8013		INX H	23			1	6
13	8014		MOV M, D	72			1	7
14	8015		HLT	76			1	5
							TOTAL = 22	

So it is observed in the above program, the total execution time is 430 T-states which was only 401 T-states in the previous program. Hence this above program takes a very large time to complete in comparison with that of the previous program. That's why this above program may be rejected in comparison to the first program, although this current program takes less memory than the first program.

Method 2: In this method, the same technique like method 1 is applied, only the exception is that the instruction "JNC XXXX" is not used and instead of that the instruction "ADC A" is used. From the assembly language program in this method 2 it is clear that if we compare the first program in method 1 and this program in method 2, the execution time is very much less in the first program in method 1. That's why the first program is the best case to add ten 8-bit numbers.

Assembly Language Program 3.2 (Method 2):

SL.	Addresses	Label	Mnemonics	Hex Codes			No. of Bytes	No. of T-States
1	8000		MVI D, 00	16	00		2	7
2	8002		MVI C, 09	0E	09		2	7
3	8004		LXI H, 8050	21	50	80	3	10
4	8007		MOV B, M	46			1	7
5	8008	Repeat	INX H	23			1	6
6	8009		MOV A, M	7E			1	7
7	800A		ADD B	80			1	4
8	800B		MOV B, A	47			1	4
9	800C		MVI A, 00	3E			2	7
10	800E		ADC D	8A			1	4
11	800F		MOV D, A	57			1	4
12	8010		DCR C	0D			1	4
13	8011		JNZ Repeat	C2			3	10 (True) / 7 (False)
14	8014		INX H	23			1	6
15	8015		MOV M, B	70			1	7
16	8016		INX H	23			1	6
17	8017		MOV M, D	72			1	7
18	8018		HLT	76			1	5
							TOTAL = 25	

Result of Program3.2:

SET1 ►

Input

Mem. Address	Content	Remarks
8050	05	No1
8051	0D	No2
8052	DD	No3
8053	AA	No4
8054	12	No5
8055	32	No6
8056	01	No7
8057	0A	No8
8058	1F	No9
8059	0A	No10

Output

Mem. Address	Content	Remarks
805A	11	Lower Byte of Result
805B	02	Higher Byte of Result

SET2 ►

Input

Mem. Address	Content	Remarks
8050	05	No1
8051	06	No2
8052	07	No3
8053	08	No4
8054	09	No5
8055	0A	No6
8056	0B	No7
8057	0C	No8
8058	0D	No9
8059	0E	No10

Output

Mem. Address	Content	Remarks
805A	5F	Lower Byte of Result
805B	00	Higher Byte of Result

3.3: Write a program to add two 16-bit binary numbers which are stored at the memory locations starting from 8050 to 8053 i.e. 1st number at 8050 and 8051 and 2nd number at 8052 and 8053. Store the result of the addition starting from memory location 8054 onward.

In this program, we are performing 16-bit addition. Therefore maximum size of the result of 16-bit addition must be determined. Maximum value of the result for 16-bit addition will be 1FFFE, when both of the 16-bit numbers are maximum in value i.e. FFFF. Hence we need atleast 3 bytes of memory to store the result. In this case lower byte, higher byte and carry byte of the result will be stored at addresses 8054, 8055 and 8056 respectively. The total memory mapping of the above mentioned case is shown in Fig-3.5.

Addresses	Contents
8050	Lower byte of No 1
8051	Higher byte of No 1
8052	Lower byte of No 2
8053	Higher byte of No 2
8054	Lower byte of result
8055	Higher byte of result
8056	Carry byte of result

Fig-3.5: Two 16-bit numbers and the addition of them are stored consecutively from 8050

Now the above program can be done in two methods – in the first method, it can be done by using the instruction "DAD Reg_Pair" and in the second method, it can be done using the instruction "ADC M". Although the first method is very simple and takes less memory and execution time, the second method must be performed. Because, whenever we perform more than 16-bit addition like 64-bit addition, 128-bit addition, the second method becomes simpler than the first method.

Method 1: The flowchart of this program in first method is shown in Fig-3.6 below.

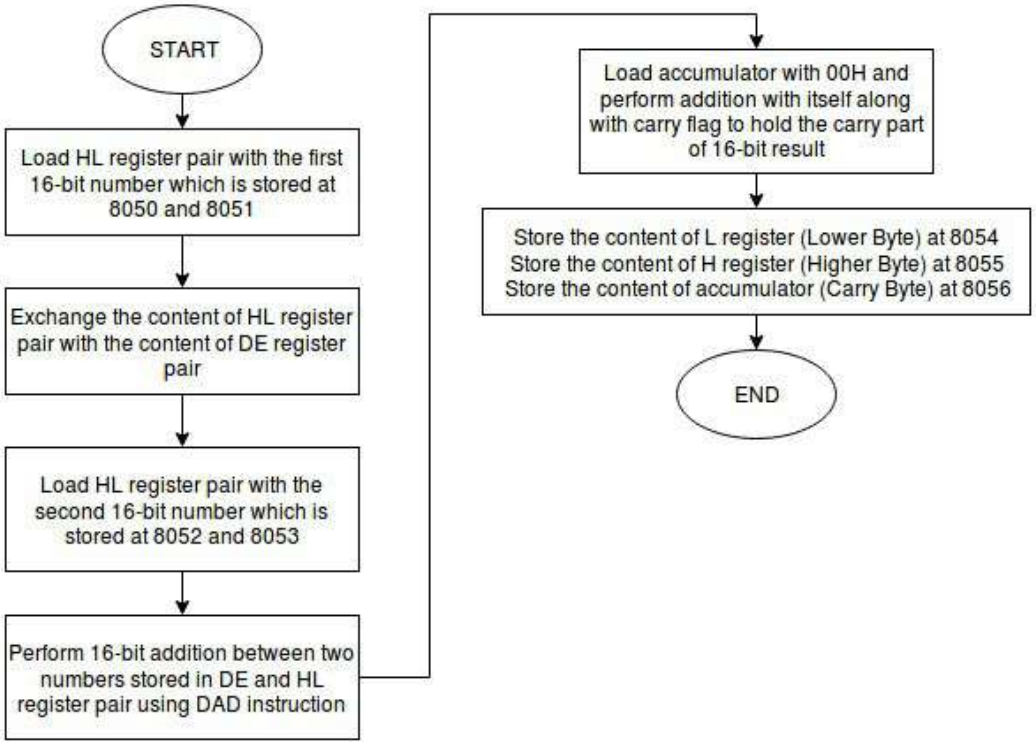


Fig-3.6: Flowchart of two 16-bit numbers addition using DAD instruction

Assembly Language Program 3.3 (Method 1):

SL.	Addresses	Label	Mnemonics	Hex Codes			No. of Bytes	No. of T-States
1	8000		XRA A	AF			1	4
2	8001		LHLD 8050	2A	50	80	3	16
3	8004		XCHG	EB			1	4
4	8005		LHLD 8052	2A	52	80	3	16
5	8008		DAD D	19			1	10
6	8009		ADC A	87			1	4
7	800A		SHLD 8054	22	54	80	3	16
8	800D		STA 8056	32	56	80	3	13
9	8010		HLT	76			1	5
							TOTAL = 17	

Method 2: In this method, for 16-bit addition the instruction "ADC M" is used in place of "DAD Reg_Pair". If we separate a 16-bit number into two bytes, one byte becomes lower byte and other becomes higher byte. So whenever addition is performed between two 16-bit numbers, first addition occurs between two lower bytes of two 16-bit numbers and the next addition is done between two higher bytes of the same two 16-bit numbers along with the carry (if occurs) propagated from the lower byte. It will be clear, if we take an example. Suppose the two 16-bit numbers which are being added, are B18C and FAF9, as shown below.

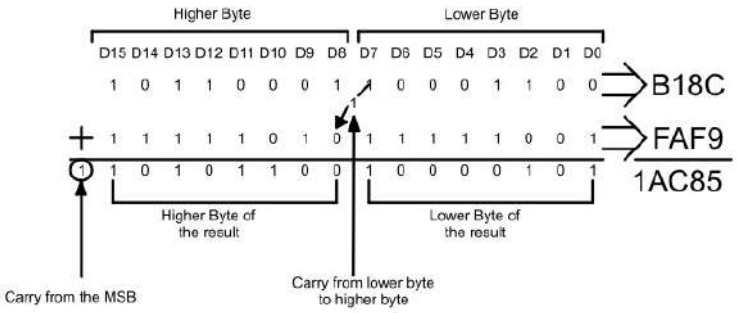


Fig-3.7: Addition between two 16-bit numbers where carry propagates from lower to higher byte

It is now obvious that whenever lower bytes of two 16-bit numbers are added, there is no chance of occuring carry from the previous stage, but during the addition of higher bytes carry may occur. Thet's why before adding lower bytes using ADC instruction, the carry flag must be made zero. The flowchart of addition of two 16-bit numbers using ADC instruction is shown below in Fig-3.8.

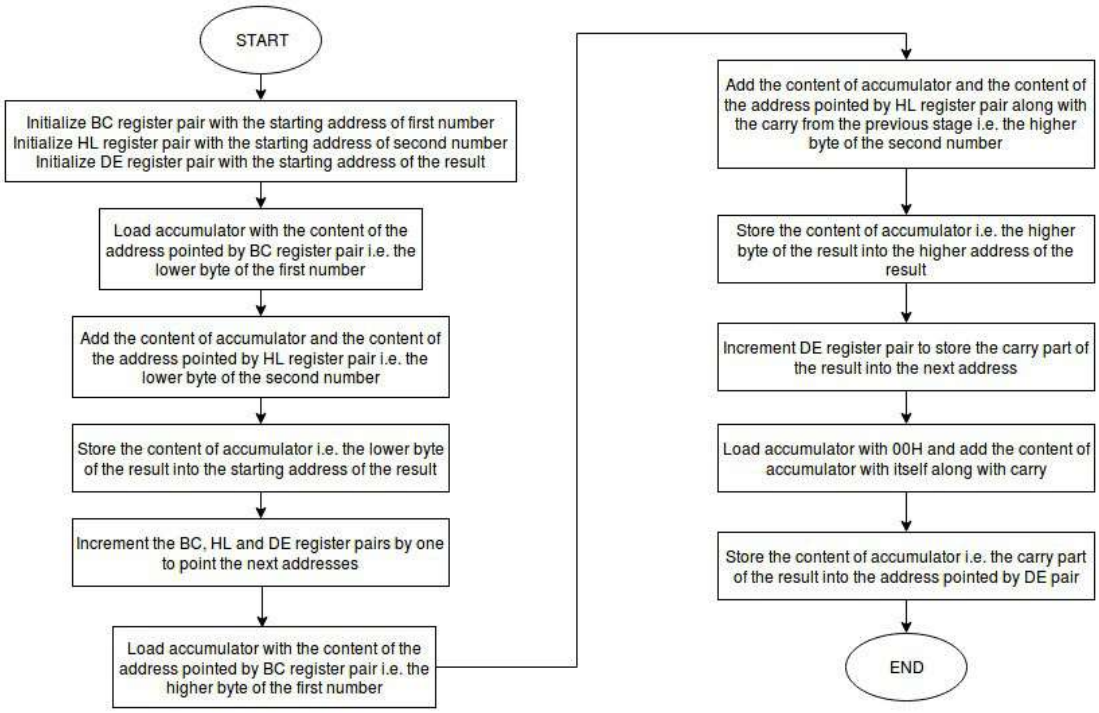


Fig-3.8: Flowchart of two 16-bit numbers addition using ADC instruction

Assembly Language Program 3.3 (Method 2):

SL.	Addresses	Label	Mnemonics	Hex Codes			No. of Bytes	No. of T-States
1	8000		LXI B, 8050	01	50	80	3	10
2	8004		LXI H, 8052	21	52	80	3	10
3	8007		LXI D, 8054	11	54	80	3	10
4	800A		LDAX B	0A			1	7
5	800B		ADD M	86			1	7
6	800C		STAX D	12			1	7
7	800D		INX B	03			1	6
8	800E		INX H	23			1	6
9	800F		INX D	13			1	6
10	8010		LDAX B	0A			1	7
11	8011		ADC M	8E			1	7
12	8012		STAX D	12			1	7
13	8013		INX D	13			1	6
14	8014		MVI A, 00	3E	00		2	7
15	8016		ADC A	8F			1	4
16	8017		STAX D	12			1	7
17	8018		HLT	76			1	5
							TOTAL = 24	

Result of Program3.3:

SET1 ►

Input

Mem. Address	Content	Remarks
8050	FF	Lower byte of No 1
8051	FE	Higher byte of No 1
8052	FC	Lower byte of No 2
8053	FD	Higher byte of No 2

Output

Mem. Address	Content	Remarks
8054	FB	Lower Byte of Result
8055	FC	Higher Byte of Result
8056	01	Carry Byte of Result

SET2 ►

Input

Mem. Address	Content	Remarks
8050	10	Lower byte of No 1
8051	20	Higher byte of No 1
8052	30	Lower byte of No 2
8053	40	Higher byte of No 2

Output

Mem. Address	Content	Remarks
8054	40	Lower Byte of Result
8055	60	Higher Byte of Result
8056	00	Carry Byte of Result

3.4: Write a program to add two 64-bit binary numbers which are stored at the memory locations starting from 8050 onwards and the memory locations starting from 8060 onwards. Store the result of the addition starting from memory location 8070 onwards.

As the two numbers are 64-bit long i.e. 8 byte long, each number occupies eight consecutive memory locations. Hence the first number starts from 8050 to 8057 and the second number starts from 8060 to 8067. Moreover, it takes atleast 9 consecutive bytes to store the result of addition starting from 8070 to 8078. The memory mapping for storing the two 64-bit numbers and their result of addition, is shown in Fig-3.9.

1St Number

Address	Content
8050	Byte1
8051	Byte2
8052	Byte3
8053	Byte4
8054	Byte5
8055	Byte6
8056	Byte7
8057	Byte8

2nd Number

Address	Content
8060	Byte1
8061	Byte2
8062	Byte3
8063	Byte4
8064	Byte5
8065	Byte6
8066	Byte7
8067	Byte8

Result of Addition

Address	Content
8070	Byte1
8071	Byte2
8072	Byte3
8073	Byte4
8074	Byte5
8075	Byte6
8076	Byte7
8077	Byte8
8078	Byte9

Fig-3.9: Memory mapping of two 64-bit numbers and their result of addition

During the addition of two 8-byte numbers, addition of each bytes from two numbers are performed starting from the lowesr byte to highest byte successively i.e. addition is done first in between Byte1 of the two numbers, then between Byte2 and so on. If carry occurs after the addition of two Byte1 of two numbers, that carry will be propagated into the addition of two Byte2 of the two numbers. Similarly if there is carry during the addition of two Byte2, that carry will be propagated into the third bytes of the two numbers. This will go on untill highest byte i.e. Byte8 addition done. In this case, one thing is important to consider that there is no chance of occuring any carry from the previous stage during the addition of lowest bytes i.e. Byte1. Hence before using ADC instruction for adding Byte1 of the two numbers, the carry flag must be reset. The flowchart of this program is shown in Fig-3.10. In this program addition will be performed for 8 times. Therefore a register should be taken as counter. But the problem here is the shortage of general purpose registers, because all the registers except Accumulator have been used to point the starting addresses of two numbers and the starting address of the destination memory block for storing the result of addition such as – BC register pair to point the addresses of 1st number, HL register pair for 2nd number and DE register pair for destination block of result. Therefore a memory location like 8080 will be reserved to store the counting value for each iteration.

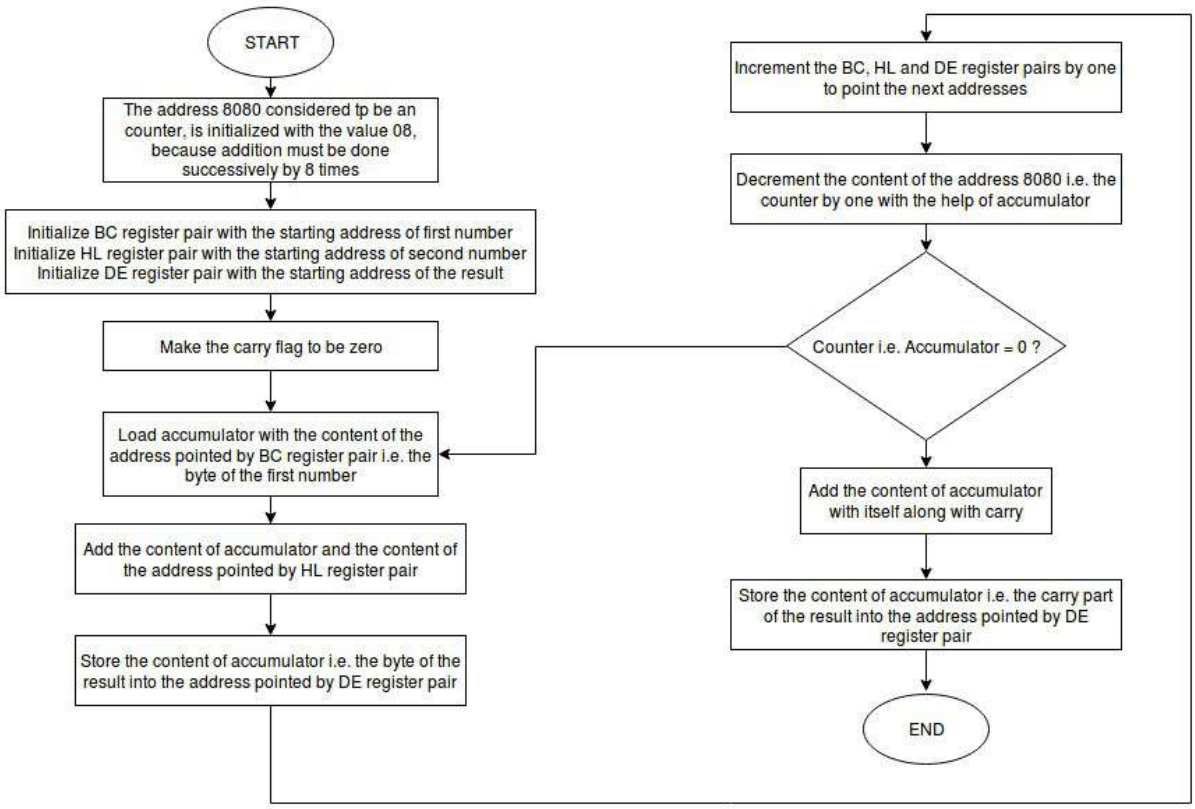


Fig-3.10: Flowchart of addition between two 64-bit numbers

Assembly Language Program 3.4:

SL.	Addresses	Label	Mnemonics	Hex Codes			No. of Bytes	No. of T-States
1	8000		MVI A, 08	3E	08		2	7
2	8002		STA 8080	32	80	80	3	13
3	8005		LXI B, 8050	01	50	80	3	10
4	8008		LXI H, 8060	21	60	80	3	10
5	800B		LXI D, 8070	11	70	80	3	10
6	800E		XRA A	AF			1	4
7	800F	Repeat	LDAX B	0A			1	7
8	8010		ADC M	8E			1	7
9	8011		STAX D	12			1	7
10	8012		INX B	03			1	6
11	8013		INX H	23			1	6

SL.	Addresses	Label	Mnemonics	Hex Codes			No. of Bytes	No. of T-States
12	8014		INX D	13			1	6
13	8015		LDA 8080	3A	80	80	3	13
14	8018		DCR A	3D			1	4
15	8019		STA 8080	32	80	80	3	13
16	801C		JNZ Repeat	C2	0F	80	3	10 (True) / 7 (False)
17	801F		ADC A	8F			1	4
18	8020		STAX D	12			1	7
19	8021		HLT	76			1	5
							TOTAL = 34	

In the above program, before starting the loop, an instruction “XRA A” is used. The reason behind it that for the addition of lowest bytes for first time, the carry flag must be reset. That's why the instruction “XRA A” is used to make the carry flag to be reset or zero. We can also use two instruction consecutively in place of the instruction “XRA A”. The instructions are - “STC” which will set the carry flag and then “CMC” which will invert the carry flag. So if we use these two instructions consecutively one after another, then the carry flag will be zero ultimately. But these two instructions take two bytes of memory whereas the instruction “XRA A” will take only a single byte of memory to do the same purpose. Therefore it is better to use the instruction “XRA A”.

Result of Program3.4:

SET1 ►

Input

No1

Addr	Content	Remarks
8050	88	Byte1
8051	99	Byte2
8052	AA	Byte3
8053	BB	Byte4
8054	CC	Byte5
8055	DD	Byte6
8056	EE	Byte7
8057	FF	Byte8

No2

Addr	Content	Remarks
8060	01	Byte1
8061	02	Byte2
8062	03	Byte3
8063	04	Byte4
8064	05	Byte5
8065	06	Byte6
8066	07	Byte7
8067	08	Byte8

Output

Result

Addr	Content	Remarks
8070	89	Byte1
8071	9B	Byte2
8072	AD	Byte3
8073	BF	Byte4
8074	D1	Byte5
8075	E3	Byte6
8076	F5	Byte7
8077	07	Byte8
8078	01	Byte9

SET2 ►

Input

No1

Addr	Content	Remarks
8050	10	Byte1
8051	20	Byte2
8052	30	Byte3
8053	40	Byte4
8054	50	Byte5
8055	60	Byte6
8056	70	Byte7
8057	80	Byte8

No2

Addr	Content	Remarks
8060	08	Byte1
8061	07	Byte2
8062	06	Byte3
8063	05	Byte4
8064	04	Byte5
8065	03	Byte6
8066	02	Byte7
8067	01	Byte8

Output

Result

Addr	Content	Remarks
8070	18	Byte1
8071	27	Byte2
8072	36	Byte3
8073	45	Byte4
8074	54	Byte5
8075	63	Byte6
8076	72	Byte7
8077	81	Byte8
8078	00	Byte9

3.5: Write a program to subtract two 8-bit binary numbers which are stored at the memory locations 8050 and 8051 and also store the result of subtraction at the memory location 8052 in signed-magnitude form. Consider the content of memory location 8051 is subtracted from the content of memory location 8050.

In this case the result of subtraction will be saved in signed-magnitude format, where the MSB is treated as the sign bit and the remaining bits represent the magnitude. If MSB is high, then the number will be treated as negative number and if it is low, then the number will be treated as positive number. Here the result of two 8-bit numbers subtraction also will be 8-bit long where MSB is the sign bit and remaining seven bits represent the magnitude of the result.

Suppose the content of 8050 is x and content of 8051 is y. That means the subtraction of (x – y) is being performed in this program. If x < y the result of the subtraction will be in 2's complement form and the result has to be 2's complemented and the MSB is made high by performing OR operation with 1000000 i.e. 80 in Hex to get the signed-magnitude form. We have represented the result in signed-magnitude form, because it is easily understandable. If x > y, the result is already in signed-magnitude form. That's why no action is taken in this case. The carry/ borrow flag is set for x < y and it is reset for x > y. So the status of the carry/ borrow flag is checked in this program to decide that whether the result of subtraction is positive or negative. Fig-3.11 shows the flowchart of the program for subtracting two 8-bit numbers.

Limitation: In this method, if the result of subtraction lies between -127 to +127, then this program works correctly, otherwise it gives erroneous result.

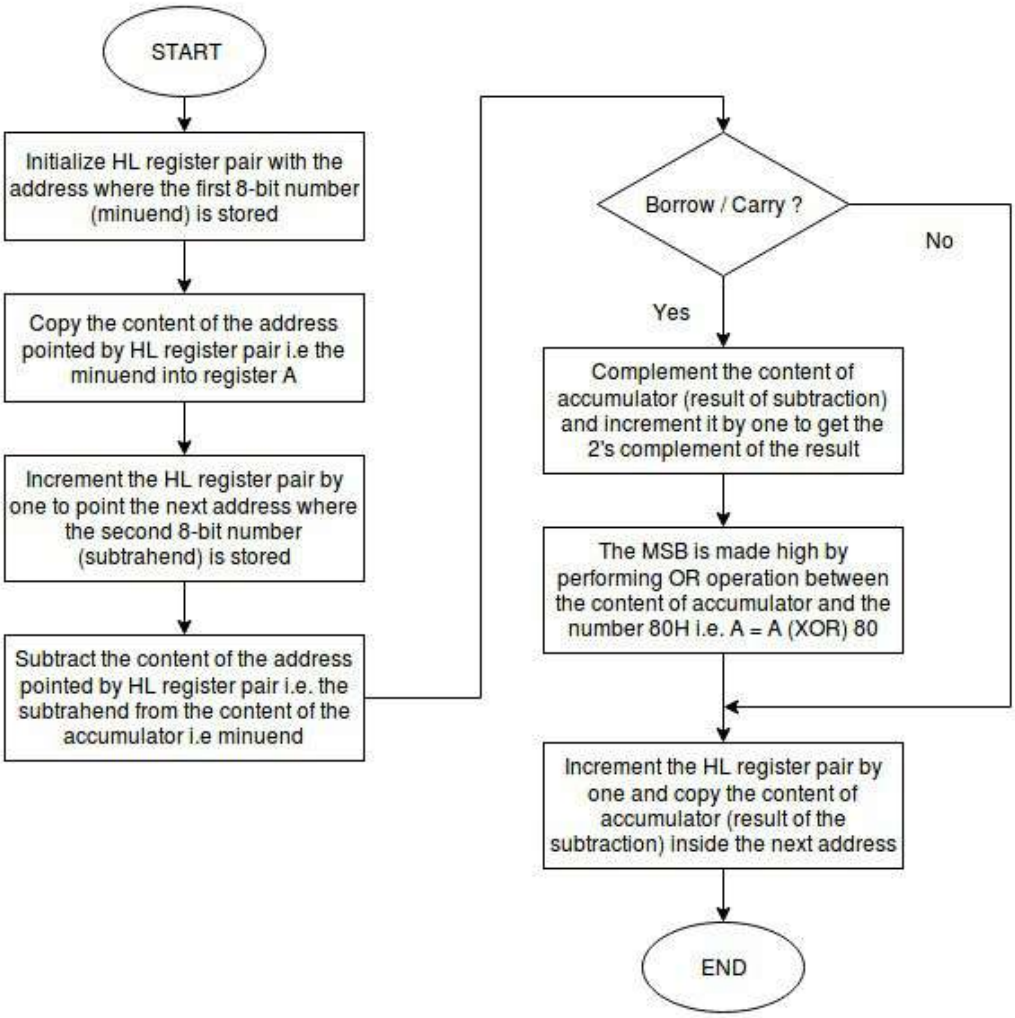


Fig-3.11: Flowchart of the program to subtract two 8-bit numbers

Assembly Language Program 3.5

SL.	Addresses	Label	Mnemonics	Hex Codes			No. of Bytes	No. of T-States
1	8000		LXI H, 8050	21	50	80	3	10
2	8003		MOV A, M	7E			1	7
3	8004		INX H	23			1	6
4	8005		SUB M	96			1	7
5	8006		JNC POSITIVE	D2	0D	80	3	10(True)/7 (False)
6	8009		CMA	2F			1	4
7	800A		INR A	3C			1	4

SL.	Addresses	Label	Mnemonics	Hex Codes			No. of Bytes	No. of T-States
8	800B		ORI 80	F6	80		2	7
9	800D	POSITIVE	INX H	23			1	6
10	800E		MOV M,A	77			1	7
11	800F		HLT	76			1	5
							TOTAL = 16	

Result of Program3.5:

SET1 ►

Input

Mem. Address	Content	Remarks
8050	AA (170)	No1 (Minuend)
8051	46 (70)	No2 (Subtrahend)

Output

Mem. Address	Content	Remarks
8052	64 (100)	Positive Result

SET2 ►

Input

Mem. Address	Content	Remarks
8050	46 (70)	No1 (Minuend)
8051	AA (170)	No2 (Subtrahend)

Output

Mem. Address	Content	Remarks
8052	E4 (-100)	Negative Result

3.6: Write a program to subtract two 16-bit binary numbers which are stored from the memory location 8050 onwards and 8052 onwards and also store the result of subtraction starting from the memory location 8054 onwards in signed-magnitude form. Consider the contents of memory locations 8052 and 8053 are subtracted from the contents of memory locations 8050 and 8051.

In case of subtraction of two 16-bit numbers, four consecutive memory locations are required to store the 16-bit minuend and 16-bit subtrahend. Therefore in this program the memory addresses 8050 and 8051 are used to store the minuend, where 8050 holds the lower byte and 8051 holds the higher byte of the minuend. Similarly the memory locations 8052 and 8053 hold the lower byte and the higher byte of the subtrahend. The result of the subtraction will be obviously 16-bit long and it takes two consecutive memory locations 8054 and 8055, where 8054 will store the lower byte of the result and 8055 will store the higher byte of the result. Now the lower byte of the subtrahend will be subtracted first from the lower byte of the minuend, then the higher byte of the subtrahend will be subtracted from the higher byte of the minuend taking account of the borrow of the lower byte

subtraction. If carry flag is set after the higher byte subtraction, it imples that the minuend is less than the subtrahend and we have to make MSB of the result to be high after taking 2's complement of the entire 16-bit result to get signed-magnitude form. If carry does not occur, we do not need to take any action. The flowchart of this program is shown in Fig-3.12.

Limitation: In this method, if the result of subtraction lies between -32767 to +32767, then this program works correctly, otherwise it gives erroneous result.

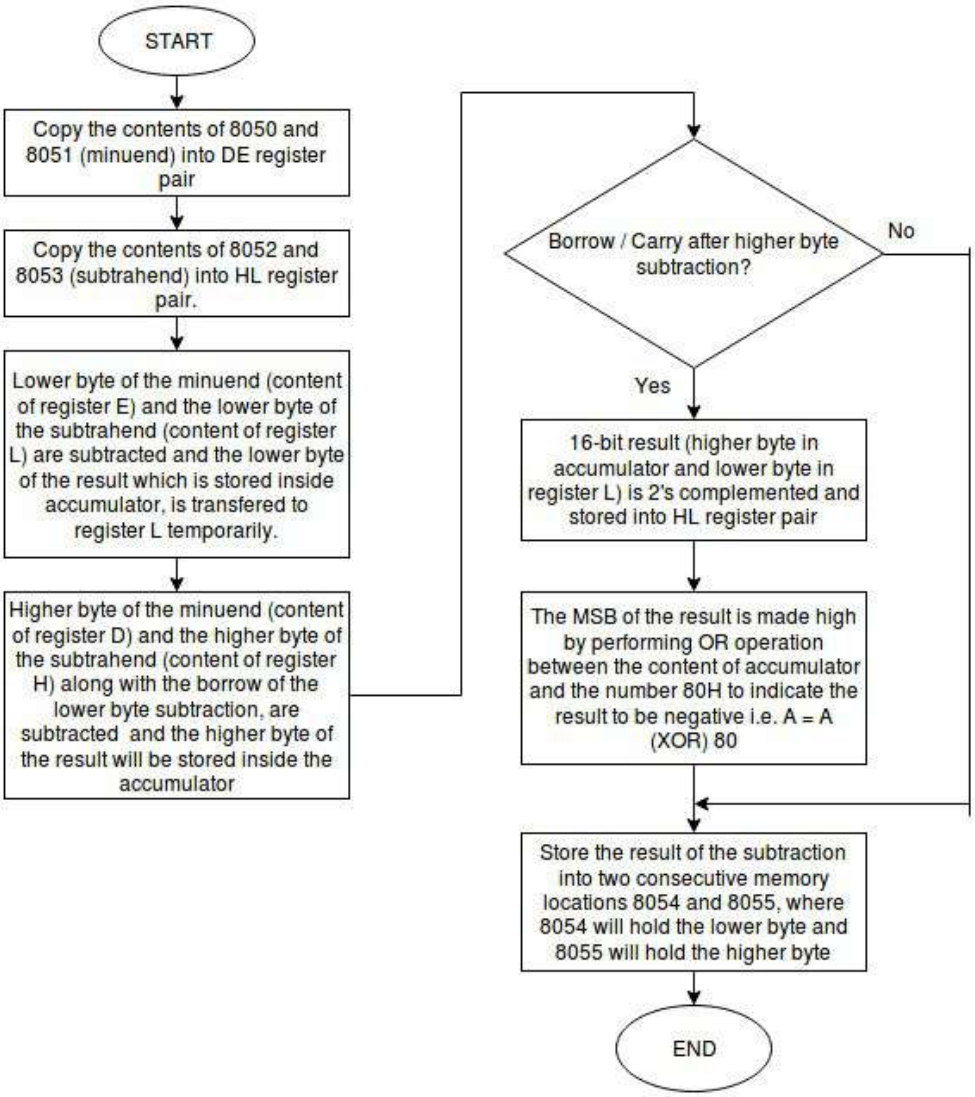


Fig-3.12: Flowchart of the subtraction between two 16-bit binary numbers

Assembly Language Program 3.6:

SL.	Addresses	Label	Mnemonics	Hex Codes			No. of Bytes	No. of T-States
1	8000		LHLD 8050	2A	50	80	3	16
2	8003		XCHG	EB			1	4
3	8004		LHLD 8052	2A	52	80	3	16
4	8007		MOV A, E	7B			1	4
5	8008		SUB L	95			1	4
6	8009		MOV L,A	6F			1	4
7	800A		MOV A,D	7A			1	4
8	800B		SBB H	9C			1	4
9	800C		JNC POSITIVE	D2	1B	80	3	10 (True) / 7 (False)
10	800F		CMA	2F			1	4
11	8010		MOV H,A	67			1	4
12	8011		MOV A,L	7D			1	4
13	8012		CMA	2F			1	4
14	8013		MOV L,A	6F			1	4
15	8014		LXI D,0001	11	01	00	3	10
16	8017		DAD D	19			1	10
17	8018		MOV A,H	7C			1	4
18	8019		ORI 80	F6	80		2	7
19	801B	POSITIVE	MOV H,A	67			1	4
20	801C		SHLD 8054	22	54	80	3	16
21	801F		HLT	76			1	5
							TOTAL = 32	

Result of Program3.6:

SET1 ►

Input

Address	Content	Remarks
8050	AA	Lower Byte of Minuend
8051	46	Higher Byte of Minuend
8052	20	Lower Byte of Subtrahend
8053	10	Higher Byte of Subtrahend

Minuend = 46AAH = 18090
Subtrahend = 1020H = 4128

Output

Mem. Address	Content	Remarks
8054	8A	Positive Result
8055	36	

Result = 368AH = 13962

SET2 ►

Input

Address	Content	Remarks
8050	20	Lower Byte of Minuend
8051	10	Higher Byte of Minuend
8052	AA	Lower Byte of Subtrahend
8053	46	Higher Byte of Subtrahend

Minuend = 1020H = 4128
Subtrahend = 46AAH = 18090

Output

Mem. Address	Content	Remarks
8054	8A	Negative Result
8055	B6	

Result = B68AH = -13962

3.7: Write a program to multiply two 8-bit binary numbers which are stored at the memory locations 8050 and 8051 and also store the result of multiplication from the memory location 8052 onwards using successive addition.

Method 1: For multiplication of two 8-bit numbers, the result of the multiplication will be maximum, if both multiplicant and multiplier will be maximum i.e. multiplicant = FFH and multiplier = FFH. In this case the result of multiplication will be FE01H which is 16-bit long. Therefore we need atleast two consecutive memory locations to store the result of multiplaction. For this reason, the result of the multiplication should be stored at two successive memory locations 8052 and 8053, where 8052 should hold the lower byte and 8053 should hold the higher byte. Here repeatative addition is utilized to implement the program. For example, suppose multiplicant = 05 and multiplier = 09, that means to get the product we have to perform (00 + 09 + 09 + 09 + 09 + 09) which implies that multiplicant should be taken as the counter and the addition of multiplier with itself should be done for several times by using the counter.

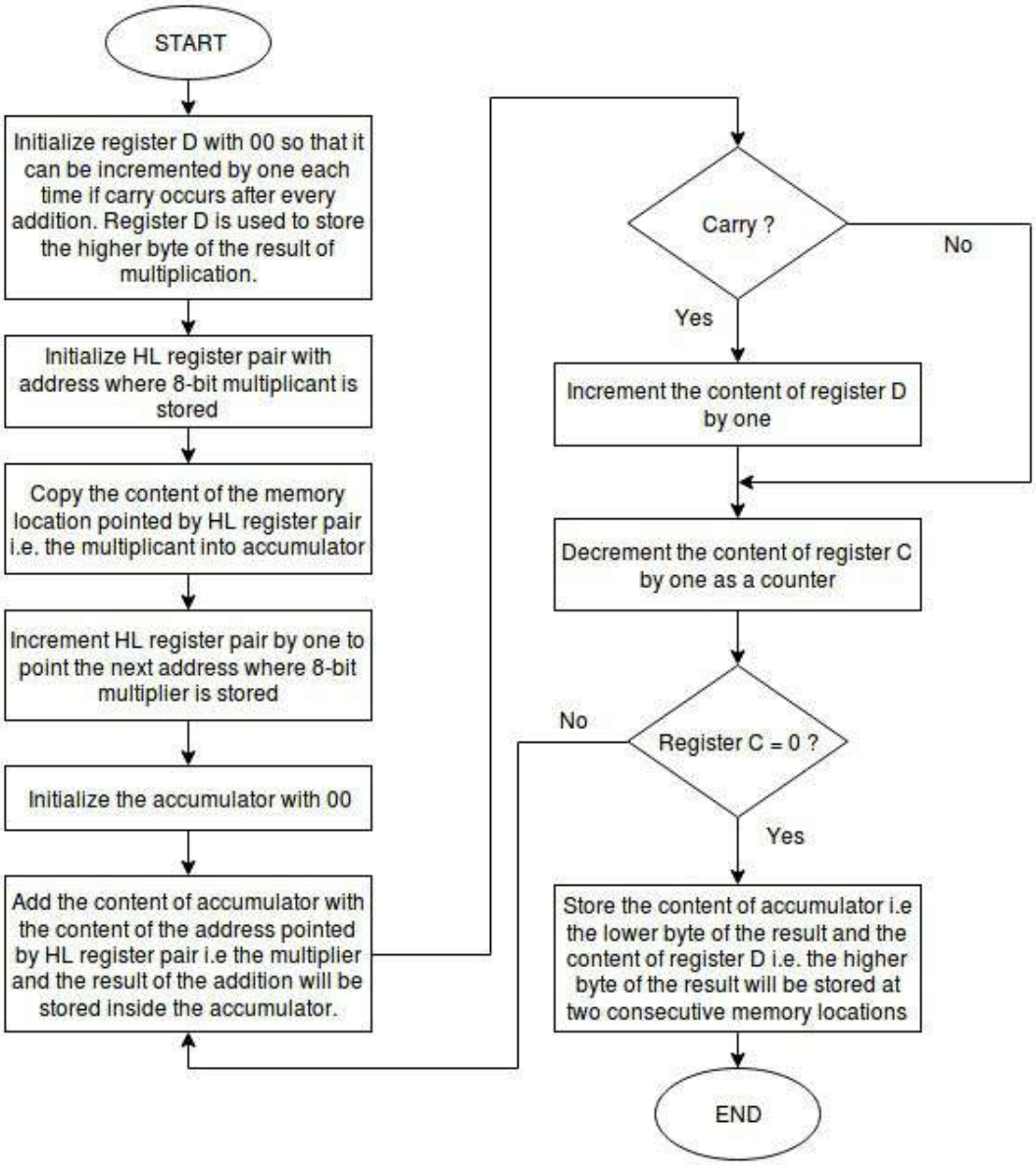


Fig-3.13: Flowchart of multiplication of two 8-bit numbers using successive addition

Assembly Language Program 3.7 (Method 1):

SL.	Addresses	Label	Mnemonics	Hex Codes			No. of Bytes	No. of T-States
1	8000		MVI D,00	16	00		2	7
2	8002		LXI H,8050	21	50	80	3	10
3	8005		MOV C,M	4E			1	7
4	8006		INX H	23			1	6
5	8007		XRA A	AF			1	4
6	8008	LOOP	ADD M	86			1	7
7	8009		JNC SKIP	D2	0D	80	3	10 (True) / 7 (False)
8	800C		INR D	14			1	4
9	800D	SKIP	DCR C	0D			1	4
10	800E		JNZ LOOP	C2	08	80	3	10 (True) / 7 (False)
11	8011		INX H	23			1	6
12	8012		MOV M,A	77			1	7
13	8013		INX H	23			1	6
14	8014		MOV M,D	72			1	7
15	8015		HLT	76			1	5
							TOTAL = 22	

Method 2: For multiplication of two 8-bit numbers, the result of the multiplication will be maximum, if both multiplicant and multiplier will be maximum i.e. multiplicant = FFH and multiplier = FFH. In this case the result of multiplication will be FE01H which is 16-bit long. Therefore we need atleast two consecutive memory locations to store the result of multiplaction. For this reason, the result of the multiplication should be stored at two successive memory locations 8052 and 8053, where the memory locations 8052 and 8053 will hold the lower byte and the higher byte of the result. Here repeatative addition is not utilized to implement the program, because it will take long time to execute the program. That's why the technique of manual multiplication of two binary numbers is used here. Here the 8-bit multiplier is copied to register B first. Then 8-bit multiplicand is also copied to register E and register D is initialized to 00H, because the aim is to store the 16-bit multiplicand or the left shifted pattern of the 16-bit multiplicand inside the register pair DE. Register pair HL is utilized to store the 16-bit result ultimately and it is initialized to 0000H at the beginning.

The bits of the divisor are searched starting from LSB to MSB one by one. If LSB is found 1, add the 16-bit multiplicand formed by adding 8 no. of 0 at the MSB side, with the content of register pair HL and the result is stored again into the same HL pair. If the next bit of the LSB of the multiplier is 1, the 16-bit multiplicand shifted one position left will be added. This process will continue upto the MSB of the multiplier. If any bit of the multiplier is found 0, no action is taken except the multiplicand will be shifted left by one position. In this program, after shifting the multiplicand left each time and inserting zero from the side of LSB, it will be stored into the DE register pair. For better understanding let's take an example of 8-bit multiplication.

8-bit Multiplicant (x) = 1110 0010
8-bit Multiplier (y) = 1110 0111

15	*14*	*13*	*12*	*11*	*10*	*9*	*8*	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*
							Multiplicant (x):	1	1	1	0	0	0	1	0
							Multiplier (y):	1	1	1	0	0	1	1	1
								1	1	1	0	0	0	1	0
							1	1	1	0	0	0	1	0	0
						1	1	1	0	0	0	1	0	0	0
					0	0	0	0	0	0	0	0	0	0	0
				0	0	0	0	0	0	0	0	0	0	0	0
			1	1	1	0	0	0	1	0	0	0	0	0	0
		1	1	1	0	0	0	1	0	0	0	0	0	0	0
	1	1	1	0	0	0	1	0	0	0	0	0	0	0	0
1	**1**	**0**	**0**	**1**	**0**	**1**	**1**	**1**	**1**	**1**	**0**	**1**	**1**	**1**	**0**

The above mentioned example clearly explains that how the multiplication of two 16-bit numbers are being done manually. Now the flowchart of this program is given in Fig-3.14.

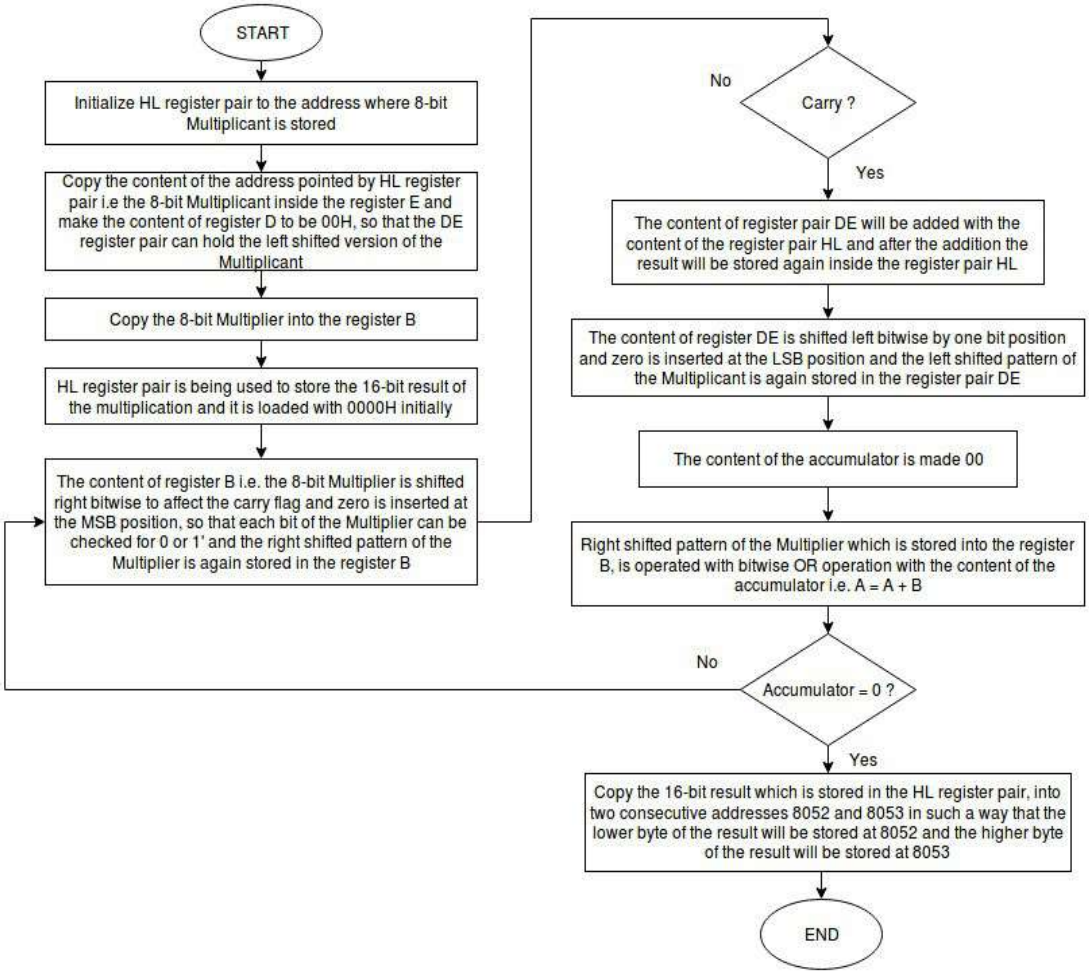


Fig-3.14: Flowchart of multiplication between two 8-bit numbers

Assembly Language Program 3.7 (Method 2):

SL.	Addresses	Label	Mnemonics	Hex Codes			No. of Bytes	No. of T-States
1	8000		LXI H,8050	21	50	80	3	10
2	8003		MOV E,M	5E			1	7
3	8004		MVI D,00	16	00		2	7
4	8006		INX H	23			1	6
5	8007		MOV B,M	46			1	7
6	8008		LXI H,0000	21	00	00	3	10

SL.	Addresses	Label	Mnemonics	Hex Codes			No. of Bytes	No. of T-States
7	800B	LOOP	MOV A,B	78			1	4
8	800C		STC	37			1	4
9	800D		CMC	3F			1	4
10	800E		RAR	1F			1	4
11	800F		MOV B,A	47			1	4
12	8010		JNC NOACTION	D2	14	80	3	10 (True) / 7 (False)
13	8013		DAD D	19			1	10
14	8014	NOACTION	MOV A,E	7B			1	4
15	8015		STC	37			1	4
16	8016		CMC	3F			1	4
17	8017		RAL	17			1	4
18	8018		MOV E,A	5F			1	4
19	8019		MOV A,D	7A			1	4
20	801A		RAL	17			1	4
21	801B		MOV D,A	57			1	4
22	801C		XRA A	AF			1	4
23	801D		ORA B	B0			1	4
24	801E		JNZ LOOP	C2	0B	80	3	10 (True) / 7 (False)
25	8021		SHLD 8052	22	52	80	3	16
26	8024		HLT	76			1	5
							TOTAL = 37	

Result of Program3.7:

SET1 ►

Input

Mem. Address	Content	Remarks
8050	FF	No1 (Multiplicand)
8051	FC	No2 (Multiplier)

Output

Mem. Address	Content	Remarks
8052	04	Lower Byte of Result
8053	FB	Higher Byte of Result

SET2 ►

Input

Mem. Address	Content	Remarks
8050	99	No1 (Multiplicand)
8051	AA	No2 (Multiplier)

Output

Mem. Address	Content	Remarks
8052	9A	Lower Byte of Result
8053	65	Higher Byte of Result

3.8: Write a program to multiply two 16-bit numbers which are stored from 8050 onwards and 8052 onwards, Result should be stored from the memory location 8054 onwards.

For multiplication of two 16-bit numbers, the result of the multiplication will be maximum, if both multiplicant and multiplier will be maximum i.e. multiplicant = FFFFH and multiplier = FFFFH. In this case the result of multiplication will be FFFE0001H which is 32-bit long. Therefore we need atleast four consecutive memory locations to store the result of multiplaction. For this reason, the result of the multiplication should be stored at four successive memory locations 8054, 8055, 8056 and 8057 from least significant byte to most significant byte. Here repeatative addition is not utilized to implement the program, because it will take long time to execute the program. That's why the technique of manual multiplication of two binary numbers is used here. As the result is four bytes long, a consecutive block of four memory locations from 8054 to 8057 will be initialized to 00000000H so that every time the multiplicant or the left shifted version of the multiplicant can be added with the content of that block of memory and the result of addition can be stored again into the same block of memory (8054 – 8057).

The bits of the divisor are searched starting from LSB to MSB one by one. If LSB is found 1, add the 32-bit multiplicant formed by adding 16 no. of 0 at the MSB side, with the content of memory block (8054 – 8057) and the result is stored again into the same block. If the next bit of the LSB of the multiplier is 1, the 32-bit multiplicant shifted one position left will be added. This process will continue upto the MSB of the multiplier. If any bit of the multiplier is found 0, no action is taken except the multiplicant will be shifted left by one position. In this program, after shifting the multiplicant left each time and inserting zero from the side of LSB, it will be stored at the memory locations starting from 8058 to 805B. For better understanding let's take an example of 16-bit multiplication which is done using the above mentioned technique.

16-bit Multiplicant (x) = 1001 0111 1110 0010 & 16-bit Multiplier (y) = 1100 0011 1110 0111

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
																Muliplicant (x):		1	0	0	1	0	1	1	1	1	1	1	0	0	0	1	0
																Multiplier (y):		1	1	0	0	0	0	1	1	1	1	1	0	0	1	1	1
																		1	0	0	1	0	1	1	1	1	1	1	0	0	0	1	0
															1	0	0	1	0	1	1	1	1	1	1	1	0	0	0	1	0	0
														1	0	0	1	0	1	1	1	1	1	1	1	0	0	0	1	0	0	0
													0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
												0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
											1	0	0	1	0	1	1	1	1	1	1	0	0	0	1	0	0	0	0	0	0	0
										1	0	0	1	0	1	1	1	1	1	1	0	0	0	1	0	0	0	0	0	0	0	0
									1	0	0	1	0	1	1	1	1	1	1	0	0	0	1	0	0	0	0	0	0	0	0	0
								1	0	0	1	0	1	1	1	1	1	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0
							1	0	0	1	0	1	1	1	1	1	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
						0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
					0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
				0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	1	0	0	1	0	1	1	1	1	1	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	1	0	1	1	1	1	1	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	**1**	**1**	**1**	**0**	**1**	**0**	**0**	**0**	**0**	**1**	**1**	**1**	**0**	**1**	**0**	**0**	**0**	**1**	**1**	**0**	**0**	**1**	**0**	**1**	**1**	**1**	**0**	**1**	**1**	**1**	**0**

The momory locations for storing the 16-bit multiplicant, 16-bit multiplier, 32-bit result and 32-bit shifted pattern of multiplicant initially are shown pictorically in Fig-3.15 below.

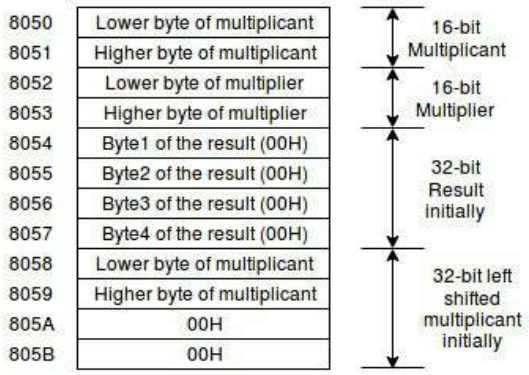


Fig-3.15: Memory locations containing Multiplicant, Multiplier, Result, Left shifted Multiplicant

The above mentioned example clearly explains that how the multiplication of two 16-bit numbers are being done manually. Now the flowchart of this program is given in Fig-3.16.

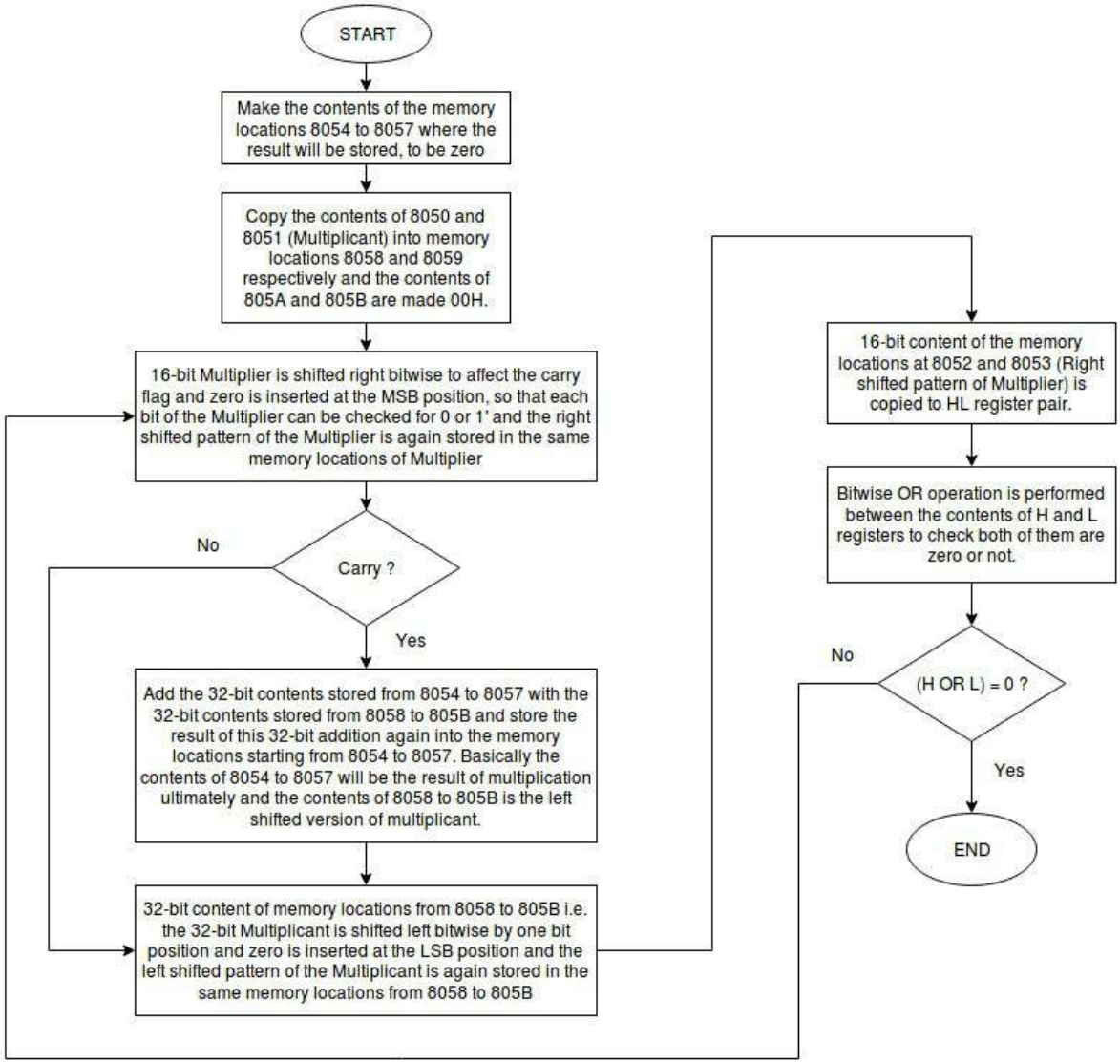


Fig-3.16: Flowchart of multiplication between two 16-bit numbers

Assembly Language Program 3.8:

SL.	Addresses	Label	Mnemonics	Hex Codes			No. of Bytes	No. of T-States
1	8000		LHLD 8050	2A	50	80	3	16
2	8003		SHLD 8058	22	58	80	3	16
3	8006		LXI H,0000	21	00	00	3	10

SL.	Addresses	Label	Mnemonics	Hex Codes			No. of Bytes	No. of T-States
4	8009		SHLD 8054	22	54	80	3	16
5	800C		SHLD 8056	22	56	80	3	16
6	800F		SHLD 805A	22	5A	80	3	16
7	8012	LOOP	STC	37			1	4
8	8013		CMC	3F			1	4
9	8014		LXI H,8053	21	53	80	3	10
10	8017		MOV A,M	7E			1	7
11	8018		RAR	1F			1	4
12	8019		MOV M,A	77			1	7
13	801A		DCX H	2B			1	6
14	801B		MOV A,M	7E			1	7
15	801C		RAR	1F			1	4
16	801D		MOV M,A	77			1	7
17	801E		JNC SKIP	D2	34	80	3	10 (True) / 7 (False)
18	8021		MVI C,04	0E	04		2	7
19	8023		LXI D,8058	11	58	80	3	10
20	8026		LXI H,8054	21	54	80	3	10
21	8029		STC	37			1	4
22	802A		CMC	3F			1	4
23	802B	REPEAT	LDAX D	1A			1	7
24	802C		ADC M	8E			1	7
25	802D		MOV M,A	77			1	7
26	802E		INX D	23			1	6
27	802F		INX H	13			1	6
28	8030		DCR C	0D			1	4
29	8031		JNZ REPEAT	C2	2B	80	3	10 (True) / 7 (False)
30	8034	SKIP	CALL SHIFT	CD	40	80	3	18
31	8037		LHLD 8052	2A	52	80	3	16

SL.	Addresses	Label	Mnemonics	Hex Codes			No. of Bytes	No. of T-States
32	803A		MOV A,L	7D			1	4
33	803B		ORA H	B4			1	4
34	803C		JNZ LOOP	C2	12	80	3	10 (True) / 7 (False)
35	803F		HLT	76			1	5
36	8040	SHIFT	STC	37			1	4
37	8041		CMC	3F			1	4
38	8042		MVI C,04	0E	04		2	7
39	8044		LXI H,8058	21	58	80	3	10
40	8047	ROTATE	MOV A,M	7E			1	7
41	8048		RAL	17			1	4
42	8049		MOV M,A	77			1	7
43	804A		INX H	23			1	6
44	804B		DCR C	0D			1	4
45	804C		JNZ ROTATE	C2	47	80	3	10 (True) / 7 (False)
46	804F		RET	C9			1	10
							TOTAL = 80	

Result of Program3.8:

SET1 ►

Input

Address	Content	Remarks
8050	FF	Lower Byte of Multiplicand
8051	FC	Higher Byte of Multiplicand
8052	FE	Lower Byte of Multiplier
8053	FB	Higher Byte of Multiplier

Output

Address	Content	Remarks
8054	02	Byte1 of Result
8055	0A	Byte2 of Result
8056	09	Byte3 of Result
8057	F9	Byte4 of Result

SET2 ►

Input

Address	Content	Remarks
8050	DD	Lower Byte of Multiplicand
8051	FC	Higher Byte of Multiplicand
8052	FE	Lower Byte of Multiplier
8053	00	Higher Byte of Multiplier

Output

Address	Content	Remarks
8054	46	Byte1 of Result
8055	E3	Byte2 of Result
8056	FA	Byte3 of Result
8057	00	Byte4 of Result

3.9: Write a program to divide two 8-bit binary numbers which are stored at the memory locations 8050 and 8051 and also store the quotient at 8052 and remainder at 8053 after the division. Assume the divident is stored at the memory location 8050 and the divisor at 8051.

In case of division of two 8-bit numbers, the quotient and the remainder both will be 8-bit long. Therefore we need two consecutive memory locations to store the result of the division. For this reason, here the result of the division should be stored at two successive memory locations 8052 and 8053, where 8052 should hold the quotient and 8053 should hold the remainder. Here repeatative subtraction is utilized to perform the division between two 8-bit numbers. For example, suppose divident = 0E and divisor = 03. Hence to get the quotient we have to perform subtraction (divident – divisor) and the subtraction will continue until divident will be less than the divisor. In this way, we will get the quotient to be 04. When the divident will be just less than the divisor, then remainder will be equal to divident and we will get the remainder to be 02 here. Therefore we have to take a counter with a initial value of 00 in this case and increment the counter by one each time the subtraction is done. Ultimately, the value of the counter will be the quotient after the completion of the division. But one thing is important to note that if the divisor = 00, the successive subtraction will go on for infinite times, because of the division-by-zero error. We have to consider that situation also. The value of the divisor should be checked and if the divisor is zero, the program must be halted immediately.

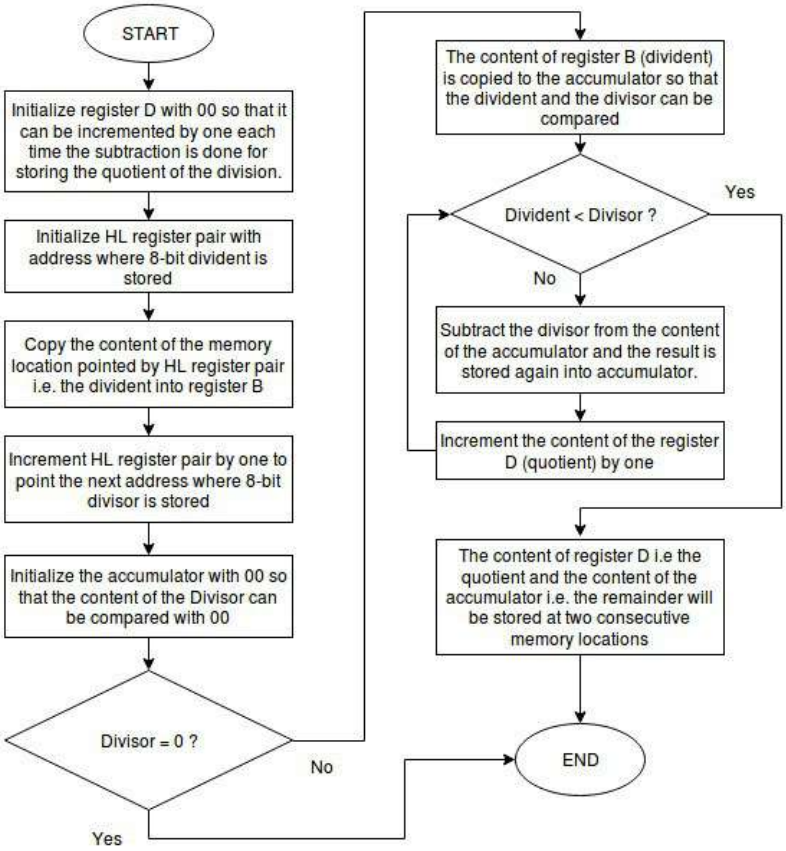


Fig-3.17: Flowchart of division of two 8-bit numbers using successive subtraction

Assembly Language Program 3.9:

SL.	Addresses	Label	Mnemonics	Hex Codes			No. of Bytes	No. of T-States
1	8000		MVI D,00	16	00		2	7
2	8002		LXI H,8050	21	50	80	3	10
3	8005		MOV B,M	46			1	7
4	8006		INX H	23			1	6
5	8007		XRA A	AF			1	4
6	8008		CMP M	BE			1	7
7	8009		JZ ZERO	CA	1A	80	3	
8	800C		MOV A,B	78			1	4
9	800D	AGAIN	CMP M	BE			1	7
10	800E		JC SKIP	DA	16	80	3	10 (True) / 7

SL.	Addresses	Label	Mnemonics	Hex Codes			No. of Bytes	No. of T-States
								(False)
11	8011		SUB M	96			1	7
12	8012		INR D	14			1	4
13	8013		JMP AGAIN	C3	0D	80	3	10 (True) / 7 (False)
14	8016	SKIP	INX H	23			1	6
15	8017		MOV M,D	72			1	7
16	8018		INX H	23			1	6
17	8019		MOV M,A	77			1	7
18	801A	ZERO	HLT	76			1	5
							TOTAL = 27	

Result of Program3.9:

SET1 ►

Input

Mem. Address	Content	Remarks
8050	FF	No1 (Divident)
8051	0A	No2 (Divisor)

Output

Mem. Address	Content	Remarks
8052	19	Quotient
8053	05	Remainder

SET2 ►

Input

Mem. Address	Content	Remarks
8050	0F	No1 (Divident)
8051	10	No2 (Divisor)

Output

Mem. Address	Content	Remarks
8052	00	Quotient
8053	0F	Remainder

Program 3.10: Write a program to divide two 16-bit binary numbers which are stored from the memory locations 8050 onwards and 8052 onwards and also store the quotient from the memory location 8054 onwards and the remainder from the memory location 8056 onwards after the division. Assume the divident is stored from the memory location 8050 onwards and the divisor from the memory location 8052 onwards.

In case of division of two 16-bit numbers, the quotient and the remainder both will be 16-bit long. Therefore we need four consecutive memory locations to store the result of the division. For this

reason, here the result of the division should be stored at four successive memory locations 8054, 8055, 8056 and 8057, where 8054 and 8055 should hold the quotient and 8056 and 8057 should hold the remainder. Here repetitive subtraction is utilized to perform the division between two 16-bit numbers. Therefor, the same concept like division of two 8-bit numbers will be applied here. The flowchart of the program to implement 16-bit division is shown in Fig-3.18 below.

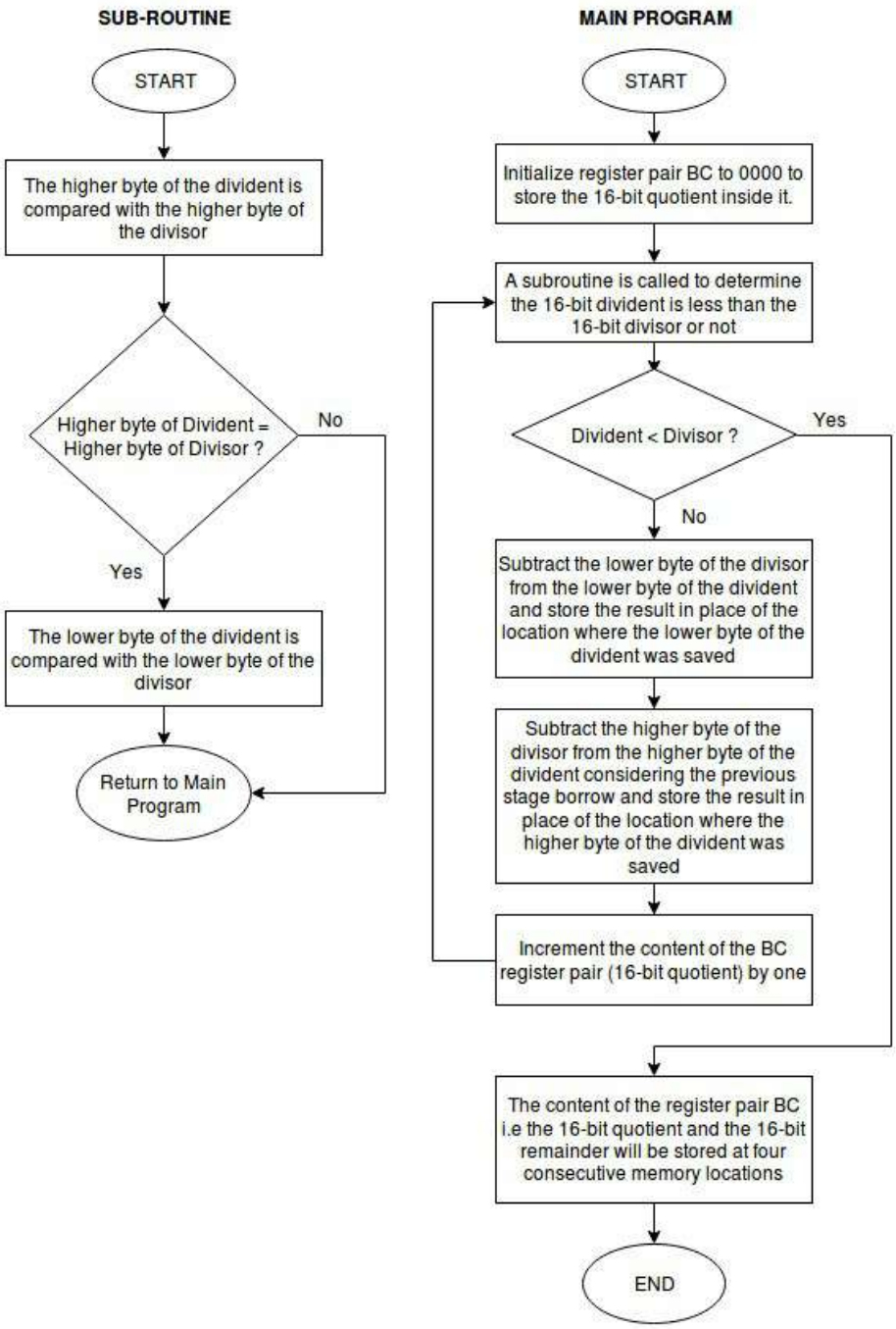


Fig-3.18: Flowchart of the program performing division between two 16-bit numbers

Assembly Language Program 3.10:

SL.	Addresses	Label	Mnemonics	Hex Codes			No. of Bytes	No. of T-States
1	8000		LXI B,0000	01	00	00	3	10
2	8003	LOOP	CALL COMP	CD	26	80	3	18
3	8006		JC SKIP	DA	1B	80	3	10 (True) / 7 (False)
4	8009		LXI D,8050	11	50	80	3	10
5	800C		LXI H,8052	21	50	80	3	10
6	800F		LDAX D	1A			1	7
7	8010		SUB M	96			1	7
8	8011		STAX D	12			1	7
9	8012		INX D	13			1	6
10	8013		INX H	23			1	6
11	8014		LDAX D	1A			1	7
12	8015		SBB M	9E			1	7
13	8016		STAX D	12			1	7
14	8017		INX B	03			1	6
15	8018		JMP LOOP	C3	03	80	3	10 (True) / 7 (False)
16	801B	SKIP	INX H	23			1	6
17	801C		MOV M,C	71			1	7
18	801D		INX H	23			1	6
19	801E		MOV M,B	70			1	7
20	801F		LHLD 8050	2A	50	80	3	16
21	8022		SHLD 8056	22	56	80	3	16
22	8025		HLT	76			1	5
23	8026	COMP	LXI D,8051	11	51	80	3	10
24	8029		LXI H,8053	21	53	80	3	10
25	802C		LDAX D	1A			1	7
26	802D		CMP M	BE			1	7
27	802E		JNZ NOEQU	C2	35	80	3	10 (True) / 7 (False)
28	8031		DCX D	1B			1	6

SL.	Addresses	Label	Mnemonics	Hex Codes			No. of Bytes	No. of T-States
29	8032		DCX H	2B			1	6
30	8033		LDAX D	1A			1	7
31	8034		CMP M	BE			1	7
32	8035	NOEQU	RET	C9			1	10
							TOTAL = 54	

Result of Program3.10:

SET1 ►

Input

Address	Content	Remarks
8050	FF	Lower Byte of Divident
8051	FC	Higher Byte of Divident
8052	FE	Lower Byte of Divisor
8053	01	Higher Byte of Divisor

Output

Address	Content	Remarks
8054	7E	Lower Byte of Quotient
8055	00	Higher Byte of Quotient
8056	FB	Lower Byte of Remainder
8057	01	Higher Byte of Remainder

SET2 ►

Input

Address	Content	Remarks
8050	AA	Lower Byte of Divident
8051	BB	Higher Byte of Divident
8052	55	Lower Byte of Divisor
8053	22	Higher Byte of Divisor

Output

Address	Content	Remarks
8054	05	Lower Byte of Quotient
8055	00	Higher Byte of Quotient
8056	01	Lower Byte of Remainder
8057	10	Higher Byte of Remainder

Exercise

1) Write a program to multiply an 8-bit number stored at memory location 8050H with 2, 4 and 8. Store the three results of multiplication starting from 8060 onward.

2) Write a program to add first ten natural numbers and store the result at memory location 9000H.

3) Write a program to add first ten odd numbers and store the result at memory location 8050H.

4) Write a program to add first ten even numbers and store the result at memory location 8050H.

5) Write a program to find out the sum of the following series and store the result of summation in DE register pair. 1 + 2 + 4 + 7 + 11 + …… up to 10 no. of terms

6) Write a program to count no. of 1s in an 8-bit binary number stored at memory location 8050.

7) Write a program to check whether an 8-bit number stored at memory location 8060 is odd or even. Store 0DH at memory location 8061 if the number is odd, otherwise store EEH at the same memory location.

8) Write a program to check a number stored at 8050H is divisible by 4 or not. If it is divisible by 4, store 01H at 8051H, otherwise store 00H at the same memory location.

9) Write a program to find out 1's complement of a number stored at memory location 8050H without using CMA instruction. [Hint: XRI FFH]

10) Write a program to determine the sum of two 8-bit numbers stored at memory locations 8050H and 8051H respectively without using ADD instruction.

11) Write a program to find out 2's complement of an 8-bit number stored at 9000H.

12) Write a program to find out 2's complement of a 16-bit number which is stored at memory location 9000H (lower byte of the number) and 9001H (higher byte of the number). The 2's complement of the number is to be stored at DE register pair.

13) Write a program to add two 32-bit binary numbers which are stored at the memory locations starting from 9050 onward and the memory locations starting from 9060 onwards. Store the result of the addition starting from memory location 9070 onward.

14) Write a program to check whether a number is positive or negative without using CMP/ CPI instruction.

15) Write a program to subtract two 8-bit numbers stored at memory locations 8050H and 8051 respectively using 2's complement.

16) Write a program to determine the value of 2^n where n is stored at memory location 8050H.

17) Write a program to swap the nibbles of an 8-bit number stored at memory location 8060H.

18) Write a program to find the mean of two 8-bit numbers stored at memory locations 8050H and 8051 respectively. Store the mean value at memory location 8052H. Consider both of the numbers either even or odd to get the mean value to be integer.

19) Write a program to determine the n^{th} term of an AP series, where the value of n, first term a and common difference d are stored at memory locations 8050H, 8051H and 8052H respectively. Store the n^{th} term at memory location 8053H.

20) Write a program to determine the half of an 8-bit even number stored at memory location 8050H. (Do not use division by 2)

21) Write a program to check whether a number stored at 8050H is equal or greater or less than 100. If the number is equal store EAH, if greater than store ABH and if less than store BEH at memory location 8051H, where EA represents EQUAL, AB represents ABOVE and BE represents BELOW.

22) Memory location 8050H stores the marks of a student out of 100. Write a program to store the Grade of the student depending upon the following criteria at memory location 8051H.
    1. $90 \leq$ Marks ≤ 100 → Grade O → Store 00H
    2. $80 \leq$ Marks < 90 → Grade E → Store EEH
    3. $70 \leq$ Marks < 80 → Grade A → Store AAH
    4. $60 \leq$ Marks < 70 → Grade B → Store BBH
    5. $50 \leq$ Marks < 60 → Grade C → Store CCH
    6. $40 \leq$ Marks < 50 → Grade D → Store DDH
    7. Marks < 40 → Grade F → Store FFH

4. Programs on Data Transfer and Data Separation

4.1: Write a program to transfer a block of ten data stored starting from the memory location 8050 onward to the memory location 8060 onward in forward direction.

This program basically performs the copy operation of a block of some data which are stored in the memory locations consecutively. Here a set of ten 8-bit numbers are to be transferred from one memory locations to another memory locations. The memory locations where the ten numbers are stored, is called source block and the memory locations where the ten numbers have to be transferred is called destination block. In this program the source block starts from the address 8050 to 8059 and the destination block starts from the address 8060 to 8069, which implies that the number of 8050 will be copied to 8060, the number of 8051 will be copied to 8061, the number of 8052 will be copied to 8062 and so on. Therefore the structure of the source block and the destination block before the execution of the program and after the execution of the program is shown in Fig-4.1 below for clear conception.

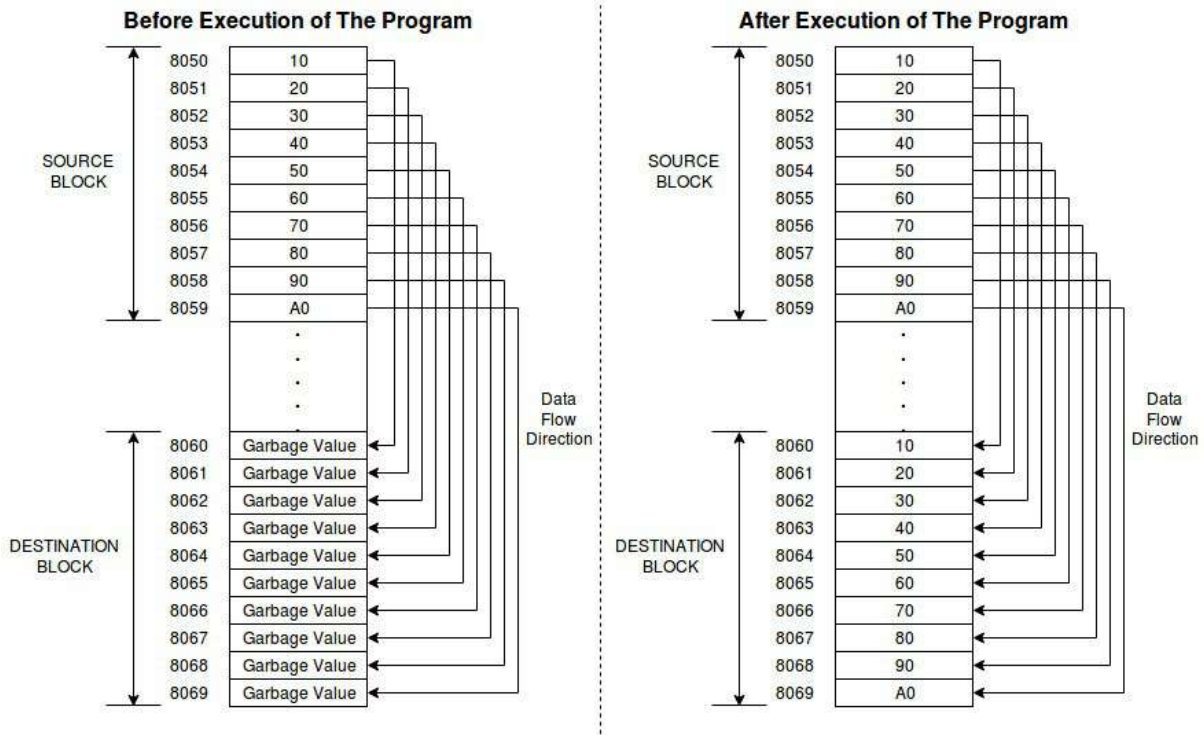


Fig-4.1: The source block and the destination block before and after execution of the program

It is being seen from the above figure that the numbers stored in the source block remain unchanged after the execution of the program. Hence it is exactly similar to the copy operation where source remains unchanged but destination is changed with the contents of the source. Now the flowchart of this program is shown in Fig-4.2.

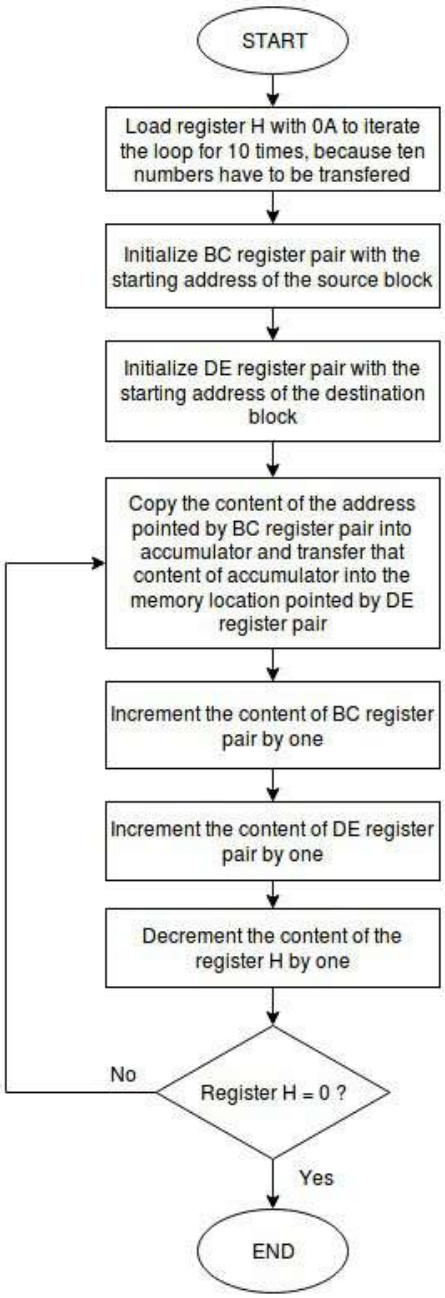


Fig-4.2: Flowchart of transfering a block of ten data from one locations to another locations

Assembly Language Program 4.1:

SL.	Addresses	Label	Mnemonics	Hex Codes			No. of Bytes	No. of T-States
1	8000		MVI H,0A	26	0A		2	7
2	8002		LXI B,8050	01	50	80	3	10
3	8005		LXI D,8060	11	60	80	3	10
4	8008	LOOP	LDAX B	0A			1	7
5	8009		STAX D	12			1	7
6	800A		INX B	03			1	6
7	800B		INX D	13			1	6
8	800C		DCR H	25			1	4
9	800D		JNZ LOOP	C2	08	80	3	10 (True) / 7 (False)
10	8010		HLT	76			1	5
							TOTAL = 17	

Result of Program 4.1:

SET1 ►

Input

Source Block

Mem. Address	Content	Remarks
8050	11	No1
8051	22	No2
8052	33	No3
8053	44	No4
8054	55	No5
8055	66	No6
8056	77	No7
8057	88	No8
8058	99	No9
8059	AA	No10

Output

Destination Block

Mem. Address	Content	Remarks
8060	11	No1
8061	22	No2
8062	33	No3
8063	44	No4
8064	55	No5
8065	66	No6
8066	77	No7
8067	88	No8
8068	99	No9
8069	AA	No10

4.2: Write a program to transfer a block of ten data stored starting from the memory location 8050 onward to the memory location 8055 onward in forward direction.

Although this program seems to be same as the previous program, but it is different from the first program. Because here the source block extends from the memory location 8050 to 8059 and the destination block extends from 8055 to 805E. Therefore some locations (8055 to 8059) of the source block are common to the destination block i.e. there is a overlapping region between the source block and the destination block. Now if we start to copy the numbers from the starting address of the source block to the starting address of the destination block, there will be a complete mishap, some numbers of the source block stored from 8055 to 8059 will be completely lost before they transfered to the destination block. Here our aim is to copy the contents of the entire source block to the destination block as it is, though the source block will not remain intact. What will happen if we follow the procedure of the first program, is shown pictorially in Fig-4.3 below.

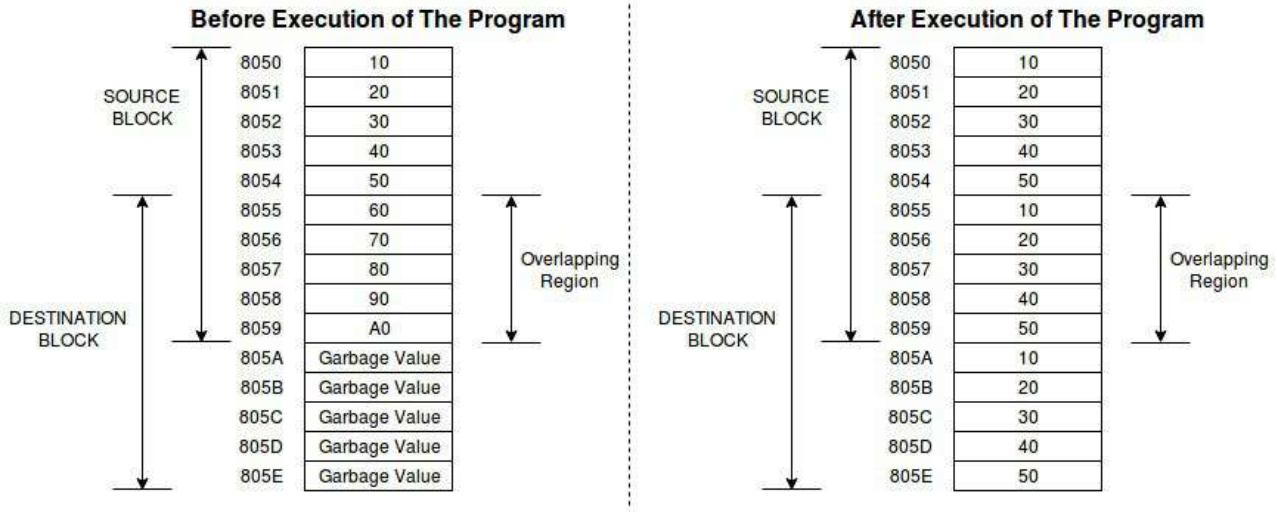


Fig-4.3: The source block and the destination block following the procedure of the first program

To solve the above problem, we have to start the copy operation from the last address of the source block to the last address of the destination block and go upward for the source block as well as the destination block to transfer the numbers one by one. The status of the source block and the destination block is shown pictorially before execution of the program and after execution of the program in Fig-4.4 below.

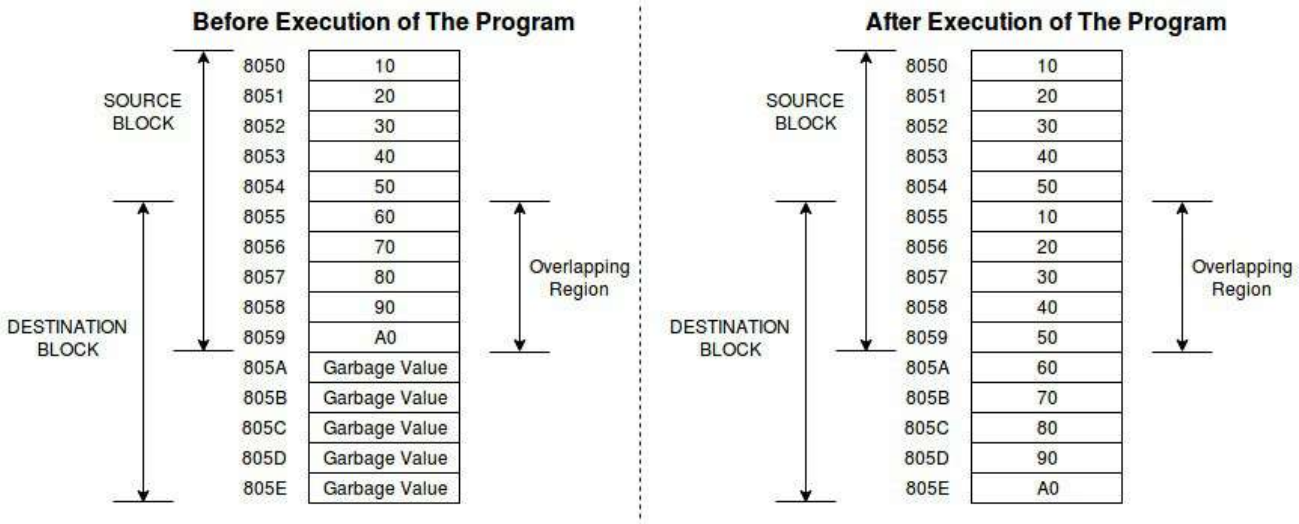


Fig-4.4: Status of the source block and the destination block following the modified procedure

Hence it is clear from the above figure that the whole data of the source block is now transfered successfully in the destination block. The flowchart of this program is shown in Fig-4.5.

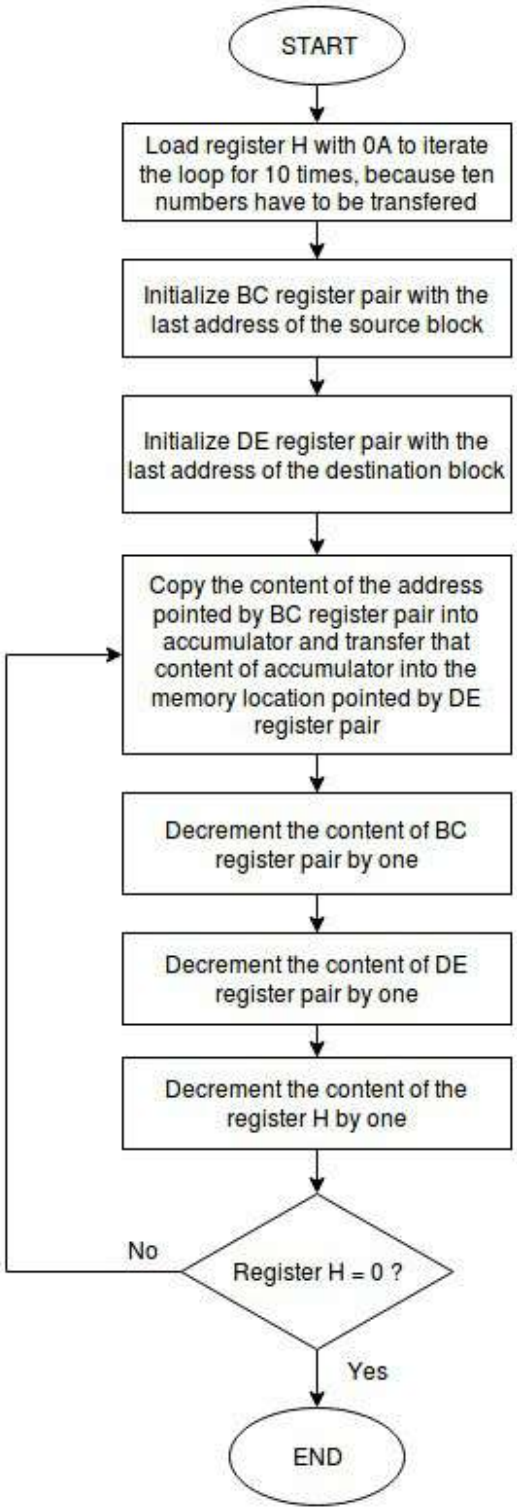


Fig-4.5: Flowchart of data transfer from source block to destination block with overlapping area

Assembly Language Program 4.2:

SL.	Addresses	Label	Mnemonics	Hex Codes			No. of Bytes	No. of T-States
1	8000		MVI H,0A	26	0A		2	7
2	8002		LXI B,8059	01	59	80	3	10
3	8005		LXI D,805E	11	5E	80	3	10
4	8008	LOOP	LDAX B	0A			1	7
5	8009		STAX D	12			1	7
6	800A		DCX B	0B			1	6
7	800B		DCX D	1B			1	6
8	800C		DCR H	25			1	4
9	800D		JNZ LOOP	C2	08	80	3	10 (True) / 7 (False)
10	8010		HLT	76			1	5
							TOTAL = 17	

Result of Program 4.2:

SET1 ►

Input

Source Block

Mem. Address	Content	Remarks
8050	11	No1
8051	22	No2
8052	33	No3
8053	44	No4
8054	55	No5
8055	66	No6
8056	77	No7
8057	88	No8
8058	99	No9
8059	AA	No10

Output

Destination Block

Mem. Address	Content	Remarks
8055	11	No1
8056	22	No2
8057	33	No3
8058	44	No4
8059	55	No5
805A	66	No6
805B	77	No7
805C	88	No8
805D	99	No9
805E	AA	No10

4.3: Write a program to separate positive numbers and negative numbers into two different memory blocks from a set of ten 8-bit signed numbers which are stored consecutively starting from the memory location 8050 onward. The positive block starts from 8060 onward and the negative block starts from 8070 onward, where positive count and negative count will be stored at the starting address of each block.

We know, if the MSB of a binary number is high, the number will be treated as negative number and if the MSB is low, the number is considered as positive number. So, the MSB of each of the ten 8-bit binary numbers which are stored at the source block starting from 8050 to 8059, is checked for high or low and is separated into two blocks of memory depending upon the status of MSB. The memory block which is storing the positive numbers, is called the positive block and the memory block which is holding the negative numbers, is called the negative block. The positive block starts from 8060 onward, where the first memory location 8060 holds the number of count of positive numbers i.e. how many positive numbers and all the positive numbers begins to be stored from 8061 onward. Similarly the negative block starts from 8070 onward, where the first location 8070 stores the number of count of negative numbers and all the negative numbers will be stored starting from the memory location 8071 onward. For better understanding we have taken a set of ten data and separated them accordingly.

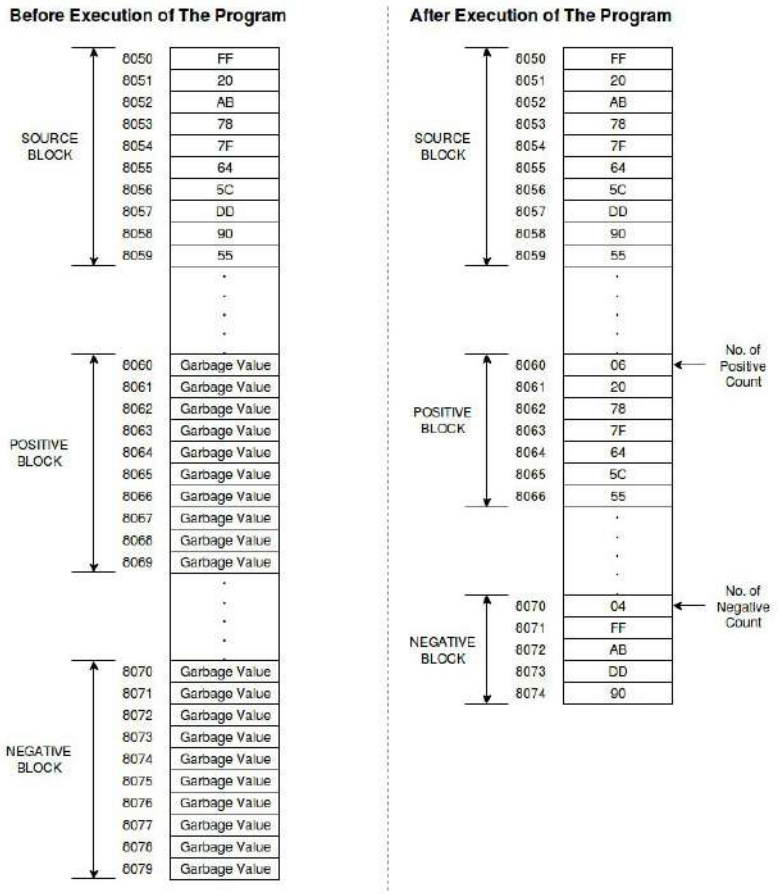


Fig-4.6: Source block, positive block and negative block before and after execution of the program

As there are three memory blocks named as Source block, Positive block and Negative block, three register pairs (HL, BC, DE) are used to point them. For example the source block, the positive block and the negative block are pointed by HL, BC and DE register pair respectively. Therefore all the general purpose resisters except accumulator are already used in this program and we have to perform all the jobs required in this program should be accomplished by using accumuilator only. That's why the memory location 804F, just before the strating address of the source block, is being used as a counter to iterate the loop for ten times to separate ten signed binary numbers. Moreover the starting address of the positive block and the starting address of the negative block are initialized with 00H to store the positive count and the negative count respectively. The flowchart of this program is given in Fig-4.7.

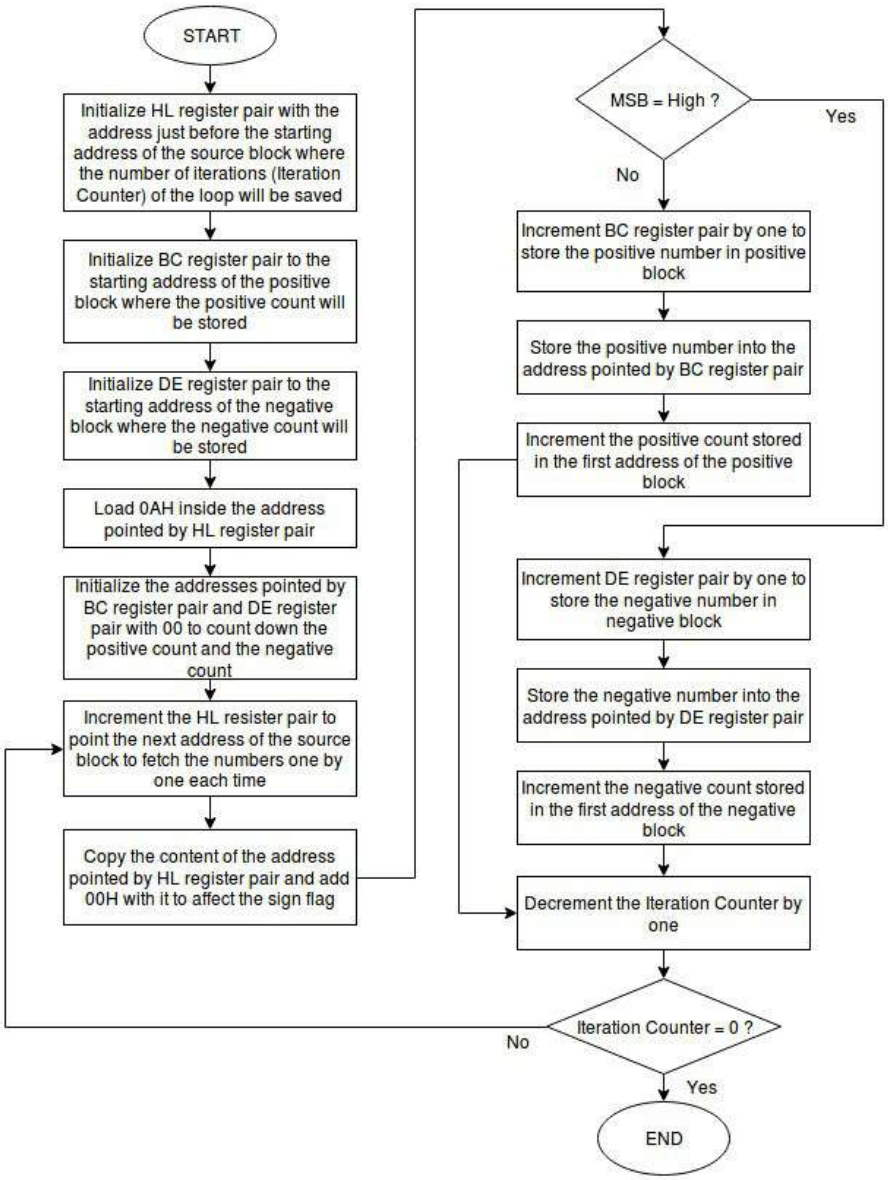


Fig-4.7: Flowchart of separating positive and negative numbers from a set of ten signed numbers

Assembly Language Program 4.3:

SL.	Addresses	Label	Mnemonics	Hex Codes			No. of Bytes	No. of T-States
1	8000		LXI H,804F	21	4F	80	3	10
2	8003		LXI B,8060	01	60	80	3	10
3	8006		LXI D,8070	11	70	80	3	10
4	8009		MVI A,0A	3E	0A		2	7
5	800B		MOV M,A	77			1	7
6	800C		XRA A	AF			1	4
7	800D		STAX B	02			1	7
8	800E		STAX D	12			1	7
9	800F	LOOP	INX H	23			1	6
10	8010		MOV A,M	7E			1	7
11	8011		ADI 00	C6	00		2	7
12	8013		JM NEGATIVE	FA	22	80	3	10 (True) / 7 (False)
13	8016		INX B	03			1	6
14	8017		STAX B	02			1	7
15	8018		LDA 8060	3A	60	80	3	13
16	801B		INR A	3C			1	4
17	801C		STA 8060	32	60	80	3	13
18	801F		JMP SKIP	C3	2B	80	3	10 (True) / 7 (False)
19	8022	NEGATIVE	INX D	13			1	6
20	8023		STAX D	12			1	7
21	8024		LDA 8070	3A	70	80	3	13
22	8027		INR A	3C			1	4
23	8028		STA 8070	32	70	80	3	13
24	802B	SKIP	LDA 804F	3A	4F	80	3	13
25	802E		DCR A	3D			1	4
26	802F		STA 804F	32	4F	80	3	13

SL.	Addresses	Label	Mnemonics	Hex Codes			No. of Bytes	No. of T-States
27	8032		JNZ LOOP	C2	0F	80	3	10 (True) / 7 (False)
28	8035		HLT	76			1	5
							TOTAL = 54	

Result of Program 4.3:

SET1 ►

Input

Source Block

Address	Content	Remarks
8050	05	No1
8051	0D	No2
8052	DD	No3
8053	AA	No4
8054	12	No5
8055	32	No6
8056	71	No7
8057	0A	No8
8058	8F	No9
8059	0A	No10

Output

Positive Block

Address	Content	Remarks
8060	07	Positive Count
8061	05	+No1
8062	0D	+No2
8063	12	+No5
8064	32	+No6
8065	71	+No7
8066	0A	+No8
8067	0A	+No10

Negative Block

Address	Content	Remarks
8070	03	Negative Count
8071	DD	-No3
8072	AA	-No4
8073	8F	-No9

Exercise

1) Write a program to transfer a block of ten data stored starting from the memory location 8050 onward to the memory location 8060 onward in reverse order.

2) Write a program to separate odd numbers and even numbers into two different memory blocks from a set of ten 8-bit numbers which are stored consecutively starting from the memory location 8050 onward. The block of odd numbers starts from 8060 onward and the block of even numbers starts from 8070 onward, where number of odd count and even count will be stored at the starting address of each block.

3) Suppose a set of ten 8-bit numbers are stored consecutively from memory location 8050H onward. Write a program to insert an element stored at memory location 804FH into the memory location 8053H.

4) Suppose a set of ten 8-bit numbers are stored consecutively from memory location 8050H onward. Write a program to delete the element which is stored at memory location 8055H.

5) Write a program to store AAH and BBH alternately for 100 times starting from memory location 9000H. Also store the last address where BBH is stored into DE register pair.

6) Write a program to store first ten natural numbers consecutively from memory location 8050H.

5. Programs on Searching and Sorting

5.1: Write a program to find the largest number from a list of ten 8-bit numbers which are stored from the memory location 8050 onward and store the largest number in register D.

If there are N no. of 8-bit numbers in a data set, then (N – 1) no. of comparisons should be performed taking two consecutive numbers at a time to find out the largest number. For each comparison the larger one among the two numbers will be stored in a register and ultimately we shall get the largest number saved inside that register after (N – 1) comparisons. Therefore nine comparisons will be done in this program, because here we have to find out the largest number from a set of ten numbers i.e. N = 10. During every comparison the larger one is to be stored inside accumulator which will hold the largest number after the completion of nine comparisons finally. For clear understanding, an example for determining the largest number from a set of six numbers in this way is given pictorially in Fig-5.1 below.

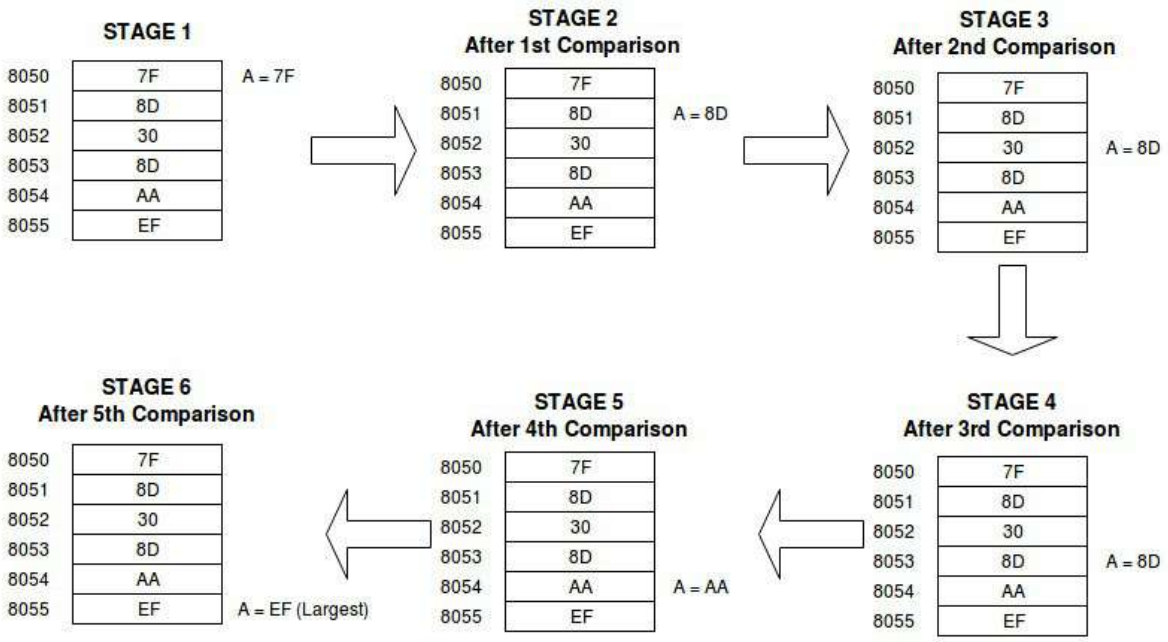


Fig-5.1: Sequences to find out the largest number from a set of six numbers

It is being seen from the above figure that the largest number (EF) is stored inside register A ultimately. The flowchart of the above mentioned program is given in Fig-5.2 below.

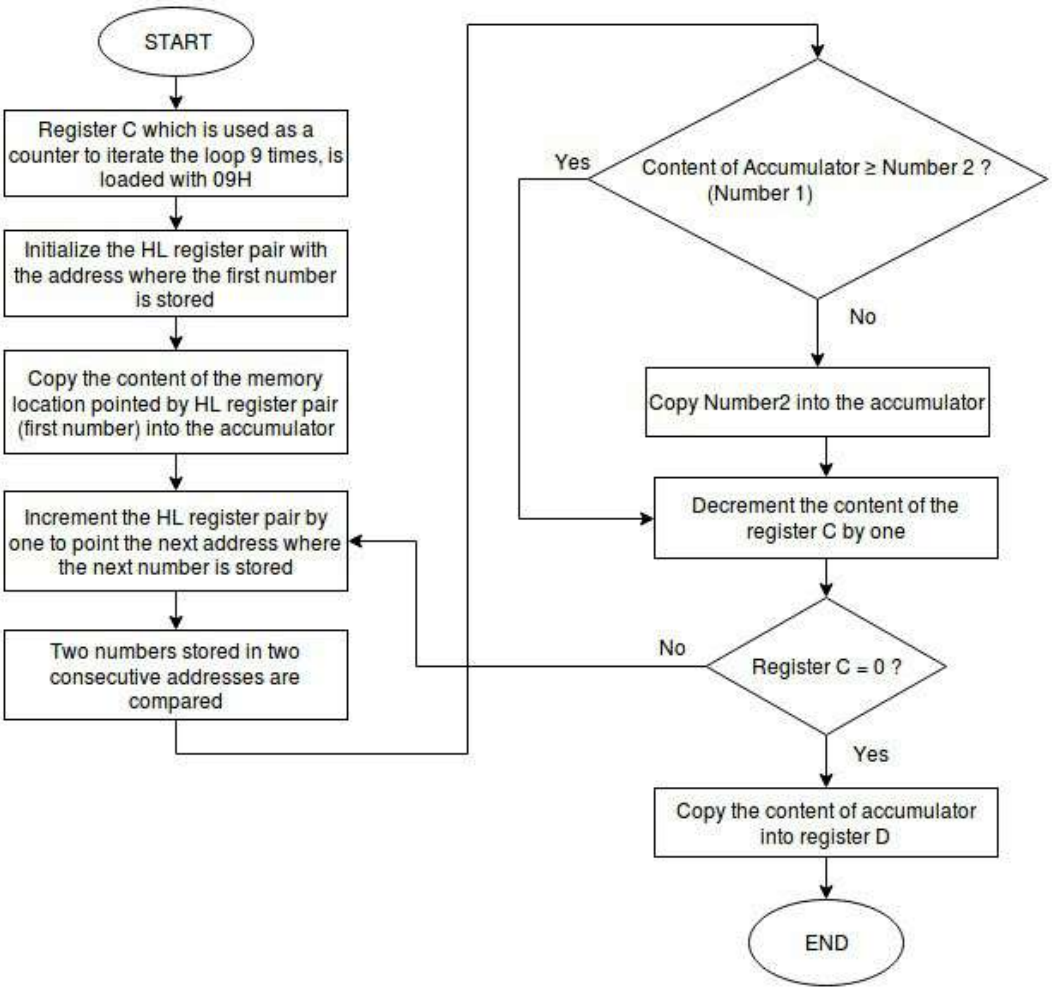


Fig-5.2: Flowchart of finding the largest number from a set of ten numbers

Assembly Language Program 5.1:

SL.	Addresses	Label	Mnemonics	Hex Codes			No. of Bytes	No. of T-States
1	8000		MVI C,09	0E	09		2	7
2	8002		LXI H,8050	21	50	80	3	10
3	8005		MOV A,M	7E			1	7
4	8006	LOOP	INX H	23			1	6
5	8007		CMP M	BE			1	7
6	8008		JNC SMALL	D2	0C	80	3	10 (True) / 7 (False)
7	800B		MOV A,M	7E			1	7
8	800C	SMALL	DCR C	0D			1	4
9	800D		JNZ LOOP	C2	06	80	3	10 (True) / 7 (False)
10	8010		MOV D,A	57			1	4
11	8011		HLT	76			1	5
							TOTAL = 18	

Result of Program 5.1:

SET1 ►

Input

Mem. Address	Content	Remarks
8050	05	No1
8051	0D	No2
8052	DD	No3
8053	AA	No4
8054	12	No5
8055	32	No6
8056	71	No7
8057	0A	No8
8058	8F	No9
8059	0A	No10

Output

D → DD (Largest No.)

5.2: Write a program to find the largest and the smallest number from a list of ten 8-bit numbers which are stored from the memory location 8050 onward and store the largest and the smallest numbers in register D and E respectively.

This program is a combination of Program 1 (To find the largest number) and Program 2 (To find the smallest number) where the largest and the smallest both numbers are determined simultaneously in a single program and stored inside the registers D and E respectively. Once the previous two programs are clearly understood, then it will be easy to understand this program. Therefore only the flowchart is sufficient to clarify the concept behind this program here. The flowchart of this program is given in Fig-5.3.

In this program two comparisons between two numbers are done consecutively, one for checking the larger number which will be stored in D register always and another for checking the smaller number which will be saved in E register always. Thus we get the largest number inside D register and the smallest number inside E register after the completion of the execution of the program.

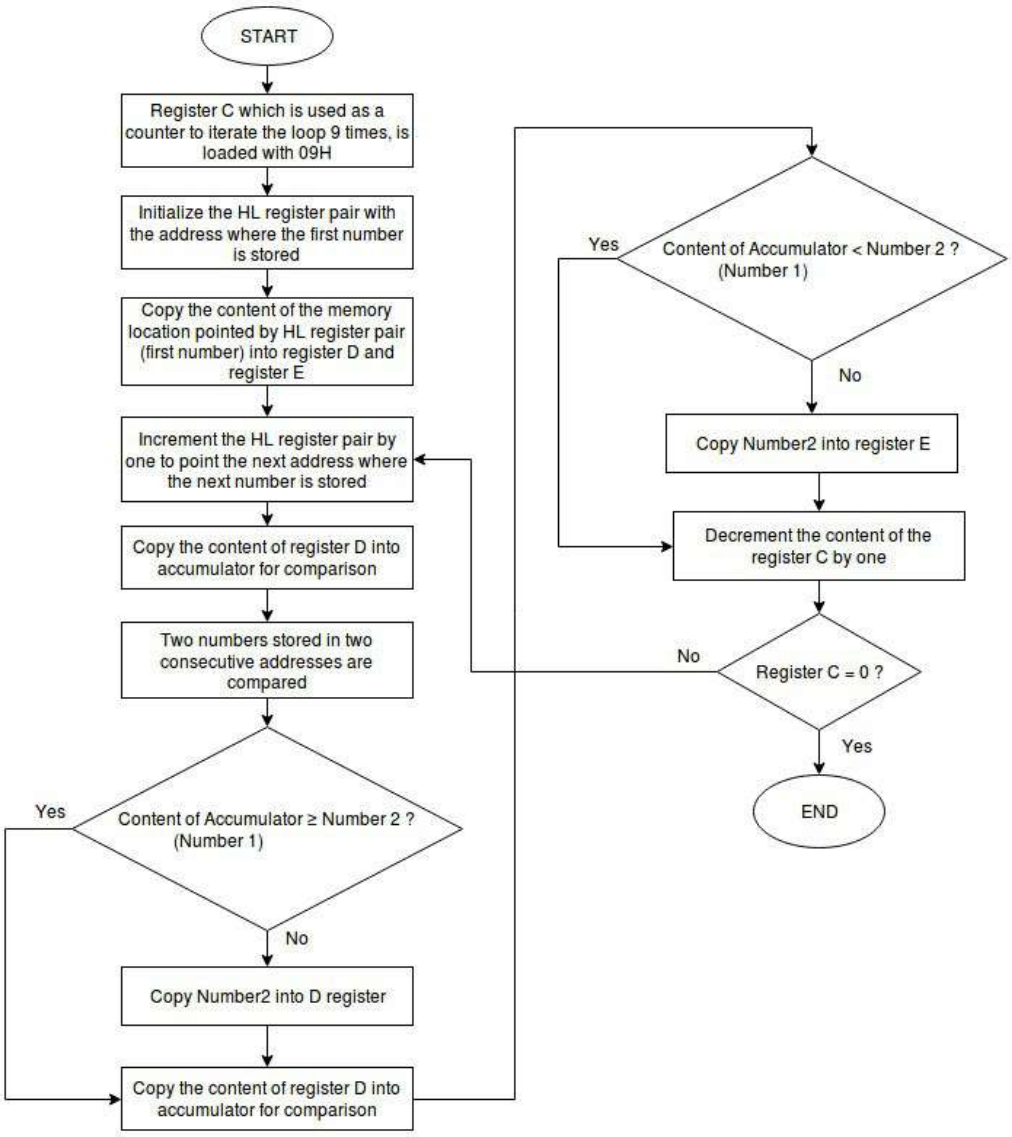


Fig-5.3: Flowchart of finding the largest and the smallest numbers from a set of ten numbers

Assembly Language Program 5.2:

SL.	Addresses	Label	Mnemonics	Hex Codes			No. of Bytes	No. of T-States
1	8000		MVI C,09	0E	09		2	7
2	8002		LXI H,8050	21	50	80	3	10
3	8005		MOV D,M	56			1	7
4	8006		MOV E,D	5A			1	4

SL.	Addresses	Label	Mnemonics	Hex Codes			No. of Bytes	No. of T-States
5	8007	LOOP	INX H	23			1	6
6	8008		MOV A,D	7A			1	4
7	8009		CMP M	BE			1	7
8	800A		JNC SMALL	D2	0E	80	3	10 (True) / 7 (False)
9	800D		MOV D,M	56			1	7
10	800E	SMALL	MOV A,E	7B			1	4
11	800F		CMP M	BE			1	7
12	8010		JC LARGE	DA	14	80	3	10 (True) / 7 (False)
13	8013		MOV E,M	5E			1	7
14	8014	LARGE	DCR C	0D			1	4
15	8015		JNZ LOOP	C2	07	80	3	10 (True) / 7 (False)
16	8018		HLT	76			1	5
							TOTAL = 25	

Result of Program 5.2:

SET1 ►

Input

Mem. Address	Content	Remarks
8050	05	No1
8051	0D	No2
8052	DD	No3
8053	AA	No4
8054	12	No5
8055	32	No6
8056	71	No7
8057	0A	No8
8058	8F	No9
8059	0A	No10

Output

D → DD (Largest No.)
E → 05 (Smallest No.)

5.3: Write a program to arrange a set of ten 8-bit numbers stored from the memory location 8050 onward in ascending order.

It is a program of sorting and in this case, the Bubble sort technique is used to arrange the numbers. In the scheme of Bubble sort, there will be (N – 1) no. of passes for N no. of 8-bit numbers and number of comparisons done between two consecutive numbers decreases by one for every pass. Comparisons among the two successive numbers are always started from the first number corresponding to all passes. If there are five numbers, for 1st pass there will be four comparisons, for 2nd pass there will be three comparisons, for 3rd pass two comparisons and for 4th pass single comparisons will be done. In each comparison, if first number is greater than the second one, they are interchanged i.e. the first number goes in the position of second number and the second number comes in the position of the first number. In this way the largest number will occupy the last position after the completion of 1st pass. Similarly the second largest number will be placed at the last but one position after the completion of 2nd pass. If this process continues, we get completely sorted numbers in ascending order after the completion of all the passes. Now it is necessary to take an example to sort five numbers in ascending order for better clarification shown in Fig-5..4.

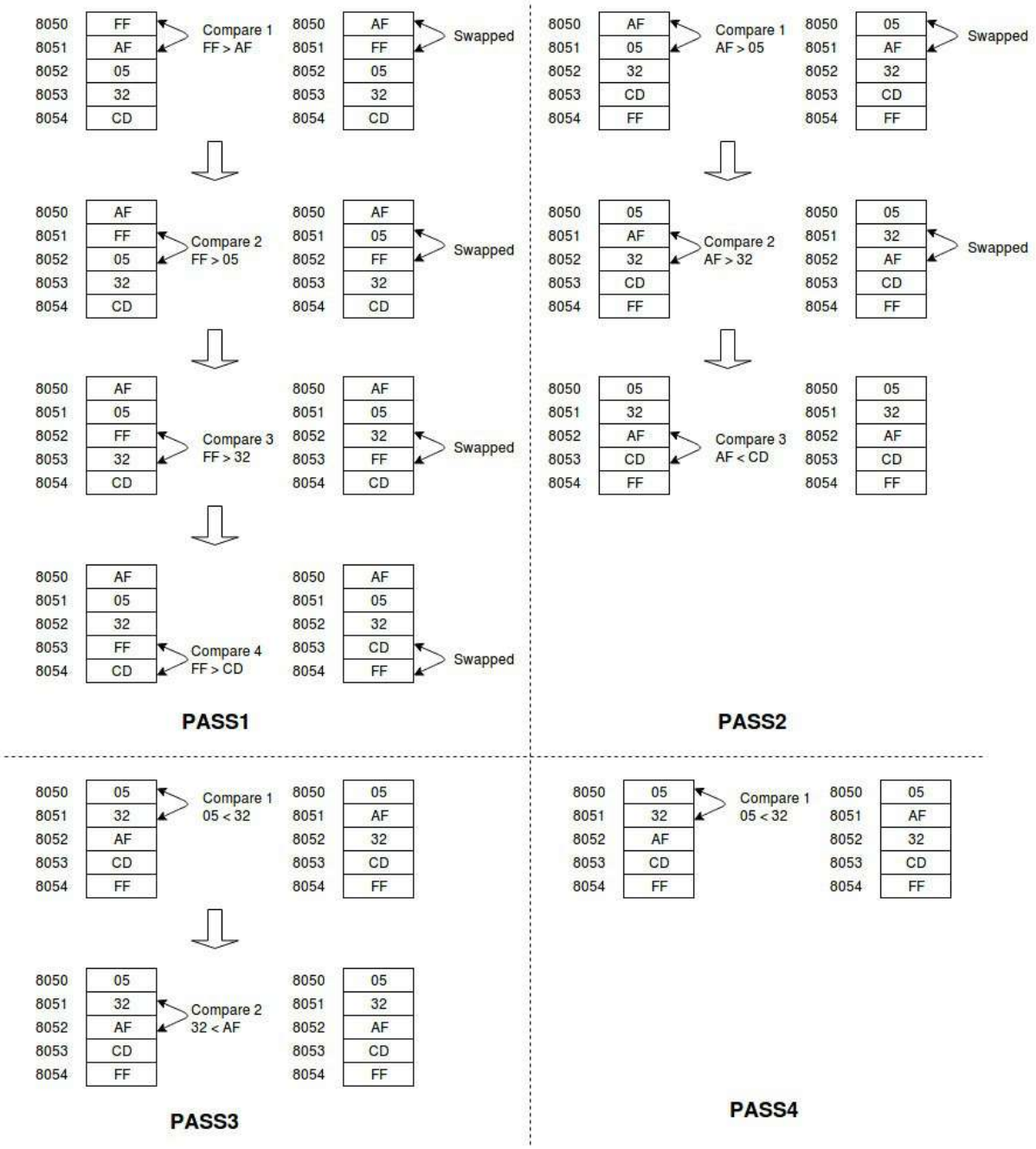


Fig-5.4: Sequences to find out the largest number from a set of six numbers

The flowchart of this program is shown in Fig-5.5 below.

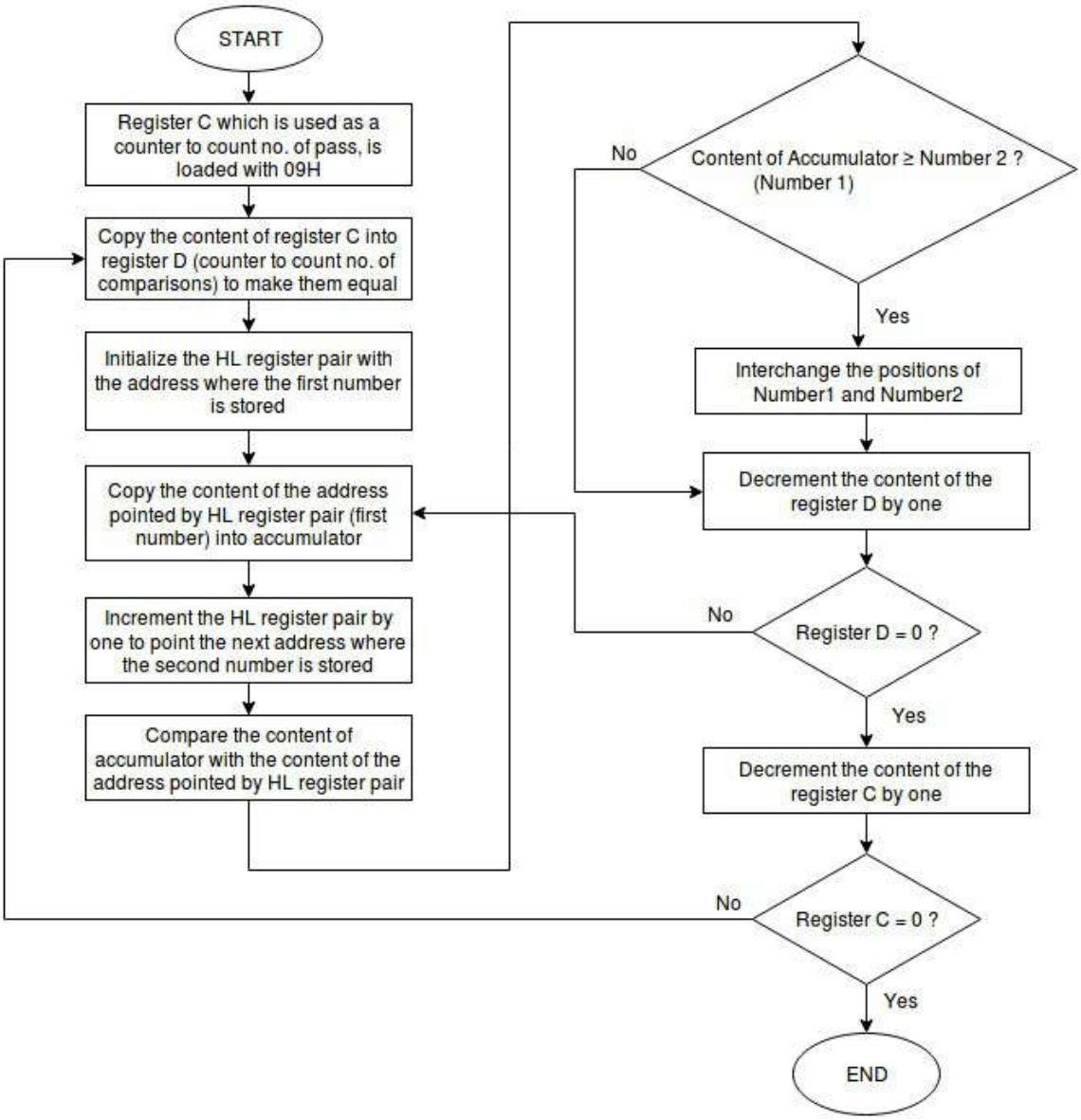


Fig-5.5: Flowchart of Bubble sort to arrange ten numbers in ascending order

Assembly Language Program 5.3:

SL.	Addresses	Label	Mnemonics	Hex Codes			No. of Bytes	No. of T-States
1	8000		MVI C,09	0E	09		2	7
2	8002	LOOP1	MOV D,C	51			1	4
3	8003		LXI H,8050	21	50	80	3	10
4	8006	LOOP2	MOV A,M	7E			1	7
5	8007		INX H	23			1	6
6	8008		CMP M	BE			1	7
7	8009		JC SKIP	DA	11	80	3	10 (True) / 7 (False)
8	800C		MOV B,M	46			1	7
9	800D		MOV M,A	77			1	7
10	800E		DCX H	2B			1	6
11	800F		MOV M,B	70			1	7
12	8010		INX H	23			1	6
13	8011	SKIP	DCR D	15			1	4
14	8012		JNZ LOOP2	C2	06	80	3	10 (True) / 7 (False)
15	8015		DCR C	0D			1	4
16	8016		JNZ LOOP1	C2	02	80	3	10 (True) / 7 (False)
17	8019		HLT	76			1	5
							TOTAL = 26	

Result of Program 5.3:

SET1 ►

Input

Before Sorting

Mem. Address	Content	Remarks
8050	05	
8051	0D	
8052	DD	
8053	AA	
8054	12	
8055	32	
8056	71	
8057	0A	
8058	8F	
8059	0A	

Output

After Sorting

Mem. Address	Content	Remarks
8050	05	
8051	0A	
8052	0A	
8053	0D	
8054	12	
8055	32	
8056	71	
8057	8F	
8058	AA	
8059	DD	

5.4: Suppose two sorted lists of ten and five numbers are stored starting from memory location 8060H onward and 8070H onward respectively. Write a program to merge these two sorted lists into a separate list in such a way that the generated list also will be in sorted form and will be stored from 8080H onward. Assume all the lists are sorted in ascending order in this program.

In this case 1st sorted list is stored from 8060H and 2nd sorted list is stored from 8070H. If the 1st and 2nd list consist of m and n no. of elements, the 3rd list after merging will consist (m + n) no. of elements. Here one element from the 1st list and another element from the 2nd list will be compared to each other. Between these two elements which one is smaller will be copied into the 3rd list. Thus this procedure will continue until any one list becomes exhausted i.e. all the elements of any one list are transferred to the 3rd list. After this, the remaining elements of the other list will be copied to 3rd list consecutively until it becomes exhausted. Finally the 3rd list of (m + n) elements thus formed starting from memory location 8080H, becomes automatically sorted in ascending order. The above mentioned procedure is explained pictorially as shown below with two lists of 5 and 2 elements respectively.

Iteration 1:

1st Sorted List

Address	Content
8060	05 (Smaller)
8061	0D
8062	DD
8063	DF
8064	EE

↔

2nd Sorted List

Address	Content
8070	0A
8071	32

3rd Sorted List

Address	Content
8080	05

Iteration 2:

1st Sorted List

Address	Content
8060	05
8061	0D
8062	DD
8063	DF
8064	EE

↔

2nd Sorted List

Address	Content
8070	0A (Smaller)
8071	32

3rd Sorted List

Address	Content
8080	05
8081	0A

Iteration 3:

1st Sorted List

Address	Content
8060	05
8061	0D (Smaller)
8062	DD
8063	DF
8064	EE

↔

2nd Sorted List

Address	Content
8070	0A
8071	32

3rd Sorted List

Address	Content
8080	05
8081	0A
8082	0D

Iteration 4:

1st Sorted List

Address	Content
8060	05
8061	0D
8062	DD
8063	DF
8064	EE

↔

2nd Sorted List	
Address	Content
8070	0A
8071	32 (Smaller)

2nd Sorted List is exhausted

3rd Sorted List

Address	Content
8080	05
8081	0A
8082	0D
8083	32

Iteration 5:

1st Sorted List	
Address	Content
8060	05
8061	0D
8062	DD
8063	DF
8064	EE

2nd Sorted List	
Address	Content
8070	0A
8071	32

3rd Sorted List

Address	Content
8080	05
8081	0A
8082	0D
8083	32
8084	DD

Iteration 6:

1st Sorted List	
Address	Content
8060	05
8061	0D
8062	DD
8063	DF
8064	EE

2nd Sorted List	
Address	Content
8070	0A
8071	32

3rd Sorted List

Address	Content
8080	05
8081	0A
8082	0D
8083	32
8084	DD
8085	DF

Iteration 7:

1st Sorted List	
Address	Content
8060	05
8061	0D
8062	DD
8063	DF
8064	EE

2nd Sorted List	
Address	Content
8070	0A
8071	32

3rd Sorted List

Address	Content
8080	05
8081	0A
8082	0D
8083	32
8084	DD
8085	DF
8086	EE

Note: Gray colored cells are indicating that they have already been transferred to destination memory locations.

In this program register pair BC and HL acts as memory pointer of 1st sorted list and 2nd sorted list respectively and DE register pair is the memory pointer of 3rd merged list. Three memory locations (805FH, 806FH and 807FH) will be used as counters of 1st, 2nd and 3rd list respectively. After every comparison the smaller element will be added to the 3rd list and the memory pointer DE of 3rd list along with any one memory pointer (either BC or HL register pair) will be incremented by 1 to get access of the next memory location. This process will be repeated until any one counter of 1st or 2nd list becomes zero. As soon as the particular counter of one list becomes zero, the remaining elements of the other list will be added to the 3rd list one by one. Thus a merged 3rd list whose all the elements are arranged in ascending order is formed ultimately.

Assembly Language Program 5.4:

SL.	Addresses	Label	Mnemonics	Hex Codes			No. of Bytes	No. of T-States
1	8000		LXI B,8060	01	60	80	3	10
2	8003		LXI H,8070	21	70	80	3	10
3	8006		LXI D,8080	11	80	80	3	10
4	8009		MVI A,0A	3E	0A		2	7
5	800B		STA 805F	32	5F	80	3	13
6	800E		MVI A,05	3E	05		2	7
7	8010		STA 806F	32	6F	80	3	13
8	8013		MVI A,0F	3E	0F		2	7
9	8015		STA 807F	32	7F	80	3	13

SL.	Addresses	Label	Mnemonics	Hex Codes			No. of Bytes	No. of T-States
10	8018	REPEAT	LDA 805F	3A	5F	80	3	13
11	801B		CPI 00	FE	00		2	7
12	801D		JZ L1	CA	3B	80	3	10/7
13	8020		LDA 806F	3A	6F	80	3	13
14	8023		CPI 00	FE	00		2	7
15	8025		JZ L2	CA	2D	80	3	10/7
16	8028		LDAX B	0A			1	7
17	8029		CMP M	BE			1	7
18	802A		JNC L1	D2	3B	80	3	10/7
19	802D	L2	LDAX B	0A			1	7
20	802E		STAX D	12			1	7
21	802F		INX B	03			1	6
22	8030		INX D	13			1	6
23	8031		LDA 805F	3A	5F	80	3	13
24	8034		DCR A	3D			1	4
25	8035		STA 805F	32	5F	80	3	13
26	8038		JMP END	C3	46	80	3	10
27	803B	L1	MOV A,M	7E			1	7
28	803C		STAX D	12			1	7
29	803D		INX H	23			1	6
30	803E		INX D	13			1	6
31	803F		LDA 806F	3A	6F	80	3	13
32	8042		DCR A	3D			1	4
33	8043		STA 806F	32	6F	80	3	13
34	8046	END	LDA 807F	3A	7F	80	3	13
35	8049		DCR A	3D			1	4
36	804A		STA 807F	32	7F	80	3	13
37	804D		JNZ REPEAT	C2	18	80	3	10/7
38	8050		HLT	76			1	5
							TOTAL = 81	

Result of Program 5.4:

SET1 ►

Input

1st Sorted List

Address	Content
8060	05
8060	0D
8062	A5
8063	AA
8064	AF
8065	B1
8066	CC
8067	D6
8068	DA
8069	DD

2nd Sorted List

Address	Content
8070	0A
8071	1F
8072	32
8073	A9
8074	B9

Output

3rd Sorted List

Address	Content
8080	05
8081	0A
8082	0D
8083	1F
8084	32
8085	A5
8086	A9
8087	AA
8088	AF
8089	B1
808A	B9
808B	CC
808C	D6
808D	DA
808E	DD

Exercise

1) Write a program to find the smallest number from a list of sixteen 8-bit numbers which are stored from the memory location 8050 onward and store the smallest number in register E.

2) Write a program to arrange a set of ten 8-bit numbers stored from the memory location 8050 onward in descending order using bubble sort.

3) Write a program to determine the number of times FF present in a set of 20 8-bit numbers which are stored from memory location 9000H.

4) Write a program to count the number of times 55H repeated in a set of 20 numbers stored consecutively starting from 8060 onward. Store the count value at the memory location 805FH.

5) Suppose two sorted lists of eight and five numbers are stored in ascending order starting from memory location 9000H onward and 9020H onward respectively. Write a program to merge these two sorted lists into a separate list in such a way that the generated list will be in descending order and will be stored from 9050H onward.

6. Programs on Data Conversion

6.1: Write a program to convert a 2-digit packed BCD number stored at memory location 8050H to unpacked BCD numbers which will be stored at memory locations 8051H and 8052H.

We know that a 2-digit packed BCD number is 8 bits long where lower 4 bits (lower nibble) forms LSD (Least significant digit) and upper 4 bits (upper nibble) forms MSD (Most significant digit). Now these two digits should be separated to form two unpacked BCD numbers. For example – 52 is a packed BCD and the corresponding unpacked BCD numbers are 05 and 02.

Now to extract out the LSD the packed BCD should be AND operated with 0FH. On the contrary the MSD will be separated after performing AND operation with F0H and the result of AND operation has to be shifted right 4 times. How a packed BCD 52H will be converted to unpacked BCDs are shown below.

	B_7	B_6	B_5	B_4	B_3	B_2	B_1	B_0
2-digit packed BCD (52H) →	0	1	0	1	0	0	1	0
0FH →	0	0	0	0	1	1	1	1
Bitwise AND operation →	---	---	---	---	---	---	---	---
Unpacked BCD with LSD (02H) →	0	0	0	0	0	0	1	0

	B_7	B_6	B_5	B_4	B_3	B_2	B_1	B_0
2-digit packed BCD (52H) →	0	1	0	1	0	0	1	0
F0H →	1	1	1	1	0	0	0	0
Bitwise AND operation →	---	---	---	---	---	---	---	---
Result of AND operation (50H) →	0	1	0	1	0	0	0	0
After 1st right shift →	0	0	1	0	1	0	0	0
After 2nd right shift →	0	0	0	1	0	1	0	0
After 3rd right shift →	0	0	0	0	1	0	1	0
Unpacked BCD with MSD (05H) → (After 4th right shift)	0	0	0	0	0	1	0	1

Assembly Language Program 6.1:

SL.	Addresses	Label	Mnemonics	Hex Codes			No. of Bytes	No. of T-States
1	8000		LXI H,8050	21	50	80	3	10
2	8003		MOV A,M	7E			1	7
3	8004		MOV B,A	47			1	4
4	8005		ANI 0F	E6	0F		2	7
5	8007		INX H	23			1	6
6	8008		MOV M,A	77			1	7
7	8009		MOV A,B	78			1	4
8	800A		ANI F0	E6	F0		2	7
9	800C		RRC	0F			1	4
10	800D		RRC	0F			1	4
11	800E		RRC	0F			1	4
12	800F		RRC	0F			1	4
13	8010		INX H	23			1	6
14	8011		MOV M,A	77			1	7
15	8012		HLT	76			1	5
							TOTAL = 19	

Result of Program 6.1:

SET1 ►

Input

Mem. Address	Content	Remarks
8050	52	2 digit packed BCD

Output

Address	Content	Remarks
8051	02	Unpacked BCD with LSD
8052	05	Unpacked BCD with MSD

SET2 ►

Input

Mem. Address	Content	Remarks
8050	94	2 digit packed BCD

Output

Address	Content	Remarks
8051	04	Unpacked BCD with LSD
8052	09	Unpacked BCD with MSD

6.2: Write a program to convert two unpacked BCD numbers stored at memory locations 9050H and 9051H to a two digits packed BCD number which will be stored at memory locations 9052H. Assume that the memory locations 9050H and 9051H is holding the unpacked BCD containing MSD and the unpacked BCD containing LSD respectively.

In this program two unpacked BCD numbers – one containing LSD and other containing MSD are joined together to a two digits packed BCD numbers. To do this the unpacked BCD consisting of MSD are shifted left for four times and then it will be OR-operated with the unpacked BCD consisting of LSD to construct the packed BCD number. Two unpacked BCD numbers 04 (LSD) and 08 (MSD) are converted to 2-digit packed BCD using the following technique as shown below.

	B_7	B_6	B_5	B_4	B_3	B_2	B_1	B_0
Unpacked BCD containing MSD (08H) →	0	0	0	0	1	0	0	0
After 1st left shift →	0	0	0	1	0	0	0	0
After 2nd left shift →	0	0	1	0	0	0	0	0
After 3rd left shift →	0	1	0	0	0	0	0	0
After 4th left shift →	1	0	0	0	0	0	0	0
Unpacked BCD containing LSD (04H) →	0	0	0	0	0	1	0	0
Bitwise OR operation →	---	---	---	---	---	---	---	---
2-digit Packed BCD (84H) →	1	0	0	0	0	1	0	0

Assembly Language Program 6.2:

SL.	Addresses	Label	Mnemonics	Hex Codes			No. of Bytes	No. of T-States
1	8000		LXI H,9050	21	50	90	3	10
2	8003		MOV A,M	7E			1	7
3	8004		RLC	07			1	4
4	8005		RLC	07			1	4
5	8006		RLC	07			1	4
6	8007		RLC	07			1	4
7	8008		INX H	23			1	6
8	8009		ORA M	B6			1	7
9	800A		INX H	23			1	6
10	800B		MOV M,A	77			1	7
11	800C		HLT	76			1	5
							TOTAL = 13	

Result of Program 6.2:

SET1 ►

Input

Address	Content	Remarks
9050	08	Unpacked BCD containing MSD
9051	04	Unpacked BCD containing LSD

Output

Address	Content	Remarks
9052	84	Packed BCD

SET2 ►

Input

Address	Content	Remarks
9050	06	Unpacked BCD containing MSD
9051	09	Unpacked BCD containing LSD

Output

Address	Content	Remarks
9052	69	Packed BCD

6.3: Write a program to convert a 2-digit packed BCD number stored at memory location 8050H to its equivalent Hexadecimal number and store the converted Hexadecimal number into memory location 8051H.

Method 1: The two digit BCD number is converted to two unpacked BCD numbers first. For example if the packed BCD number is 25, the unpacked BCD numbers will be 02 and 05 respectively, where 02 is MSD (Most significant digit) and 05 is LSD (Least significant digit). Here basically the two digits are separated and LSD is added with 10 times of MSD to get the equivalent Hexadecimal number. Therefore Hexadecimal number = 10 × MSD + LSD.

In this program 10 × MSD is stored in register D and LSD is stored in register C. Finally register D and register C are added together to get the Hexadecimal number. Basically 10 × MSD = 8 × MSD + 2 × MSD. If a number is shifted left 3 times, it will be multiplied with 8 and if a number is shifted left 1 time, it will be multiplied with 2. Here initially MSD is in the upper nibble and the lower nibble is zero. If it is shifted right 1 time, it is equivalent to shifting left 3 times for getting 8 × MSD and if it is shifted right 3 times to get 2 × MSD. Here the number masked with F0H is shifted right one time to get 8 × MSD and shifted right 3 times to get 2 × MSD. Let's take an example.

The packed BCD number = 25
Masked with 0F = 05 and masked with F0 = 20 = 0010 0000
After shifted right 1 time = 0001 0000 = 16 = 8 × 2
After shifted right 3 times = 0000 0100 = 4 = 2 × 2
Now 10 × 2 = 8 × 2 + 2 × 2
Therefore equivalent HEX number = 10 × 2 + 05

Assembly Language Program 6.3 (Method 1):

SL.	Addresses	Label	Mnemonics	Hex Codes			No. of Bytes	No. of T-States
1	8000		LXI H,8050	21	50	80	3	10
2	8003		MOV A,M	7E			1	7
3	8004		MOV B,A	47			1	4
4	8005		ANI 0F	E6	0F		2	7
5	8007		MOV C,A	4F			1	4
6	8008		MOV A,B	78			1	4
7	8009		ANI F0	E6	F0		2	7
8	800B		RRC	0F			1	4
9	800C		MOV D,A	57			1	4
10	800D		RRC	0F			1	4
11	800E		RRC	0F			1	4
12	800F		ADD D	82			1	4
13	8010		ADD C	81			1	4
14	8011		INX H	23			1	6
15	8012		MOV M,A	77			1	7
16	8013		HLT	76			1	5
							TOTAL = 20	

Method 2: In this alternate method the equivalent Hexadecimal number is achieved in the same way, only the difference in the technique to get 10 × MSD. Here packed BCD number is unpacked first to store MSD and LSD into two separate registers. Suppose register A is holding MSD. Now A is added with itself to hold 2 × MSD, then A is added with itself once again to store 4 × MSD and finally A is added with itself to get 8 × MSD. Now 8 × MSD + 2 × MSD is performed to have 10 × MSD which is ultimately added with LSD to get the equivalent Hexadecimal number.

Assembly Language Program 6.3(Method 2):

SL.	Addresses	Label	Mnemonics	Hex Codes			No. of Bytes	No. of T-States
1	8000		LXI H,8050	21	50	80	3	10
2	8003		MOV A,M	7E			1	7
3	8004		MOV B,A	47			1	4

SL.	Addresses	Label	Mnemonics	Hex Codes			No. of Bytes	No. of T-States
4	8005		ANI 0F	E6	0F		2	7
5	8007		MOV C,A	4F			1	4
6	8008		MOV A,B	78			1	4
7	8009		ANI F0	E6	F0		2	7
8	800B		RRC	0F			1	4
9	800C		RRC	0F			1	4
10	800D		RRC	0F			1	4
11	800E		RRC	0F			1	4
12	800F		ADD A	87			1	4
13	8010		MOV D,A	57			1	4
14	8011		ADD A	87			1	4
15	8012		ADD A	87			1	4
16	8013		ADD D	82			1	4
17	8014		ADD C	81			1	4
18	8015		INX H	23			1	6
19	8016		MOV M,A	77			1	7
20	8017		HLT	76			1	5
							TOTAL = 24	

Result of Program 6.3:

SET1 ►

Input

Mem. Address	Content	Remarks
8050	91	2 digit packed BCD

Output

Mem. Address	Content	Remarks
8051	5B	Equivalent Hex No.

SET2 ►

Input

Mem. Address	Content	Remarks
8050	15	2 digit packed BCD

Output

Mem. Address	Content	Remarks
8051	0F	Equivalent Hex No.

6.4: Write a program to convert an 8-bit Hexadecimal number stored at memory location 8050H to unpacked BCDs which will be stored starting from memory location 8051H.

Method 1: In this case the Hexadecimal number is converted to unpacked BCDs i.e. three digits are separated and saved into three different memory locations. For example – if the Hexadecimal number is FEH (254 in Decimal), then three unpacked BCD digits 02, 05 and 04 will be stored into three consecutive memory locations. For this purpose the Hexadecimal number is divided by 100 (64 in HEX) first, where quotient gives the 1st unpacked BCD. The remainder is again divided by 10 (0A in HEX) to get 2nd unpacked BCD in the quotient and 3rd unpacked BCD in the remainder. These three unpacked BCDs are stored consecutively in the memory locations starting from 8051.

Assembly Language Program 6.4(Method 1):

SL	Addresses	Label	Mnemonics	Hex Codes			No. of Bytes	No. of T-States
1	8000		LXI H,8050	21	50	80	3	10
2	8003		MOV A,M	7E			1	7
3	8004		MVI B,64	06	64		2	7
4	8006		CALL HEX2BCD	CD	11	80	3	18
5	8009		MVI B,0A	06	0A		2	7
6	800B		CALL HEX2BCD	CD	11	80	3	18
7	800E		INX H	23			1	6
8	800F		MOV M,A	77			1	7
9	8010		HLT	76			1	5
10	8011	HEX2BCD	INX H	23			1	6
11	8012		MVI M,FF	36	FF		2	10
12	8014	LOOP	INR M	34			1	10
13	8015		SUB B	90			1	4
14	8016		JNC LOOP	D2	14	80	3	10
15	8019		ADD B	80			1	4
16	801A		RET	C9			1	10
							TOTAL= 27	

Result of Program 6.4 (Method 1):

SET1 ►

Input

Mem. Address	Content	Remarks
8050	FD	2 digit Hex No.

Hex No. = FD Equivalent Decimal No. = 253

Output

Mem. Address	Content	Remarks
8051	02	Unpacked BCD1
8052	05	Unpacked BCD2
8053	03	Unpacked BCD3

SET2 ►

Input

Mem. Address	Content	Remarks
8050	E1	2 digit Hex No.

Hex No. = E1 Equivalent Decimal No. = 225

Output

Mem. Address	Content	Remarks
8051	02	Unpacked BCD1
8052	02	Unpacked BCD2
8053	05	Unpacked BCD3

Method 2: In this method the Hexadecimal number is converted to packed BCD number using DAA instruction. That means, if the Hexadecimal number is AFH (175 in Decimal), after conversion we have two packed BCD numbers, one is 01 and other is 75 which will be stored successively into two memory locations 8051H and 8052H. We know DAA converts the result of two BCD addition into BCD. In 8085 all the numbers are represented in Hexadecimal form i.e. if we want to represent BCD number, it is basically a HEX number. For example BCD number 15 is basically a Hexadecimal number whose value is 21 and BCD number 18 is another Hexadecimal number with value 24. If we add them, then we get the following results.

BCD Addition	We get the following
15	15
18	18
--------	-------
33	2D
Desired Result	Wrong Result

From the above example it is clear that the result of the BCD addition may be incorrect. DAA instruction rectifies this error and generate the correct result in BCD. In the above example if DAA is used after the addition, it will give 33 as a result. In this program if a Hexadecimal number is n, it will be converted to the corresponding BCD number by initializing accumulator with 00H, adding 1 with itself n times and using DAA after every addition. Thus the accumulator will hold the packed BCD number finally. Here one thing is important to mention that DAA instruction is used after ADD instruction normally.

Assembly Language Program 6.4(Method 2):

SL	Addresses	Label	Mnemonics	Hex Codes			No. of Bytes	No. of T-States
1	8000		LXI H,8050	21	50	80	3	10
2	8003		MOV C,M	4E			1	7
3	8004		MVI B,00	06	00		2	7
4	8006		XRA A	AF			1	4
5	8007	LOOP	ADI 01	C6	01		2	7
6	8009		DAA	27			1	4
7	800A		JNC SKIP	D2	0E	80	3	10
8	800D		INR B	04			1	4
9	800E	SKIP	DCR C	0D			1	4
10	800F		JNZ LOOP	C2	07	80	3	10
11	8012		INX H	23			1	6
12	8013		MOV M,B	70			1	7
13	8014		INX H	23			1	6
14	8015		MOV M,A	77			1	7
15	8016		HLT	76			1	5
							TOTAL= 23	

Result of Program 6.4 (Method 2):

SET1 ►

Input

Mem. Address	Content	Remarks
8050	FD	2 digit Hex No.

Hex No. = FD Equivalent Decimal No. = 253

Output

Mem. Address	Content	Remarks
8051	02	Higher Byte of BCD
8052	53	Lower Byte of BCD

SET2 ►

Input

Mem. Address	Content	Remarks
8050	E1	2 digit Hex No.

Hex No. = E1 Equivalent Decimal No. = 225

Output

Mem. Address	Content	Remarks
8051	02	Higher Byte of BCD
8052	25	Lower Byte of BCD

6.5: Write a program to add two BCD numbers stored at memory locations 8050H and 8051H respectively. Store the result of the BCD addition at memory locations 8052H and 8053H respectively.

The concept of BCD addition has been already discussed in Program 6.2 Method 2. How the DAA instruction is used after addition to generate correct result is also explained. Therefore this program is self-explanatory.

Assembly Language Program 6.5:

SL	Addresses	Label	Mnemonics	Hex Codes			No. of Bytes	No. of T-States
1	8000		LXI H,8050	21	50	80	3	10
2	8003		MOV A,M	7E			1	7
3	8004		INX H	23			1	6
4	8005		MVI B,00	06	00		2	7
5	8007		ADD M	86			1	7
8	8008		DAA	27			1	4
9	8009		JNC SKIP	D2	0D	80	3	10
10	800C		INR B	04			1	4
11	800D	SKIP	INX H	23			1	6
12	800E		MOV M,B	70			1	7
13	800F		INX H	23			1	6
14	8010		MOV M,A	77			1	7
15	8011		HLT	76			1	5
							TOTAL= 18	

Result of Program6.5:

Input

Mem. Address	Content	Remarks
8050	34	2 digit BCD No1
8051	74	2 digit BCD No2

Output

Address	Content	Remarks
8052	01	Higher Byte of Result
8053	08	Lower Byte of Result

Note: This program will not run properly in Jubin's 8085 simulator due to wrong implementation of DAA instruction. In simulator if CY flag is already set before the execution of DAA, DAA clears CY flag, which happens for BCD addition between 98 and 97. This program will run perfectly in 8085 Trainer Kit.

6.6: Write a program to convert a Hexadecimal number to its equivalent ASCII numbers. Store the Hexadecimal number at 8060H and corresponding ASCII numbers at 8061 and 8062 respectively.

A single digit Hexadecimal number is represented by any digit from 0 to 9 and any alphabet from A to F. Here the two digit Hexadecimal number is divided into two single digit Hexadecimal number by masking the upper nibble and lower nibble. Now each single digit Hexadecimal number will be converted to its equivalent ASCII numbers. The ASCII values of 0 to 9 and A to F are given below.

Hexadecimal Number	ASCII Value
0	30H
1	31H
2	32H
3	33H
4	34H
5	35H
6	36H
7	37H
8	38H
9	39H
A	41H
B	42H
C	43H
D	44H
E	45H
F	46H

From the ASCII chart it is clear that if Hexadecimal number lies between 0 to 9, 30H should be added with the Hexadecimal number and if Hexadecimal number lies between A to F, then 37H should be added with it to get the corresponding ASCII value.

Assembly Language Program 6.6:

SL	Addresses	Label	Mnemonics	Hex Codes			No. of Bytes	No. of T-States
1	8000		LXI H, 8060	21	60	80	3	10
2	8003		MOV B,M	46			1	7
3	8004		MOV A,B	78			1	4
4	8005		ANI 0F	E6	0F		2	7
5	8007		CALL HEX2ASC	CD	19	80	3	18
6	800A		INX H	23			1	6
7	800B		MOV M,A	77			1	7
8	800C		MOV A,B	78			1	4
9	800D		ANI F0	E6	F0		2	7
10	800F		RRC	0F			1	4
11	8010		RRC	0F			1	4
12	8011		RRC	0F			1	4
13	8012		RRC	0F			1	4
14	8013		CALL HEX2ASC	CD	19	80	3	18
15	8016		INX H	23			1	6
16	8017		MOV M,A	77			1	7
17	8018		HLT	76			1	5
18	8019	HEX2ASC	CPI 0A	FE	0A		2	7
19	801B		JC SKIP	DA	20	80	3	10/7
20	801E		ADI 07	C6	07		2	7
21	8020	SKIP	ADI 30	C6	30		2	7
22	8022		RET	C9			1	10
							TOTAL= 35	

Result of Program 6.6:

SET1 ►

Input

Mem. Address	Content	Remarks
8060	5F	2 digit Hex No.

Output

Address	Content	Remarks
8061	35	ASCII Value of 5
8062	46	ASCII Value of F

SET2 ►

Input

Mem. Address	Content	Remarks
8060	A0	2 digit Hex No.

Output

Address	Content	Remarks
8061	41	ASCII Value of A
8062	30	ASCII Value of 0

6.7: Write a program to convert an 8-bit Hexadecimal number stored at memory location 8050H to its equivalent gray code which will be stored at memory location 8051H.

To determine the corresponding gray code of a binary number the rule is to take the MSB of the binary number unchanged and all the other bits of the gray code is achieved by performing EXOR operation between two consecutive bits of the binary number. If an 8-bit binary number is represented as $B_7B_6B_5B_4B_3B_2B_1B_0$, then the corresponding gray code can be determined as follows.

$G_7 = 0 \oplus B_7 = B_7$ $\quad$ $G_3 = B_4 \oplus B_3$

$G_6 = B_7 \oplus B_6$ $\quad$ $G_2 = B_3 \oplus B_2$

$G_5 = B_6 \oplus B_5$ $\quad$ $G_1 = B_2 \oplus B_1$

$G_4 = B_5 \oplus B_4$ $\quad$ $G_0 = B_1 \oplus B_0$

The above mentioned process can be implemented by right shifting the binary number one bit position, which appends a zero at the MSB position and then performing bit-wise XOR operation between the actual binary number and the right shifted version of the binary number as shown below.

Binary Number →	B_7	B_6	B_5	B_4	B_3	B_2	B_1	B_0
	$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$
Right shifted Binary Number →	0	B_7	B_6	B_5	B_4	B_3	B_2	B_1
	---	---	---	---	---	---	---	---
Gray Code →	G_7	G_6	G_5	G_4	G_3	G_2	G_1	G_0

Assembly Language Program 6.7:

SL	Addresses	Label	Mnemonics	Hex Codes			No. of Bytes	No. of T-States
1	8000		LXI H, 8050	21	50	80	3	10
2	8003		MOV A,M	7E			1	7
3	8004		STC	37			1	4
4	8005		CMC	3F			1	4
5	8006		RAR	1F			1	4
6	8007		XRA M	AE			1	7
7	8008		INX H	23			1	6
8	8009		MOV M,A	77			1	7
9	800A		HLT	76			1	5
							TOTAL = 11	

Result of Program 6.7:

SET1 ►

Input

Mem. Address	Content	Remarks
8050	25	8-bit Hex Number

Output

Address	Content	Remarks
8051	37	8-bit Gray Code

SET2 ►

Input

Mem. Address	Content	Remarks
8050	C2	8-bit Hex Number

Output

Address	Content	Remarks
8051	A3	8-bit Gray Code

6.8: Write a program to convert an 8-bit gray code stored at memory location 8050H to its equivalent hex code which will be stored at memory location 8051H.

Suppose an 8-bit gray code is denoted as $G_7G_6G_5G_4G_3G_2G_1G_0$. Now this gray code can be converted to corresponding binary number using the following process.

$B_7 = 0 \oplus G_7 = G_7$ $\quad$ $B_3 = B_4 \oplus G_3$
$B_6 = B_7 \oplus G_6 = G_7 \oplus G_6$ $\quad$ $B_2 = B_3 \oplus G_2$
$B_5 = B_6 \oplus G_5$ $\quad$ $B_1 = B_2 \oplus G_1$
$B_4 = B_5 \oplus G_4$ $\quad$ $B_0 = B_1 \oplus G_0$

The above expressions to convert gray to binary are shown pictorially in Fig-6.1 for 4-bit representation.

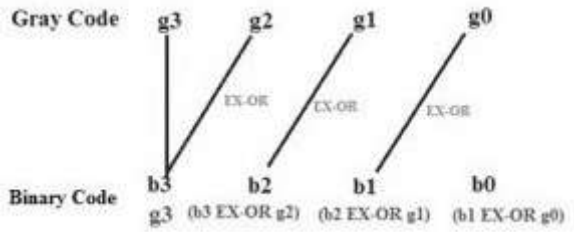


Fig-6.1: Gray to binary conversion

It is being observed that any bit in the converted binary number depends on the previous binary bit. Due to this reason B_6 binary bit can not be determined unless B_7 bit is calculated, B_5 bit can only be determined after the evaluation of B_6 bit and so on. In this program a loop is iterated for 7 times to convert the gray code to binary as shown below.

Iteration 1:

Gray code →	G_7	G_6	G_5	G_4	G_3	G_2	G_1	G_0
	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕
Right shifted Gray code →	0	G_7	G_6	G_5	G_4	G_3	G_2	G_1
Binary code1 →	$G_7 = B_7$	B_6	D_5	D_4	D_3	D_2	D_1	D_0
	↑	↑	↑	↑	↑	↑	↑	↑
	Valid	Valid	Invalid	Invalid	Invalid	Invalid	Invalid	Invalid

Iteration 2:

Gray code →	G_7	G_6	G_5	G_4	G_3	G_2	G_1	G_0
	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕
Right shifted Binary code1 →	0	B_7	B_6	D_5	D_4	D_3	D_2	D_1
Binary code2 →	$G_7 = B_7$	B_6	B_5	D_4	D_3	D_2	D_1	D_0
	↑	↑	↑	↑	↑	↑	↑	↑
	Valid	Valid	Valid	Invalid	Invalid	Invalid	Invalid	Invalid

Iteration 3:

Gray code →	G_7	G_6	G_5	G_4	G_3	G_2	G_1	G_0
	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕
Right shifted Binary code2 →	0	B_7	B_6	B_5	D_4	D_3	D_2	D_1
Binary code3 →	$G_7 = B_7$	B_6	B_5	B_4	D_3	D_2	D_1	D_0
	↑	↑	↑	↑	↑	↑	↑	↑
	Valid	Valid	Valid	Valid	Invalid	Invalid	Invalid	Invalid

Iteration 4:

Gray code →	G_7	G_6	G_5	G_4	G_3	G_2	G_1	G_0
	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕
Right shifted Binary code3 →	0	B_7	B_6	B_5	B_4	D_3	D_2	D_1
Binary code4 →	$G_7 = B_7$	B_6	B_5	B_4	B_3	D_2	D_1	D_0
	↑	↑	↑	↑	↑	↑	↑	↑
	Valid	Valid	Valid	Valid	Valid	Invalid	Invalid	Invalid

Iteration 5:

Gray code →	G_7	G_6	G_5	G_4	G_3	G_2	G_1	G_0
	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕
Right shifted Binary code4 →	0	B_7	B_6	B_5	B_4	B_3	D_2	D_1
Binary code5 →	$G_7 = B_7$	B_6	B_5	B_4	B_3	B_2	D_1	D_0
	↑	↑	↑	↑	↑	↑	↑	↑
	Valid	Valid	Valid	Valid	Valid	Valid	Invalid	Invalid

Iteration 6:

Gray code →	G_7	G_6	G_5	G_4	G_3	G_2	G_1	G_0
	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕
Right shifted Binary code5 →	0	B_7	B_6	B_5	B_4	B_3	B_2	D_1
Binary code6 →	$G_7 = B_7$	B_6	B_5	B_4	B_3	B_2	B_1	D_0
	↑	↑	↑	↑	↑	↑	↑	↑
	Valid	Valid	Valid	Valid	Valid	Valid	Valid	Invalid

Iteration 7:

Gray code →	G_7	G_6	G_5	G_4	G_3	G_2	G_1	G_0
	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕
Right shifted Binary code6 →	0	B_7	B_6	B_5	B_4	B_3	B_2	B_1
Binary code7 →	$G_7 = B_7$	B_6	B_5	B_4	B_3	B_2	B_1	B_0
	↑	↑	↑	↑	↑	↑	↑	↑
	Valid	Valid	Valid	Valid	Valid	Valid	Valid	Valid

It is being observed clearly that the Binary code7 thus achieved finally after 7th iteration is valid.

Assembly Language Program 6.8:

SL	Addresses	Label	Mnemonics	Hex Codes			No. of Bytes	No. of T-States
1	8000		LXI H,8050	21	50	80	3	10
2	8003		MOV A,M	7E			1	7
3	8004		MVI C,07	0E	07		2	7
4	8006	LOOP	STC	37			1	4
5	8007		CMC	3F			1	4
6	8008		RAR	1F			1	4
7	8009		XRA M	AE			1	7
8	800A		DCR C	0D			1	4
9	800B		JNZ LOOP	C2	06	80	3	10/7
10	800E		INX H	23			1	6
11	800F		MOV M,A	77			1	7
12	8010		HLT	76			1	5
							TOTAL = 17	

Result of Program 6.8:

SET1 ►

Input

Mem. Address	Content	Remarks
8050	37	8-bit Gray code

Output

Address	Content	Remarks
8051	25	8-bit hexadecimal number

SET2 ►

Input

Mem. Address	Content	Remarks
8050	A3	8-bit Gray code

Output

Address	Content	Remarks
8051	C2	8-bit hexadecimal number

Exercise

1) Write a program to convert an ASCII character stored at 8050H to its equivalent Hexadecimal number which should be placed at memory location 8051H.

2) Write a program to convert a 2-digit packed BCD number stored at 8050H to its equivalent packed Excess 3 codes which should be placed at memory location 8051.
[Example: Packed 2-digit BCD: 92 → Packed 2-digit Excess 3 Code: C5]

3) Write a program to convert a 2-digit packed Excess 3 code stored at 8050H to its equivalent 2-digit packed BCD number which should be placed at memory location 8051H.

7. Programs on Look up Table

7.1: Write a program to determine the square of a number which is stored at memory location 8050H using Look up Table. Also store the square value at memory location 8051H.

Although to determine the square of a number can be evaluated by multiplying the number with itself, but the square of a number is determined by using look up table to develop the concept of the look up table. Here a portion of the memory has been used to store the square of the numbers 00H to 0FH. We can not store the square of a number beyond 0F (15 in Decimal), because it exceeds the maximum range of a 8-bit number, FFH (255 in Decimal). In this program the look up table has been started from the memory location 8060H onward as shown below.

Look up Table

Memory Address	Square of 8-bit number	
8060H	00H (0)	←square of 0
8061H	01H (1)	←square of 1
8062H	04H (4)	←square of 2
8063H	09H (9)	←square of 3
8064H	10H (16)	←square of 4
8065H	19H (25)	←square of 5
8066H	24H (36)	←square of 6
8067H	31H (49)	←square of 7
8068H	40H (64)	←square of 8
8069H	51H (81)	←square of 9
806AH	64H (100)	←square of 10
806BH	79H (121)	←square of 11
806CH	90H (144)	←square of 12
806DH	A9H (169)	←square of 13
806EH	C4 (196)	←square of 14
806FH	E1H (225)	←square of 15

To get the square of a number, that particular number is added with the starting address of the look up table to get the memory location where the square of that number is saved. Now the content of that memory address is retrieved to get the square of the number.

Assembly Language Program 7.1:

SL	Addresses	Label	Mnemonics	Hex Codes			No. of Bytes	No. of T-States
1	8000		LDA 8050	3A	50	80	3	13
2	8003		MOV L,A	6F			1	4
3	8004		MVI H,00	26	00		2	7
4	8006		LXI D,8060	11	60	80	3	10
5	8009		DAD D	19			1	10
6	800A		MOV A,M	7E			1	7
7	800B		STA 8051	32	51	80	3	13
8	800E		HLT	76			1	5
							TOTAL= 15	

Result of Program 7.1:

SET1 ►

Input

Mem. Address	Content	Remarks
8050	03	8-bit No.

Output

Address	Content	Remarks
8051	09	Square of No. 3

SET2 ►

Input

Mem. Address	Content	Remarks
8050	0F	8-bit No.

Output

Address	Content	Remarks
8051	E1	Square of No. F

Exercise

1) Suppose a Common Cathode 7-segment display is connected to data bus of 8085 via a 74373 latch and the latch will be enabled with the port address FFH. The different pins of the 7-segment display is connected to the data bus as follows.
   D_0 → a segment
   D_1 → b segment
   D_2 → c segment
   D_3 → d segment
   D_4 → e segment
   D_5 → f segment
   D_6 → g segment
   D_7 → h dot point

   Now write a program to convert a single digit BCD number stored at memory location 8050H to its equivalent 7 segment display code using look up table and send the 7-segment equivalent code through data bus using port address FFH to show the BCD number on the 7-segment display.

2) Suppose a Common Anode 7-segment display is connected to data bus of 8085 via a 74373 latch and the latch will be enabled with the port address FFH. The different pins of the 7-segment display is connected to the data bus as follows.
   D_0 → a segment
   D_1 → b segment
   D_2 → c segment
   D_3 → d segment
   D_4 → e segment
   D_5 → f segment
   D_6 → g segment
   D_7 → h dot point

   Now write a program to convert a single digit BCD number stored at memory location 8050H to its equivalent 7 segment display code using look up table and send the 7-segment equivalent code through data bus using port address FFH to show the BCD number on the 7-segment display.

3) Write a program to find out the factorial of a 8-bit number stored at memory location 8050 and store the factorial at memory location 8051 using look up table.

8. Programs on String Manipulation

A string consists of some characters which may be alphabets or numbers. The characters of a string are stored in consecutive memory locations. In 8085 a string is basically composed of a series of hexadecimal numbers which are stored in successive memory locations. For example – a string "66778090AABBCCDDEEFF" stored consecutively from memory location 9000H as shown below.

Mem. Addres	Content
9000	66
9001	77
9002	80
9003	90
9004	AA
9005	BB
9006	CC
9007	DD
9008	EE
9009	FF

Here the programs related to string manipulation will be explained.

8.1: Suppose a string is stored from memory location 8050H to 8057H. Write a program to reverse the string and store the reversed string starting from 8060H.

Suppose a string "0123456789ABCDEF" is stored from memory location 8050 to 8057 as shown in Fig-8.1. After the execution of the program the string will be reversed and the reversed string "FEDCBA9876543210" will be stored starting from 8060 onward as shown in Fig-8.2.

Mem. Address	Content
8050	01
8051	23
8052	45
8053	67
8054	89
8055	AB
8056	CD
8057	EF

Fig-8.1: Source string

Mem. Address	Content
8060	FE
8061	DC
8062	BA
8063	98
8064	76
8065	54
8066	32
8067	10

Fig-8.2: Reversed string

Here every 8-bit data is to be copied starting from memory location 8057 to accumulator, swap the nibbles of the accumulator by using RAL instruction and save the swapped data starting from memory location 8060 onward. That means the source string should be copied from memory location 8057 to 8050 and the reversed string should be stored from memory location 8060 to 8067.

Assembly Language Program 8.1:

SL	Addresses	Label	Mnemonics	Hex Codes			No. of Bytes	No. of T-States
1	8000		LXI H,8057	21	57	80	3	10
2	8003		LXI D,8060	11	60	80	3	10
3	8006		MVI C,08	0E	08		2	7
4	8008	LOOP	MOV A,M	7E			1	7
5	8009		RLC	07			1	4
6	800A		RLC	07			1	4
7	800B		RLC	07			1	4
8	800C		RLC	07			1	4

SL	Addresses	Label	Mnemonics	Hex Codes			No. of Bytes	No. of T-States
9	800D		STAX D	12			1	7
10	800E		DCX H	2B			1	6
11	800F		INX D	13			1	6
12	8010		DCR C	0D			1	4
13	8011		JNZ LOOP	C2	06	80	3	10/7
14	8014		HLT	76			1	5
							TOTAL= 21	

Result of Program 8.1:

SET1 ►

Input

Source String

Mem. Address	Content	Remarks
8050	01	
8051	23	
8052	45	
8053	67	
8054	89	
8055	AB	
8056	CD	
8057	EF	

Output

Reversed String

Mem. Address	Content	Remarks
8060	FE	
8061	DC	
8062	BA	
8063	98	
8064	76	
8065	54	
8066	32	
8067	10	

SET2 ►

Input

Source String

Mem. Address	Content	Remarks
8050	1F	
8051	2E	
8052	3D	
8053	4C	

Output

Reversed String

Mem. Address	Content	Remarks
8060	88	
8061	97	
8062	A6	
8063	B5	

Mem. Address	Content	Remarks
8054	5B	
8055	6A	
8056	79	
8057	88	

Mem. Address	Content	Remarks
8064	C4	
8065	D3	
8066	E2	
8067	F1	

8.2: Suppose a string is stored from memory location 8050H to 8058H. Write a program to check whether the string is palindrome or not. If the string is palindrome, 01H should be stored at memory location 8059H, otherwise 00H should be stored in the same memory location.

A string is said to be palindrome when it matches exactly with its reversed form. For example – a string "ABCDEF99FEDCBA" is palindrome, because if it is written in reverse order it will be the same string "ABCDEF99FEDCBA". Now for 8085 microprocessor a string always consists of even no. of characters, because each memory location stores 8-bit data which includes two characters. Hence for 8085 architecture it is not possible to store a string which comprises odd no. of characters. Now the question arises how to check it. One thing is important to observe that every pair of characters from starting position is just reverse of the pair of characters from end position. In case of the above string "ABCDEF99FEDCBA" AB from starting positions is just reverse of BA from end positions. Similarly CD is reversed of DC and EF is also reversed form of FE.

Here two cases arise – 1) no. of memory locations consumed by the string is even and 2) no. of memory locations consumed by the string is odd. This implies that every pair of characters is reversed and compared with its counterpart pair of characters up to n/2 for even no. of memory locations and ($\lfloor n/2 \rfloor+1$) for odd no. of memory locations where n is the no. of memory locations consumed by the string. For examples - the string "ABCDEF99FEDCBA" takes 7 (odd) consecutive memory locations. That's why the checking has to be performed up to 4^{th} memory location. At any stage if the reversed pair of characters does not match with its corresponding pair of characters, the string will not be palindrome. If the reversed pair of characters matches with its corresponding pair of characters up to n/2 or ($\lfloor n/2 \rfloor+1$), the string will be a palindrome. According to the condition of the program 01H will be stored at the memory location 8059H, if the string is palindrome and 00H will be stored at 8059H if the string is not palindrome.

Assembly Language Program 8.2:

SL	Addresses	Label	Mnemonics	Hex Codes			No. of Bytes	No. of T-States
1	8000		LXI D,8050	11	50	80	3	10
2	8003		LXI H,8058	21	58	80	3	10
3	8006		MVI C,05	0E	05		2	7
4	8008	LOOP	LDAX D	1A			1	7
5	8009		RLC	07			1	4
6	800A		RLC	07			1	4
7	800B		RLC	07			1	4
8	800C		RLC	07			1	4
9	800D		CMP M	BE			1	7
10	800E		JNZ NOTPALIN	C2	1C	80	3	10
11	8011		INX D	13			1	6
12	8012		DCX H	2B			1	6
13	8013		DCR C	0D			1	4
14	8014		JNZ LOOP	C2	08	80	3	10
15	8017		XRA A	AF			1	4
16	8018		INR A	3C			1	4
17	8019		JMP PALIN	C3	1D	80	3	10
18	801C	NOTPALIN	XRA A	AF			1	4
19	801D	PALIN	STA 8059	32	59	80	3	13
20	8020		HLT	76			1	5
							TOTAL= 33	

Result of Program 8.2:

SET1 ►

Input

Source String

Mem. Address	Content	Remarks
8050	AB	
8051	CD	
8052	EF	
8053	12	
8054	33	
8055	21	
8056	FE	
8057	DC	
8058	BA	

Output

Mem. Address	Content	Remarks
8059	01	Palindrome

SET2 ►

Input

Source String

Mem. Address	Content	Remarks
8050	AB	
8051	CD	
8052	EF	
8053	12	
8054	34	
8055	21	
8056	FE	
8057	DC	
8058	BA	

Output

Mem. Address	Content	Remarks
8059	00	Not Palindrome

8.3: Suppose two strings are stored into two memory blocks - 8050H to 8059H and 8060H to 8065H respectively. Write a program to concatenate these two strings and store the concatenated string starting from memory location 8050H onward.

As per the program objective 1st string of 20 characters is stored starting from memory location 8050H to 8059H and 2nd string of 12 characters is stored from 8060H to 8065H. The target of this program is to join or concatenate these two strings into one string. Therefore in this program the 2nd string is appended at the end of the 1st string to form a 3rd string of 32 characters (20 characters of 1st string + 12 characters of 2nd string) which will occupy 16 no. of memory locations from 8050H to 805FH. To understand the above mentioned illustration let's take an example.

Suppose "0123456789ABCDEF9988" is the 1st string which is stored into the memory locations from 8050H to 8059H and "A1B2C3D4E5F6" is the 2nd string which is stored into the memory locations from 8060H to 8065H. After the execution of this program 1st and 2nd string will be joined together to form a 3rd string " 0123456789ABCDEF9988A1B2C3D4E5F6" which will be stored from the memory location 8050H to the memory location 805FH as shown in Fig-8.3.

1st String

Address	Content
8050	01
8051	23
8052	45
8053	67
8054	89
8055	AB
8056	CD
8057	EF
8058	99
8059	88

2nd String

Address	Content
8060	A1
8061	B2
8062	C3
8063	D4
8064	E5
8065	F6

3rd Concatenated String

Address	Content
8050	01
8051	23
8052	45
8053	67
8054	89
8055	AB
8056	CD
8057	EF
8058	99
8059	88
805A	A1
805B	B2
805C	C3
805D	D4
805E	E5
805F	F6

Fig-8.3: Two strings are concatenated to form a 3rd string

Assembly Language Program 8.3:

SL	Addresses	Label	Mnemonics	Hex Codes			No. of Bytes	No. of T-States
1	8000		LXI H,8060	21	60	80	3	10
2	8003		LXI D,805A	11	5A	80	3	10
3	8006		MVI C,06	0E	06		2	7
4	8008	LOOP	MOV A,M	7E			1	7
5	8009		STAX D	12			1	7
6	800A		INX H	23			1	6
7	800B		INX D	13			1	6
8	800C		DCR C	0D			1	4
9	800D		JNZ LOOP	C2	08	80	3	10/7
10	8010		HLT	76			1	5

Result of Program 8.3:

SET1 ► **Input**

1st String

Address	Content
8050	01
8051	23
8052	45
8053	67
8054	89
8055	AB
8056	CD
8057	EF
8058	99
8059	88

2nd String

Address	Content
8060	A1
8061	B2
8062	C3
8063	D4
8064	E5
8065	F6

Output

Concatenated 3rd String

Address	Content
8050	01
8051	23
8052	45
8053	67
8054	89
8055	AB
8056	CD
8057	EF
8058	99
8059	88
805A	A1
805B	B2
805C	C3
805D	D4
805E	E5
805F	F6

SET2 ►

Input

1st String

Address	Content
8050	11
8051	22
8052	33
8053	44
8054	55
8055	66
8056	77
8057	88
8058	99
8059	AA

2nd String

Address	Content
8060	BB
8061	CC
8062	DD
8063	EE
8064	FF
8065	32

Output

Concatenated 3rd String

Address	Content
8050	11
8051	22
8052	33
8053	44
8054	55
8055	66
8056	77
8057	88
8058	99
8059	AA
805A	BB
805B	CC
805C	DD
805D	EE
805E	FF
805F	32

8.4: Write a program to check whether a string stored from 8050 onward contains another sub-string stored from 8060 onward or not. Store 01H into the memory location 8070H if the main string contains the sub-string, otherwise store 02H into the memory location 8070H.

Here one string known as main string is stored from the memory location 8050H and another string known as sub-string is stored starting from memory location 8060H. Obviously the length of the sub-string will be less or equal to the length of the main string. Here four cases may happen.

Case 1: In this case no matching happens between the main string and sub-string. For example – if the main string is "1234567890ABCDEF9988" and the sub-string is "2233445566", it is observed that there is no matching between the main string and the sub-string. Therefore 02H should be stored into the memory location 8070H to indicate the mismatch between the two strings.

Case 2: Here partial matching occurs between the main string and the sub-string. For example – if the main string is "1234567890ABCDEF9988" and the sub-string is "ABCDEF1122", it is observed that a portion of sub-string "ABCDEF" is found into the main string. As entire sub-string is not found into the main string, it results mismatch between the main string and the sub-string. Therefore 02H will be stored into the memory location 8070H.

Case 3: In this case the entire sub-string is found into the main string which results successful matching between the two strings. Hence 01H should be stored in the memory location 8070H. For example – complete matching occurs if the main string becomes " 1234567890ABCDEF9988" and the sub-string is "ABCDEF9988".

Case 4: This case consists of both partial matching and complete matching. As complete matching is found finally, 01H will be stored into the same memory location according to the program criteria. For example – if main string is " 12ABCD7890ABCDEF9988" and the sub-string is "ABCDEF9988", then partial matching occurs for "ABCD" from 2nd position whereas complete matching happens for "ABCDEF9988" from 6th position.

Now these above mentioned four cases must be handled in the program to check the matching of two strings. If the no. of 8-bit data in the main string is m and the no. of 8-bit data in the sub-string is n, then there will be no chance of finding the whole sub-string inside the main string beyond (m – n + 1)th data. Therefore we have to compare up to (m – n + 1)th data in the main string, beyond of that there is no chance to get the entire sub-string into the main string. The following example will reveal the above mentioned situation.

Suppose main string "1234567890ABCDEF8899" has 10 no. of 8-bit data and sub-string "ABCDEF8899" has 5 no. of 8-bit data. Therefore we have to search for matching of data up to 6th (10 – 5 + 1) position i.e. up to the data "AB" into the main string, because beyond of that there is no possibility to get the whole sub-string "ABCDEF8899".

Here 1st 8-bit data of the sub-string is started to be compared with all the 8-bit data of main string consecutively from 1st data to (m – n + 1)th data of the main string. If matching is found at any stage, the rest of the data from the sub-string are compared with the data of the main string consecutively. That means, if the 1st data of sub-string is matched with any data of main string, then the comparisons between the pairs of the data – one from sub-string and other from main string are performed successively until the end of the sub-string or a mismatch is found. If every pair of data from the sub-string and the main string are matched perfectly, then it can be concluded that the sub-string is found into the main string and if any mismatch is found, then the 1st data from sub-string and the the data of main string where mismatch was found should be compared once again to get the entire sub-string inside the remaining part of the main string. Here one important point to consider that if mismatch is found after (m – n + 1)th data of the main string, then comparisons are not carried out further to indicate the absence of the sub-string inside the main string.

In the following program we have taken a main string with 10 no. of data and the sub-string with 5 no. of data. Therefore comparisons should continue up to 6th data of the main string. To fulfill this purpose register C will act as counter of main string and initialized with 06H. Similarly register B is used as counter of sub-string and initialized with 05H. In addition to this, register pair DE has been used as memory pointer of main string and register pair HL has been used as memory pointer for sub-string in this program. If a match is found, register C and B both will be decremented by one for every iteration, otherwise register C only will be decremented by one for each iteration. There are two loops in this program – one is controlled by the counter register C and other is controlled by the counter register B. The loop of counter register C will continue until a matching between the 1st data from the sub-string and any data [up to (m – n + 1)th data] from the main string is found. On the other hand if a matching is found, then the loop of counter register B will be initiated. If the loop of register B is terminated by decreasing B to zero, it is clear that the sub-string is found inside the main string and if the loop of register C is terminated for C = 0, then the sub-string is not found into the main string.

Assembly Language Program 8.4:

SL	Addresses	Label	Mnemonics	Hex Codes			No. of Bytes	No. of T-States
1	8000		MVI C,06	0E	06		2	7
2	8002		LXI D,8050	11	50	80	3	10
3	8005		LXI H,8060	21	60	80	3	10
4	8008	REPEAT	LDAX D	1A			1	7
5	8009		CMP M	BE			1	7
6	800A		JNZ NOTEQUAL	C2	2B	80	3	10 / 7
7	800D		MVI B,05	06	05		2	7
8	800F	AGAIN	LDAX D	1A			1	7
9	8010		CMP M	BE			1	7
10	8011		JNZ NOMATCH	C2	26	80	3	10 / 7
11	8014		INX D	13			1	6
12	8015		INX H	23			1	6
13	8016		MOV A,C	79			1	4
14	8017		CPI 01	FE	01		2	7
15	8019		JC BYPASS	DA	1D	80	3	10 / 7
16	801C		DCR C	0D			1	4
17	801D	BYPASS	DCR B	05			1	4
18	801E		JNZ AGAIN	C2	0F	80	3	10 / 7
19	8021		MVI A,01	3E	01		2	7

SL	Addresses	Label	Mnemonics	Hex Codes			No. of Bytes	No. of T-States
20	8023		JMP FINAL	C3	32	80	3	10
21	8026	NOMATCH	DCX D	1B			1	6
22	8027		INR C	0C			1	4
23	8028		LXI H,8060	21	60	80	3	10
24	802B	NOTEQUAL	INX D	13			1	6
25	802C		DCR C	0D			1	4
26	802D		JNZ REPEAT	C2	08	80	3	10 / 7
27	8030		MVI A,02	3E	02		2	7
28	8032	FINAL	STA 8070	32	70	80	3	13
29	8035		HLT	76			1	5
							TOTA = 54	

Result of Program 8.4:

SET1 ► (Corresponds to Case 1)

Input

Main String

Address	Content
8050	12
8051	34
8052	56
8053	78
8054	87
8055	65
8056	43
8057	21
8058	CD
8059	EF

Sub-String

Address	Content
8060	AB
8061	CD
8062	EF
8063	88
8064	99

Output

Address	Content	Remarks
8070	02	Sub-string not found

SET2 ► ***(Corresponds to Case 1)***

Input

Main String

Address	Content
8050	12
8051	34
8052	56
8053	78
8054	87
8055	65
8056	AB
8057	CD
8058	EF
8059	88

Sub-String

Address	Content
8060	AB
8061	CD
8062	EF
8063	88
8064	99

Output

Address	Content	Remarks
8070	02	Sub-string not found

SET3 ► ***(Corresponds to Case 2)***

Input

Main String

Address	Content
8050	12
8051	34
8052	56
8053	78
8054	AB
8055	CD
8056	AB
8057	CD
8058	EF
8059	88

Sub-String

Address	Content
8060	AB
8061	CD
8062	EF
8063	88
8064	99

Output

Address	Content	Remarks
8070	02	Sub-string not found

SET4 ► ***(Corresponds to Case 2)***

Input

Main String

Address	Content
8050	12
8051	34
8052	AB
8053	CD
8054	56
8055	78
8056	87
8057	EF
8058	88
8059	99

Sub-String

Address	Content
8060	AB
8061	CD
8062	EF
8063	88
8064	99

Output

Address	Content	Remarks
8070	02	Sub-string not found

SET5 ► ***(Corresponds to Case 2)***

Input

Main String

Address	Content
8050	12
8051	34
8052	56
8053	AB
8054	CD
8055	AB
8056	AB
8057	CD
8058	EF
8059	88

Sub-String

Address	Content
8060	AB
8061	CD
8062	EF
8063	88
8064	99

Output

Address	Content	Remarks
8070	02	Sub-string not found

SET6 ► ***(Corresponds to Case 3)***

Input

Main String

Address	Content
8050	12
8051	34
8052	56
8053	78
8054	87
8055	AB
8056	CD
8057	EF
8058	88
8059	99

Sub-String

Address	Content
8060	AB
8061	CD
8062	EF
8063	88
8064	99

Output

Address	Content	Remarks
8070	01	Sub-string found

SET7 ► ***(Corresponds to Case 3)***

Input

Main String

Address	Content
8050	AB
8051	CD
8052	EF
8053	88
8054	99
8055	12
8056	34
8057	56
8058	78
8059	87

Sub-String

Address	Content
8060	AB
8061	CD
8062	EF
8063	88
8064	99

Output

Address	Content	Remarks
8070	01	Sub-string found

SET8 ► ***(Corresponds to Case 4)***

Input

Main String

Address	Content
8050	12
8051	AB
8052	CD
8053	AB
8054	CD
8055	EF
8056	88
8057	99
8058	34
8059	56

Sub-String

Address	Content
8060	AB
8061	CD
8062	EF
8063	88
8064	99

Output

Address	Content	Remarks
8070	01	Sub-string found

SET9 ► ***(Corresponds to Case 4)***

Input

Main String

Address	Content
8050	12
8051	34
8052	56
8053	AB
8054	CD
8055	AB
8056	CD
8057	EF
8058	88
8059	99

Sub-String

Address	Content
8060	AB
8061	CD
8062	EF
8063	88
8064	99

Output

Address	Content	Remarks
8070	01	Sub-string found

8.5: Suppose two strings are stored into two memory blocks - 8050H to 8059H and 8060H to 8063H respectively. Write a program to insert the second string into the first string starting from the memory location 8053H.

It can be observed that the second string stored from 8060H to 8063H has four 8-bit data and the first string stored from 8050H to 8059H has ten 8-bit data. To insert the second string at the memory location 8053H of the first string, all the data from 8053H to 8059H must be shifted into the memory locations 8057H to 805DH first to make a space of 4 bytes so that the second string can be accommodated into that memory space. After shifting the data the entire second string should be copied from the memory locations 8060H to 8063H into the memory locations 8053H to 8056H.

Now to make a space of four consecutive memory locations starting from 8053H to 8056H, seven 8-bit data of first string from the memory locations 8053H to 8059H should be shifted to the memory locations 8057H to 805DH. Hence register C has been considered as a counter and initialized with 07H. After shifting these seven data, the four 8-bit data of the second string stored from the memory location 8060H to 8063H should be copied into the memory locations 8053H to 8056H. For this purpose the register C will be initialized with 04H to act as a counter and will be used to transfer these four data. Thus the second string will be inserted into the first string from the memory location 8053H.

Assembly Language Program 8.5:

SL	Addresses	Label	Mnemonics	Hex Codes			No. of Bytes	No. of T-States
1	8000		LXI H,8059	21	59	80	3	10
2	8003		LXI D,805D	11	5D	80	3	10
3	8006		MVI C,07	0E	07		2	7
4	8008	REPEAT	MOV A,M	7E			1	7
5	8009		STAX D	12			1	7
6	800A		DCX H	2B			1	6
7	800B		DCX D	1B			1	6
8	800C		DCR C	0D			1	4
9	800D		JNZ REPEAT	C2	08	80	3	10 / 7
10	8010		MVI C,04	0E	04		2	7
11	8012		LXI D,8060	11	60	80	3	10
12	8015	AGAIN	INX H	23			1	6
13	8016		LDAX D	1A			1	7
14	8017		MOV M,A	77			1	7

SL	Addresses	Label	Mnemonics	Hex Codes			No. of Bytes	No. of T-States
15	8018		INX D	13			1	6
16	8019		DCR C	0D			1	4
17	801A		JNZ AGAIN	C2	15	80	3	10 / 7
18	801D		HLT	76			1	5
							TOTAL = 30	

Result of Program 8.5:

SET1 ►

Input

1st String

Address	Content
8050	11
8051	22
8052	33
8053	44
8054	55
8055	66
8056	77
8057	88
8058	99
8059	AA

2nd String

Address	Content
8060	BB
8061	CC
8062	DD
8063	EE

Output

Address	Content
8050	11
8051	22
8052	33
8053	BB
8054	CC
8055	DD
8056	EE
8057	44
8058	55
8059	66
805A	77
805B	88
805C	99
805D	AA

SET2 ►

Input

1st String

Address	Content
8050	12
8051	23
8052	34
8053	45
8054	56
8055	67
8056	78
8057	89
8058	9A
8059	AB

2nd String

Address	Content
8060	BC
8061	CD
8062	DE
8063	EF

Output

Address	Content
8050	12
8051	23
8052	34
8053	BC
8054	CD
8055	DE
8056	EF
8057	45
8058	56
8059	67
805A	78
805B	89
805C	9A
805D	AB

Exercise

1) Write a program to check whether two strings are identical or not. Consider the two strings having same length of 16 characters are stored from memory location 8050 onward and 8060 onward respectively.

2) Write a program to replace all the characters ‘A’ with the character ‘D’ in a string which is stored from the memory location 9000H onward.

9. Details of 8255 peripheral in 8085 trainer kit
Interfacing programs of 8255 in 8085 trainer kit

The 8255 is a widely used programmable, parallel I/O device. It can be programmed to transfer data under various conditions, from simple I/O to interrupt I/O. 8255 has 24 I/O pins that can be grouped primarily in two 8-bit parallel ports: A and B with the remaining 8-bits as port C. 8 bits of port C can be used as individual bits or be grouped in two 4 bit ports: C_{UPPER} (C_U) and C_{LOWER} (C_L) as in Fig-9.1. The functions of these ports are defined by writing a control word in the control register.

Fig-9.2 shows all the functions of the 8255, classified according to two modes: the bit set/ reset (BSR) mode and the I/O mode. The BSR mode is used to set or reset the bits in port C. The I/O mode is further divided into three modes: Mode0, Mode1, Mode2. In Mode0, all ports function as simple I/O ports. Mode1 is a handshaking mode whereby ports A and/ or port B use bits from port C as handshaking signals. In the handshaking mode, two types of I/O data transfer can be implemented: status check and interrupt. In Mode2, port A can be set up for bidirectional data transfer using handshaking signals from port C and port B can be set up either in Mode0 or Mode1.

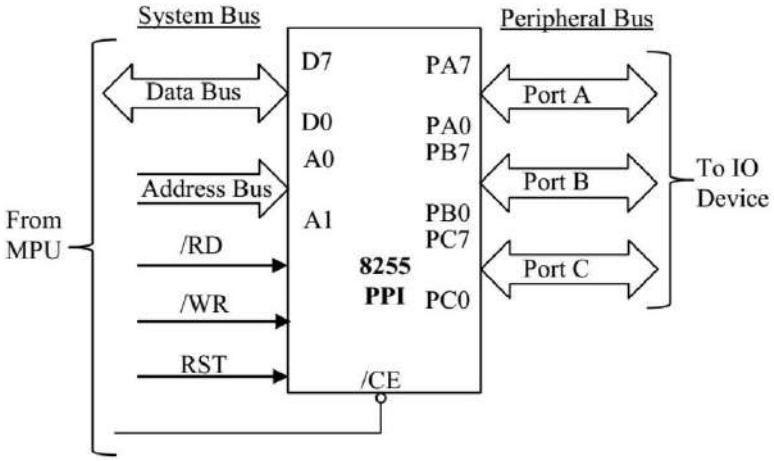


Fig-9.1: 8255 three I/O ports with address bus, data bus and control lines

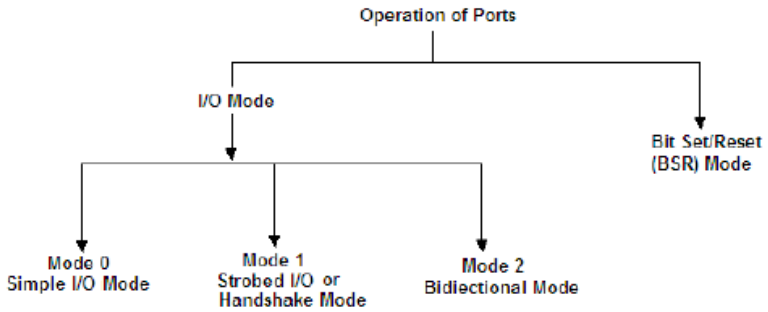


Fig-9.2: 8255 different modes of operation

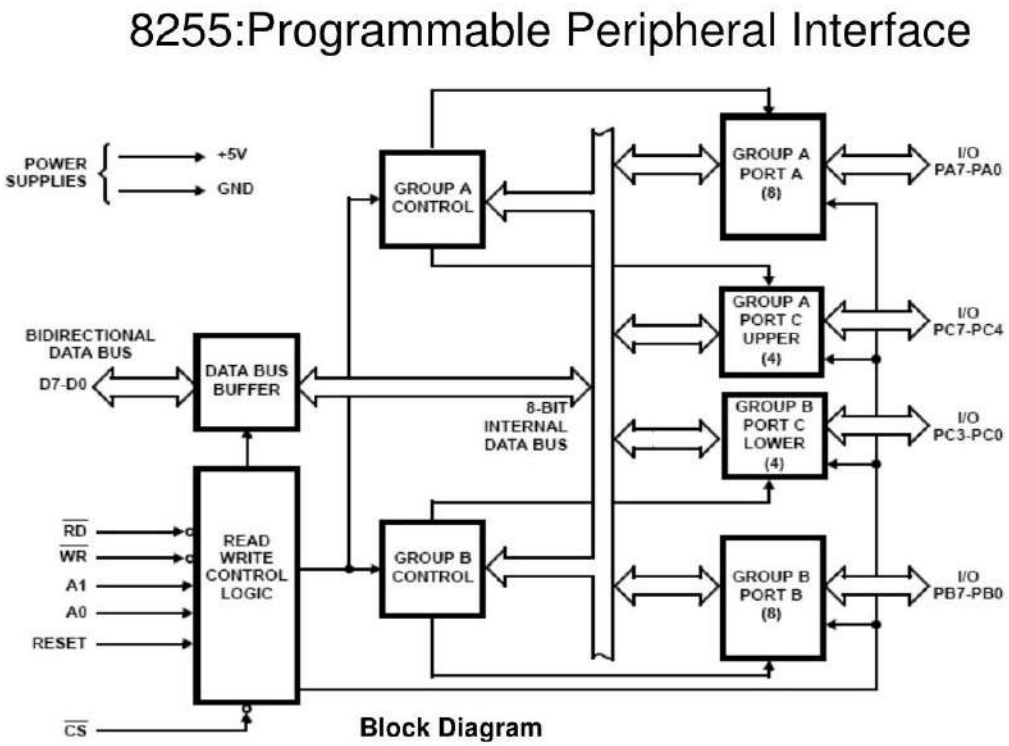


Fig-9.3: Block diagram of 8255 PPI

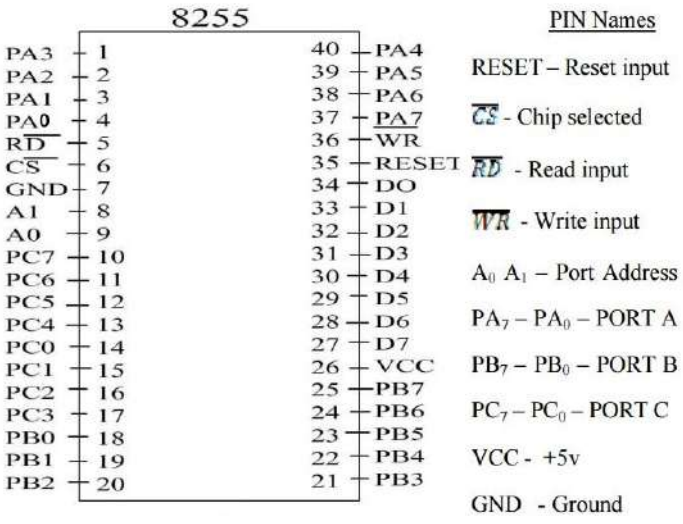


Fig-9.4: Pin diagram of 8255 PPI

The block diagram and IC pin diagram of 8255 is shown in Fig-9.3 and Fig-9.4 respectively. The block diagram of Fig-9.1 shows two 8-bit ports: A and B along with two 4-bit ports C_L and C_U. In addition to this, there are 8-bit data bus (D0 -D7), two address lines (A0 and A1) and four control lines: RESET, RD′, WR′ and CS′. The description of these control lines, address lines and data bus are given below.

1) CS′ , A0, A1: CS′ (Chip Select) line is used to enable the 8255 IC. CS′ is normally connected to a decoded address and and A0 and A1 are generally connected to the A0 and A1 address lines of the 8085 microprocessor. These three lines gives the range of I/O addresses for which any one of the three ports (port A, port B, port C) and the control register will be selected for operation.

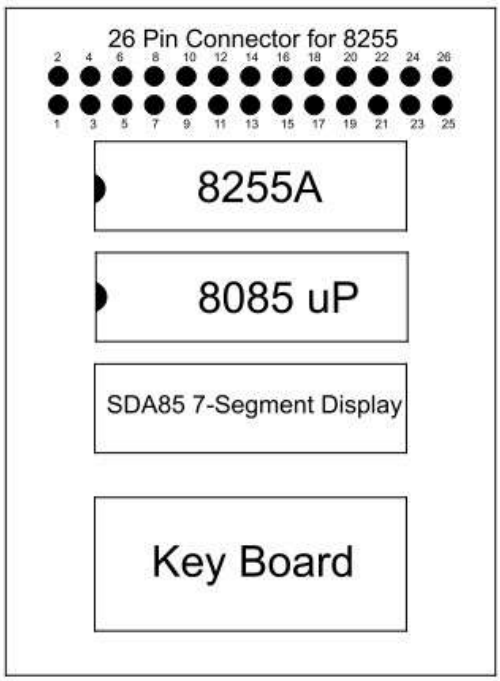
CS′	A1	A0	Selected
0	0	0	Port A
0	0	1	Port B
0	1	0	Port C
0	1	1	Control Register
1	X	X	8255 chip disabled

2) RESET: It is an active high signal which clears the control register and sets all the ports in input mode.

3) RD′: This control signal enables read operation. When this signal is low, the microprocessor 8085 reads data from a selected I/O port of 8255.

4) WR′: This control signal enables write operation. When this signal is low, the microprocessor 8085 send/ write a data/ bit pattern into a selected I/O port or control register of 8255.

From the above discussion it is clear that the range of address by which I/O ports or control register is selected depends upon the hardware connection of CS′ of 8255 with the 8085 microprocessor. 8255 chip is connected into the board of 8085 Trainer Kit. The schematic top views of 8255 in the trainer kit SDA85H and SDA85M along with 26 pins sockets is shown in Fig-9.5. Among these 26 pins of the socket, 8 pins for port A, 8 pins for port B, 8 pins for port C and remaining 2 pins are used for Vcc (+5V) and GND.



TOP VIEW OF SDA85 8085 MICROPROCESSOR KIT

Fig-9.5: Top schematic views of SDA85H and SDA85M Trainer kit with 26 pin connector for 8255

In top schematic views of 8085 kit only 8085 IC, 8255 IC, Kit 7-Segment display with keyboard and 26 pin connector for 8255 peripheral are shown for simplicity. There is a three pin jumper JP4 in the trainer kit SDA85H and a two pin jumper JP4 in the trainer kit SDA85M by which the address ranges of different PORT and Control registers are selected. The details of these addresses for selecting different port registers of 8255 is given in the following Table.

Jumper No.	Description	Addresses for selected PORTs & Control Register	
3 pin JP4 (1 & 2 pins shorted)	Select address range (D8H – DFH) for 8255 IC in SDA85H	D8	PORT A
		D9	PORT B
		DA	PORT C
		DB	Control Register
		DC	PORT A
		DD	PORT B
		DE	PORT C
		DF	Control Register

Jumper No.	Description	Addresses for selected PORTs & Control Register	
3 pin JP4 (2 & 3 pins shorted)	Select address range (F0H – F7H) for 8255 IC in SDA85H	F0	PORT A
		F1	PORT B
		F2	PORT C
		F3	Control Register
		F4	PORT A
		F5	PORT B
		F6	PORT C
		F7	Control Register
2 pin JP4 shorted	Select address range (D8H – DFH) for 8255 IC (U4) in SDA85M	D8	PORT A
		D9	PORT B
		DA	PORT C
		DB	Control Register
		DC	PORT A
		DD	PORT B
		DE	PORT C
		DF	Control Register
2 pin JP4 shorted	Select address range (F0H – F7H) for 8255 IC (U3) in SDA85M	F0	PORT A
		F1	PORT B
		F2	PORT C
		F3	Control Register
		F4	PORT A
		F5	PORT B
		F6	PORT C
		F7	Control Register

The 8255 Control Word is given in the following Fig-9.6. Now 8255 can be operated in two modes, one is I/O Mode and another is BSR Mode. The selection of the mode is done with the help of D7 bit of the control word. If D7 = 1, 8255 will be operated in simple I/O Mode and if D7 = 0, 8255 will be operated in BSR (Bit Set Reset) Mode.

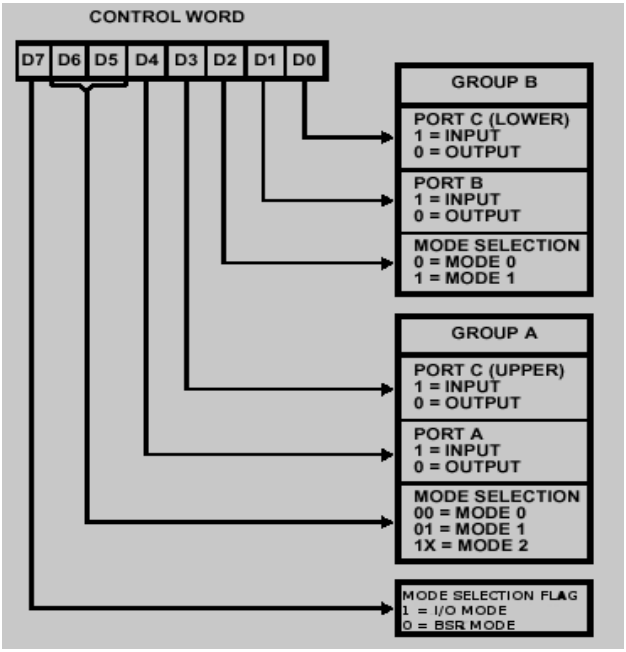


Fig-9.6: Control Word of 8255 Peripheral

In BSR Mode we can access each bit of PORT C individually. In BSR Mode, the control word is shown in Fig-9.7.

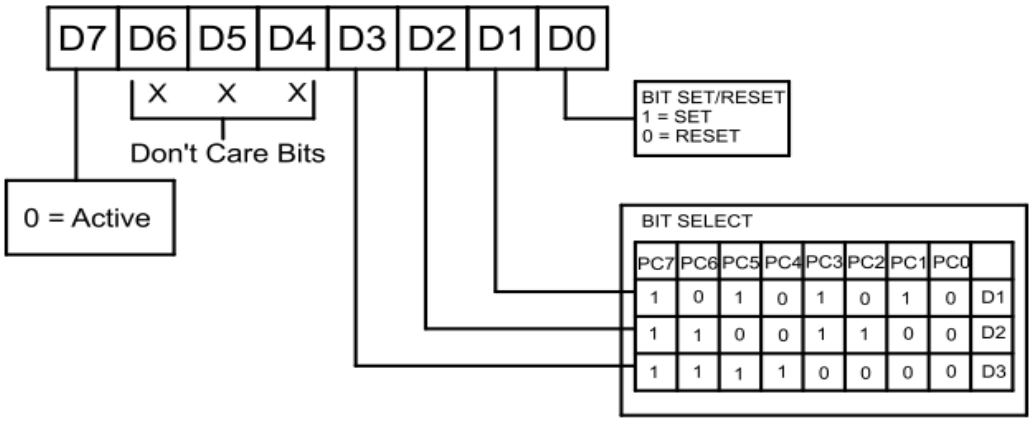


Fig-9.7: Control Word of 8255 Peripheral for BSR mode

There is a 26 pin connector which is used to access port A, port B, port C of 8255 IC in the SDA85 Trainer Kit. The pin connection details of this connector with the 8255 IC is given in the following table.

Pin No. of 26 pin connector	Connection Description
1	PORT line PC4, IC Pin 13
2	PORT line PC5, IC Pin 12
3	PORT line PC2, IC Pin 16
4	PORT line PC3, IC Pin 17
5	PORT line PC0, IC Pin 14
6	PORT line PC1, IC Pin 15
7	PORT line PB6, IC Pin 24
8	PORT line PB7, IC Pin 25
9	PORT line PB4, IC Pin 22
10	PORT line PB5, IC Pin 23
11	PORT line PB2, IC Pin 20
12	PORT line PB3, IC Pin 21
13	PORT line PB0, IC Pin 18
14	PORT line PB1, IC Pin 19
15	PORT line PA6, IC Pin 38
16	PORT line PA7, IC Pin 37
17	PORT line PA4, IC Pin 40
18	PORT line PA5, IC Pin 39
19	PORT line PA2, IC Pin 2
20	PORT line PA3, IC Pin 1
21	PORT line PA0, IC Pin 4
22	PORT line PA1, IC Pin 3
23	PORT line PC6, IC Pin 11
24	PORT line PC7, IC Pin 10
25	JP1 Pin 2 in 85M & 85H
26	GND

8255 Port Connector Board – The pins of Port A, Port B and Port C of 8255 in SDA85H/ SDA85M/ SDA86 kits are randomly placed in the 26 pin connector, which is very difficult to connect to any external board/ device. To avoid this problem we have designed a PCB named 8255 PORT CONNECTOR SDA8085/86 T. KIT with model no. RJ8255PCONN V1.0 to separate these three ports Port A, Port B and Port C. Basically this board separates and segregates the ports of 8255 in SDA85 Trainer Kit into three 8 pin connectors for Port A, Port B and Port C to ease the connection of Ports. These board can be used for 8085 as well as 8086 Trainer Kit. In case of 8086 Trainer Kit +5V supply can be provided to any external device by shorting JP1 and +5V pins of J7 connector in 8255 Port Connector Board. Again if any external power supply is connected to the header pin "EXT PWR_SUPPLY" of the 8255 Port Connector Board, +5V can also be delivered to any external device by shorting JP2 and+5V pins of J7 connector. If power supply is connected properly the LED in the board will glow as an indicator. There are three 2 pin headers placed inside the PWR_OUT section which can be used to provide +5V supply to three different external devices which are to be interfaced with the 8255 in Trainer Kit. In addition to this there is 26 pin Clip-based FRC male connector which is to be connected to the 26 pin 8255 Port connector in the Trainer Kit. The 3D top view of 8255 Port Connector Board is shown in Fig-9.8.

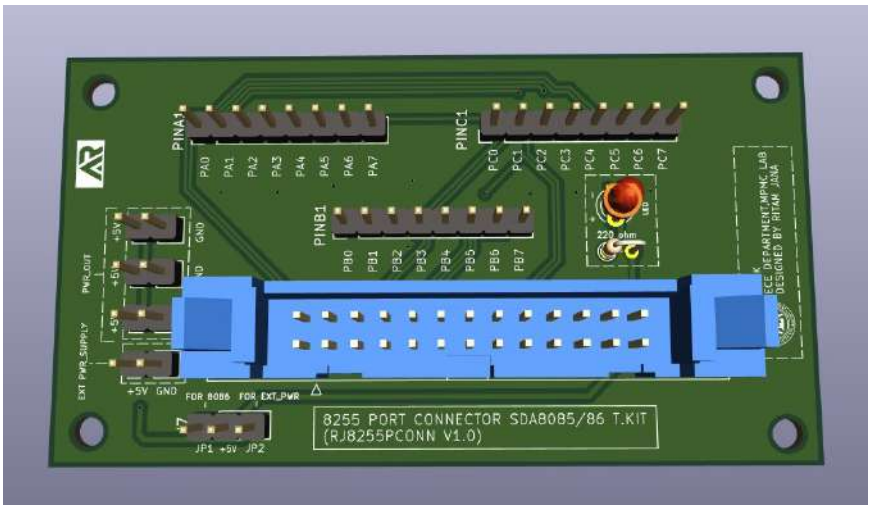


Fig-9.8: 3D top view of 8255 Port Connector Board

Steps to connect external device via 8255 Port Connector Board:

Step 1: Connect 26 pin 8255 connector on SDA85H/ SDA85M/ SDA86 Trainer Kit with the 26 pin male FRC connector on 8255 Port Connector Board using a F-F FRC ribbon cable.

Step 2: If SDA85H/ SDA85M Trainer Kit is used, external power supply is connected through EXT PWR_SUPPLY header on 8255 Port Connector Board and JP2 and +5V pin of J7 connector are to be shorted to provide +5V to the external device.

Step 3: If SDA86 Trainer Kit is used, external power supply may be used as described in Step 2 or +5V may be supplied from the Trainer Kit by shorting JP1 and +5V pins of J7 connector on 8255 Port Connector Board. As soon as +5V is connected, the LED will glow on 8255 Board.

Step 4: Now +5V is supplied to the external device via any one of three PWR_OUT headers on 8255 Port Connector Board.

Step 5: Finally one or two or three ports among Port A, Port B and Port C of 8255 Port Connector Board are connected to the external device as required.

After connecting the external device/ circuitry to the 8085 trainer kit via 8255 Port Connector Board the hex codes of 8085 assembly language program is to be loaded on the SDA85H/ SDA85M Trainer Kit to drive the external device/ circuitry according to the program objective.

9.1: Write a program to blink a set of eight LEDs in a particular pattern using 8255 peripheral chip of SDA85H/ SDA85M Trainer Kit with a time delay of 1 sec.

Eight LEDs are connected to Port B of 8255 IC on 8085 Trainer Kit SDA85H/ SDA85M via 8255 Port Connector Board with current limiting resistors of 220Ω and ICs ULN2803. The necessary +5V power supply is provided to the external circuitry via 8255 Port Connector Board. Now the hex codes of assembly language program is loaded in the memory of 8085 microprocessor of SDA85H/ SDA85M Trainer Kit to blink the LEDs with a time delay of 1 sec. As soon as the program loaded in the memory will be executed, the 8 LEDs will start to blink with a given patterns. To generate a delay of 1 sec a subroutine is written for 8085 microprocessor. The calculation of this delay subroutine is given below.

The crystal of 6.144 MHz is connected to the SDA85H/ SDA85M Trainer Kit.

Therefore operating frequency of 8085 = (6.144 / 2) MHz = 3.072 MHz.

As a result every T-state becomes 0.325 μs i.e. T = 0.325 μs.

The delay subroutine along with no. of T states corresponding to each instruction is given below where BC register pair will be loaded with a 16-bit initial value and this value is unknown initially. Depending upon this value of BC register pair the subroutine will create a delay of 1 sec. Let the initial value of BC pair is n which has been calculated as follows.

SL.	Label	Mnemonics of Delay Subroutine	No. of T States
1	DELAY:	LXI B, n	10 T
2	AGAIN:	NOP	4 T
3		NOP	4 T

4		NOP	4 T
5		NOP	4 T
6		NOP	4 T
7		NOP	4 T
8		DCX B	6 T
9		MOV A,C	4 T
10		ORA B	4 T
11		JNZ AGAIN	10 T / 7 T
12		RET	10 T

Calculation:

Total time taken by the subroutine = 10T + n x 48T – 3T + 10T = (48n + 17) x 0.325 μs

According to the criterion of the program the delay subroutine should be 1 sec.

$\therefore$ (48n + 17) x 0.325 μs = 10^6 μs

or, 48n + 17 = 3076923

or, 48n = 3076906

$\therefore$ n = 64102 = FA66H

Now the delay subroutine will be same with a initial value of FA66H to be stored in BC register pair.

As port B of 8255 is being interfaced with the LED circuit to send the required bit patterns, the Port B will be operated as output port in Mode 0 of simple I/O mode. For that purpose the control word will be evaluated as follows.

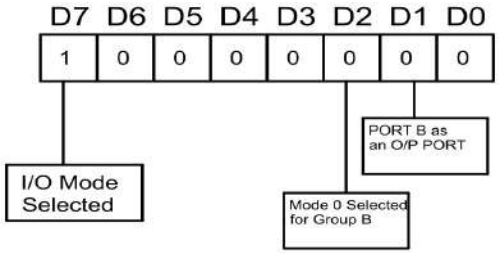


Fig-9.9: PORT B of 8255 selected as an output PORT in Mode 0 in Group B using simple I/O Mode

As we have used SDA85H for implementing this program, the address range selected here is F4H – F7H. Accordingly the address of Port B and control register becomes F5H and F7H respectively. Here the blinking pattern we have chosen that all LEDs will be on simultaneously and after 1 sec delay all LEDs will be off simultaneously, which results the two bit patterns as FFH and 00H respectively. The entire assembly language program is given below.

Assembly language program 9.1:

SL.	Addresses	Label	Mnemonics	No. of Bytes	Hex Codes		
1	8000		MVI A,80	2	3E	80	
2	8002		OUT F7	2	D3	F7	
3	8004		MVI A,FF	2	3E	FF	
4	8006	LOOP:	OUT F5	2	D3	F5	
5	8008		MOV D,A	1	57		
6	8009		CALL DELAY	3	CD	12	80
7	800C		MOV A,D	1	7A		
8	800D		CMA	1	2F		
9	800E		JMP LOOP	3	C3	06	80
10	8011		HLT	1	76		
11	8012	DELAY:	LXI B,FA66	3	01	66	FA
12	8015	AGAIN:	NOP	1	00		
13	8016		NOP	1	00		
14	8017		NOP	1	00		
15	8018		NOP	1	00		
16	8019		NOP	1	00		
17	801A		NOP	1	00		
18	801B		DCX B	1	0B		
19	801C		MOV A,C	1	79		
20	801D		ORA B	1	B0		
21	801E		JNZ AGAIN	3	C2	15	80
22	8021		RET	1	C9		
				TOTAL = 34			

9.2: Write a program to display a single digit BCD number (0 to 9) in a 7-segment display using 8255 peripheral chip of SDA85H/ SDA85M Trainer Kit. Assume that the single digit number will be stored at the address 8060H.

Before implementing this experiment along with program the detail explanation of 7-segment display is required. A 7-segment display is commonly used in electronic display devices for decimal numbers from 0 to 9 and in some cases, basic characters. The use of LEDs in seven-segment displays made it popular, bright and clear, easy to interface and cost effective. There are 7 illuminating segments (named as a, b, c, d, e, f, g) and a dot (named as DP) in a 7-segment display. Corresponding to each segment and dot there is a LED inside the 7-segment display. A particular segment in a 7-segment display becomes illuminated if the corresponding LED of that segment glows due to the forward biasing. The pin-out of a 7-segment display is shown in Fig-9.10.

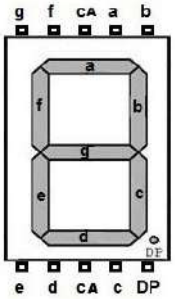


Fig-9.10(a): Pin-out of Common Anode 7-segment display

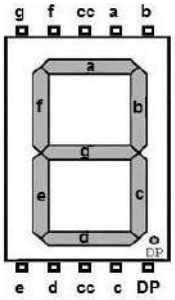


Fig-9.10(b): Pin-out of Common Cathode 7-segment display

Basically there are two types of 7-segment display namely 1) Common Anode 7-segment display and 2) Common Cathode 7-segment display.

1) Common Anode 7-segment display – In this construction all the anodes of eight LEDs are connected together to form a common terminal CA as shown in Fig-9.8(a). Other eight cathode terminals are connected to eight pins namely a, b, c, d, e, f, g and DP. The internal schematic diagram of a common anode 7-segment display is shown in Fig-9.11.

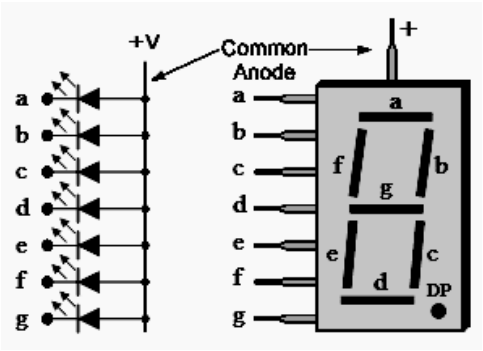


Fig-9.11: Internal schematic diagram of common anode 7-segment display

2) Common Cathode 7-segment display – In this construction all the cathodes of eight LEDs are shorted together to form a common terminal CC as shown in Fig-9.8(b). Other eight anode terminals are connected to eight pins namely a, b, c, d, e, f, g and DP. The internal schematic diagram of a common cathode 7-segment display is shown in Fig-9.12.

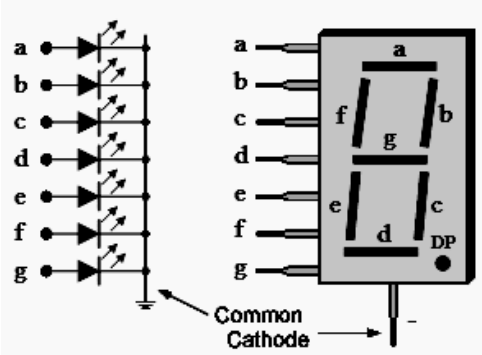


Fig-9.12: Internal schematic diagram of common cathode 7-segment display

Suppose in this case PORT B of 8255 is selected for sending the 8-bit data to display a single digit BCD number on a 7-segment display and the address range F0 – F3 is selected for different PORT registers and Control register. For this purpose the following circuit is to be connected. The interfacing circuit to display a single digit BCD number on a 7-segment display through 8255 chip, is shown in Fig-9.13.

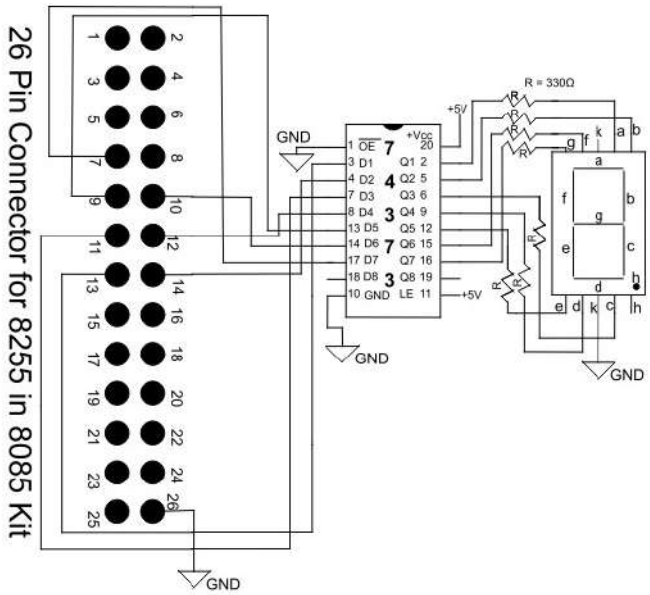


Fig-9.13: Circuit diagram to display single digit BCD on 7-segment display using 8255

Here PORT B is used as an output PORT for sending corresponding code to 7-segment display for the single digit number in the range of 0 to 9. Hence the control word of 8255 is given in Fig-9.14.

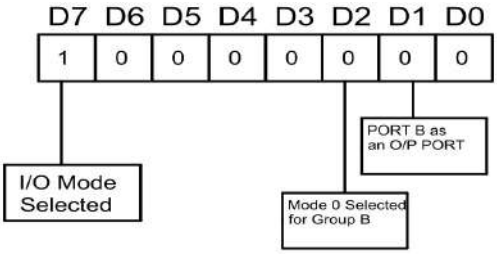


Fig-9.14: PORTB of 8255 selected as output PORT in Mode 0 in Group B using simple I/O Mode

So we can see from Fig-9.9 that the control word will be 80H to operate PORT B as an output PORT and this control word is to stored inside the control register using address F3. After storing the specified control word, the PORT B is configured as an output PORT and ready to send data to the inputs of the LATCH 74373. The LE pin should be high and $\overline{OE}$ pin should be low to transfer a 8-bit data from its input pins (D1, D2, D3,.....) to output pins (Q1, Q2, Q3,.....) and depending on these data the single digit number is glown on the 7-segment display. Now there are 10 8-bit codes corresponding to 10 single digit numbers which will be displayed and because of that these codes are to be stored in a look-up table which is basically a block of memory to store these code for 7-segment display. Suppose the look-up table starts from memory address 8050 to 8059 and is shown in the following Fig-9.15. The single digit number to be displayed, should be stored in the memory address 8060.

PB7	PB6	PB5	PB4	PB3	PB2	PB1	PB0	Code in Hex	Memory Addresses to store the codes for 7-seg display	Displayed Single Digit Number
DP	g	f	e	d	c	b	a			
0	0	1	1	1	1	1	1	3F	8050	0
0	0	0	0	0	1	1	0	06	8051	1
0	1	0	1	1	0	1	1	5B	8052	2
0	1	0	0	1	1	1	1	4F	8053	3
0	1	1	0	0	1	1	0	66	8054	4
0	1	1	0	1	1	0	1	6D	8055	5
0	1	1	1	1	1	0	1	7D	8056	6
0	0	0	0	0	1	1	1	07	8057	7
0	1	1	1	1	1	1	1	7F	8058	8
0	1	1	0	1	1	1	1	6F	8059	9

Fig-9.15: Look-up table for single digit number display on a 7-segment display

The flowchart of the assembly language program for 8085 kit to display a single digit number on a 7-segment display is shown in the following Fig-9.16.

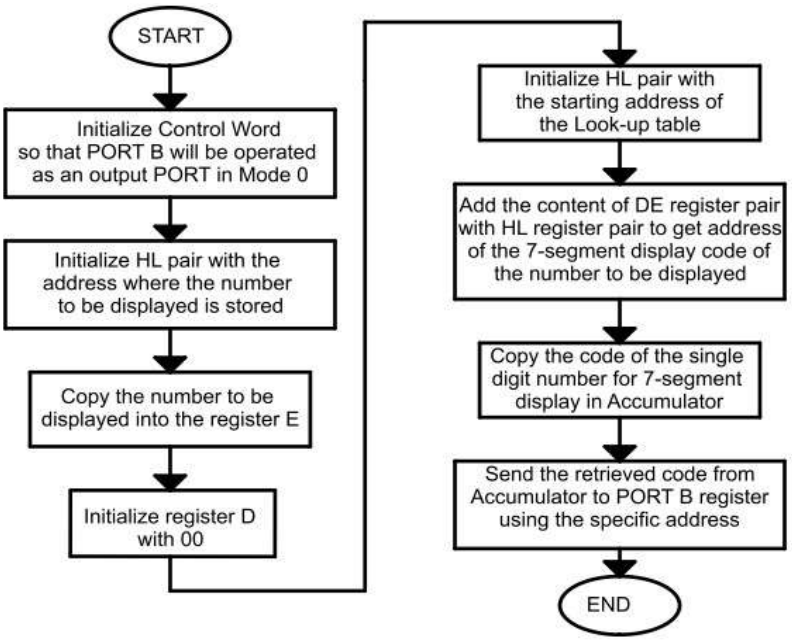


Fig-9.16: Flowchart of the program to display a single digit number on a 7-segment display

Assembly Language Program 9.2:

Note: 1) Look-up table for 7-segment display codes starts from the address 8050
2) The single digit BCD number to be displayed, is stored at address 8060

SL.	Addresses	Label	Mnemonics	No. of Bytes	Hex Codes		
1	8000		MVI A, 80	2	3E	80	
2	8002		OUT F3	2	D3	F3	
3	8004		LXI H, 8060	3	21	60	80
4	8007		MOV E, M	1	5E		
5	8008		MVI D, 00	2	16	00	
6	800A		LXI H, 8050	3	21	50	80
7	800D		DAD D	1	19		
8	800E		MOV A, M	1	7E		
9	800F		OUT F1	2	D3	F1	
10	8011		RST 5	1	EF		
				TOTAL = 18			

10. Familiarization with 8051 Simulator

The simulation of 8051 microcontroller programs are done by using a simulation software "Keil". Keil is a well-known integrated development environment (IDE) used primarily for embedded systems software development. It is developed by ARM, a leading provider of semiconductor intellectual property. Here's an overview of Keil highlighting its key features and functionalities:

1. **IDE Interface**: Keil provides a user-friendly interface that integrates various tools required for embedded systems development. It includes a text editor, project manager, and debugger, among other tools, all within a unified environment.
2. **Support for ARM Architecture**: Keil primarily targets the ARM architecture, which is widely used in embedded systems ranging from microcontrollers to sophisticated system-on-chip (SoC) designs. It supports various ARM cores and families, including Cortex-M, Cortex-R, and Cortex-A series processors.
3. **Compiler and Debugger**: Keil includes a robust C/C++ compiler optimized for ARM architectures, capable of generating highly efficient code for embedded applications. It also features a powerful debugger that supports source-level debugging, real-time data visualization, and various debugging techniques like breakpoints, watch points, and trace.
4. **Peripheral Simulation**: Keil provides simulation capabilities that allow developers to simulate the behavior of various peripherals commonly found in microcontrollers and other embedded devices. This feature is valuable for testing and debugging embedded software without the need for physical hardware.
5. **RTOS Support**: Keil offers support for various real-time operating systems (RTOS), including popular ones like FreeRTOS and RTX (Keil's own RTOS). This allows developers to efficiently develop and debug embedded applications that utilize multitasking and real-time scheduling.
6. **Extensive Device Support**: Keil provides extensive device support, including a wide range of microcontrollers and development boards from various manufacturers. This makes it easier for developers to select the appropriate target device for their projects and ensures compatibility with Keil's tool chain.
7. **Middleware and Libraries**: Keil offers a suite of middleware components and libraries that simplify common tasks in embedded systems development. This includes libraries for communication protocols (e.g., UART, SPI, I2C), file systems, graphics, and more.

8. **Integration with ARM Ecosystem**: As part of ARM's ecosystem, Keil seamlessly integrates with other ARM development tools and technologies, such as ARM Development Studio, CMSIS (Cortex Microcontroller Software Interface Standard), and ARM's IP portfolio.
9. **Community and Support**: Keil has a large user community and extensive documentation, tutorials, and forums available to developers. This ensures that developers have access to resources and assistance when using the IDE and tackling embedded systems development challenges.

Overall, Keil is a comprehensive IDE used for embedded systems development, offering powerful tools, extensive device support, and integration with ARM's ecosystem to streamline the development process for embedded applications. In addition to these supports it also supports 8051, 8052, 8031 microcontroller that assembly language as well as C programming for 8051 microcontroller can be executed using this IDE. Keil is only compatible with Windows operating system, but it can also be installed in Ubuntu Linux using "Wine" application.

10.1 Installation of Keil Uvision

Installation on Windows:
Download Keil uVision5 and install the free version or Trial version of this software. Only limitation of this Trial version is that maximum 2KB of program code can be compiled and executed in Keil Trial Version.

Installation on Ubuntu:
Step1 – Install the application Wine using the following command.

$ sudo apt install wine

Step2 – After successful installation of wine, configure it running the following command.

$ winecfg

Step3 – Now copy the setup file of Keil IDE in a director, navigate into that directory and execute the following command.

$ wine KeilSetupFile.exe

Step4 – Continue the installation procedure as Windows.

10.2 How to Use: The screenshots of Keil uVision5 are shown in Fig-10.1.

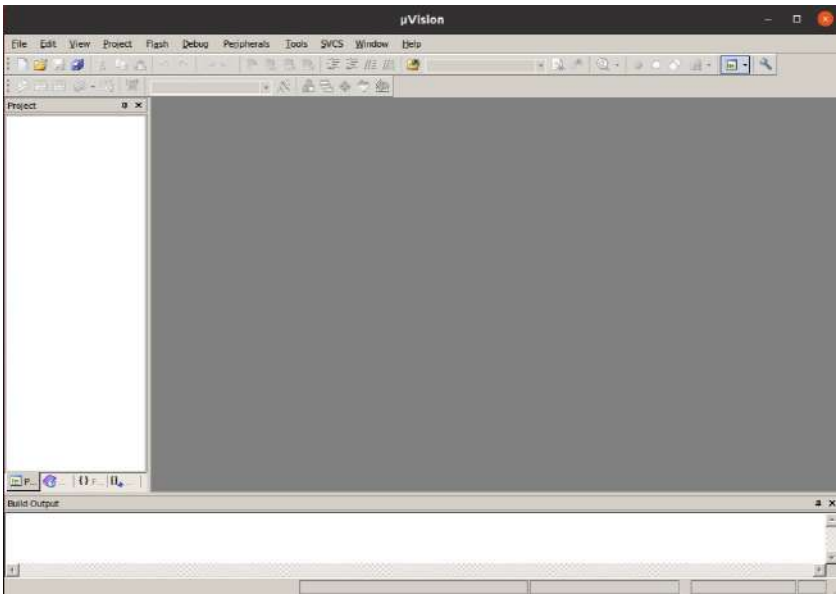


Fig-10.1: Screenshot of Keil after opening first time

To write a 8051 program in Keil a directory or folder should be created with relevant name. A project file with relevant name and .uvproj extension should be created in the same directory. Generally the project name and the name of the folder are kept same. After creating the project the program of 8051 can be written in either assembly language or C language. The program written in assembly/ C language should be saved on an file with .asm/ .c extension. Now this file is to be added to the created project to execute it. The step-by-step procedure of writing 8051 assembly language program in Keil is given below.

10.3 Procedure to write 8051 assembly language program:

- Click the option "New uVision Project" under the Project menu, enter a name of the project (normally the same name of the directory) and save the project file with .uvproj in the previously created directory.

- A window will appear where the specific microcontroller from a list of microcontrollers should be selected. In this laboratory AT89S51 or AT89S51 should be chosen as device.

- After that a message "Copy 'STARTUP.A51' to Project Folder and Add File to Project" will appear where 'No' option is to be selected for program in assembly language and 'Yes' option to be selected for program in C language.

- After clicking 'No' option a 'Target' option will be generated in the project window at left. If we expand it, 'Source Group 1" option will appear as shown in Fig-10.2.

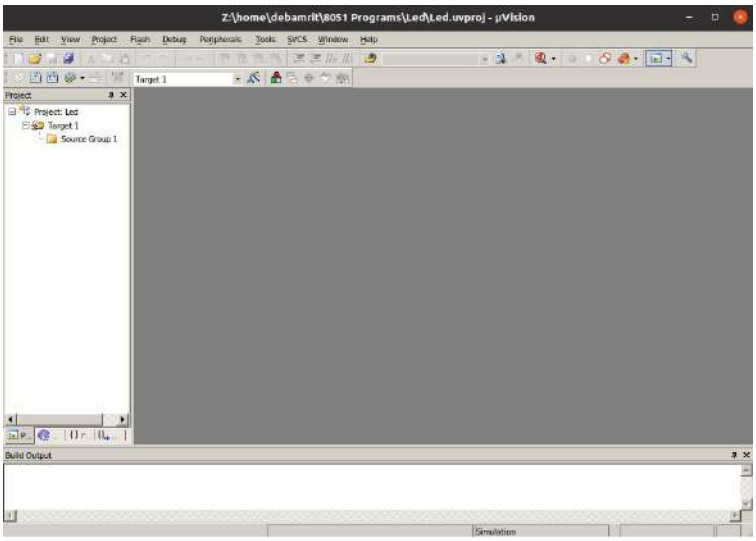


Fig-10.2: Screenshot of Keil after creating a new project

- Now right click the option 'Target' and select 'Option for Target Target1' to configure the following options for the project.
    1. Verify the microcontroller as AT89S51/ AT89S52 under 'Device' Tab
    2. Change the crystal frequency in the text box 'Xtal (MHz)' as per the crystal used in your circuit. For example – if the crystal of 11.0592MHz is used, enter the frequency in the text box as 11.0592.
    3. Check the option "Create HEX File" under 'Output' Tab.

  After selecting the above mentioned configurations, press ok button.

- Now open a new file from 'File' menu which will appear as in Fig-10.3.

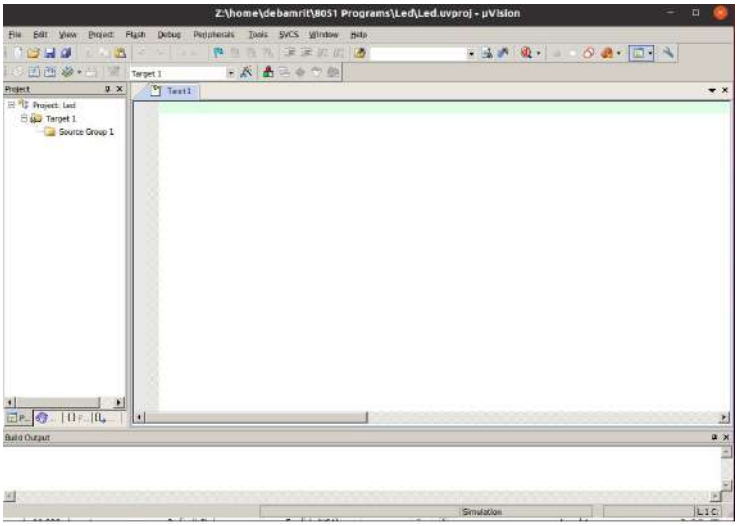


Fig-10.3: Screenshot of Keil after opening a new file

- Now the assembly language code is written in the white space and saved with the same filename of project with an extension .asm in the same directory where the project file has been stored also. As soon as the file is saved with .asm extension, the text highlighting is activated in the program code which helps to find out the syntax error. A sample program with text highlighting in Keil is shown in Fig-10.4. In Keil every assembly language program should be started with the directive "ORG Starting Address of the Program" and terminated with another directive "END".

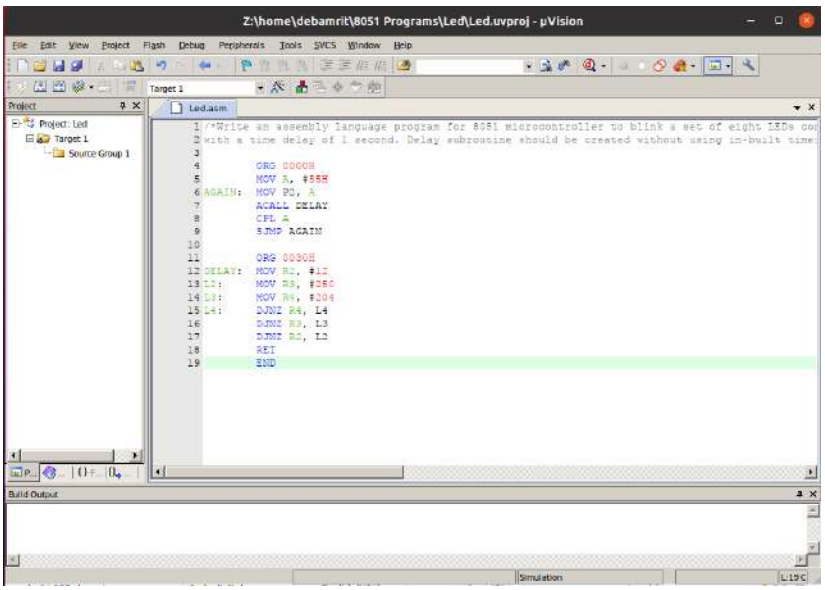


Fig-10.4: Sample 8051 assembly language program with text highlighting

- Finally this .asm file will be added to the project by right clicking on the 'Source Group 1' and selecting the option 'Add Existing Files to Group Source Group 1'. The program writing is complete and now the program can be compiled, built and executed.

- ***Comment inside program:*** To give a comment inside the program, the comment line must be preceded by '/*' and terminated by '*/'. The simulator excludes these comment lines during debug. Giving comment in the program is a good practice to specify explanation of the program. This practice helps the programmer to recapitulate the logic of a complicated program in future.

- ***Storing Data inside Program Memory:*** To store the data inside program memory prior to the execution of the program the following assembler directives should be used.

  ORG Address of the Memory from where data can be stored consecutively
  DB Data1, Data2, Data3, Data4, ……..

  DB (Data Byte) is a directive which stores all the 8-bit data (Data1, Data2, Data3,…..) consecutively starting from the address specified by "ORG".

```
ORG 200
DB 3FH, 06H, 5BH, 4FH, 66H, 6DH, 7DH, 07H,7FH, 6FH
```

In the above example the simulator will load ten 8-bit data (3FH, 06H, 5BH, 4FH, 66H, 6DH, 7DH, 07H,7FH, 6FH) consecutively starting from memory location 200 (in decimal) in the program memory.

- Keil is not case-sensitive. So the program code can be written either in capital or small letter.
- In Keil the decimal value is represented only by number like 15,100 etc and Hexadecimal value is represented by the number followed by 'H' like 0FH, 64H etc.
- In Hexadecimal representation, if the first digit is alphabet, zero must be appended before it to avoid the syntax error in Keil. For example – MOV R2, #0E2H and MOV R2, #2EH.

10.4 Procedure to Build/ Rebuild 8051 project:

- After completing the program writing the project is to be built by clicking the 'Build' option in Keil as shown in Fig-10.5. Actually Build will perform multiple tasks like generation of List file (.lst), creation of object file (.obj), generation of Hex file (.hex) etc. In addition to this it will check errors in program. If *0 warning 0 error* occurs, Build process is completed successfully.
- After completion of Build process a list file with .lst extension is created into the folder "Listings" and a Hex file is created into the folder "Objects". The list file contains the entire program code along with the memory mapping, opcodes, operands etc. The Hex file contains the program code in Hexadecimal numbers. Basically this Hex file is used by the Programmer Software to burn the program code into the microcontroller chip.
- If there is some modification in the program, the project should be rebuilt by using the option 'Rebuilt' as shown in Fig-10.5. This will modify all the files like list file, hex file etc.

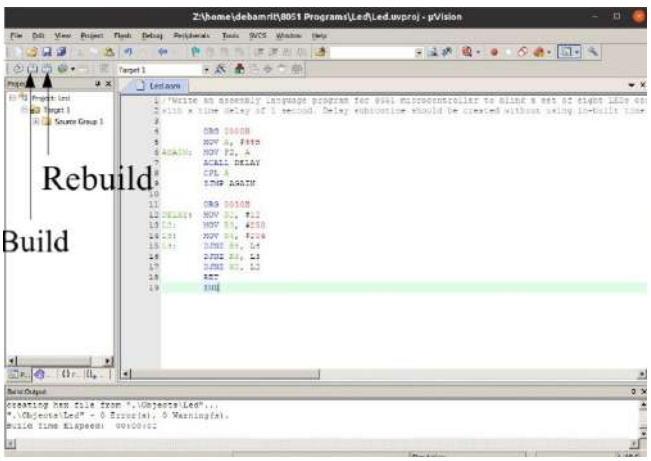


Fig-10.5: Build and Rebuild option in Keil

10.5 Procedure to execute 8051 program:

- To execute the program in Keil the option "Start/ Stop Debug Session" should be clicked to start Debug Session. In Keil any program can execute only in this session. That's why to run any program we must enter into this Debug Session first.

- If the program involves any peripherals like I/O ports, then the corresponding I/O port (Port 0, Port1, Port2, Port3) should be selected from Peripherals → IO Ports. It will display the I/O port wizard in front of the user as shown in Fig-10.6.

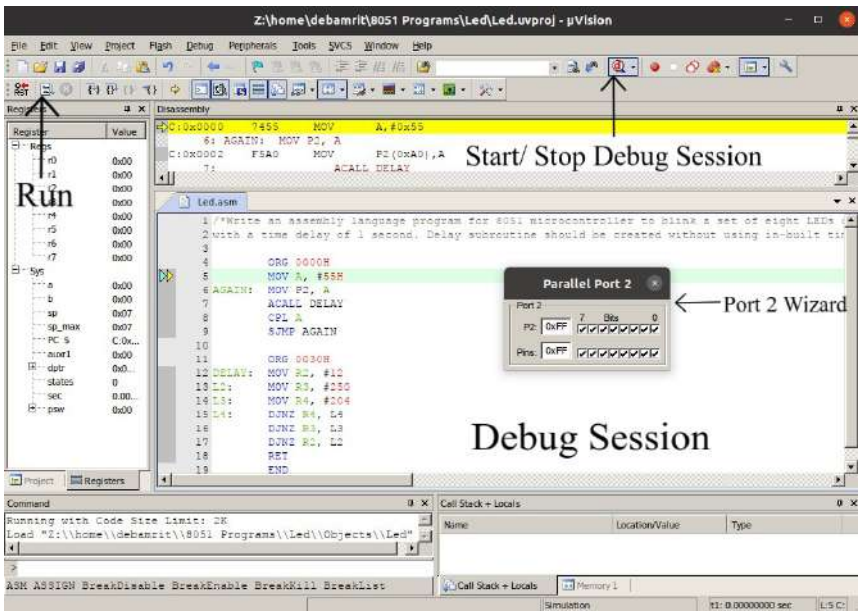


Fig-10.6: Debug Session in Keil

From the above figure we can see 8 pins of Port 2 is marked with ✓ which indicates On and if ✓ is not present, it indicates Off at that pin.

- Now the Run button is to be clicked to execute the program. During execution the port wizard functions according to the program.

- In 8051 most of the time the programs run indefinitely. Therefore Stop button should be clicked to finish the program execution.

- Finally click 'Start/ Stop Debug Session' button once again to return back to the Editor Mode of Keil where any modification of the program may be done.

10.6 Procedure to store Data inside RAM during program execution: We know 8051 has 128 bytes of onchip RAM. To access 128 bytes (2^7 bytes) of RAM seven address lines are required. So each address of onchip RAM of 8051 is 8 bits long. The address range of 8051 RAM is 00H – 7FH. The address range 00H – 1FH is dedicated for four memory banks namely Bank0, Bank1, Bank2 and Bank3, the address range 20H – 2FH is used as bit addressable memory and 30H – 7FH is used as scratchpad where programmer can store some data temporarily as per his requirement.

Keil has given the facility to store data inside scratchpad of onchip RAM by the use of the following process.

- Open debug session by clicking the button "Start/ Stop Debug Session" as shown in Fig-10.7.

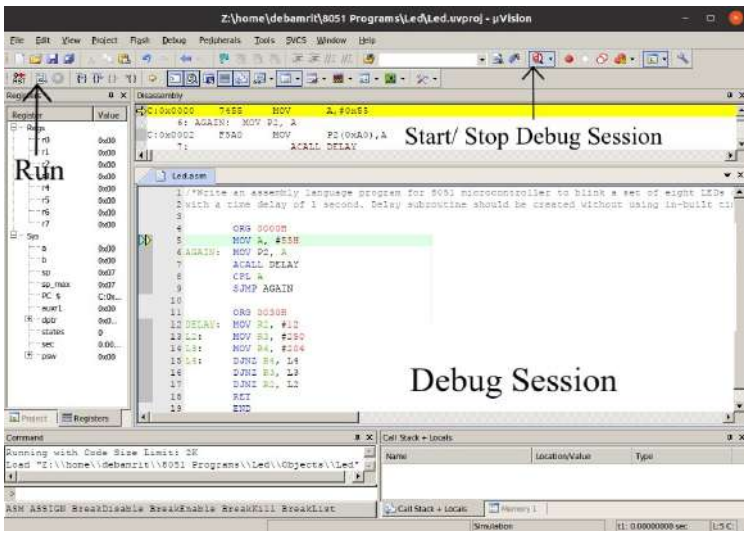


Fig-10.7: Debug Session in Keil

- Now click the tab 'Memory1' at the bottom right corner of the debug window which will open the memory window as shown in Fig-10.8.

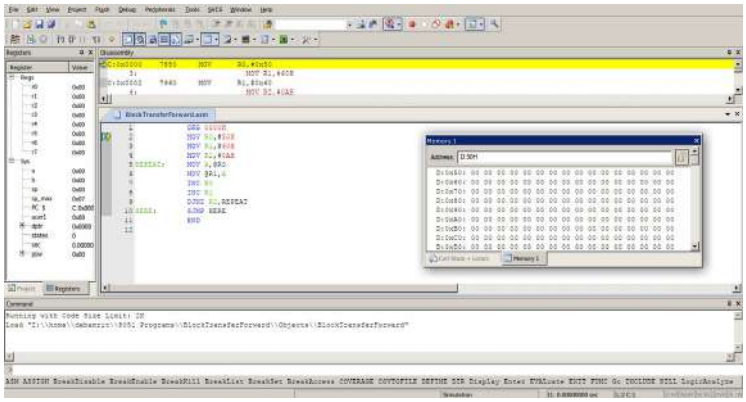


Fig-10.8: Memory window for 8051

- In this memory window you can insert data for onchip RAM as well as for Program ROM (flash memory). To store data inside onchip RAM we have to follow insert the syntax into the text box of the memory window as given below.

  D:8 bit address in Hexadecimal

  For example – D:50H which will allow to insert data starting from the address 50H onward.

  Similarly to insert data into the onchip PROM the syntax "C:16bit address in Hexadecimal" should be given in the text box. For example C:0000H will allow to store data starting from the memory address 0000H.

- After entering the above mentioned syntax when the enter button is pressed, the contents of all the addresses from the given address will be displayed in the memory wizard. Now the content which is to be changed, is double clicked to select the particular address inside the RAM. The content of the selected location is altered as per the need of the user and finally 'Enter' button is pressed to save the data into the selected memory location. The sequences to store FFH into the memory location 53H inside the scratchpad is shown in Fig-10.9, Fig-10.10 and Fig-10.11 respectively.

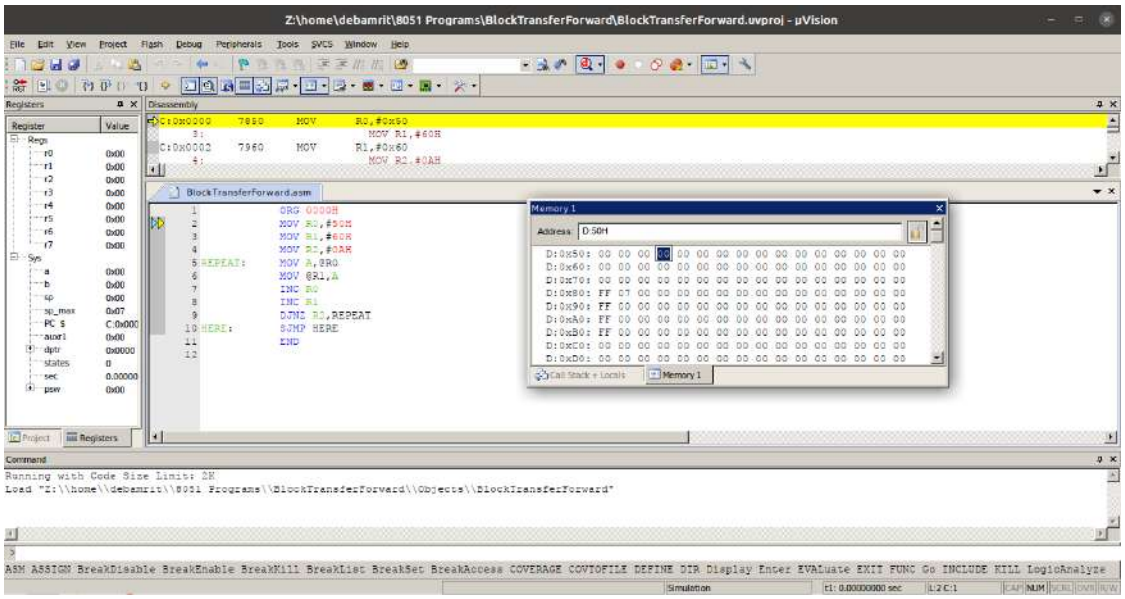


Fig-10.9: Memory location 53H is selected by double click

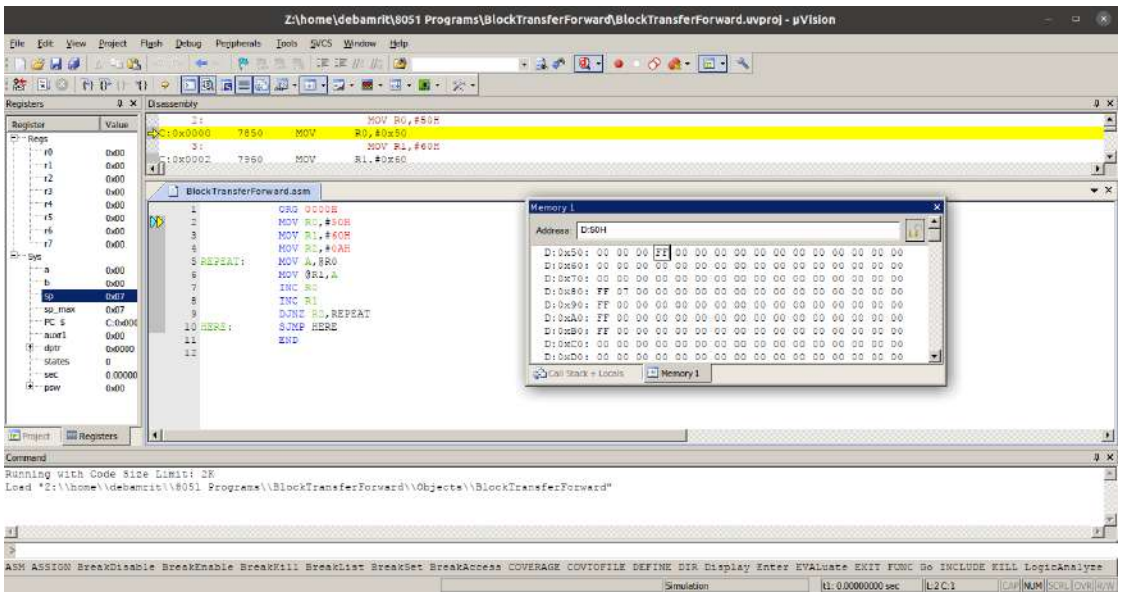


Fig-10.10: Content of memory location 53H has been changed to FFH

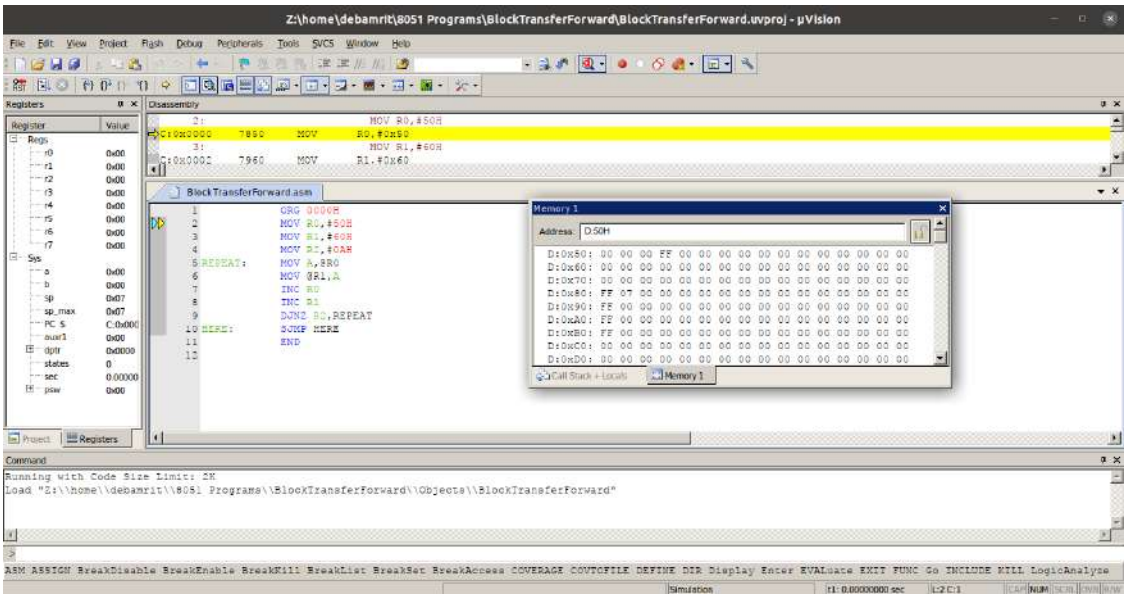


Fig-10.11: New data FFH has been saved into memory location 53H after pressing 'Enter'

11. Procedure to burn 8051 microcontroller

8051 microcontroller can decode only binary numbers or machine level language. That's why only the hex codes should be written into the onchip program memory or flash memory. The Keil simulator converts the assembly language codes to its equivalent hex codes which are stored into a Hex file with .hex extension. Now this Hex file is used by the programmer software to dump all the hex codes into the flash memory of the 8051 microcontroller. The procedure to dump the hex codes of a program onto the flash memory of a microcontroller is called burning/ programming the microcontroller chip.

11.1: Hardware Description: For this purpose we require a programmer device which will be connected to the computer via Serial port or USB port. After successful connection with PC a burner software will be used to dump the hex codes to the microcontroller through the programmer device. Here we have used USBASP device and ProgISP software to burn the 8051 microcontroller chip. The USBASP circuit board is shown in Fig-11.1.

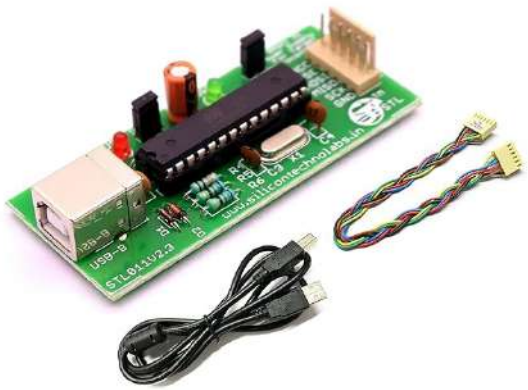


Fig-11.1: 8051 USBASP Programmer board

There are 6 pins in the circuit board of USBASP programmer shown above namely Vcc (+5V of USB port of PC), GND, MOSI (Master Out Slave In), MISO (Master In Slave Out), SCK and RESET. Now the 8051 microcontroller which is to be burnt is connected with the USBASP and +5V power supply as below.

SL	Pins of 8051	Connection	SL	Pins of 80511	Connection
1	Vcc, IC pin 40→	Vcc of USBASP	5	MISO, IC pin 7→	MISO of USBASP
2	GND, IC pin 20→	GND of USBASP	6	SCK, IC pin 8→	SCK of USBASP
3	EA′, IC pin 31→	Vcc of USBASP	7	RESET, IC pin 9→	RESET of USBASP
4	MOSI, IC pin 6→	MOSI of USBASP	8	XTAL2, IC pin 18→ XTAL1, IC pin 19→	Crystal connected between these 2 pins

To establish the above mentioned connection an 8051 microcontroller mounting board (KSR805152-MB1) has been designed where a ZIF socket has been introduced into the board to easily connect and disconnect the 8051 chip frequently. The circuit diagram of this mounting board is shown in Fig-11.2.

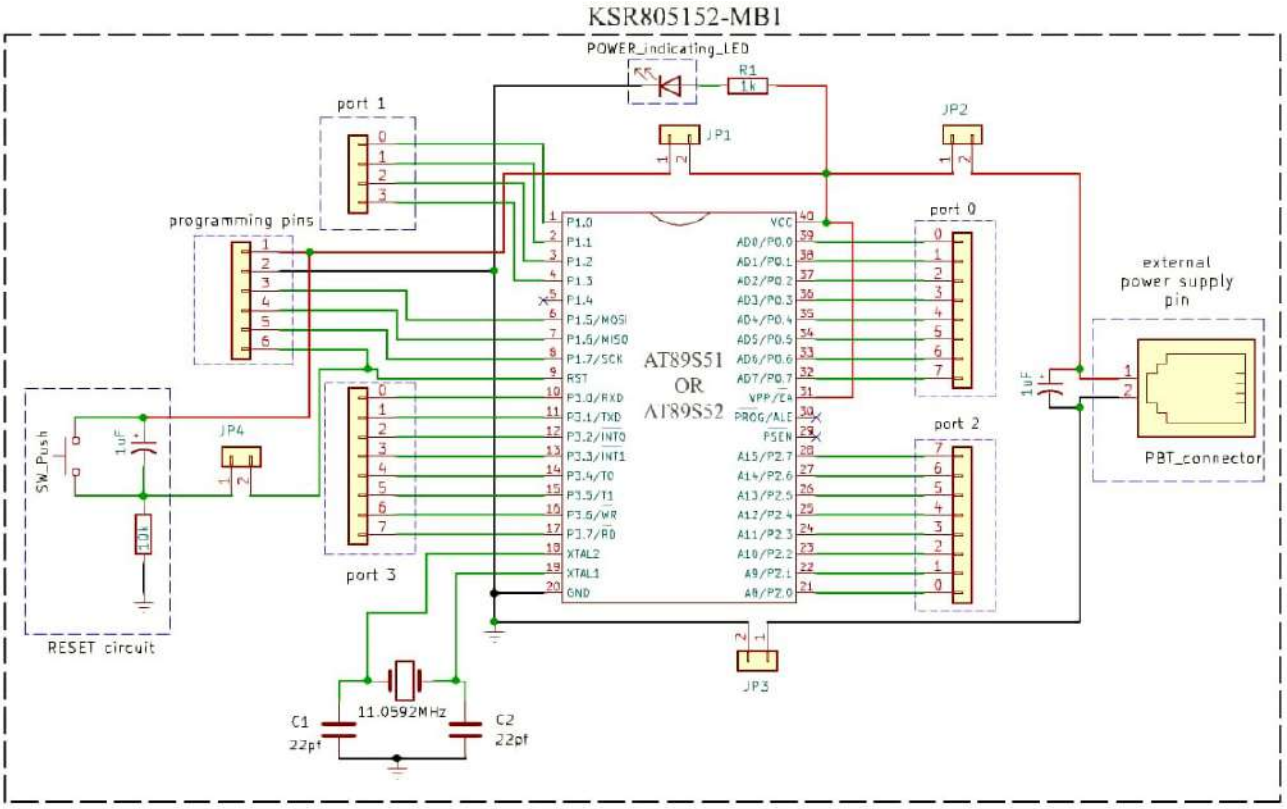


Fig-11.2: Circuit diagram of KSR805152-MB1 mounting board

AT89S51 microcontroller has ISP (In-System Programming) feature which enables the burner to flash the memory of the microcontroller when it is connected to the actual system or in circuit. That means, it is not required to pull out the microcontroller from the system for burning. Keeping the microcontroller chip into the system, it is possible to flash the program memory. The USBASP programmer along with the KSR805152-MB1 mounting board can do the In-System Programming. Moreover the KSR805152-MB1 board gives the facility to provide power from external supply through a PBT connector. There are four jumpers namely JP1, JP2, JP3, JP4 having different functionalities given in the following table.

Purposes	Status of the jumpers
To burn the 8051/ 8052 chip disconnecting from system	JP1 → shorted JP2 → open JP3 → shorted/ open JP4 → open (RESET circuit disconnected)
To perform ISP (In-System Programming) on 8051/ 8052 chip *Note: External power supply must be disconnected from PBT connector*	JP1 → shorted JP2 → shorted JP3 → shorted JP4 → open (RESET circuit disconnected)
Run the 8051/ 8052 chip along with its I/O circuits from USB power supply *Note: External power supply must be disconnected from PBT connector*	JP1 → shorted JP2 → shorted JP3 → shorted JP4 → shorted (RESET circuit connected)
Run the 8051/ 8052 chip along with its I/O circuits from external power supply *Note: External power supply must be connected at PBT connector*	JP1 → open JP2 → shorted JP3 → shorted JP4 → shorted (RESET circuit connected)

The Mounting Board has a RESET circuit which is used to reset the 8051 microcontroller. When the push button switch is pressed momentarily, the microcontroller will be reset. There are three 8 pin jumper headers connected to port 0, port 2, port 3 and a 4 pin jumper header connected to port1 (P1.0 – P1.3). These jumper headers are utilized to connect the ports of the microcontroller to the external circuitry. A 6 pin jumper header connected to MOSI, MISO, SCK, RESET, Vcc and GND of the microcontroller chip is used to flash/ burn the program memory of the 8051 chip with the help of USBASP burner. The KSR805152-MB1 and Embeddinator's 8051/ 8052 Mounting Board are shown in Fig-11.3(a) and Fig-11.3(b) respectively.

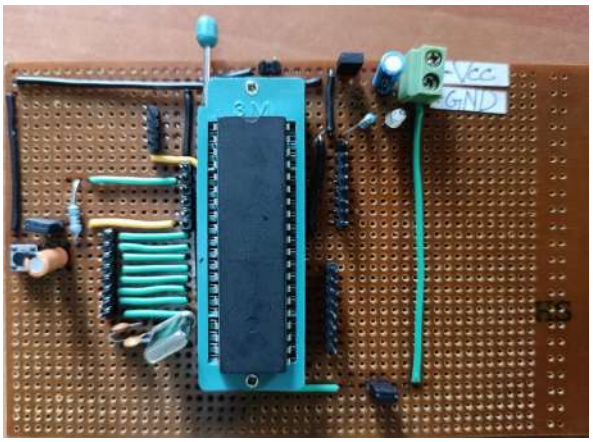


Fig-11.3(a): KSR805152-MB1
8051 microcontroller mounting board

Fig-11.3(b): Embeddinator 8051/ 8052
microcontroller mounting board

11.2: Software Description: The software used to burn 8051 program into the chip is ProgISP. It is a software tool commonly used for programming microcontrollers, particularly those manufactured by Atmel, such as 8051, 8052, AVR series microcontrollers. It provides an interface for users to program and configure microcontrollers for various applications. ProgISP typically supports a range of programming hardware, such as ISP (In-System Programming) programmers, which allow users to program microcontrollers while they are already installed in a circuit. The software often provides features like reading, writing, verifying, and erasing microcontroller memory, as well as configuring fuses and other settings. It's a crucial tool for embedded systems developers and hobbyists working with Atmel microcontrollers. This software is only compatible in Windows operating system. The procedure to install the software is given below.

Step1 – To run ProgISP in Windows it is necessary to install the Windows Driver first. If the driver is not installed, the USBASP programmer device is not detected under Device Manager even after the device is connected to PC via USB port. After the successful installation of the Windows Driver, 'Usbasp' is automatically displayed under 'LibUSB' tab in the Device Manager of Windows as shown in Fig-11.4.

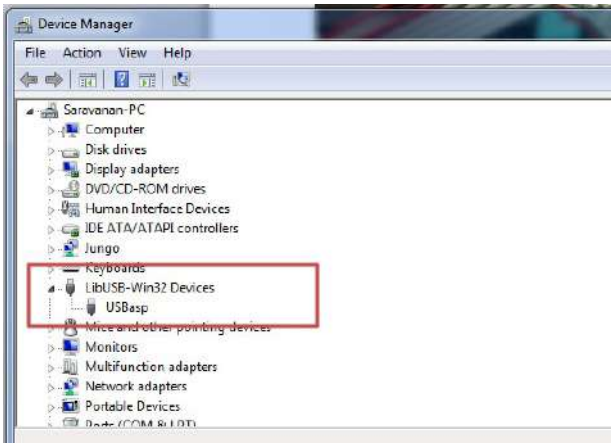


Fig-11.4: USBasp device is detected after the installation of Windows Driver of ProgISP

The windows driver for ProgISP is normally found inside the win-driver folder of ProgISP as shown in Fig-11.5.

Fig-11.5: Windows Driver of ProgISP inside the folder win-driver

Open the folder win-driver and install the driver software for your operating system (Windows 10/ Windows 7).

Step2 – Now ProgISP software can be opened directly by double clicking on the progisp.exe file as shown in Fig-11.5. To do this the directory containing progisp.exe along with other files should be copied into the system and double click on progisp.exe to open and run the software. The screenshot of the software is shown in Fig-11.6. The PRG ISP & USB ASP buttons should be bright. If it's greyed out , then check the USB cable connection & the driver installation.

Fig-11.6: ProgISP Software after opening

11.3 Procedure to burn hex code using ProgISP:

Step1 – Select the target chip (AT89S51 for 8051 microcontroller or AT89S52 for 8052 microcontroller) from the "Select Chip " drop down menu and ensure that following options are enabled as shown in Fig-11.7. If you enable the LOCK CHIP button , others can't make a copy of your chip.

1) Verify Signature 2) Chip Erase 3) Program Flash 4) Verify Flash

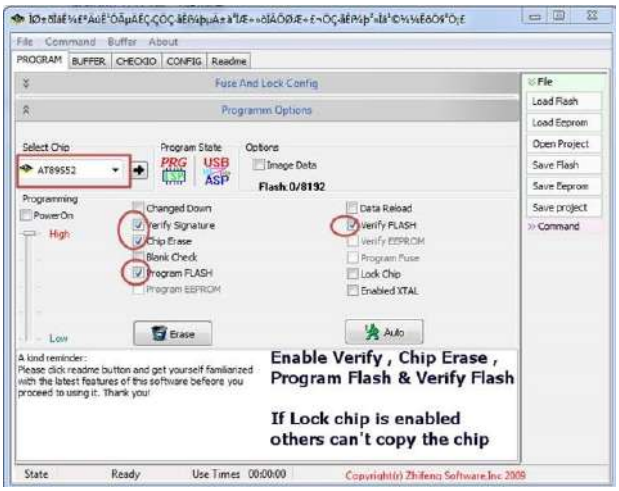


Fig-11.7: Target chip selected with some options enabled

Step2 – Click on File → Load Flash or directly click on the button 'Load Flash' at the right side panel and browse to the location of the Hex file you have created using Keil as shown in Fig-11.8 and Fig-11.9 respectively.

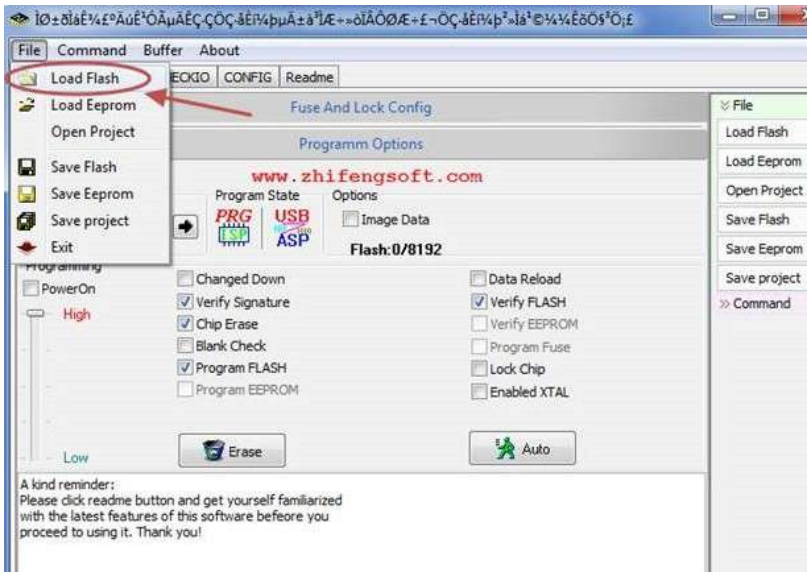


Fig-11.8: Load Flash button to select and load Hex file

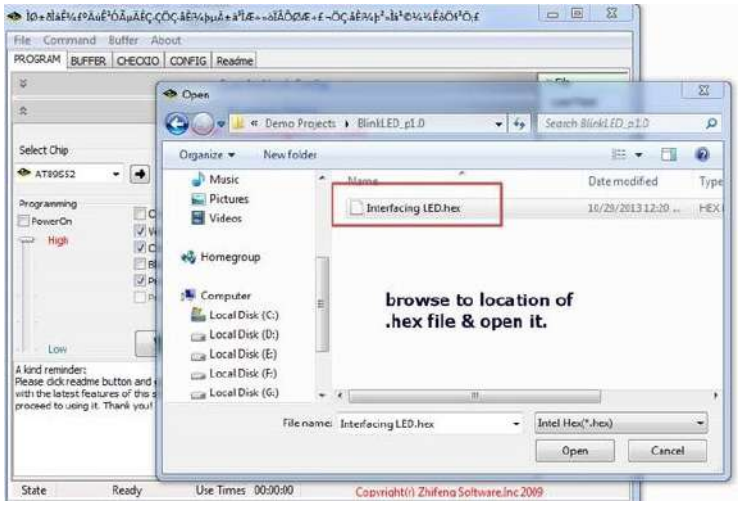


Fig-11.9: Browse the Hex file

Step3 – After selecting the Hex file click the button 'Auto' to flash the program memory of 8051 chip with the Hex file selected. Finally a message "Erase, Write Flash, Verify Flash successfully done" appears to indicate that 8051 program has been loaded into the 8051 chip successfully as shown in Fig-11.10 and Fig-11.11 respectively.

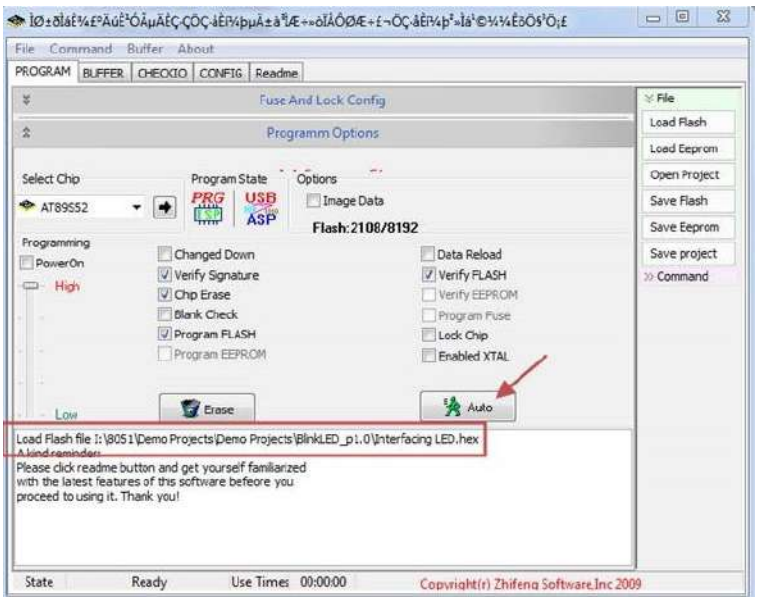


Fig-11.10: Auto button to flash the hex code into the microcontroller chip

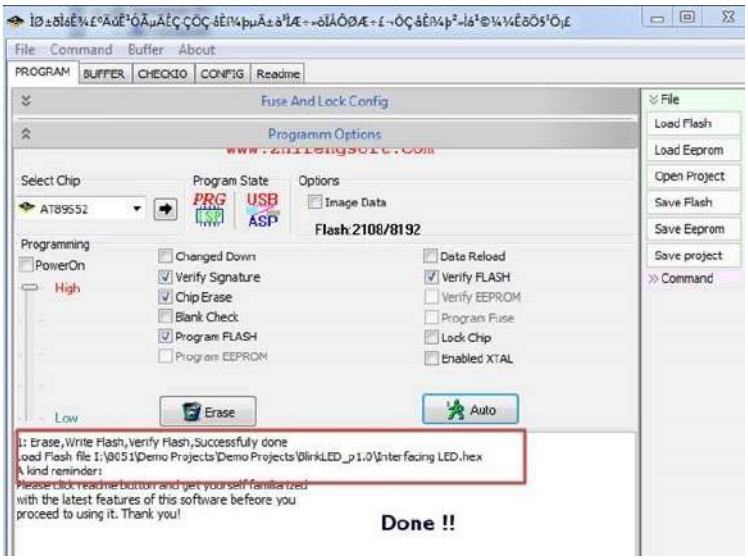


Fig-11.11: Hex file is burnt successfully

12. Programs on Arithmetic and Logical Operations

12.1: Write a program to add two 8-bit binary numbers which are stored at the memory locations 50H and 51H and also store the result of addition into memory locations 52H and 53H where 52H and 53H will hold lower and higher byte respectively.

Method 1: In case of addition of two 8-bit binary numbers, the maximum result will be 1FE when both of the numbers are maximum i.e. FF (FF + FF = 01FE). Hence it is clear that we need an extra bit to store the result which implies that two bytes are required to store the result where carry part will constitute the higher byte and the remaining 8-bit of the result will constitute lower byte. To store this two bytes of result two consecutive memory locations with addresses 52H and 53H are required.

Assembly Language Program 12.1 (Method 1):

SL.	Label	Instructions of 8051
1		MOV B,#00H
2		MOV R0,#50H
3		MOV A,@R0
4		INC R0
5		ADD A,@R0
6		JNC SKIP
7		INC B
8	SKIP	INC R0
9		MOV @R0,A
10		INC R0
11		MOV @R0,B
12	HERE	SJMP HERE

Method 2: In this alternate method the addition is done using “ADD A,Data” instruction, but carry is considered using “ADDC A,Data” instruction instead of conditional branching instruction “JNC”.

Assembly Language Program 12.1 (Method 2):

SL.	Label	Instructions of 8051
1		MOV R0,#50H
2		MOV A,@R0
3		INC R0
4		ADD A,@R0
5		INC R0
6		MOV @R0,A
7		MOV A,#00H
8		ADDC A,A
9		INC R0
10		MOV @R0,A
11	HERE	SJMP HERE

Result of Program 12.1:

SET1 ►

Input

Mem. Address	Content	Remarks
50	0A	No1
51	DD	No2

Output

Mem. Address	Content	Remarks
52	E7	Lower Byte of Result
53	00	Higher Byte of Result

SET2 ►

Input

Mem. Address	Content	Remarks
50	FF	No1
51	FE	No2

Output

Mem. Address	Content	Remarks
52	FD	Lower Byte of Result
53	01	Higher Byte of Result

12.2: Write a program to add ten 8-bit binary numbers which are stored at the memory locations starting from 50H to 59H and also store the result of addition at memory locations 5AH and 5BH whereas 5AH and 5BH will hold the lower and higher byte of the result respectively.

It is to determine first, what will be the maximum value of the result of ten 8-bit numbers addition so that it can be decided that how many bytes are required to store the result. Naturally the result of addition will be maximum, if all the ten 8-bit numbers having their maximum value i.e FF.
Hence FF + FF + FF + FF + FF + FF + FF + FF + FF + FF = 9F6 i.e 09F6

Therefore it is clear that atleast 2 bytes are required to store the result of ten 8-bit numbers addition. We have to use two consecutive memory locations – one 5AH and another 5BH for storing the lower byte and higher byte of the result respectively.

The concept of this program is that addition should be performed repeatedly for n times for addition of n no. of 8-bit numbers and a register is to be taken for counting the no of carries occurred for these multiple no. of addition. In this case register B has been taken to hold how many times the carry occurred during 9 times addition of ten 8-bit numbers. Each time if a carry occurs the content of register B is to be incremented by one. The ten 8-bit numbers are stored in consecutive memory locations starting from 50H to 59H and the lower byte and the higher byte of the result will be stored at address 5AH and 5BH respectively, which is shown pictorially in the following Fig-12.1.

Addresses	Contents
50	No 1
51	No 2
52	No 3
53	No 4
54	No 5
55	No 6
56	No7
57	No 8
58	No 9
59	No 10
5A	Lower Byte of Result
5B	Higher Byte of Result

Fig-12.1: Ten 8-bit numbers and the result of addition are stored consecutively from 50H

Assembly Language Program 12.2:

SL.	Label	Instructions of 8051
1		MOV R2,#09H
2		MOV B,#00H
3		MOV R0,#50H
4		MOV A,@R0
5	REPEAT	INC R0
6		ADD A,@R0
7		JNC SKIP
8		INC B
9	SKIP	DJNZ R2,REPEAT
10		INC R0
11		MOV @R0,A
12		INC R0
13		MOV @R0,B
14	HERE	SJMP HERE

Result of Program 12.2:

SET1 ►

Input

Mem. Address	Content	Remarks
50	05	No1
51	0D	No2
52	DD	No3
53	AA	No4
54	12	No5
55	32	No6
56	01	No7
57	0A	No8
58	1F	No9
8059	0A	No10

Output

Mem. Address	Content	Remarks
5A	11	Lower Byte of Result
5B	02	Higher Byte of Result

SET2 ►

Input

Mem. Address	Content	Remarks
50	05	No1
51	06	No2
52	07	No3
53	08	No4
54	09	No5
55	0A	No6
56	0B	No7
57	0C	No8
58	0D	No9
59	0E	No10

Output

Mem. Address	Content	Remarks
5A	5F	Lower Byte of Result
5B	00	Higher Byte of Result

12.3: Write a program to add two 64-bit binary numbers which are stored at the memory locations starting from 50H onward and the memory locations starting from 60H onward. Store the result of the addition starting from memory location 70H onward.

As the two numbers are 64-bit long i.e. 8 byte long, each number occupies eight consecutive memory locations. Hence the first number starts from 50H to 57H and the second number starts from 60H to 67H. Moreover, it takes at least 9 consecutive bytes to store the result of addition starting from 70H to 78H as shown in Fig-12.2.

1St Number

Address	Content
50	Byte1
51	Byte2
52	Byte3
53	Byte4
54	Byte5
55	Byte6
56	Byte7
57	Byte8

2nd Number

Address	Content
60	Byte1
61	Byte2
62	Byte3
63	Byte4
64	Byte5
65	Byte6
66	Byte7
67	Byte8

Result of Addition

Address	Content
70	Byte1
71	Byte2
72	Byte3
73	Byte4
74	Byte5
75	Byte6
76	Byte7
77	Byte8
78	Byte9

Fig-12.2: Memory mapping of two 64-bit numbers and their result of addition

During the addition of two 8-byte numbers, addition of each bytes from two numbers are performed starting from the lowest byte to highest byte successively i.e. addition is done first in between Byte1 of the two numbers, then between Byte2 and so on. If carry occurs after the addition of two Byte1 of two numbers, that carry will be propagated into the addition of two Byte2 of the two numbers. Similarly if there is carry during the addition of two Byte2, that carry will be propagated into the third bytes of the two numbers. This will go on until highest byte i.e. Byte8 addition is done. In this case, one thing is important to consider that there is no chance of occurring any carry from the previous stage during the addition of lowest bytes i.e. Byte1. Hence before using ADDC instruction for adding Byte1 of the two numbers, the carry flag must be zero. In this program addition will be performed for 8 times. Therefore the register R2 should be taken as counter.

But the problem here is the shortage of memory pointing registers, because the registers R0 and R1 of Bank0 are already being consumed to point the starting address of the two memory blocks where minuend and subtrahend are stored. To point the memory block of the result of the addition it is required another memory pointing register, which is served by the register R0 of Bank2. Therefore in this program it is required to switch the banks of 8051 using the two flags RS0 and RS1 of PSW (Program Status Word). Switching of the banks from Bank0 to Bank2 is done with the help of the instruction “SETB PSW.4” and the switching from Bank2 to Bank0 is performed by using “CLR PSW.4”.

Assembly Language Program 12.3:

SL.	Label	Instructions of 8051
1		MOV R2,#08H
2		SETB PSW.4
3		MOV R0,#70H
4		CLR PSW.4
5		MOV R0,#50H
6		MOV R1,#60H
7		CLR C
8	REPEAT	MOV A,@R0
9		ADDC A,@R1
10		INC R0
11		INC R1
12		SETB PSW.4
13		MOV @R0,A
14		INC R0
15		CLR PSW.4
16		DJNZ R2,REPEAT
17		MOV A,#00H
18		ADDC A,#00H
19		SETB PSW.4
20		MOV @R0,A
21	HERE	SJMP HERE

Result of Program 12.3:

SET1 ►

Input

No1

Addr	Content	Remarks
50	88	Byte1
51	99	Byte2
52	AA	Byte3
53	BB	Byte4
54	CC	Byte5
55	DD	Byte6
56	EE	Byte7
57	FF	Byte8

No2

Addr	Content	Remarks
60	01	Byte1
61	02	Byte2
62	03	Byte3
63	04	Byte4
64	05	Byte5
65	06	Byte6
66	07	Byte7
67	08	Byte8

Output

Result

Addr	Content	Remarks
70	89	Byte1
71	9B	Byte2
72	AD	Byte3
73	BF	Byte4
74	D1	Byte5
75	E3	Byte6
76	F5	Byte7
77	07	Byte8
78	01	Byte9

SET2 ►

Input

No1

Addr	Content	Remarks
50	10	Byte1
51	20	Byte2
52	30	Byte3
53	40	Byte4
54	50	Byte5
55	60	Byte6
56	70	Byte7
57	80	Byte8

No2

Addr	Content	Remarks
60	08	Byte1
61	07	Byte2
62	06	Byte3
63	05	Byte4
64	04	Byte5
65	03	Byte6
66	02	Byte7
67	01	Byte8

Output

Result

Addr	Content	Remarks
70	18	Byte1
71	27	Byte2
72	36	Byte3
73	45	Byte4
74	54	Byte5
75	63	Byte6
76	72	Byte7
77	81	Byte8
78	00	Byte9

12.4: Write a program to subtract two 64-bit binary numbers which are stored at the memory locations starting from 50H onward and the memory locations starting from 60H onward. Store the result of the subtraction starting from memory location 70H onward.

As the two numbers are 64-bit long i.e. 8 byte long, each number occupies eight consecutive memory locations. Hence the first number starts from 50H to 57H and the second number starts from 60H to 67H. In this case the 2nd number (subtrahend) will be subtracted from the 1st number (minuend). Moreover, it takes 8 consecutive bytes to store the magnitude of the subtraction starting from 70H to 77H and an extra byte is required to store the polarity of the result into the memory location 78H as shown in Fig-12.3. If the result is negative, 01H will be stored at 78H to indicate the negative result and 00H will be stored at the same memory location, if the result is positive.

Minuend

Address	Content
50	Byte1
51	Byte2
52	Byte3
53	Byte4
54	Byte5
55	Byte6
56	Byte7
57	Byte8

Subtrahend

Address	Content
60	Byte1
61	Byte2
62	Byte3
63	Byte4
64	Byte5
65	Byte6
66	Byte7
67	Byte8

Result of Subtraction

Address	Content
70	Byte1
71	Byte2
72	Byte3
73	Byte4
74	Byte5
75	Byte6
76	Byte7
77	Byte8
78	00H/01H (Polarity)

Fig-12.3: Memory mapping of two 64-bit numbers and their result of subtraction

During the subtraction of two 8-byte numbers, subtraction of each byte from two numbers are performed starting from the lowest byte to highest byte successively i.e. subtraction is done first in between Byte1 of the two numbers, then between Byte2 and so on. If borrow occurs after the subtraction of two Byte1 of two numbers, that borrow will be propagated into the subtraction of two Byte2 of the two numbers. Similarly if there is borrow during the subtraction of two Byte2, that borrow will be propagated into the third bytes of the two numbers. This will go on until highest byte i.e. Byte8. In this case, one thing is important to consider that there is no chance of occurring any borrow from the previous stage during the subtraction of lowest bytes i.e. Byte1. Hence before using SUBB instruction for subtracting Byte1 of the two numbers, the carry flag must be zero. In this program subtraction will be performed for 8 times. Therefore the register R2 should be taken as counter.

Here we have followed the signed magnitude convention to improve the readability of the user. In this convention to represent the sign of the number an extra bit is taken and the magnitude is represented always in normal form. For example -2 will be 1 00000010 and +2 will be 0 00000010 in signed-magnitude form. To follow this convention the following method is carried out to represent the result of subtraction in signed-magnitude form. If the result of subtraction is negative using the instruction SUBB, the magnitude will be in 2's complement form and the carry flag will be set. To present the negative result in normal form the magnitude will be again 2's complemented and 01H will be stored as 9^{th} byte to show the negative result. On the contrary if the result is positive, the instruction SUBB will give the magnitude in normal form. Hence there is no requirement to perform 2's complement on the magnitude and the magnitude part of the result will be stored directly in the normal form and 00H will be saved as 9^{th} byte to indicate that the result is positive.

But the problem here is the shortage of memory pointing registers, because the registers R0 and R1 of Bank0 have been already consumed to point the starting address of the two memory blocks where minuend and subtrahend are stored. To point the memory block of the result of the subtraction it is essential to use another memory pointing register, which is served by the register R0 of Bank2. Therefore in this program it is necessary to switch the banks of 8051 using the two flags RS0 and RS1 of PSW (Program Status Word). Switching of the banks from Bank0 to Bank2 and vice versa is done with the help of two instructions "SETB PSW.4" and "CLR PSW.4".

Assembly Language Program 12.4:

SL.	Label	Instructions of 8051
1		MOV R2,#08H
2		SETB PSW.4
3		MOV R0,#70H
4		CLR PSW.4
5		MOV R0,#50H

SL.	Label	Instructions of 8051
6		MOV R1,#60H
7		CLR C
8	REPEAT	MOV A,@R0
9		SUBB A,@R1
10		INC R0
11		INC R1
12		SETB PSW.4
13		MOV @R0,A
14		INC R0
15		CLR PSW.4
16		DJNZ R2,REPEAT
17		MOV A,#00H
18		SETB PSW.4
19		MOV @R0,A
20		JNC HERE
21		MOV R2,#08H
22		MOV R0,#70H
23	LOOP	MOV A,@R0
24		CPL A
25		ADDC A,#00H
26		MOV @R0,A
27		INC R0
28		DJNZ R2,LOOP
29		MOV A,#01H
30		MOV @R0,A
31	HERE	SJMP HERE

Result of Program 12.4:

SET1 ►

Input

Minuend

Addr	Content	Remarks
50	88	Byte1
51	99	Byte2
52	AA	Byte3
53	BB	Byte4
54	CC	Byte5
55	DD	Byte6
56	EE	Byte7
57	FF	Byte8

Subtrahend

Addr	Content	Remarks
60	01	Byte1
61	02	Byte2
62	03	Byte3
63	04	Byte4
64	05	Byte5
65	06	Byte6
66	07	Byte7
67	08	Byte8

Output

Result

Addr	Content	Remarks
70	87	Byte1
71	97	Byte2
72	A7	Byte3
73	B7	Byte4
74	C7	Byte5
75	D7	Byte6
76	E7	Byte7
77	F7	Byte8
78	00	Positive

SET2 ►

Input

Minuend

Addr	Content	Remarks
50	08	Byte1
51	07	Byte2
52	06	Byte3
53	05	Byte4
54	04	Byte5
55	03	Byte6
56	02	Byte7
57	01	Byte8

Subtrahend

Addr	Content	Remarks
60	10	Byte1
61	20	Byte2
62	30	Byte3
63	40	Byte4
64	50	Byte5
65	60	Byte6
66	70	Byte7
67	80	Byte8

Output

Result

Addr	Content	Remarks
70	08	Byte1
71	19	Byte2
72	2A	Byte3
73	3B	Byte4
74	4C	Byte5
75	5D	Byte6
76	6E	Byte7
77	7F	Byte8
78	01	Negative

12.5: Write a program to perform algebraic sum of two signed 8-bit binary numbers which are stored at the memory locations 50H and 51H and also store the result of addition into memory location 52H.

In this program two 8-bit signed numbers will be added and the result of addition which is basically 8-bit long will be stored into the memory location 52H. As both the numbers are signed, there may be four possible cases as follows.

Case-1: Both the numbers are positive and result also will be positive.
Case-2: Both the numbers are negative and the result also will be negative.
Case-3: 1st number is positive and 2nd number is negative, which gives positive result if 1st number is larger than 2nd number and negative result occurs if 1st number is smaller than 2nd number.
Case-4: 1st number is negative and 2nd number is positive, which gives negative result if 1st number is larger than 2nd number and positive result occurs if 1st number is smaller than 2nd number.

In 8051 microcontroller the MSB is used to represent the sign bit and the remaining bits are used as magnitude of the signed number. If MSB is 0, the number will be considered as positive and if MSB is 1, the number will be treated as negative. Therefore for 8-bit signed number if D7 = 0, the number is positive and if D7 = 1, the number is negative. To fulfill this condition, a negative number is represented in 2's complement form and a positive one is represented in normal form in 8051. To represent a negative number the number is complemented at first omitting its negative sign and then 1 is added to it to get the 2's complement form. For example, -7 is first taken as 07 and converted to its binary equivalent which is 0000 0111. Now it is complemented to 1111 1000 and added 1 to it to get 1111 1001 which is the 2's complement of 07 and it is used as -7 in 8051. Therefore we can say -7 is represented as F9H in 8051 microcontroller. Therefore in signed convention the range of 8-bit positive numbers is 0 to $(2^7 - 1)$ i.e. 0 to 127 and the range of 8-bit negative numbers is -1 to -2^7 i.e. -1 to -128. Now all the positive and negative numbers along with its binary and hexadecimal forms are given in the following table.

Decimal	Binary	Hexadecimal
127	0111 1111	7F
126	0111 1110	7E
:	:	:
1	0000 0001	01
0	0000 0000	00
-1	1111 1111 (2's complement of 1)	FF
-2	1111 1110 (2's complement of 2)	FE
:	:	:
-128	1000 0000 (2's complement of 128)	80

Hence it is clear that if the result of signed number addition is not in the range as per the above table, it causes erroneous result and this is called overflow problem. For example, if -128 and -2 are added it should result -130. But -130 is out of range (-1 to -128) for negative numbers in 8051 microcontroller. That's why this addition will result overflow error. To encounter this overflow problem 8051 microcontroller has an overflow flag (OV) in PSW register of 8051. If overflow occurs, OV flag becomes 1, otherwise it remains 0. The PSW flag of 8051 microcontroller is shown below.

CY	AC	F0	RS1	RS0	OV	-	P
PSW.7	*PSW.6*	*PSW.5*	*PSW.4*	*PSW.3*	*PSW.2*	*PSW.1*	*PSW.0*

Program Status Word (PSW) of 8051 microcontroller

In the above PSW register each flag is accessed by its bit position like PSW.0, PSW.2 etc. Hence overflow flag (OV) is accessed by PSW.2. Whether OV is set or not, is checked by the instruction "JB PSW.2" where JB stands for "jump for bit" i.e. if the bit is set then it will jump, otherwise it will not jump. Now if the following two conditions are satisfied, then OV flag will be set.

Condition 1 → If there is no carry out of D7 (CY = 0) and a carry from D6 to D7, then OV = 1.

Condition 2 → If there is a carry out of D7 (CY = 0) and no carry from D6 to D7, then OV = 1.

Therefore the status of the overflow flag can be represented as EX-OR operation between CY and carry from D6 to D7 bit.

In this program the negative numbers are given in 2's complement form in the memory locations 50H and 51H for algebraic addition. After performing addition the status of the overflow flag is checked. If it is set, then the program is terminated by storing 0EH at the memory location 52H where '0E' represents the overflow error. If overflow problem does not arise, the MSB (D7 bit) of the result is checked. If it is 1, then the result is negative and also in 2's complement form. To increase the readability of the user, the 2's complemented result is converted to its normal form and the MSB (D7) is made high to indicate that the result is negative. For example, if the result is FE, then D7 bit of the result is 1. This implies that the result is negative. Now FEH is 2's complemented to 02H and ORed with 80H, which results 82H. Therefore in this case the result (-2) is converted to 82H and stored at the memory location 52H. If the MSB of the result is zero, then it is positive and the result already is in normal form. This normal form of the result is stored at the RAM location 52H.

Assembly Language Program 12.5:

SL.	Label	Instructions of 8051
1		MOV R0,#50H
2		MOV A,@R0
3		INC R0
4		ADD A,@R0
5		JB PSW.2,ERROR
6		MOV B,A
7		ANL A,#80H
8		CJNE A,#00H,NEG
9		MOV A,B
10		SJMP RESULT
11	NEG	MOV A,B
12		CPL A
13		ADD A,#01H
14		ORL A,#80H
15	RESULT	INC R0
16		MOV @R0,A
17		SJMP HERE
18	ERROR	INC R0
19		MOV A,#0EH
20		MOV @R0,A
21	HERE	SJMP HERE

Result of Program 12.5:

SET1 ►

Input

Mem. Address	Content	Remarks
50H	07H	Positive No1 (+7)
51H	12H	Positive No2 (+18)

Output

Mem. Address	Content	Remarks
52H	19H	OV=0, Result is +25

SET2 ►

Input

Mem. Address	Content	Remarks
50H	FEH	Negative No1 (-2)
51H	FBH	Negative No2 (-5)

Output

Mem. Address	Content	Remarks
52H	F9H	OV=0, Result is -7

SET3 ►

Input

Mem. Address	Content	Remarks
50H	80H	Negative No1 (-128)
51H	FEH	Negative No2 (-2)

Output

Mem. Address	Content	Remarks
52H	7EH	OV=1, Result is +126 (Error)

SET4 ►

Input

Mem. Address	Content	Remarks
50H	60H	Positive No1 (+96)
51H	46H	Positive No2 (+70)

Output

Mem. Address	Content	Remarks
52H	A6H	OV=1, Result is -90 (Error)

SET5 ►

Input

Mem. Address	Content	Remarks
50H	7FH	Positive No1 (+127)
51H	FEH	Negative No2 (-2)

Output

Mem. Address	Content	Remarks
52H	7DH	OV=0, Result is +125

12.6: Write a program to multiply two 8-bit binary numbers which are stored at the memory locations 50H and 51H and also store the result of multiplication into memory locations 52H (lower byte) and 53H (higher byte) respectively.

Unlike 8085 microprocessor, there is an instruction to perform multiplication between two unsigned 8-bit numbers for 8051 microcontroller. The instruction for multiplication is given below.

"MUL AB" where one number is stored in the register A and other number is stored in register B. After the multiplication the 16-bit result is stored in register B and A respectively. Register B holds the higher byte and the register A holds the lower byte of the result. Here the number stored at the memory location 50H will be copied to register A and the number stored at the memory location 51H will be copied to register B. Using the instruction "MUL AB" the multiplication is done and the content of A and B are stored at the memory locations 52H and 53H respectively.

Assembly Language Program 12.6:

SL.	Label	Instructions of 8051
1		MOV R0,#50H
2		MOV A,@R0
3		INC R0
4		MOV B,@R0
5		MUL AB
6		INC R0
7		MOV @R0,A
8		INC R0
9		MOV @R0,B
10	HERE	SJMP HERE

Result of Program 12.6:

SET1 ►

Input

Mem. Address	Content	Remarks
50H	07H	No1 (Multiplicand)
51H	12H	No2 (Multiplier)

Output

Mem. Address	Content	Remarks
52H	7EH	Lower byte of result
53H	00H	Higher byte of result

12.7: Write a program to divide a 8-bit binary number stored at the memory locations 50H by another 8-bit number stored at the memory location 51H and store the quotient at the location 52H and the remainder at the location 53H.

In 8051 microcontroller a instruction is present for 8-bit division. The instruction is given below.

*"DIV AB"*where the content of A is the dividend and the content of B is the divisor. After the execution of this instruction the content of A becomes the quotient and the content of B becomes the remainder. If the content of B is zero i.e. the divisor is equal to zero, then divide-by-zero error occurs and OV flag is set to indicate this error. In this program after the DIV instruction the OV is checked and if it is 1, FFH is stored at both the memory locations 52H and 53H.

Assembly Language Program 12.7:

SL.	Label	Instructions of 8051
1		MOV R0,#50H
2		MOV A,@R0
3		INC R0
4		MOV B,@R0
5		DIV AB
6		JB PSW.2,ERROR
7		INC R0
8		MOV @R0,A
9		INC R0
10		MOV @R0,B
11		SJMP HERE
12	ERROR	MOV A,#0FFH
13		INC R0
14		MOV @R0,A
15		INC R0
16		MOV @R0,A
17	HERE	SJMP HERE

Result of Program 12.7:

SET1 ►

Input

Mem. Address	Content	Remarks
50H	5FH	No1 (Dividend)
51H	0AH	No2 (Divisor)

Output

Mem. Address	Content	Remarks
52H	09H	Quotient
53H	05H	Remainder

SET2 ►

Input

Mem. Address	Content	Remarks
50H	50H	No1 (Dividend)
51H	00H	No2 (Divisor)

Output

Mem. Address	Content	Remarks
52H	FFH	Divide-By-Zero error
53H	FFH	Divide-By-Zero error

Exercise

1) Write a program to multiply two 8-bit binary numbers which are stored at the memory locations 50H and 51H respectively using successive addition method and also store the result of multiplication into memory locations 52H (lower byte) and 53H (higher byte) respectively.

2) Write a program to divide a 8-bit binary number stored at the memory locations 50H by another 8-bit number stored at the memory location 51H using successive subtraction method and also store the quotient at the location 52H and the remainder at the location 53H.

13. Programs on Data Transfer and Data Separation

13.1: Write a program to transfer a block of ten data stored starting from the onchip RAM location 50H onward to the onchip RAM location 60H onward in forward direction.

This program basically performs the copy operation of a set of ten 8-bit data from one memory location to another memory location consecutively in forward direction. The memory locations where the ten numbers are stored, is called source block and the memory locations where the ten numbers have to be transferred is called destination block. In this program the source block starts from the address 50H to 59H and the destination block starts from the address 60H to 69H inside the scratchpad area of the onchip RAM. As the data are copied in forward direction the number at 50H will be copied to 60H, the number of 51H will be copied to 61H, the number of 52H will be copied to 62H and so on. The pictorial representation of the above mentioned procedure has been already given for 8085 microprocessor.

Assembly Language Program 13.1:

SL.	Label	Instructions of 8051
1		MOV R0,#50H
2		MOV R1,#60H
3		MOV R2,#0AH
4	REPEAT	MOV A,@R0
5		MOV @R1,A
6		INC R0
7		INC R1
8		DJNZ R2,REPEAT
9	HERE	SJMP HERE

Result of Program 13.1:

SET1 ►

Input **Output**

Source Block

RAM Address	Content	Remarks
50	10	No1
51	20	No2
52	30	No3
53	40	No4
54	50	No5
55	60	No6
56	70	No7
57	80	No8
58	90	No9
59	A0	No10

Destination Block

RAM Address	Content	Remarks
60	10	No1
61	20	No2
62	30	No3
63	40	No4
64	50	No5
65	60	No6
66	70	No7
67	80	No8
68	90	No9
69	A0	No10

13.2: Write a program to transfer a block of ten data stored starting from the onchip RAM location 50H onward to the onchip RAM location 54H onward in forward direction.

In this program the source block extends from the RAM location 50H to 59H and the destination block extends from 54H to 5DH. Therefore six memory locations starting from 54H to 59H of the source block are common to the destination block i.e. there is a overlapping region between the source block and the destination block. Now if we start to copy the numbers from the starting address of the source block to the starting address of the destination block, the numbers of the source block stored from 54H to 59H will be completely lost before they are transferred to the destination block. Here our aim is to copy the contents of the entire source block to the destination block as it is, though the source block will not remain intact. That means the six data from 54H to 59H of the source block will not remain intact, but the entire source block will be copied to the destination block from 54H to 5DH without any data loss. To accomplish this, the data of the source block should be copied starting from the last address of the source block to the last address of the destination block. Therefore data of 59H of source block will be copied to 5DH of destination block, data of 58H of source block will be copied to 5CH of destination block, data of 57H of source block will be copied to 5BH of destination block and so on.

Assembly Language Program 13.2:

SL.	Label	Instructions of 8051
1		MOV R0,#59H
2		MOV R1,#5DH
3		MOV R2,#0AH
4	REPEAT	MOV A,@R0
5		MOV @R1,A
6		DEC R0
7		DEC R1
8		DJNZ R2,REPEAT
9	HERE	SJMP HERE

Result of Program 13.2:

SET1 ►

Input

Source Block

RAM Address	Content	Remarks
50	11	No1
51	22	No2
52	33	No3
53	44	No4
54	55	No5
55	66	No6
56	77	No7
57	88	No8
58	99	No9
59	AA	No10

Output

Destination Block

RAM Address	Content	Remarks
54	11	No1
55	22	No2
56	33	No3
57	44	No4
58	55	No5
59	66	No6
5A	77	No7
5B	88	No8
5C	99	No9
5D	AA	No10

13.3: Write a program to transfer a block of ten data stored starting from the onchip RAM location 50H onward to the onchip RAM location 60H onward in reverse direction.

In this program ten data of source block starting from RAM location 50H to 59H will be copied to the destination block starting from RAM location 60H to 69H in reverse direction. Therefore the data of source block at RAM location 50H will be copied to RAM location 69H of destination block, the data of source block at RAM location 51H will be copied to RAM location 68H of destination block, the data of source block at RAM location 52H will be copied to RAM location 67H of destination block and so on. To implement this the memory pointer of source block will be incremented by one whereas the memory pointer of destination block will be decremented by one after every data transfer.

Assembly Language Program 13.3:

SL.	Label	Instructions of 8051
1		MOV R0,#50H
2		MOV R1,#69H
3		MOV R2,#0AH
4	REPEAT	MOV A,@R0
5		MOV @R1,A
6		INC R0
7		DEC R1
8		DJNZ R2,REPEAT
9	HERE	SJMP HERE

Result of Program 13.3:

SET1 ►

Input **Output**

Source Block

RAM Address	Content	Remarks
50	12	No1
51	23	No2
52	34	No3
53	45	No4
54	56	No5
55	67	No6
56	78	No7
57	89	No8
58	9A	No9
59	AB	No10

Destination Block

RAM Address	Content	Remarks
60	AB	No10
61	9A	No9
62	89	No8
63	78	No7
64	67	No6
65	56	No5
66	45	No4
67	34	No3
68	23	No2
69	12	No1

13.4: Write a program to separate positive numbers and negative numbers into two different memory blocks from a set of ten 8-bit signed numbers which are stored consecutively starting from the memory location 50H onward. The positive block starts from 60H onward and the negative block starts from 70H onward in the scratchpad area of onchip RAM, where positive count and negative count will be stored at the starting address of each block.

We know, if the MSB of a binary number is high, the number will be treated as negative number and if the MSB is low, the number is considered as positive number. So, the MSB of each of the ten 8-bit binary numbers which are stored at the source block starting from 50H to 59H, is checked for high or low and is separated into two blocks of memory depending upon the status of MSB. The memory block which is storing the positive numbers, is called the positive block and the memory block which is holding the negative numbers, is called the negative block. So here the positive block starts from 60H onward, where the first memory location 60H holds the number of count of positive numbers i.e. how many positive numbers and all the positive numbers begins to be stored from 61H onward. Similarly the negative block starts from 70H onward, where the first location 70H stores the number of count of negative numbers and all the negative numbers will be stored starting from the memory location 71H onward.

In case of 8051 microcontroller there is no sign flag present in the PSW. So we have no option to check whether a number is positive or negative directly in 8051. To accomplish this the number will be copied to accumulator and the content of accumulator rotated left through carry (using RLC

instruction) for one time to get the MSB in the carry flag. Now the status of the carry flag is checked (using JNC/ JC instruction) to decide the polarity of the number. That means if the carry flag CY = 1, the number will be negative and if CY = 0, the number will be positive. We need three memory pointers in this case, 1st memory pointer for source block, 2nd memory pointer for positive block and 3rd memory pointer for negative block. In case of 8051 only two memory pointers R0 and R1 are available for register indirect addressing from Bank0. Therefore they will be used as 2nd and 3rd memory pointers for positive and negative block respectively. Now the register R0 from Bank2 will be utilized as 1st memory pointer for source block. Hence it is clear that the bank switching between Bank0 and Bank2 will be required here to implement this program.

Assembly Language Program 13.4:

SL.	Label	Instructions of 8051
1		SETB PSW.4
2		MOV R0,#50H
3		CLR PSW.4
4		MOV R2,#0AH
5		MOV R0,#61H
6		MOV R1,#71H
7		MOV R3,#00H
8		MOV R4,#00H
9	REPEAT	CLR C
10		SETB PSW.4
11		MOV A,@R0
12		INC R0
13		RLC A
14		CLR PSW.4
15		JNC POSITIVE
16		INC R4
17		RRC A
18		MOV @R1,A
19		INC R1
20		SJMP SKIP
21	POSITIVE	INC R3

SL.	Label	Instructions of 8051
22		RRC A
23		MOV @R0,A
24		INC R0
25	SKIP	DJNZ R2,REPEAT
26		MOV 60H,R3
27		MOV 70H,R4
28	HERE	SJMP HERE

Result of Program 13.4:

SET1 ►

Input

Source Block

Address	Content	Remarks
50	05	No1
51	0D	No2
52	DD	No3
53	AA	No4
54	12	No5
55	32	No6
56	71	No7
57	0A	No8
58	8F	No9
59	0A	No10

Output

Positive Block

Address	Content	Remarks
60	07	Positive Count
61	05	+No1
62	0D	+No2
63	12	+No5
64	32	+No6
65	71	+No7
66	0A	+No8
67	0A	+No10

Negative Block

Address	Content	Remarks
70	03	Negative Count
71	DD	-No3
72	AA	-No4
73	8F	-No9

13.5: Write a program to separate odd numbers and even numbers into two different memory blocks from a set of ten 8-bit numbers which are stored consecutively starting from the memory location 50H onward. The odd block starts from 60H onward and the even block starts from 70H onward in the scratchpad area of onchip RAM, where odd count and even count will be stored at the starting address of each block.

We know, if the LSB of a binary number is high, the number will be treated as odd number and if the LSB is low, the number is considered as even number. So, the LSB of each of the ten 8-bit binary numbers which are stored at the source block starting from 50H to 59H, is checked for high or low and is separated into two blocks of memory depending upon the status of LSB. The memory block which is storing the odd numbers, is called the odd block and the memory block which is holding the even numbers, is called the even block. So here the odd block starts from 60H onward, where the first memory location 60H holds the number of count of odd numbers and all the odd numbers begin to be stored from 61H onward. Similarly the even block starts from 70H onward, where the first location 70H stores the number of count of even numbers and all the even numbers will be stored starting from the memory location 71H onward.

In case of 8051 microcontroller the number will be copied to accumulator and the content of accumulator rotated right through carry (using RRC instruction) for one time to get the LSB in the carry flag. Now the status of the carry flag is checked (using JNC/ JC instruction) to decide whether the number is odd or even. That means if the carry flag CY = 1, the number will be odd and if CY = 0, the number will be even. We need three memory pointers in this case, 1st memory pointer for source block, 2nd memory pointer for odd block and 3rd memory pointer for even block. In case of 8051 only two memory pointers R0 and R1 are available for register indirect addressing from Bank0. Therefore they will be used as 2nd and 3rd memory pointers for odd and even block respectively. Now the register R0 from Bank2 will be utilized as 1st memory pointer for source block. Hence it is clear that the bank switching between Bank0 and Bank2 will be required here to implement this program.

Assembly Language Program 13.5:

SL.	Label	Instructions of 8051
1		SETB PSW.4
2		MOV R0,#50H
3		CLR PSW.4
4		MOV R2,#0AH
5		MOV R0,#61H
6		MOV R1,#71H
7		MOV R3,#00H

SL.	Label	Instructions of 8051
8		MOV R4,#00H
9	REPEAT	CLR C
10		SETB PSW.4
11		MOV A,@R0
12		INC R0
13		RRC A
14		CLR PSW.4
15		JC ODD
16		INC R4
17		RLC A
18		MOV @R1,A
19		INC R1
20		SJMP SKIP
21	ODD	INC R3
22		RLC A
23		MOV @R0,A
24		INC R0
25	SKIP	DJNZ R2,REPEAT
26		MOV 60H,R3
27		MOV 70H,R4
28	HERE	SJMP HERE

Result of Program 13.5:

SET1 ►

Input

Source Block

Address	Content	Remarks
50	05	No1
51	0D	No2
52	DD	No3
53	AA	No4
54	12	No5
55	32	No6
56	71	No7
57	0A	No8
58	8E	No9
59	0A	No10

Output

Odd Block

Address	Content	Remarks
60	04	Odd Count
61	05	No1
62	0D	No2
63	DD	No3
64	71	No7

Even Block

Address	Content	Remarks
70	06	Even Count
71	AA	No4
72	12	No5
73	32	No6
74	0A	No8
75	8E	No9
76	0A	No10

Exercise

3) Suppose a set of ten 8-bit numbers are stored consecutively from memory location 40H onward in the onchip RAM of 8051. Write a program to insert an element stored at memory location 4FH into the memory location 43H.

4) Suppose a set of ten 8-bit numbers are stored consecutively from memory location 50H onward in the onchip RAM of 8051. Write a program to delete the element which is stored at memory location 55H.

5) Write a program to store AAH and BBH alternately for 100 times starting from memory location 30H onward in the scratchpad area of 8051. Also store the last address where BBH is stored into the register R7 of Bank3 of 8051.

6) Write a program to store first ten natural numbers consecutively from memory location 50H in the onchip RAM of 8051.

7) Write a program to store first ten numbers of Fibonacci series consecutively from memory location 60H in the onchip RAM of 8051.

[Hint: 1st no. = 0 and 2nd no. = 1 for Fibonacci series
After that nth no. = (n-1)th no. + (n-2)th no.]

14. Programs on Searching and Sorting

14.1: Write a program to find the largest and the smallest number from a list of ten 8-bit numbers which are stored from the memory location 50H onward and store the largest and the smallest numbers at memory location 5AH and 5BH respectively.

In this program the largest number will be stored in register R3 whereas the smallest number will be stored in register R4 primarily, them the largest value stored inside R3 will be transferred to memory location 5AH and the smallest value inside R4 will be transferred to memory location 5BH. To accomplish this, the 1st number at memory location 50H will be copied into R3 and R4 both. Then the next numbers stored consecutively from 51H will be compared with R3 as well as R4 one by one. If the number is larger than the content of R3, the number will be copied to R3 to overwrite the previous value. On the contrary, if the number is smaller than the content of R4, it will be replaced by the number. Thus we get the largest value inside R3 and the smallest value inside R4 finally after scanning all the ten numbers which are stored starting from the memory location 50H to 59H.

Assembly Language Program 14.1:

SL.	Label	Instructions of 8051
1		MOV R0,#50H
2		MOV A,@R0
3		MOV R3,A
4		MOV R4,A
5		MOV R2,#09H
6	REPEAT	INC R0
7		MOV B,@R0
8		MOV A,R3
9		CJNE A,B,NEXT1
10	NEXT1	JNC SKIP1
11		MOV R3,B
12	SKIP1	MOV A,R4
13		CJNE A,B,NEXT2
14	NEXT2	JC SKIP2
15		MOV R4,B

SL.	Label	Instructions of 8051
16	SKIP2	DJNZ R2,REPEAT
17		INC R0
18		MOV A,R3
19		MOV @R0,A
20		INC R0
21		MOV A,R4
22		MOV @R0,A
23	HERE	SJMP HERE

Result of Program 14.1:

SET1 ►

Input

Mem. Address	Content	Remarks
50	05	No1
51	0D	No2
52	DD	No3
53	AA	No4
54	12	No5
55	32	No6
56	71	No7
57	0A	No8
58	8F	No9
59	0A	No10

Output

5A → DD (Largest No.)
5B → 05 (Smallest No.)

14.2: Write a program to find the number DDH from a list of ten 8-bit numbers which are stored from the memory location 50H onward and store the number of times DDH found into the memory location 5AH.

In this program each and every number from the list of ten numbers stored consecutively from memory location 50H to 59H is compared with the key number DDH. If there is a matching, the counter register R3 will be incremented by one, otherwise the content of R3 remains unchanged. Finally the register will hold the number times DDH found in the list of ten numbers and will be stored at RAM location 5AH.

Assembly Language Program 14.2:

SL.	Label	Instructions of 8051
1		MOV R0,#50H
2		MOV R2,#0AH
3		MOV R3,#00H
4	REPEAT	MOV A,@R0
5		CJNE A,#0DDH,SKIP
6		INC R3
7	SKIP	INC R0
8		DJNZ R2,REPEAT
9		MOV A,R3
10		MOV @R0,A
11	HERE	SJMP HERE

Result of Program 14.2:

SET1 ►

Input

Mem. Address	Content	Remarks
50	05	No1
51	0D	No2
52	DD	No3
53	AA	No4
54	12	No5
55	32	No6
56	DD	No7
57	0A	No8
58	8F	No9
59	0A	No10

Output

5A → 02 (No. of times DDH found)

SET2 ►

Input

Mem. Address	Content	Remarks
50	05	No1
51	0D	No2
52	D0	No3
53	AA	No4
54	12	No5
55	32	No6
56	D0	No7
57	0A	No8
58	8F	No9
59	0A	No10

Output

5A → 00 (No. of times DDH found)

14.3: Write a program to arrange a set of ten 8-bit numbers stored from the memory location 50H onward in ascending order.

It is a program the Bubble sort technique is used to arrange the numbers. In Bubble sort, there will be (N – 1) no. of passes for N no. of 8-bit numbers and number of comparisons between two consecutive numbers decreases by one for every pass. Comparisons between two successive numbers are always started from the first number corresponding to all passes. If there are five numbers, for 1st pass there will be four comparisons, for 2nd pass there will be three comparisons, for 3rd pass two comparisons and for 4th pass single comparison will be done. In each comparison, if first number is greater than the second one, they are interchanged i.e. the first number goes in the position of second number and the second number comes in the position of the first number. In this way the largest number will occupy the last position after the completion of 1st pass. Similarly the second largest number will be placed at the last but one position after the completion of 2nd pass. If this process continues, we get completely sorted numbers in ascending order after the completion of all the passes. Now it is necessary to take an example to sort five numbers in ascending order for better clarification which is explained previously in the program of sorting for 8085 microprocessor.

Assembly Language Program 14.3:

SL.	Label	Instructions of 8051
1		MOV R2,#09H
2	LOOP1	MOV R0,#50H
3		MOV A,R2
4		MOV R3,A
5	LOOP2	MOV A,@R0
6		INC R0
7		MOV B,@R0
8		CJNE A,B,NEXT
9	NEXT	JC SKIP
10		DEC R0
11		MOV @R0,B
12		INC R0
13		MOV @R0,A
14	SKIP	DJNZ R3,LOOP2
15		DJNZ R2,LOOP1
16	HERE	SJMP HERE

Result of Program 14.3:

SET1 ►

Input **Output**

Before Sorting

Mem. Address	Content	Remarks
50	05	
51	0D	
52	DD	
53	AA	
54	12	
55	32	
56	71	
57	0A	
58	8F	
59	0A	

After Sorting

Mem. Address	Content	Remarks
50	05	
51	0A	
52	0A	
53	0D	
54	12	
55	32	
56	71	
57	8F	
58	AA	
59	DD	

14.4: Suppose two sorted lists of ten and five numbers are stored starting from memory location 40H onward and 50H onward respectively. Write a program to merge these two sorted lists into a separate list in such a way that the generated list also will be in sorted form and will be stored from 60H onward. Assume all the lists are sorted in ascending order in this program.

In this case 1st sorted list is stored from 40H and 2nd sorted list is stored from 50H. If the 1st and 2nd list consist of m and n no. of elements, the 3rd list after merging will consist (m + n) no. of elements. Here one element from the 1st list and another element from the 2nd list will be compared to each other. Between these two elements which one is smaller will be copied into the 3rd list. Thus this procedure will continue until any one list becomes exhausted i.e. all the elements of that list are transferred to the 3rd list. After this, the remaining elements of the other list will be copied to 3rd list consecutively until it becomes exhausted. Finally the 3rd list of (m + n) elements thus formed starting from memory location 60H, becomes automatically sorted in ascending order. The above mentioned procedure is explained pictorially as shown below with two lists of 5 and 2 elements respectively where 7 iterations (5 + 2) are needed to create the 3rd sorted list.

Iteration 1:

1st Sorted List

Address	Content
40	05 (Smaller)
41	0D
42	DD
43	DF
44	EE

↔

2nd Sorted List

Address	Content
50	0A
51	32

3rd Sorted List

Address	Content
60	05

Iteration 2:

1st Sorted List

Address	Content
40	05
41	0D
42	DD
43	DF
44	EE

↔

2nd Sorted List

Address	Content
50	0A (Smaller)
51	32

3rd Sorted List

Address	Content
60	05
61	0A

Iteration 3:

1st Sorted List

Address	Content
40	05
41	0D (Smaller)
42	DD
43	DF
44	EE

↔

2nd Sorted List

Address	Content
50	0A
51	32

3rd Sorted List

Address	Content
60	05
61	0A
62	0D

Iteration 4:

1st Sorted List

Address	Content
40	05
41	0D
42	DD
43	DF
44	EE

↔

2nd Sorted List	
Address	Content
50	0A
51	32 (Smaller)

2nd Sorted List is exhausted

3rd Sorted List

Address	Content
60	05
61	0A
62	0D
63	32

Iteration 5:

1st Sorted List	
Address	Content
40	05
41	0D
42	DD
43	DF
44	EE

2nd Sorted List	
Address	Content
50	0A
51	32

3rd Sorted List

Address	Content
60	05
61	0A
62	0D
63	32
64	DD

Iteration 6:

1st Sorted List	
Address	Content
40	05
41	0D
42	DD
43	DF
44	EE

2nd Sorted List	
Address	Content
50	0A
51	32

3rd Sorted List

Address	Content
60	05
61	0A
62	0D
63	32
64	DD
65	DF

Iteration 7:

1st Sorted List	
Address	Content
40	05
41	0D
42	DD
43	DF
44	EE

2nd Sorted List	
Address	Content
50	0A
51	32

3rd Sorted List

Address	Content
60	05
61	0A
62	0D
63	32
64	DD
65	DF
66	EE

Note: Gray colored cells are indicating that they have already been transferred to destination memory locations.

In this program registers R2 and R3 acts as memory pointer of 1st sorted list and 2nd sorted list respectively and R1 register is the memory pointer of 3rd merged list. Three registers (R4, R5 and R6) will be used as counters of 1st, 2nd and 3rd list respectively. After every comparison the smaller element will be added to the 3rd list and the memory pointer R1 of 3rd list along with any one memory pointer (either R2 or R3) will be incremented by 1 to get access of the next memory location. This process will be repeated until any one counter of 1st or 2nd list becomes zero. As soon as the particular counter of one list becomes zero, the remaining elements of the other list will be added to the 3rd list one by one. Thus a merged 3rd list whose all the elements are arranged in ascending order is formed ultimately.

Assembly Language Program 14.4:

SL.	Label	Instructions of 8051
1		MOV R1,#60H
2		MOV R2,#40H
3		MOV R3,#50H
4		MOV R4,#0AH
5		MOV R5,#05H
6		MOV R6,#0FH
7	REPEAT	MOV A,R4
8		CJNE A,#00H,L1
9	L4	MOV A,R3

SL.	Label	Instructions of 8051
10		MOV R0,A
11		MOV A,@R0
12		MOV @R1,A
13		INC R3
14		INC R1
15		DEC R5
16		SJMP L3
17	L1	MOV A,R5
18		CJNE A,#00H,L2
19	L5	MOV A,R2
20		MOV R0,A
21		MOV A,@R0
22		MOV @R1,A
23		INC R2
24		INC R1
25		DEC R4
26		SJMP L3
27	L2	MOV A,R2
28		MOV R0,A
29		MOV A,@R0
30		MOV B,R3
31		MOV R0,B
32		MOV B,@R0
33		CJNE A,B,NEXT
34	NEXT	JNC L4
35		SJMP L5
36	L3	DJNZ R6,REPEAT
37	HERE	SJMP HERE

Result of Program 14.4:

SET1 ►

Input **Output**

1st Sorted List

Address	Content
40	05
41	0D
42	A5
43	AA
44	AF
45	B1
46	CC
47	D6
48	DA
49	DD

2nd Sorted List

Address	Content
50	0A
51	1F
52	32
53	A9
54	B9

3rd Sorted List

Address	Content
60	05
61	0A
62	0D
63	1F
64	32
65	A5
66	A9
67	AA
68	AF
69	B1
6A	B9
6B	CC
6C	D6
6D	DA
6E	DD

Exercise

1) Write a program to find the largest number from a list of sixteen 8-bit numbers which are stored from the memory location 50H onward and store the largest number in register R3.

2) Write a program to find the smallest number from a list of ten 8-bit numbers which are stored from the memory location 50H onward and store the smallest number in register R4.

3) Write a program to arrange a set of ten 8-bit numbers stored from the memory location 50H onward in descending order using bubble sort.

4) Write a program to determine the no. of times FF present in a set of 20 8-bit numbers which are stored from memory location 60H. Store the count value at the memory location 5FH.

5) Suppose two sorted lists of eight and five numbers are stored in ascending order starting from memory location 40H onward and 50H onward respectively. Write a program to merge these two sorted lists into a separate list in such a way that the generated list will be in descending order and will be stored from 9050H onward.

15. Programs on Data Conversion

15.1: Write a program to convert a 2-digit packed BCD number stored at memory location 50H to unpacked BCD numbers which will be stored at memory locations 51H and 52H.

We know that a 2-digit packed BCD number is 8 bits long where lower 4 bits (lower nibble) forms LSD (Least significant digit) and upper 4 bits (upper nibble) forms MSD (Most significant digit). Now these two digits should be separated to form two unpacked BCD numbers. For example – 52 is a packed BCD and the corresponding unpacked BCD numbers are 05 (MSD) and 02 (LSD).

Now to extract out the LSD the packed BCD should be AND operated with 0FH. On the contrary the MSD will be separated after performing AND operation with F0H and the result of AND operation has to be shifted right 4 times. How a packed BCD 52H will be converted to unpacked BCDs are shown below.

	B_7	B_6	B_5	B_4	B_3	B_2	B_1	B_0
2-digit packed BCD (52H) →	0	1	0	1	0	0	1	0
0FH →	0	0	0	0	1	1	1	1
Bitwise AND operation →	---	---	---	---	---	---	---	---
Unpacked BCD with LSD (02H) →	0	0	0	0	0	0	1	0

	B_7	B_6	B_5	B_4	B_3	B_2	B_1	B_0
2-digit packed BCD (52H) →	0	1	0	1	0	0	1	0
F0H →	1	1	1	1	0	0	0	0
Bitwise AND operation →	---	---	---	---	---	---	---	---
Result of AND operation (50H) →	0	1	0	1	0	0	0	0
After 1st right shift →	0	0	1	0	1	0	0	0
After 2nd right shift →	0	0	0	1	0	1	0	0
After 3rd right shift →	0	0	0	0	1	0	1	0
Unpacked BCD with MSD (05H) → (After 4th right shift)	0	0	0	0	0	1	0	1

Assembly Language Program 15.1:

SL.	Label	Instructions of 8051
1		MOV R0,#50H
2		MOV A,@R0
3		MOV R2,A
4		ANL A,#0FH
5		INC R0
6		MOV @R0,A
7		MOV A,R2
8		ANL A,#0F0H
9		RR A
10		RR A
11		RR A
12		RR A
13		INC R0
14		MOV @R0,A
15	HERE	SJMP HERE

Result of Program 15.1:

SET1 ►

Input

Address	Content	Remarks
50	68	2 digit packed BCD

Output

Address	Content	Remarks
51	08	Unpacked BCD with LSD
52	06	Unpacked BCD with MSD

SET2 ►

Input

Address	Content	Remarks
50	94	2 digit packed BCD

Output

Address	Content	Remarks
51	04	Unpacked BCD with LSD
52	09	Unpacked BCD with MSD

15.2: Write a program to convert two unpacked BCD numbers stored at memory locations 50H and 51H to a two digits packed BCD number which will be stored at memory locations 52H. Assume that the memory locations 50H and 51H is holding the unpacked BCD numbers containing MSD and the unpacked BCD number containing LSD respectively.

In this program two unpacked BCD numbers – one containing LSD and other containing MSD are joined together to a two digits packed BCD numbers. To do this the unpacked BCD consisting of MSD are shifted left for four times and then it will be OR-operated with the unpacked BCD consisting of LSD to construct the packed BCD number. Two unpacked BCD numbers 04 (LSD) and 08 (MSD) are converted to 2-digit packed BCD using the following technique as shown below.

	B_7	B_6	B_5	B_4	B_3	B_2	B_1	B_0
Unpacked BCD containing MSD (08H) →	0	0	0	0	1	0	0	0
After 1st left shift →	0	0	0	1	0	0	0	0
After 2nd left shift →	0	0	1	0	0	0	0	0
After 3rd left shift →	0	1	0	0	0	0	0	0
After 4th left shift →	1	0	0	0	0	0	0	0
Unpacked BCD containing LSD (04H) →	0	0	0	0	0	1	0	0
Bitwise OR operation →	---	---	---	---	---	---	---	---
2-digit Packed BCD (84H) →	1	0	0	0	0	1	0	0

Assembly Language Program 15.2:

SL.	Label	Instructions of 8051
1		MOV R0,#50H
2		MOV A,@R0
3		RL A
4		RL A
5		RL A
6		RL A
7		INC R0
8		MOV B,@R0
9		ORL A,B
10		INC R0
11		MOV @R0,A
12	HERE:	SJMP HERE

Result of Program 15.2:

SET1 ►

Input

Address	Content	Remarks
50	07	Unpacked BCD containing MSD
51	02	Unpacked BCD containing LSD

Output

Address	Content	Remarks
52	72	Packed BCD

SET2 ►

Input

Address	Content	Remarks
50	06	Unpacked BCD containing MSD
51	05	Unpacked BCD containing LSD

Output

Address	Content	Remarks
52	65	Packed BCD

15.3: Write a program to convert a 2-digit packed BCD number stored at memory location 50H to its equivalent Hexadecimal number which will be stored into memory location 51H.

Method 1: The two packed digit BCD number is converted to two unpacked BCD numbers first. For example if the packed BCD number is 25, the unpacked BCD numbers will be 02 and 05 respectively, where 02 is MSD (Most significant digit) and 05 is LSD (Least significant digit). Here basically the two digits are separated and LSD is added with 10 times of MSD to get the equivalent Hexadecimal number. Therefore Hexadecimal number = 10 × MSD + LSD.

In this program 10 × MSD is stored in register A and LSD is stored in register R3. Finally register A and register R3 are added together to get the Hexadecimal number. We know 10 × MSD = 8 × MSD + 2 × MSD. If a number is shifted left 3 times, it will be multiplied with 8 and if a number is shifted left 1 time, it will be multiplied with 2. Here initially MSD is in the upper nibble and the lower nibble is zero. If it is shifted right 1 time, it is equivalent to shifting left 3 times for getting 8 × MSD and if it is shifted right 3 times we shall get 2 × MSD. Here the number masked with F0H is shifted right one time to get 8 × MSD and shifted right 3 times to get 2 × MSD. Finally there two are added together to get 10 × MSD. This is explained in the following example.

The packed BCD number = 25
Masked with 0F = 05 and masked with F0 = 20 = 0010 0000
After shifted right 1 time = 0001 0000 = 16 = 8 × 2
After shifted right 3 times = 0000 0100 = 4 = 2 × 2
Now 10 × 2 = 8 × 2 + 2 × 2
Therefore equivalent HEX number = 10 × 2 + 05

Assembly Language Program 15.3 (Method 1):

SL.	Label	Instructions of 8051
1		MOV A,50H
2		MOV R2,A
3		ANL A,#0FH
4		MOV R3,A
5		MOV A,R2
6		ANL A,#0F0H
7		RR A
8		MOV R2,A
9		RR A
10		RR A
11		ADD A,R2
12		ADD A,R3
13		MOV 51H,A
14	HERE	SJMP HERE

Method 2: In this method the two digit packed BCD number is unpacked into LSD (least significant digit) and MSD (most significant digit) first, then the MSD is multiplied by 10 with the help of "MUL AB" instruction of 8051 microcontroller. Thus the achieved 10 times of MSD is added with the unpacked LSD to get the equivalent hexadecimal number.

Assembly Language Program 15.3 (Method 2):

SL.	Label	Instructions of 8051
1		MOV A,50H
2		MOV R2,A
3		ANL A,#0FH
4		MOV R3,A
5		MOV A,R2
6		ANL A,#0F0H
7		RR A

SL.	Label	Instructions of 8051
8		RR A
9		RR A
10		RR A
11		MOV B,#10
12		MUL AB
13		ADD A,R3
14		MOV 51H,A
15	HERE	SJMP HERE

Result of Program 15.3:

SET1 ►

Input

RAM Address	Content	Remarks
50	99	2 digit packed BCD

Output

Mem. Address	Content	Remarks
51	63	Equivalent Hex No.

SET2 ►

Input

RAM Address	Content	Remarks
8050	15	2 digit packed BCD

Output

RAM Address	Content	Remarks
51	0F	Equivalent Hex No.

15.4: Write a program to convert an 8-bit Hexadecimal number stored at memory location 50H to unpacked BCD numbers which will be stored starting from memory location 51H.

Method 1: In this case the Hexadecimal number is converted to three unpacked BCDs i.e. three digits are separated and saved into three different memory locations. For example – if the Hexadecimal number is FEH (254 in Decimal), then three unpacked BCD digits 02, 05 and 04 will be stored into three consecutive memory locations starting from 51H. That means MSD (most significant digit) will be stored at 51H, ID (Intermediate digit) will be stored at 52H and LSD (least significant digit) will be stored at 53H. For this purpose the Hexadecimal number is divided by 100 (64 in HEX) first, where quotient gives the 1st unpacked BCD (MSD). The remainder is again divided by 10 (0A in HEX) to get 2nd unpacked BCD (ID) in the quotient and 3rd unpacked BCD (LSD) in the remainder. These three unpacked BCDs are stored consecutively in the memory locations starting from 51H to 53H.

Assembly Language Program 15.4 (Method 1):

SL.	Label	Instructions of 8051
1		MOV R0,#50H
2		MOV A,@R0
3		MOV B,#100
4		DIV AB
5		INC R0
6		MOV @R0,A
7		MOV A,B
8		MOV B,#10
9		DIV AB
10		INC R0
11		MOV @R0,A
12		INC R0
13		MOV @R0,B
14	HERE	SJMP HERE

Method 2: In this case the Hexadecimal number is converted to three unpacked BCDs i.e. three digits are separated and saved into three different memory locations. For example – if the Hexadecimal number is FEH (254 in Decimal), then three unpacked BCD digits 04, 05 and 02 will be stored into three consecutive memory locations starting from 51H. That means LSD (least significant digit) will be stored at 51H, ID (Intermediate digit) will be stored at 52H and MSD (most significant digit) will be stored at 53H. To do this the hexadecimal number is divided by 10, which will give 3rd unpacked BCD (LSD) as remainder, after that the quotient is again divided by 10 to give 2nd unpacked BCD (ID) as remainder and 1st unpacked BCD (MSD) as quotient.

Assembly Language Program 15.4 (Method 2):

SL.	Label	Instructions of 8051
1		MOV R0,#50H
2		MOV A,@R0
3		MOV B,#10
4		DIV AB

SL.	Label	Instructions of 8051
5		INC R0
6		MOV @R0,B
7		MOV B,#10
8		DIV AB
9		INC R0
10		MOV @R0,B
11		INC R0
12		MOV @R0,A
13	HERE	SJMP HERE

Result of Program 15.4:

SET1 ►

Input

RAM Address	Content	Remarks
50	FD	2 digit Hex No.

Hex No. = FD Equivalent Decimal No. = 253

Output

RAM Address	Content	Remarks
51	02	Unpacked BCD1
52	05	Unpacked BCD2
53	03	Unpacked BCD3

SET2 ►

Input

RAM Address	Content	Remarks
50	E1	2 digit Hex No.

Hex No. = E1 Equivalent Decimal No. = 225

Output

RAM Address	Content	Remarks
51	02	Unpacked BCD1
52	02	Unpacked BCD2
53	05	Unpacked BCD3

15.5: Write a program to convert a Hexadecimal number to its equivalent ASCII numbers. Store the Hexadecimal number at 60H and corresponding ASCII numbers at 61H and 62H respectively.

A single digit Hexadecimal number is represented by any digit from 0 to 9 and any alphabet from A to F. Here the two digit Hexadecimal number is separated into two single digit Hexadecimal number by masking the upper nibble, right shifting and masking the lower nibble. Now each single digit Hexadecimal number will be converted to its equivalent ASCII numbers. The ASCII values of 0 to 9 and A to F are given below.

Hexadecimal Number	ASCII Value
0	30H
1	31H
2	32H
3	33H
4	34H
5	35H
6	36H
7	37H
8	38H
9	39H
A	41H
B	42H
C	43H
D	44H
E	45H
F	46H

From the ASCII chart it is clear that if Hexadecimal number lies between 0 to 9, 30H will be added with the Hexadecimal number and if Hexadecimal number lies between A to F, then 37H should be added with it to get the corresponding ASCII value.

For example – the hexadecimal number 4EH is separated into 04 (MSD) and 0E (LSD) whereas 04 is achieved by masking with F0H and right shifting it four times and 0EH is achieved by masking with 0FH. Now 30H is added with 04H to get the corresponding ASCII value 34H. On the other hand 37H is added with 0EH to get the ASCII value 45H.

Assembly Language Program 15.5:

SL.	Label	Instructions of 8051
1		MOV R0,#60H
2		MOV A,@R0
3		MOV B,A
4		ANL A,#0FH
5		ACALL HEX2ASCII
6		INC R0
7		MOV @R0,A
8		MOV A,B
9		ANL A,#0F0H
10		RR A
11		RR A
12		RR A
13		RR A
14		ACALL HEX2ASCII
15		INC R0
16		MOV @R0,A
17	HERE	SJMP HERE
18	HEX2ASCII	CJNE A,#0AH,NEXT
19	NEXT	JC DIGIT
20		ADD A,#07H
21	DIGIT	ADD A,#30H
22		RET

Result of Program 15.5:

SET1 ►

Input

Address	Content	Remarks
60	5F	2 digit Hex No.

Output

Address	Content	Remarks
61	35	ASCII Value of 5
62	46	ASCII Value of F

SET2 ►

Input

Address	Content	Remarks
60	A0	2 digit Hex No.

Output

Address	Content	Remarks
61	41	ASCII Value of A
62	30	ASCII Value of 0

15.6: Write a program to construct a Hexadecimal number from two ASCII numbers which are stored at RAM locations 60H and 61H respectively. Store the Hexadecimal number at RAM location 62H.

A single digit Hexadecimal number is represented by any digit from 0 to 9 or any alphabet from A to F. We know that the ASCII values of 0 to 9 lies between 30H to 39H and the ASCII values of A to F lies between 41H to 46H according the ASCII table given below. As a hexadecimal number will be constructed using the two ASCII values, the ASCII values should be provided between 30H to 39H or 41H to 46H. In this program the ASCII value stored at 60H will be used to form the MSD of the hexadecimal number and the ASCII value stored at 61H will be utilized to construct LSD of the hexadecimal number. Now the generated MSD will be shifted left for four times and OR-operated with the LSD to construct the packed 2-digit hexadecimal number.

Hexadecimal Number	ASCII Value
0	30H
1	31H
2	32H
3	33H
4	34H
5	35H
6	36H
7	37H
8	38H
9	39H
A	41H
B	42H
C	43H
D	44H
E	45H
F	46H

From the ASCII chart it is clear that if ASCII value lies between 30H to 39H, 30H will be subtracted from the ASCII value and if ASCII value lies between 41H to 46H, then 37H will be subtracted from it to get the corresponding single digit hexadecimal number. In this way two single digit hexadecimal numbers, one MSD and other LSD, are formed and finally the LSD is OR-operated with 4 times left shifted version of MSD to generate the hexadecimal number.

Assembly Language Program 15.6:

SL.	Label	Instructions of 8051
1		MOV R0,#60H
2		MOV A,@R0
3		ACALL ASCII2HEX
4		RL A
5		RL A
6		RL A
7		RL A
8		MOV B,A
9		INC R0
10		MOV A,@R0
11		ACALL ASCII2HEX
12		ORL A,B
13		INC R0
14		MOV @R0,A
15	HERE	SJMP HERE
16	ASCII2HEX	CLR C
17		SUBB A,#30H
18		CJNE A,#0AH,NEXT
19	NEXT	JC NOACTION
20		SUBB A,#07H
21	NOACTION	RET

Result of Program 15.6:

SET1 ►

Input

Address	Content	Remarks
60	35	ASCII value for MSD
61	46	ASCII value for LSD

Output

Address	Content	Remarks
62	5F	Hexadecimal number

SET2 ►

Input

Address	Content	Remarks
60	41	ASCII value for MSD
61	31	ASCII value for LSD

Output

Address	Content	Remarks
62	A1	Hexadecimal number

15.7: Write a program to convert an 8-bit Hexadecimal number stored at RAM location 50H to its equivalent gray code which will be stored at RAM location 51H.

To determine the corresponding gray code of a binary number the rule is to take the MSB of the binary number unchanged and all the other bits of the gray code is achieved by performing EXOR operation between two consecutive bits of the binary number. If an 8-bit binary number is represented as $B_7B_6B_5B_4B_3B_2B_1B_0$, then the corresponding gray code can be determined as follows.

$G_7 = 0 \oplus B_7 = B_7$ $\quad$ $G_3 = B_4 \oplus B_3$
$G_6 = B_7 \oplus B_6$ $\quad$ $G_2 = B_3 \oplus B_2$
$G_5 = B_6 \oplus B_5$ $\quad$ $G_1 = B_2 \oplus B_1$
$G_4 = B_5 \oplus B_4$ $\quad$ $G_0 = B_1 \oplus B_0$

The above mentioned process can be implemented by right shifting the binary number one bit position, which appends a zero at the MSB position and then performing bit-wise XOR operation between the actual binary number and the right shifted version of the binary number as shown below.

Binary Number →	B_7	B_6	B_5	B_4	B_3	B_2	B_1	B_0
	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕
Right shifted Binary Number →	0	B_7	B_6	B_5	B_4	B_3	B_2	B_1
Gray Code →	G_7	G_6	G_5	G_4	G_3	G_2	G_1	G_0

Assembly Language Program 15.7:

SL.	Label	Instructions of 8051
1		MOV R0,#50H
2		MOV A,@R0
3		MOV B,A
		CLR C
4		RRC A
5		XRL A,B
6		INC R0
7		MOV @R0,A
8	HERE	SJMP HERE

Result of Program 15.7:

SET1 ►

Input

Mem. Address	Content	Remarks
50	25	8-bit Hex Number

Output

Address	Content	Remarks
51	37	8-bit Gray Code

SET2 ►

Input

Mem. Address	Content	Remarks
50	C2	8-bit Hex Number

Output

Address	Content	Remarks
51	A3	8-bit Gray Code

15.8: Write a program to convert an 8-bit gray code stored at RAM location 50H to its equivalent hexadecimal code which will be stored at RAM location 51H.

Suppose an 8-bit gray code is denoted as $G_7G_6G_5G_4G_3G_2G_1G_0$. Now this gray code can be converted to corresponding binary number using the following process.

$B_7 = 0 \oplus G_7 = G_7$
$B_6 = B_7 \oplus G_6 = G_7 \oplus G_6$
$B_5 = B_6 \oplus G_5$
$B_4 = B_5 \oplus G_4$
$B_3 = B_4 \oplus G_3$
$B_2 = B_3 \oplus G_2$
$B_1 = B_2 \oplus G_1$
$B_0 = B_1 \oplus G_0$

The above expressions to convert gray to binary are shown pictorially in Fig-6.1 for 4-bit representation.

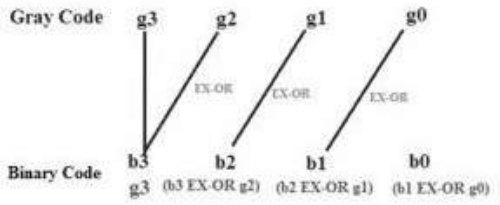


Fig-6.1: Gray to binary conversion

It is being observed that any bit in the converted binary number depends on the previous binary bit. Due to this reason B_6 binary bit can not be determined unless B_7 bit is calculated, B_5 bit can only be determined after the evaluation of B_6 bit and so on. In this program a loop is iterated for 7 times to convert the gray code to binary as shown below.

Iteration 1:

Gray code →	G_7	G_6	G_5	G_4	G_3	G_2	G_1	G_0
	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕
Right shifted Gray code →	0	G_7	G_6	G_5	G_4	G_3	G_2	G_1
Binary code1 →	$G_7 = B_7$	B_6	D_5	D_4	D_3	D_2	D_1	D_0
	↑	↑	↑	↑	↑	↑	↑	↑
	Valid	Valid	Invalid	Invalid	Invalid	Invalid	Invalid	Invalid

Iteration 2:

Gray code →	G_7	G_6	G_5	G_4	G_3	G_2	G_1	G_0
	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕
Right shifted Binary code1 →	0	B_7	B_6	D_5	D_4	D_3	D_2	D_1
Binary code2 →	$G_7 = B_7$	B_6	B_5	D_4	D_3	D_2	D_1	D_0
	↑	↑	↑	↑	↑	↑	↑	↑
	Valid	Valid	Valid	Invalid	Invalid	Invalid	Invalid	Invalid

Iteration 3:

Gray code →	G_7	G_6	G_5	G_4	G_3	G_2	G_1	G_0
	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕
Right shifted Binary code2 →	0	B_7	B_6	B_5	D_4	D_3	D_2	D_1
Binary code3 →	$G_7 = B_7$	B_6	B_5	B_4	D_3	D_2	D_1	D_0
	↑	↑	↑	↑	↑	↑	↑	↑
	Valid	Valid	Valid	Valid	Invalid	Invalid	Invalid	Invalid

Iteration 4:

Gray code →	G_7	G_6	G_5	G_4	G_3	G_2	G_1	G_0
	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕
Right shifted Binary code3 →	0	B_7	B_6	B_5	B_4	D_3	D_2	D_1
Binary code4 →	$G_7 = B_7$	B_6	B_5	B_4	B_3	D_2	D_1	D_0
	↑	↑	↑	↑	↑	↑	↑	↑
	Valid	Valid	Valid	Valid	Valid	Invalid	Invalid	Invalid

Iteration 5:

Gray code →	G_7	G_6	G_5	G_4	G_3	G_2	G_1	G_0
	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕
Right shifted Binary code4 →	0	B_7	B_6	B_5	B_4	B_3	D_2	D_1
Binary code5 →	$G_7 = B_7$	B_6	B_5	B_4	B_3	B_2	D_1	D_0
	↑	↑	↑	↑	↑	↑	↑	↑
	Valid	Valid	Valid	Valid	Valid	Valid	Invalid	Invalid

Iteration 6:

Gray code →	G_7	G_6	G_5	G_4	G_3	G_2	G_1	G_0
	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕
Right shifted Binary code5 →	0	B_7	B_6	B_5	B_4	B_3	B_2	D_1
Binary code6 →	$G_7 = B_7$	B_6	B_5	B_4	B_3	B_2	B_1	D_0
	↑	↑	↑	↑	↑	↑	↑	↑
	Valid	Valid	Valid	Valid	Valid	Valid	Valid	Invalid

Iteration 7:

Gray code →	G_7	G_6	G_5	G_4	G_3	G_2	G_1	G_0
	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕
Right shifted Binary code6 →	0	B_7	B_6	B_5	B_4	B_3	B_2	B_1
Binary code7 →	$G_7 = B_7$	B_6	B_5	B_4	B_3	B_2	B_1	B_0
	↑	↑	↑	↑	↑	↑	↑	↑
	Valid	Valid	Valid	Valid	Valid	Valid	Valid	Valid

It is being observed clearly that the Binary code7 thus achieved finally after 7^{th} iteration is valid.

Assembly Language Program 15.8:

SL.	Label	Instructions of 8051
1		MOV R0,#50H
2		MOV A,@R0
3		MOV R2,A
4		MOV R3,#07H
	REPEAT	CLR C
5		RRC A
6		XRL A,R2
7		DJNZ R3,REPEAT
8		INC R0
9		MOV @R0,A
10	HERE	SJMP HERE

Result of Program 15.8:

SET1 ►

Input

Mem. Address	Content	Remarks
50	37	8-bit Gray code

Output

Address	Content	Remarks
51	25	8-bit hexadecimal number

SET2 ►

Input

Mem. Address	Content	Remarks
50	A3	8-bit Gray code

Output

Address	Content	Remarks
51	C2	8-bit hexadecimal number

15.9: Write a program to add two 8-bit BCD numbers stored at memory locations 50H and 51H respectively. Store the result of the BCD addition at memory locations 52H and 53H respectively where 52H will hold the lower byte of the result and 53H will hold the higher byte of the result.

In this program two BCD numbers stored at RAM locations 50H and 51H are added together. As the maximum value of 2-digit BCD number is 99, the maximum result of BCD addition will be 198 here. Although we are considering the BCD numbers, but a BCD number is basically a hexadecimal number to 8051 microcontroller. For example the BCD number 99 is naturally considered as 99H by the microcontroller. Therefore the addition of two BCD numbers such as (99 + 99) is basically the addition of two hexadecimal numbers such as (99H + 99H) which gives 132H. But we should get the BCD number 198 as a result of BCD addition. To convert 132H to our desired result (198H) the instruction "DA A" (Decimal Adjust Accumulator) should be used just after performing addition between 99H and 99H using the instruction "ADD", because we know "DA A" converts the result of two BCD addition into a BCD number. For example - if we add two BCD numbers 15 and 18, then we get the following results.

BCD Addition	We get the following
15	15
18	18
--------	-------
33	2D
Desired Result	Wrong Result

From the above example it is clear that the result of the BCD addition may be incorrect. DAA instruction rectifies this error and generate the correct result in BCD. In the above example if DAA is used after the addition, it will give 33 as a result. Here one thing is important to mention that DAA instruction should be used after ADD instruction.

Assembly Language Program 15.9:

SL.	Label	Instructions of 8051
1		MOV B,#00H
2		MOV R0,#50H
3		MOV A,@R0
4		INC R0
5		ADD A,@R0
6		DA A
7		JNC NOCARRY
8		INC B

SL.	Label	Instructions of 8051
9	NOCARRY	INC R0
10		MOV @R0,B
11		INC R0
12		MOV @R0,A
13	HERE	SJMP HERE

Result of Program 15.9:

SET1 ►

Input

Mem. Address	Content	Remarks
50	34	2 digit BCD No1
51	15	2 digit BCD No2

Output

Address	Content	Remarks
52	00	Higher Byte of Result
53	49	Lower Byte of Result

SET2 ►

Input

Mem. Address	Content	Remarks
50	98	2 digit BCD No1
51	97	2 digit BCD No2

Output

Address	Content	Remarks
52	01	Higher Byte of Result
53	95	Lower Byte of Result

15.10: Write a program to add two 32-bit BCD numbers stored from memory locations 50H onward and 54H onward respectively. Store the result of the BCD addition from memory locations 58H onward where 58H will hold the least significant byte of the result and 5CH will hold the most significant byte of the result.

Here 1st 32-bit BCD number will occupy 4 consecutive locations starting from 50H to 53H and 2nd 32-bit BCD number will occupy 4 consecutive locations starting from 54H to 57H. After the BCD addition the result will be 5 bytes long, so it will be stored into 5 consecutive RAM locations starting from 58H to 5CH. 5CH location will hold the most significant byte of the result which is basically the carry of the BCD addition and 58H memory location will hold the least significant byte of the result.

The 1st bytes from both numbers are added and converted to BCD using "DA A" instruction, then 2nd bytes from both numbers are added along with the carry from 1st BCD addition and converted to BCD using "DA A" instruction, after that 3rd bytes are added along with the carry from 2nd BCD addition and converted to corresponding BCD and finally 4th bytes from both numbers are added along with the carry from 3rd BCD addition and converted to BCD to achieve the 5 bytes long result. Hence the BCD addition is performed 4 times and to implement this, a loop should be iterated for 4 times as given in the following program.

Assembly Language Program 15.10:

SL.	Label	Instructions of 8051
1		MOV R3,#50H
2		MOV R4,#54H
3		MOV R1,#58H
4		MOV R2,#04H
5		CLR C
6	REPEAT	MOV B,R3
7		MOV R0,B
8		MOV A,@R0
9		MOV B,R4
10		MOV R0,B
11		ADDC A,@R0
12		DA A
13		MOV @R1,A
14		INC R3
15		INC R4
16		INC R1
17		DJNZ R2,REPEAT
18		XRL A,A
19		ADDC A,#00H
20		MOV @R1,A
21	HERE	SJMP HERE

Result of Program 15.10:

SET1 ►

Input

BCD No1

Addr	Content	Remarks
50	66	Byte1
51	77	Byte2
52	88	Byte3
53	99	Byte4

BCD No2

Addr	Content	Remarks
54	99	Byte1
55	88	Byte2
56	77	Byte3
57	66	Byte4

Output

Result

Addr	Content	Remarks
58	65	Byte1
59	66	Byte2
5A	66	Byte3
5B	66	Byte4
5C	01	Byte5

SET2 ►

Input

No1

Addr	Content	Remarks
50	44	Byte1
51	89	Byte2
52	57	Byte3
53	12	Byte4

No2

Addr	Content	Remarks
54	17	Byte1
55	20	Byte2
56	49	Byte3
57	52	Byte4

Output

Result

Addr	Content	Remarks
58	61	Byte1
59	09	Byte2
5A	07	Byte3
5B	65	Byte4
5C	00	Byte5

Exercise

1) Write a program to convert a 2-digit packed BCD number stored at 50H to its equivalent packed Excess 3 codes which should be placed at RAM location 51H.
[Example: Packed 2-digit BCD: 92 → Packed 2-digit Excess 3 Code: C5]

2) Write a program to convert a 2-digit packed Excess 3 code stored at 50H to its equivalent 2-digit packed BCD number which should be placed at RAM location 51H.

16. Programs on Look up Table

16.1: Write a program to determine the square of a number which is stored at RAM location 50H using Look up Table. Also store the square of the number at RAM location 51H.

Although the square of a number can be evaluated by multiplying the number with itself, but here the square of a number is determined by using look up table to develop the concept of the look up table. Here a portion of the program memory has been used to store the square of the numbers 00H to 0FH. We can not store the square of a number beyond 0F (15 in Decimal), because it exceeds the maximum range of a 8-bit number, FFH (255 in Decimal). In this program the look up table has been started from the memory location 400H onward in the code memory or the program memory of 8051 microcontroller as shown below.

Look up Table

Program Memory Address	Square of 8-bit number	
400H	00H (0)	←square of 0
401H	01H (1)	←square of 1
402H	04H (4)	←square of 2
403H	09H (9)	←square of 3
404H	10H (16)	←square of 4
405H	19H (25)	←square of 5
406H	24H (36)	←square of 6
407H	31H (49)	←square of 7
408H	40H (64)	←square of 8
409H	51H (81)	←square of 9
40AH	64H (100)	←square of 10
40BH	79H (121)	←square of 11
40CH	90H (144)	←square of 12
40DH	A9H (169)	←square of 13
40EH	C4 (196)	←square of 14
40FH	E1H (225)	←square of 15

The above mentioned square of numbers 00H – 0FH will be stored from 400H – 40FH into the flash memory of 8051 microcontroller using the following assembler directives at the end of the program in Keil.

ORG 0400H
DB 00H,01H,04H,09H,10H,19H,24H,31H,40H,51H,64H,79H,90H,0A9H,0C4H,0E1H

To get the square of a number, that particular number is added with the starting address of the look up table to get the memory location where the square of that number is saved. Now the content of that memory address is retrieved to get the square of the number. To retrieve the square number from Look up table into code/ program memory the instruction "MOVC A,@A+DPTR" will be used, where DPTR will hold the starting address of the look up table and the number whose square is to be determined will be stored into accumulator. (A + DPTR) will give the address where the square of a given number is stored. After the execution of "MOVC A,@A+DPTR" the accumulator will hold the square value which will be stored at the desired memory location.

Assembly Language Program 16.1:

SL.	Label	Instructions of 8051
1		MOV R0,#50H
2		MOV A,@R0
3		MOV DPTR,#400H
4		MOVC A,@A+DPTR
5		INC R0
6		MOV @R0,A
7	HERE	SJMP HERE

Result of Program 16.1:

SET1 ►

Input

Mem. Address	Content	Remarks
50	09	Number

Output

Address	Content	Remarks
51	51	Square of the number

SET2 ►

Input

Mem. Address	Content	Remarks
50	0F	Number

Output

Address	Content	Remarks
51	E1	Square of the number

Exercise

1) Suppose a Common Cathode 7-segment display is connected to Port2 of 8051 microcontroller via a 74373 latch which is made enabled already by setting LE pin high. The different pins of the 7-segment display is connected to the Port2 as follows.
   $P_{2.0}$ → a segment
   $P_{2.1}$ → b segment
   $P_{2.2}$ → c segment
   $P_{2.3}$ → d segment
   $P_{2.4}$ → e segment
   $P_{2.5}$ → f segment
   $P_{2.6}$ → g segment
   $P_{2.7}$ → h dot point

   Now write a program to convert a single digit BCD number stored at memory location 50H to its equivalent 7 segment display code using look up table and send the 7-segment equivalent code through Port2 to show the BCD number on the 7-segment display.

2) Suppose a Common Anode 7-segment display is connected to Port2 of 8051 microcontroller via a 74373 latch which is made enabled already by setting LE pin high. The different pins of the 7-segment display is connected to the Port2 as follows.
   $P_{2.0}$ → a segment
   $P_{2.1}$ → b segment
   $P_{2.2}$ → c segment
   $P_{2.3}$ → d segment
   $P_{2.4}$ → e segment
   $P_{2.5}$ → f segment
   $P_{2.6}$ → g segment
   $P_{2.7}$ → h dot point

   Now write a program to convert a single digit BCD number stored at memory location 50H to its equivalent 7 segment display code using look up table and send the 7-segment equivalent code through Port2 to show the BCD number on the 7-segment display.

17. Programs on String Manipulation

17.1: Suppose a string is stored from memory location 50H to 57H. Write a program to reverse the string and store the reversed string starting from 60H onward.

Suppose a string "0123456789ABCDEF" is stored from memory location 50H to 57H as shown in Fig-8.1. After the execution of the program the string will be reversed and the reversed string "FEDCBA9876543210" will be stored starting from 60H onward as shown in Fig-8.2.

Mem. Address	Content
50	01
51	23
52	45
53	67
54	89
55	AB
56	CD
57	EF

Fig-8.1: Source string

Mem. Address	Content
60	FE
61	DC
62	BA
63	98
64	76
65	54
66	32
67	10

Fig-8.2: Reversed string

Here every 8-bit data is to be copied starting from memory location 57H to accumulator, swap the nibbles of the accumulator by using SWAP A instruction and save the swapped data starting from memory location 60H onward. That means the source string should be copied from memory location 57H to 50H and the reversed string should be stored from memory location 60H to 67H.

Assembly Language Program 17.1:

SL.	Label	Instructions of 8051
1		MOV R2,#0AH
2		MOV R0,#57H
3		MOV R1,#60H
4	REPEAT	MOV A,@R0
5		SWAP A
6		MOV @R1,A

SL.	Label	Instructions of 8051
7		DEC R0
8		INC R1
9		DJNZ R2,REPEAT
10	HERE	SJMP HERE

Result of Program 17.1:

SET1 ►

Input

Source String

Mem. Address	Content	Remarks
50	01	
51	23	
52	45	
53	67	
54	89	
55	AB	
56	CD	
57	EF	

Output

Reversed String

Mem. Address	Content	Remarks
60	FE	
61	DC	
62	BA	
63	98	
64	76	
65	54	
66	32	
67	10	

SET2 ►

Input

Source String

Mem. Address	Content	Remarks
50	1F	
51	2E	
52	3D	
53	4C	
54	5B	
55	6A	
56	79	
57	88	

Output

Reversed String

Mem. Address	Content	Remarks
60	88	
61	97	
62	A6	
63	B5	
64	C4	
65	D3	
66	E2	
67	F1	

17.2: Suppose a string is stored from memory location 50H to 57H. Write a program to check whether the string is palindrome or not. If the string is palindrome, 01H should be stored at memory location 58H, otherwise 00H should be stored in the same memory location.

A string is said to be palindrome when it matches exactly with its reversed form. For example – a string "ABCDEF99FEDCBA" is palindrome, because if it is written in reverse order it will be the same string "ABCDEF99FEDCBA". Now for 8051 microcontroller a string always consists of even no. of characters, because each memory location stores 8-bit data which includes two characters. Hence for 8051 it is not possible to store a string which comprises odd no. of characters. Now the question arises how to check it. One thing is important to observe that every pair of characters from starting position is just reverse of the pair of characters from end position. In case of the above string "ABCDEF99FEDCBA" AB from starting positions is just reverse of BA from end position. Similarly CD is reversed of DC and EF is also reversed form of FE.

Here two cases may arise – 1) no. of memory locations consumed by the string is even and 2) no. of memory locations consumed by the string is odd. This implies that every pair of characters is reversed and compared with its counterpart pair of characters up to n/2 for even no. of memory locations and ($\lfloor n/2 \rfloor + 1$) for odd no. of memory locations where n is the no. of memory locations consumed by the string. For examples - the string "ABCDEF99FEDCBA" takes 7 (odd) consecutive memory locations. That's why the checking has to be performed up to 4^{th} memory location. At any stage if the reversed pair of characters does not match with its corresponding pair of characters, the string will not be palindrome. If the reversed pair of characters matches with its corresponding pair of characters up to n/2 or ($\lfloor n/2 \rfloor + 1$), the string will be a palindrome. According to the condition of the program 01H will be stored at the memory location 58H, if the string is palindrome and 00H will be stored if the string is not palindrome. In this program the string occupies 8 consecutive memory locations. That's why the comparisons will be carried out up to n/2 no. of pair of characters and to accomplish this the counter register should be initialized with 4 (8 / 2).

Assembly Language Program 17.2:

SL.	Label	Instructions of 8051
1		MOV R2,#04H
2		MOV R0,#50H
3		MOV R1,#57H
4	REPEAT	MOV A,@R0
5		SWAP A
6		MOV B,@R1
7		CJNE A,B,NOTEQUAL

SL.	Label	Instructions of 8051
8		INC R0
9		DEC R1
10		DJNZ R2, REPEAT
11		MOV A,#01H
12		MOV 58H,A
13		SJMP HERE
14	NOTEQUAL	MOV A,#00H
15		MOV 58H,A
16	HERE	SJMP HERE

Result of Program 17.2:

SET1 ►

Input

Source String

Mem. Address	Content	Remarks
50	AB	
51	CD	
52	EF	
53	12	
54	21	
55	FE	
56	DC	
57	BA	

Output

Mem. Address	Content	Remarks
58	01	Palindrome

SET2 ►

Input

Source String

Mem. Address	Content	Remarks
50	AB	
51	CD	
52	EF	
53	12	
54	34	
55	FE	
56	DC	
57	BA	

Output

Mem. Address	Content	Remarks
58	00	Not Palindrome

17.3: Write a program to check whether a string stored from RAM location 50H onward contains another sub-string stored from RAM location 60H onward or not. Store 01H into the memory location 70H if the main string contains the sub-string, otherwise store 02H into the same memory location.

Here one string known as main string is stored from the memory location 50H and another string known as sub-string is stored starting from memory location 60H. Obviously the length of the sub-string will be less or equal to the length of the main string. Here four cases may happen.

Case 1: In this case no matching happens between the main string and sub-string. For example – if the main string is "1234567890ABCDEF9988" and the sub-string is "2233445566", it is observed that there is no matching between the main string and the sub-string. Therefore 02H should be stored into the memory location 70H to indicate the mismatch between the two strings.

Case 2: Here partial matching occurs between the main string and the sub-string. For example – if the main string is "1234567890ABCDEF9988" and the sub-string is "ABCDEF1122", it is observed that a portion of sub-string "ABCDEF" is found into the main string. As entire sub-string is not found into the main string, it results mismatch between the main string and the sub-string. Therefore 02H will be stored into the memory location 70H.

Case 3: In this case the entire sub-string is found into the main string which results successful matching between the two strings. Hence 01H should be stored in the memory location 70H. For example – complete matching occurs if the main string becomes " 1234567890ABCDEF9988" and the sub-string is "ABCDEF9988".

Case 4: This case consists of both partial matching and complete matching. As complete matching is found finally, 01H will be stored into the same memory location according to the program criteria. For example – if main string is " 12ABCD7890ABCDEF9988" and the sub-string is "ABCDEF9988", then partial matching occurs for "ABCD" from 2nd position whereas complete matching happens for "ABCDEF9988" from 6th position.

Now these above mentioned four cases must be handled in the program to check the matching of two strings. If the no. of 8-bit data in the main string is m and the no. of 8-bit data in the sub-string is n, then there will be no chance of finding the whole sub-string inside the main string beyond (m – n + 1)th data. Therefore we have to compare up to (m – n + 1)th data in the main string, beyond of that there is no chance to get the entire sub-string into the main string. The following example will clearly demonstrate this situation.

Suppose main string "1234567890ABCDEF8899" has 10 no. of 8-bit data and sub-string "ABCDEF8899" has 5 no. of 8-bit data. Therefore we have to search for matching of data up to 6th (10 – 5 + 1) position i.e. up to the data "AB" into the main string, because beyond of that there is no possibility to get the complete matching of sub-string "ABCDEF8899".

Here 1st 8-bit data of the sub-string is started to be compared with all the 8-bit data of main string consecutively from 1st data to (m – n + 1)th data of the main string. If matching is found at any stage, the rest of the data from the sub-string are compared with the data of the main string consecutively. That means, if the 1st data of sub-string is matched with any data of main string, then the comparisons between the pairs of the data – one from sub-string and other from main string are performed successively until the end of the sub-string or a mismatch is found. If every pair of data from the sub-string and the main string are matched perfectly, then it can be concluded that the sub-string is found into the main string and if any mismatch is found, then the 1st data from sub-string and the the data of main string where mismatch was found should be compared once again to get the entire sub-string inside the remaining part of the main string. Here one important point to consider that if mismatch is found after (m – n + 1)th data of the main string, then comparisons are not carried out further to imply the absence of the sub-string inside the main string.

In the following program we have taken a main string with 10 no. of data and the sub-string with 5 no. of data. Therefore comparisons should continue up to 6th data of the main string. To fulfill this purpose register R2 will act as counter of main string and initialized with 06H. Similarly register R3 is used as counter of sub-string and initialized with 05H. In addition to this, register R0 has been used as memory pointer of main string and register R1 has been used as memory pointer for sub-string in this program. If a match is found, register R2 and R3 both will be decremented by one for every iteration, otherwise register R2 only will be decremented by one for each iteration. There are two loops in this program – one is controlled by the counter register R2 and other is controlled by

the counter register R3. The loop of counter register R2 will continue until a matching between the 1st data from the sub-string and any data [up to (m – n + 1)th data] from the main string is found. On the other hand if a matching is found, then the loop of counter register R3 will be initiated. If the loop of register R3 is terminated by decreasing R3 to zero, it is clear that the sub-string is found inside the main string and if the loop of register R2 is terminated for R2 = 0, then the sub-string is not found into the main string.

Assembly Language Program 17.3:

SL.	Label	Instructions of 8051
1		MOV R2,#06H
2		MOV R0,#50H
3		MOV R1,#60H
4	REPEAT	MOV A,@R0
5		MOV B,@R1
6		CJNE A,B,NOTEQUAL
7		MOV R3,#05H
8	AGAIN	MOV A,@R0
9		MOV B,@R1
10		CJNE A,B,INEQAFTERMATCH
11		INC R0
12		INC R1
13		MOV A,R2
14		CJNE A,#01H,AFTER
15	AFTER	JC BYPASS
16		DEC R2
17	BYPASS	DJNZ R3,AGAIN
18		MOV A,#01H
19		SJMP FINAL
20	INEQAFTERMATCH	DEC R0
21		INC R2

SL.	Label	Instructions of 8051
22		MOV R1,#60H
23	NOTEQUAL	INC R0
24		DJNZ R2,REPEAT
25		MOV A,#02H
26	FINAL	MOV 70H,A
27	HERE	SJMP HERE

Result of Program 17.3:

SET1 ► (Corresponds to Case 1)

Input

Main String

Address	Content
50	12
51	34
52	56
53	78
54	87
55	65
56	43
57	21
58	CD
59	EF

Sub-String

Address	Content
60	AB
61	CD
62	EF
63	88
64	99

Output

Address	Content	Remarks
70	02	Sub-string not found

SET2 ► ***(Corresponds to Case 1)***

Input

Main String

Address	Content
50	12
51	34
52	56
53	78
54	87
55	65
56	AB
57	CD
58	EF
59	88

Sub-String

Address	Content
60	AB
61	CD
62	EF
63	88
64	99

Output

Address	Content	Remarks
70	02	Sub-string not found

SET3 ► ***(Corresponds to Case 2)***

Input

Main String

Address	Content
50	12
51	34
52	56
53	78
54	AB
55	CD
56	AB
57	CD
58	EF
59	88

Sub-String

Address	Content
60	AB
61	CD
62	EF
63	88
64	99

Output

Address	Content	Remarks
70	02	Sub-string not found

SET4 ► ***(Corresponds to Case 2)***

Input

Main String

Address	Content
50	12
51	34
52	AB
53	CD
54	56
55	78
56	87
57	EF
58	88
59	99

Sub-String

Address	Content
60	AB
61	CD
62	EF
63	88
64	99

Output

Address	Content	Remarks
70	02	Sub-string not found

SET5 ► ***(Corresponds to Case 2)***

Input

Main String

Address	Content
50	12
51	34
52	56
53	AB
54	CD
55	AB
56	AB
57	CD
58	EF
59	88

Sub-String

Address	Content
60	AB
61	CD
62	EF
63	88
64	99

Output

Address	Content	Remarks
70	02	Sub-string not found

SET6 ► ***(Corresponds to Case 3)***

Input

Main String

Address	Content
50	12
51	34
52	56
53	78
54	87
55	AB
56	CD
57	EF
58	88
59	99

Sub-String

Address	Content
60	AB
61	CD
62	EF
63	88
64	99

Output

Address	Content	Remarks
70	01	Sub-string found

SET7 ► ***(Corresponds to Case 3)***

Input

Main String

Address	Content
50	AB
51	CD
52	EF
53	88
54	99
55	12
56	34
57	56
58	78
59	87

Sub-String

Address	Content
60	AB
61	CD
62	EF
63	88
64	99

Output

Address	Content	Remarks
70	01	Sub-string found

SET8 ► ***(Corresponds to Case 4)***

Input

Main String

Address	Content
50	12
51	AB
52	CD
53	AB
54	CD
55	EF
56	88
57	99
58	34
59	56

Sub-String

Address	Content
60	AB
61	CD
62	EF
63	88
64	99

Output

Address	Content	Remarks
70	01	Sub-string found

SET9 ► ***(Corresponds to Case 4)***

Input

Main String

Address	Content
50	12
51	34
52	56
53	AB
54	CD
55	AB
56	CD
57	EF
58	88
59	99

Sub-String

Address	Content
60	AB
61	CD
62	EF
63	88
64	99

Output

Address	Content	Remarks
70	01	Sub-string found

17.4: Suppose two strings are stored into two memory blocks - 50H to 59H and 60H to 63H respectively. Write a program to insert the second string into the first string starting from the memory location 53H.

It can be observed that the second string stored from 60H to 63H has four 8-bit data and the first string stored from 50H to 59H has ten 8-bit data. To insert the second string at the memory location 53H of the first string, all the data of RAM locations starting from 53H to 59H must be shifted into the memory locations 57H to 5DH first to make a space of 4 bytes so that the second string can be accommodated into that memory space. After shifting the data the entire second string should be copied from the memory locations 60H to 63H into the memory locations 53H to 56H.

Now to make a space of four consecutive memory locations starting from 53H to 56H, seven 8-bit data of first string from the memory locations 53H to 59H should be shifted to the memory locations 57H to 5DH. Hence register R2 has been considered as a counter and initialized with 07H. After shifting these seven data, the four 8-bit data of the second string stored from the memory location 60H to 63H should be copied into the memory locations 53H to 56H. Again the register R2 will be initialized with 04H to act as a counter and will be used to transfer these four data. Thus the second string will be inserted into the first string from the memory location 53H.

Assembly Language Program 17.4:

SL.	Label	Instructions of 8051
1		MOV R2,#07H
2		MOV R0,#59H
3		MOV R1,#5DH
4	REPEAT	MOV A,@R0
5		MOV @R1,A
6		DEC R0
7		DEC R1
8		DJNZ R2,REPEAT
9		MOV R2,#04H
10		MOV R1,#60H
11	AGAIN	INC R0
12		MOV A,@R1
13		MOV @R0,A
14		INC R1

SL.	Label	Instructions of 8051
15		DJNZ R2,AGAIN
16	HERE	SJMP HERE

Result of Program 17.4:

SET1 ►

Input

1st String

Address	Content
50	11
51	22
52	33
53	44
54	55
55	66
56	77
57	88
58	99
59	AA

2nd String

Address	Content
60	BB
61	CC
62	DD
63	EE

Output

Address	Content
50	11
51	22
52	33
53	BB
54	CC
55	DD
56	EE
57	44
58	55
59	66
5A	77
5B	88
5C	99
5D	AA

SET2 ►

Input

1st String

Address	Content
50	12
51	23
52	34
53	45
54	56
55	67
56	78
57	89
58	9A
59	AB

2nd String

Address	Content
60	BC
61	CD
62	DE
63	EF

Output

Address	Content
50	12
51	23
52	34
53	BC
54	CD
55	DE
56	EF
57	45
58	56
59	67
5A	78
5B	89
5C	9A
5D	AB

Exercise

1) Write a program to check whether two strings are identical or not. Consider the two strings having same length of 16 characters are stored from memory location 50H onward and 60H onward respectively.

2) Write a program to replace all the characters ‘A’ with the character ‘D’ in a string which is stored from the memory location 40H onward.

3) Suppose two strings are stored into two memory blocks - 50H to 59H and 60H to 65H respectively. Write a program to concatenate these two strings and store the concatenated string starting from memory location 50H onward

18. Programs of Interfacing with LEDs

18.1: Write a program for 8051 microcontroller to blink a set of 8 LEDs connected to Port 2 with some patterns considering the dalay time of 1 second.

Eight LEDs are connected to Port 2 of 8051 microcontroller with current limiting resistors of 270Ω. The power supply +5V is delivered to the 8051 mounting board KSR85152-MB1 as well as the LED circuits by the USB power supply of PC. That's why the microcontroller 8051 is burnt here first using USBASP programmer disconnecting the RESET circuit with only JP4 jumper open and then all the jumpers JP1, JP2, JP3, JP4 in the 8051 mounting board are made shorted to execute the dumped program with +5V supply from USBASP programmer. The circuit diagram during the execution of the flashed program into the program memory of 8051 is shown in Fig-18.1.

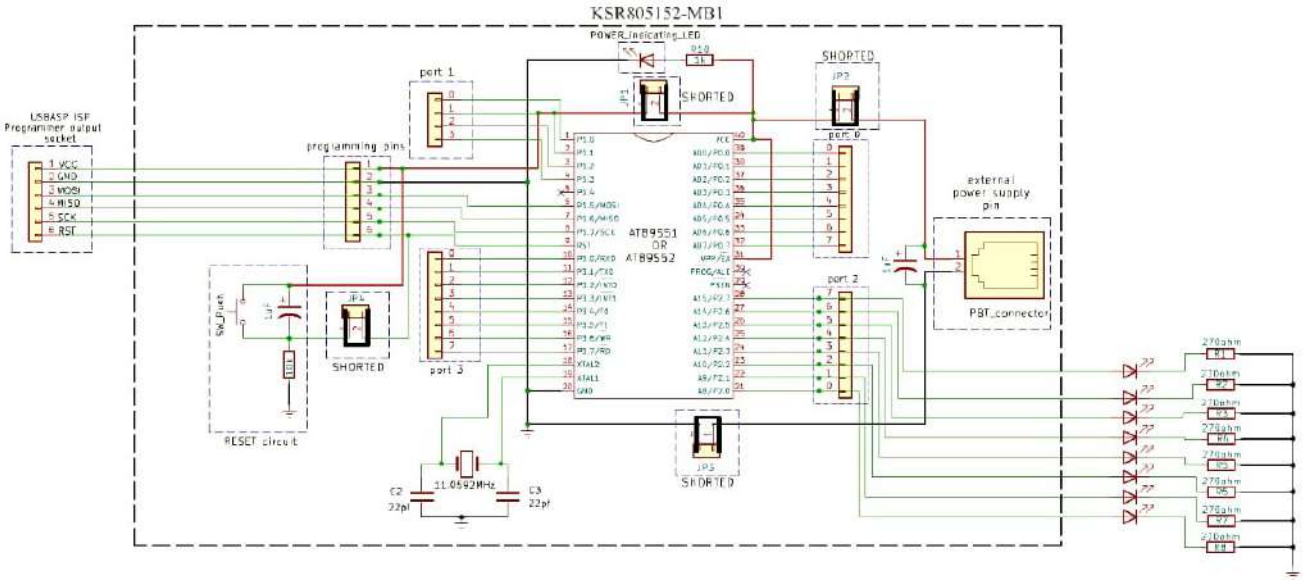


Fig-18.1: Circuit diagram of LED interfacing with 8051 after burning the program

Here two 8-bit patterns of 55H and AAH to glow the LEDs are followed with a time delay of 1 second. That means, the LEDs will glow with a pattern 55H and AAH alternately with a delay of 1 second. Now to create 1 second time delay, a delay subroutine should be written in assembly language. This delay subroutine can be written in two ways – either by using instructions of 8051 or by using in-built Timer of 8051. The delay subroutine of 1 second using the instructions of 8051 along with its calculation is given below in Method 1.

Method 1: *Delay Subroutine using the instructions of 8051*

Label	Instructions of 8051	No. of Machine Cycle
	MOV R2, #n	1
L2:	MOV R3, #255	1
L3:	MOV R4, #255	1
L4:	DJNZ R4, L4	2
	DJNZ R3, L3	2
	DJNZ R2, L2	2
	RET	2

For 8051 microcontroller 1 machine cycle (MC) = $12 \times T_{OSC} = 12 / f_{OSC}$, where f_{OSC} is the operating frequency of 8051 or the frequency of the crystal connected to 8051. In our case f_{OSC} = 11.0592 MHz. Therefore here 1 machine cycle = 1.085 μs.

In the above subroutine three nested loops are used. The inner-most loop (DJNZ R4, L4) will be iterated for 255 times, the loop (DJNZ R3, L3) including the inner-most loop will execute for 255 times and the outer-most loop (DJNZ R2, L2) including the two inner loops will execute x times.

Total no. of MCs taken by the subroutine $=1 + ((1+ ((1 + (255 \times 2) + 2) \times 255) + 2) \times n) + 2$

$$= 130818 \times n + 3$$

Now total time taken by the subroutine = $(130818 \times n +3) \times 1.085$ μs which will be equal to 1 second.

So, $(130818 \times n +3) \times 1.085 = 10^6$

or, $(130818 \times n +3) = 921659$

$\therefore n = 7$

If we replace n by 7 in the subroutine, it will generate 1 second delay when it will be executed for 1 time. To create 1 second delay this subroutine will be called using the instruction "ACALL DELAY". Now the entire 8051 program is written below including the above mentioned delay subroutine.

Assembly Language Program 18.1 (Method 1):

SL.	Label	Instructions of 8051
1		MOV A, #55H
2	AGAIN:	MOV P2, A
3		ACALL DELAY
4		CPL A
5		SJMP AGAIN
6	DELAY:	MOV R2, #7
7	L2:	MOV R3, #255
8	L3:	MOV R4, #255
9	L4:	DJNZ R4, L4
10		DJNZ R3, L3
11		DJNZ R2, L2
12		RET

Method 2: *Delay subroutine using Timer of 8051*

8051 has two timers, Timer 0 and Timer 1. They can be used as timer or event counter. The purpose of timer is to create a delay very accurately where as the counter will count the no. of pulses coming at the pin T0 (pin no. 14) or T1 (pin no. 15) of 8051 microcontroller. Timer 0 and Timer 1 both are 16 bit wide. 16 bit register of Timer 0 can be accessed as lower byte and higher byte whereas the lower byte register is called TL0 and the higher byte register is called TH0. Similarly lower byte register and higher byte register of Timer1 are called TL1 and TH1 respectively. There are four modes of timer namely Mode 0, Mode 1, Mode 2 and Mode 3. Among these four modes we are interested to Mode 1 in our laboratory purpose. Mode 1 of timer is also called 16-bit timer mode. In Mode 1 TH0 and TL0 of Timer 0 together hold the 16-bit initial value from where Timer 0 starts to increase its value i.e. with each clock pulse the 16-bit value of Timer 0 will be incremented by one. Timer 0 is started by making the bit TR0 inside TCON register high. In this way the value of Timer 0 will reach to FFFFH. With one more clock pulse the value of Timer 0 becomes 0000H from FFFFH. This incident is called roll over of the timer. When the roll over happens, a bit TF0 inside TCON register transits from low to high. The program should check the status of this bit TF0 continuously. As soon soon TF0 becomes high, the Timer 0 is to be stopped by making TR0 low. The same above mentioned incidents are happened for Timer 1 also.

Now the value of a timer takes $(12 \times T_{OSC})$ time to be incremented by 1. If the crystal frequency is 11.0592 MHz, $(12 \times T_{OSC})$ will be equal to 1.085 μs. This implies that the timer will take $1.085 \times p$ μs time to be incremented for p times. The value of p depends on the initial value of the timer. If the

timer is initialized with n, then the timer will be incremented for (65535 – n +1) times. Here the timer will take (65536 – n) × 1.085 μs which will be treated as a delay.

∴ Delay = (65536 – n) × 12 × T_{OSC}

$$= \frac{(65536-n)\times 12}{f_{OSC}}$$ where n is the initial value and f_{OSC} is the crystal frequency

If the delay to be created is known, then the initial value n of the timer can be determined using the above mentioned formula. For example if f_{OSC} = 11.0592 MHz, the initial value is considered as n and the delay to be created is 50 ms, then we get the following equation.

$$\frac{(65536-n)\times 12}{f_{OSC}} = 50 \text{ ms}$$

or, (65536 – n) × 1.085 μs = 50 × 10^3 μs

or, 65536 – n = 46083 ∴ n = 19453 = 4BFDH

Therefore the initial value of the timer is 4BFDH. If Timer 0 is used, TH0 and TL0 are to be loaded with 4BH and FDH respectively. Now the modes of Timer 0 and Timer 1 can be changed with the help of TMOD register which is given below in Fig-18.2.

TMOD: TIMER/COUNTER MODE CONTROL REGISTER. NOT BIT ADDRESSABLE.

GATE	C/$\overline{T}$	M1	M0	GATE	C/$\overline{T}$	M1	M0
TIMER 1				TIMER 0			

GATE When TRx (in TCON) is set and GATE = 1, TIMER/COUNTERx will run only while INTx pin is high (hardware control). When GATE = 0, TIMER/COUNTERx will run only while TRx = 1 (software control).

C/$\overline{T}$ Timer or Counter selector. Cleared for Timer operation (input from internal system clock). Set for Counter operation (input from Tx input pin).

M1 Mode selector bit. (NOTE 1)

M0 Mode selector bit. (NOTE 1)

NOTE 1:

M1	M0	Operating Mode	
0	0	0	13-bit Timer (MCS-48 compatible)
0	1	1	16-bit Timer/Counter
1	0	2	8-bit Auto-Reload Timer/Counter
1	1	3	(Timer 0) TL0 is an 8-bit Timer/Counter controlled by the standard Timer 0 control bits, TH0 is an 8-bit Timer and is controlled by Timer 1 control bits. (Timer 1) Timer/Counter 1 stopped.

Fig-18.2: TMOD Register of 8051 microcontroller

Therefore from Fig-18.2 it is clear that TMOD register must be initialized with 01H to operate Timer 0 in Mode1.

TCON : Timer/Counter Control Register (Bit Addressable)

TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0

TF1	TCON.7	Timer 1 overflow flag. Set by hardware when the Timer/Counter 1 overflows. Cleared by hardware as processor vectors to the interrupt service routine.
TR1	TCON.6	Timer 1 run control bit. Set/cleared by software to turn Timer/Counter ON/OFF.
TF0	TCON.5	Timer 0 overflow flag. Set by hardware when the Timer/Counter 0 overflows. Cleared by hardware as processor vectors to the service routine.
TR0	TCON.4	Timer 0 run control bit. Set/cleared by software to turn Timer/Counter 0 ON/OFF.
IE1	TCON.3	External Interrupt 1 edge flag. Set by hardware when External interrupt edge is detected. Cleared by hardware when interrupt is processed.
IT1	TCON.2	Interrupt 1 type control bit. Set/cleared by software to specify falling edge/flow level triggered External Interrupt.
IE0	TCON.1	External Interrupt 0 edge flag. Set by hardware when External Interrupt edge detected. Cleared by hardware when interrupt is processed.
IT0	TCON.0	Interrupt 0 type control bit. Set/cleared by software to specify falling edge/low level triggered External Interrupt.

Fig-18.3: TCON Register of 8051 microcontroller

In this program we are going to create delay of 1 second using Timer 0 in Mode1. The Timer 0 will generate maximum delay when it will be initialized with 00H. Therefore with crystal frequency of 11.0592 MHz and TH0 = 00H and TH0 = 00H, the Timer 0 gives maximum delay of 71 ms which is obviously less than our desired delay of 1 second. Therefore using Timer 0 only we can not generate a time delay of 1 second. To solve this problem, we have to generate a delay of 50 ms using Timer 0 and iterate the subroutine 20 times to create the delay of 1 second. The delay calculation using Timer 0 is given below.

$$\frac{(65536-n)\times 12}{f_{OSC}} = 50 \text{ ms} \qquad [\, f_{OSC} = 11.0592 \text{ MHz}]$$

or, $(65536 - n) \times 1.085\ \mu s = 50 \times 10^3\ \mu s$

or, $65536 - n = 46083$

$\therefore n = 19453 = 4BFDH$

Therefore TH0 = 4BH, TL0 = FDH and TMOD = 01H
Now the same program (already written in Method 1) is rewritten using Timer 0 in Mode1 for delay subroutine.

Assembly Language Program 18.1 (Method 2):

SL.	Label	Instructions of 8051
1		MOV P2, #55H
2	AGAIN:	ACALL DELAY
3		XRL P2, #0FFH
4		SJMP AGAIN
5	DELAY	MOV TMOD, #01H
6		MOV R2, #20
7	REPIT:	MOV TL0, #0FDH
8		MOV TH0, #4BH
9		SETB TR0
10	HERE:	JNB TF0, HERE
11		CLR TR0
12		CLR TF0
13		DJNZ R2, REPIT
14		RET

C Program 18.1 (Method 2):

```
#include<reg51.h>
void Delay();

void main()
{
        P2 = 0x55;

        while(1)
        {
                Delay();
                P2 = ~P2;
        }
}
```

C Program 18.1 (Method 2):

```
void Delay()
{
        unsigned char i;
        TMOD = 0x01;
        for(i=0;i<20;i++)
        {
                TL0 = 0xFD;
                TH0 = 0x4B;
                TR0 = 1;
                while(TF0==0);
                TR0 = 0;
                TF0 = 0;
        }
}
```

Method 3: *Delay subroutine using interrupt of Timer for 8051*

In this method the delay is created using any one of the timers (Timer 0 or Timer 1) where interrupt happens due to the occurrence of overflow. Before explaining this timer interrupt a brief introduction to the 8051 interrupts are given below.

8051 microcontroller has six interrupts including the reset which is not available to the programmer.

- ***Reset -*** When the reset pin is activated, 8051 jumps to the address location 0000H which is the starting address of the interrupt vector table (IVT) of reset. This IVT of reset ends at memory location 0002H. Therefore the size of IVT for reset is 3 bytes. Basically it is the power-up reset.

- ***Two interrupts for Timer 0 and Timer 1 –*** There are two interrupts, one for Timer 0 and other for Timer 1 which occurs due to the overflow of the timers. As soon as TF0 flag of Timer 0 is set due to the overflow of Timer 0, an interrupt occurs and 8051 jumps to the address 000BH which is the interrupt vector address of Timer 0. Similarly due to the overflow in Timer 1 the program will jumps to the address 001BH which is the starting address of the Timer 1 IVT.

- ***Two external hardware interrupts INT0 and INT1 –*** Pin no. 12 (P3.2) and 13 (P3.3) are used for external hardware interrupts INT0 and INT1 respectively. Any external device may interrupt the 8051 microcontroller through these pins. Memory locations 0003H and 0013H in the interrupt vector table are used for INT0 and INT1 respectively. INT0 and INT1 are also known as EX0 and EX1 respectively.

- ***Serial communication interrupt –*** Serial communication has a single interrupt that belongs to both receive and transmit. Interrupt vector location 0023H is used for serial communication.

Table-18.1: The entire interrupt vector table of 8051 microcontroller

Interrupt	PROM Location	8051 IC Pin	Flag Clearing
Reset	0000H	9	Auto
External hardware interrupt 0 (INT0)	0003H	12 (P3.2)	Auto
Timer 0 interrupt (TF0)	000BH		Auto
External hardware interrupt 1 (INT1)	0013H	13 (P3.3)	Auto
Timer 1 interrupt (TF1)	001BH		Auto
Serial COM interrupt (RI and TI)	0023H		Programmer clears it

Enabling and disabling interrupts – Upon reset the 8051 microcontroller disables all the five interrupts except reset. Therefore it is the responsibility of the programmer to enable one or more interrupts according to his requirement. The interrupts are enabled or disabled with the help of a register called Interrupt Enable register (IE). Note that IE is a bit-addressable register.

IE: Interrupt Enable Register							
IE.7	IE.6	IE.5	IE.4	IE.3	IE.2	IE.1	IE.0
EA	--	ET2	ES	ET1	EX1	ET0	EX0
D7	D6	D5	D4	D3	D2	D1	D0

EA: If EA = 0 all the interrupts are disabled i.e. no interrupt is acknowledged by 8051
If EA = 1 interrupts are individually enabled or disabled by setting or clearing its enable bit

ET2: If ET2 = 0 Timer 2 overflow or capture interrupt is disabled (only for 8052)
If ET = 1 Timer 2 overflow or capture interrupt is enabled (only for 8052)

ES: If ES = 0 serial port interrupt is disabled
If ES = 1 serial port interrupt is enabled

ET1: If ET1 = 0 Timer 1 overflow interrupt is disabled
If ET1 = 1 Timer 1 overflow interrupt is enabled

EX1: If EX1 = 0 external interrupt INT1 is disabled
If EX1 = 1 external interrupt INT1 is enabled

ET0: If ET0 = 0 Timer 0 overflow interrupt is disabled
If ET0 = 1 Timer 0 overflow interrupt is enabled

EX0: If EX0 = 0 external interrupt INT0 is disabled
If EX0 = 1 external interrupt INT0 is enabled

Fig.18.2: Interrupt Enable Register

If the timer interrupt in the IE register is enabled, whenever the timer rolls over, TF is raised and the microcontroller is interrupted and jumps to the interrupt vector table to service the ISR written by the programmer. When the microcontroller jumps to the ISR, it clears the TF flag automatically. After completing the ISR it returns back to the instruction which was being executed at that time.

In this case we have used Timer 0 to generate the delay of 1 sec and Timer 0 Interrupt (TF0) is used to sense the roll over of Timer 0 with the help of interrupt. As interrupt is used here, the memory location 0000H which is basically the vector address of Reset interrupt, can not the used for storing the program code. That's why a jump instruction is used at 0000H to transfer the program sequence to any other location where the program starts to be stored. At the same time another jump instruction is stored at the vector address 000BH of Timer 0 Interrupt to transfer the program sequence to the ISR when the microcontroller is being interrupted for Timer 0 overflaw. The delay subroutine is written using Timer 0 inside the ISR. The delay calculation for 1 sec is identical as described in Method 2. As a result, after every 50 ms time delay an interrupt is triggered due to the overflow of Timer 0. Thus a delay of 1 sec is created after triggering Timer 0 Interrupt for 20 times. To enable Timer 0 Interrupt the following bit pattern is to be set for IE register.

1	0	0	0	0	0	1	0
EA	--	ET2	ES	ET1	EX1	ET0	EX0
D7	D6	D5	D4	D3	D2	D1	D0

Therefore it is being observed clearly that 82H should to loaded to IE register to enable Timer 0 Interrupt. The implementation of same delay creation program using Timer 0 overflow interrupt is given below.

Assembly Language Program 18.1 (Method 3):

SL.	Label	Instructions of 8051
1		*ORG 0000H*
2		LJMP MAIN
3		
4		*ORG 000BH*
5		LJMP ISR
6		
7		*ORG 0030H*
8	MAIN:	MOV P2, #55H
9		MOV TMOD, #01H
10		MOV TL0, #0FDH

SL.	Label	Instructions of 8051
11		MOV TH0, #4BH
12		MOV IE, #82H
13		MOV R2, #20
14		SETB TR0
15	HERE:	SJMP HERE
16		
17	ISR:	CLR TR0
18		DJNZ R2, SKIP
19		XRL P2,#0FFH
20		MOV R2, #20
21	SKIP:	MOV TL0, #0FDH
22		MOV TH0, #4BH
23		SETB TR0
24		RETI
25		*END*

Writing Interrupt Service Routine (ISR) in C: 8051 C compiler (Keil) assigns a unique number to each interrupt in 8051 microcontroller. To implement ISR for 8051 using C language the unique interrupt number is placed after the keyword 'interrupt' in case of ISR (interrupt function) definition as given below.

```
void ISR_Name() interrupt x       // x is the unique interrupt number
{
        statement 1;
        statement 2;
              :
              :
        statement n;
}
```

Note: In case of ISR implementation in C for 8051 in Keil prototype declaration of ISR is not allowed, although it is permitted for normal user-defined function.

Table-18.2: Different interrupts along with its unique interrupt number

Interrupt	Unique Interrupt Number
External hardware interrupt 0 (INT0)	0
Timer 0 interrupt (TF0)	1
External hardware interrupt 1 (INT1)	2
Timer 1 interrupt (TF1)	3
Serial COM interrupt (RI and TI)	4
Timer 2 Interrupt (TF2) (only for 8052)	5

C Program 18.1 (Method 3):

```
#include<reg51.h>
#define TCOUNT 20

void Delay() interrupt 1
{
	static unsigned char i = TCOUNT;
	TR0 = 0;
	i--;
	if(i == 0)
	{
		P2 = ~P2;
		i = 20;
	}
	TL0 = 0xFD;
	TH0 = 0x4B;
	TR0 = 1;
}
void main()
{
	P2 = 0x55;
	TMOD = 0x01;
	TL0 = 0xFD;
	TH0 = 0x4B;
	IE = 0x82;
	TR0 = 1;

	for(;;);
}
```

18.2: Write a program for 8051 microcontroller to blink an LED connected to P2.0 with a dalay time of 1 second.

In this case the circuit shown in Fig-18.1 will work. The LED conneced at P2.0 (pin no. 21) will blink with a delay of 1 second. The other LEDs connected at P2.1 to P2.7 will not change. The delay calculation using Timer or without using Timer remain same as explained in Method 1 or Method 2 of program 18.1. Here the entire program with delay subroutine using Timer 0 is given below.

Assembly Language Program 18.2:

SL.	Label	Instructions of 8051
1		SETB P2.0
2	AGAIN:	ACALL DELAY
3		CPL P2.0
4		SJMP AGAIN
5	DELAY:	MOV TMOD, #01H
6		MOV R2, #20
7	REPIT:	MOV TL0, #0FDH
8		MOV TH0, #4BH
9		SETB TR0
10	HERE:	JNB TF0, HERE
11		CLR TR0
12		CLR TF0
13		DJNZ R2, REPIT
14		RET

Exercise

1) Write a program for 8051 microcontroller to blink an LED connected to P1.3 with a dalay time of 1.5 second. Implement the delay subroutine using Timer 1 in Mode 1 and without using Timer.

2) Write a program for 8051 microcontroller to implement a decade counter which will count with a delay of 1 second. The counting of the decade counter should be displayed on four LEDs connected to P2.0 – P2.3.

3) Write a program for 8051 microcontroller to implement a 4-bit counter which will count with a delay of 2 seconds. The counting of the counter should be displayed on 2 digits 7 segment displays.

4) Write a program for 8051 microcontroller to generate a square wave at P1.0 with an ON time of 3 ms and an OFF time of 10 ms using Timer 0 in Mode 1. Assume the frequency of the crystal to be 11.0592 MHz.

5) Write a program for 8051 microcontroller to blink an LED connected to P1.3 with a dalay time of 1.5 second. Implement the delay subroutine using Timer 1 in Mode 1 and using Timer.1 Overflow Interrupt (TF1).

19. Programs on reading input switch state

19.1: Write a program for 8051 microcontroller to read the states of two input switches connected at P1.0 and P1.1 and generates different blinking patterns of 8 LEDs connected to Port 2 according to the four input patterns.

Here two ON-OFF switches SW1 and SW2 are connected at P1.0 and P1.1 respectively. When the switch is pressed ON, it gives 1 to the corresponding input pin of the 8051 microcontroller and when it is OFF, it sends 0 to the port. Therefore SW1 and SW2 generates four input patterns at Port 1 and reading those patterns the 8051 microcontroller sends four different blinking patterns to Port2 as given in the following table.

SL	SW2	SW1	Output Pattern	Description of output pattern
1	0	0	PTRN0	0000 0000 → All LEDs will remain off.
2	0	1	PTRN1	0011 0011 → 1100 1100 ↑ ↓ 1100 1100 ← 0011 0011
3	1	0	PTRN2	0000 0000 → 0001 1000 → 0011 1100 →0111 1110 ↑ ↓ 0001 1000← 0011 1100 ← 0111 1110 ←1111 1111
4	1	1	PTRN3	0000 1111 → 1111 0000 ↑ ↓ 1111 0000 ← 0000 1111

To provide these four input patterns using two switches, SW1 and SW2 we have used a input module KSR-IP1 where eight ON-OFF DPDT switches are used along with a unidirectional buffer IC 74LS244. Therefore this input module is capable to deliver 8-bit binary pattern to any port of the 8051 microcontroller. The input module along with its circuit diagram are shown in Fig-19.1 and Fig-19.2 respectively.

Fig-19.1: KSR-IP1 Input Module

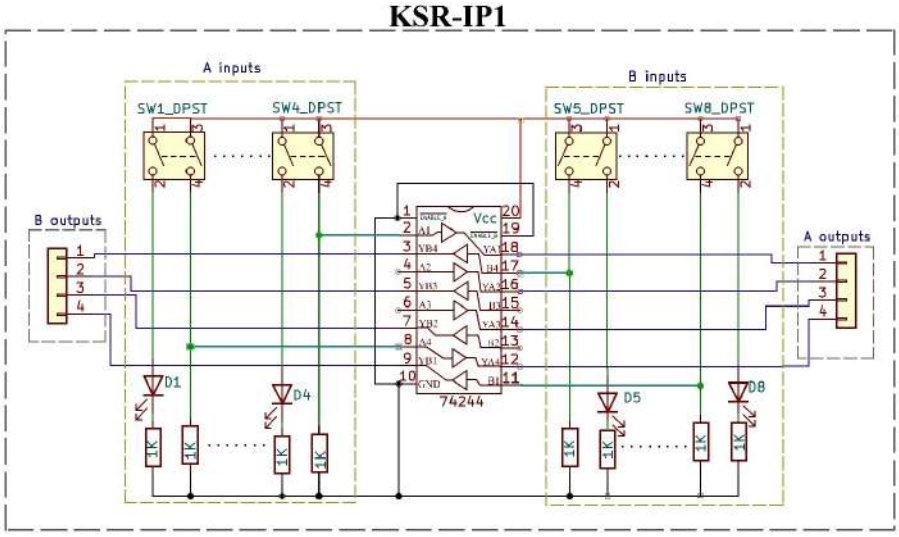


Fig-19.2: Circuit diagram of KSR-IP1 Input Module

Among these eight switches (SW1 – SW8) two switches SW1 and SW2 are connected to P1.0 and P1.1 of Port1 of 8051. The entire circuit diagram with input switches and eight LEDs connected to Port2 is shown in Fig-12.3. Here the whole circuit is driven by the external +5V power supply after burning the 8051 chip using USBASP programmer. For this reason all the jumpers (JP2, JP3, JP4) except JP1 are made shorted.

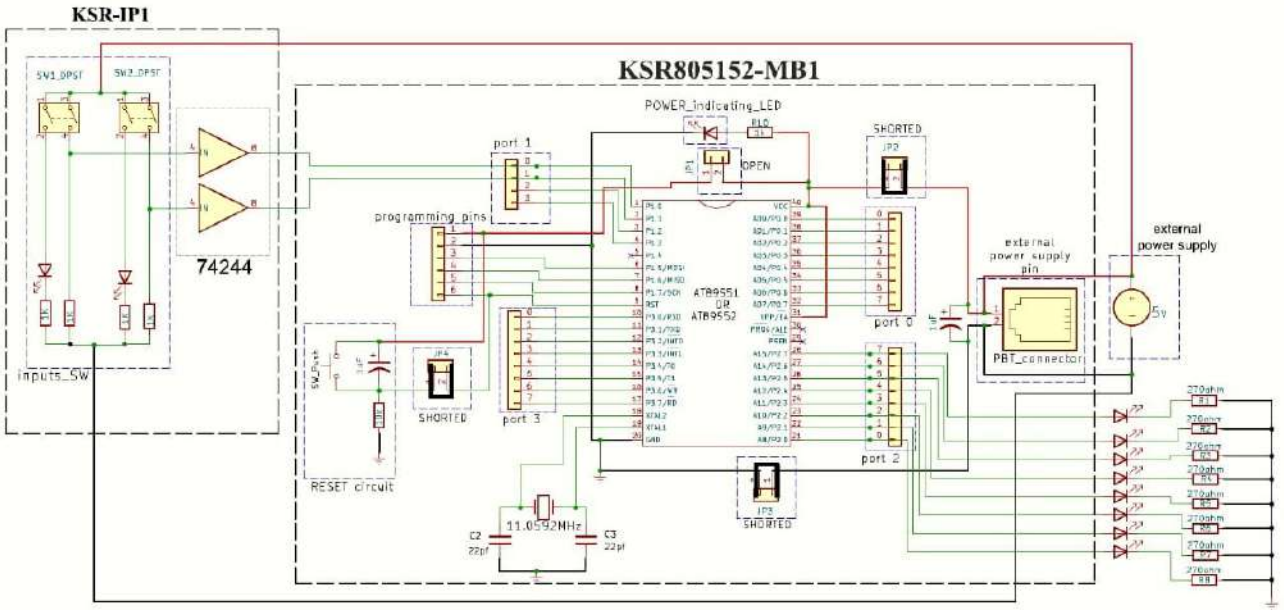


Fig-19.3: Circuit diagram of KSR805152-MB1 board, input module KSR-IP1 connected to Port1 and eight LEDs connected to Port2

To configure any port in 8051 as input port the port register is to be loaded with FFH. As Port1 here is used as an input port, it has been initialized with FFH at the beginning of the program.

Assembly Language Program 19.1:

SL.	Label	Instructions of 8051
1		MOV TMOD,#01H
2		MOV P1,#0FFH
3	AGAIN:	MOV P2,#00H
4		MOV A,P1
5		ANL A,#03H
6		CJNE A,#00,SKIP
7		SJMP AGAIN
8	SKIP:	CJNE A,#01H,L1
9		ACALL PTRN1
10	L1:	CJNE A,#02H,L2
11		ACALL PTRN2
12	L2:	CJNE A,#03H,L3
13		ACALL PTRN3
14	L3:	SJMP AGAIN
15	PTRN1:	MOV P2,#33H
16	AG1:	MOV A,P1
17		ANL A,#03H
18		CJNE A,#01H,RETN1
19		MOV R2,#20
20	REP1:	MOV TL0,#08H
21		MOV TH0,#4CH
22		ACALL DELAY
23		DJNZ R2,REP1
24		XRL P2,#0FFH
25		SJMP AG1
26	RETN1:	RET

SL.	Label	Instructions of 8051
26	PTRN2:	CLR 05H
27		MOV R6,#00H
28	AG2:	MOV A,P1
29		ANL A,#03H
30		CJNE A,#02H,RETN2
31		MOV A,R6
32		MOV C,05H
33		CPL C
34		RLC A
35		MOV 05H,C
36		MOV R6,A
37		MOV R2,#30
38	REP2:	MOV TL0,#08H
39		MOV TH0,#4CH
40		ACALL DELAY
41		DJNZ R2,REP2
42		SJMP AG2
43	RETN2:	RET
44	PTRN3:	MOV P2,#0FH
45	AG3:	MOV A,P1
46		ANL A,#03H
47		CJNE A,#03H,RETN3
48		MOV R2,#30
49	REP3:	MOV TL0,#08H
50		MOV TH0,#4CH
51		ACALL DELAY
52		DJNZ R2,REP3
53		XRL P2,#0FFH
54		SJMP AG3

SL.	Label	Instructions of 8051
55	RETN3:	RET
56	DELAY:	SETB TR0
57	HERE:	JNB TF0,HERE
58		CLR TR0
59		CLR TF0
60		RET

20. Programs of 7 segment display interfacing

A 7-segment display is commonly used in electronic display devices for decimal numbers from 0 to 9 and in some cases, basic characters. The use of LEDs in seven-segment displays made it popular, bright and clear, easy to interface and cost effective. There are 7 illuminating segments (named as a, b, c, d, e, f, g) and a dot (named as DP) in a 7-segment display. Corresponding to each segment and dot there is a LED inside the 7-segment display. A particular segment in a 7-segment display becomes illuminated if the corresponding LED of that segment glows due to the forward biasing. The pin-out of a 7-segment display is shown in Fig-20.1.

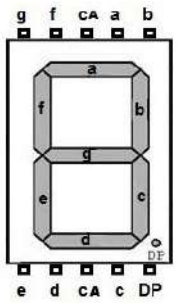


Fig-20.1(a): Pin-out of Common Anode 7-segment display

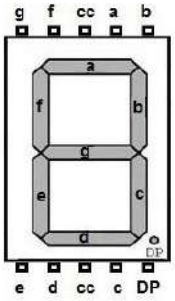


Fig-20.1(b): Pin-out of Common Cathode 7-segment display

Basically there are two types of 7-segment display namely 1) Common Anode 7-segment display and 2) Common Cathode 7-segment display.

1) Common Anode 7-segment display – In this construction all the anodes of eight LEDs are connected together to form a common terminal CA as shown in Fig-20.1(a). Other eight cathode terminals are connected to eight pins namely a, b, c, d, e, f, g and DP. The internal schematic diagram of a common anode 7-segment display is shown in Fig-20.2.

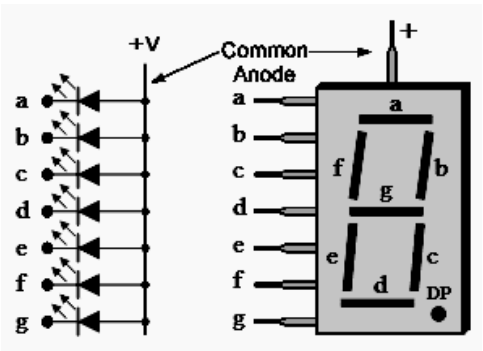


Fig-20.2: Internal schematic diagram of common anode 7-segment display

2) Common Cathode 7-segment display – In this construction all the cathodes of eight LEDs are shorted together to form a common terminal CC as shown in Fig-20.1(b). Other eight anode terminals are connected to eight pins namely a, b, c, d, e, f, g and DP. The internal schematic diagram of a common cathode 7-segment display is shown in Fig-20.3.

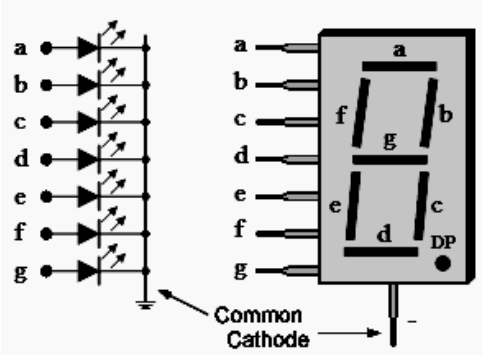


Fig-20.3: Internal schematic diagram of common cathode 7-segment display

Among these two configurations common anode 7 segment display has been used to design the 7 segment display driver board where three 7 segment displays have been used to show any three digit decimal number in the range of 0 to 999. We know that the current rating of any port except Port 0 is around 15 mA in total which is not sufficient to drive the 7 segment display. That's why a driver IC ULN2803 is used to deliver the required current to the 7 segment display to glow it prominently. ULN2803 comprises eight open collector darlington pair transistors inside it. Here each LED of the common anode 7 segment display is connected to the collector of each darlington pair via a current limiting resistance of 220 Ω. If the base of a darlington pair is made high, the darlington pair makes the collector shorted to ground. As a result a current will start to flow through the LED of a particular segment making that segment to be illuminated. Thus any decimal digit in the range of 0 to 9 can be shown on the 7 segment display. Now these three 7 segment displays are driven by a common 8-bit bus with the help of multiplexing using three 74LS373 latches. At any particular time only one latch will be enabled to store the bit pattern to display a digit on a 7 segment display. In this way all of the three 7 segment displays shows the digits one after another, which forms a complete 3 digits number finally. The circuit diagram of the 3 digit 7 segment display driver is shown in Fig-20.4.

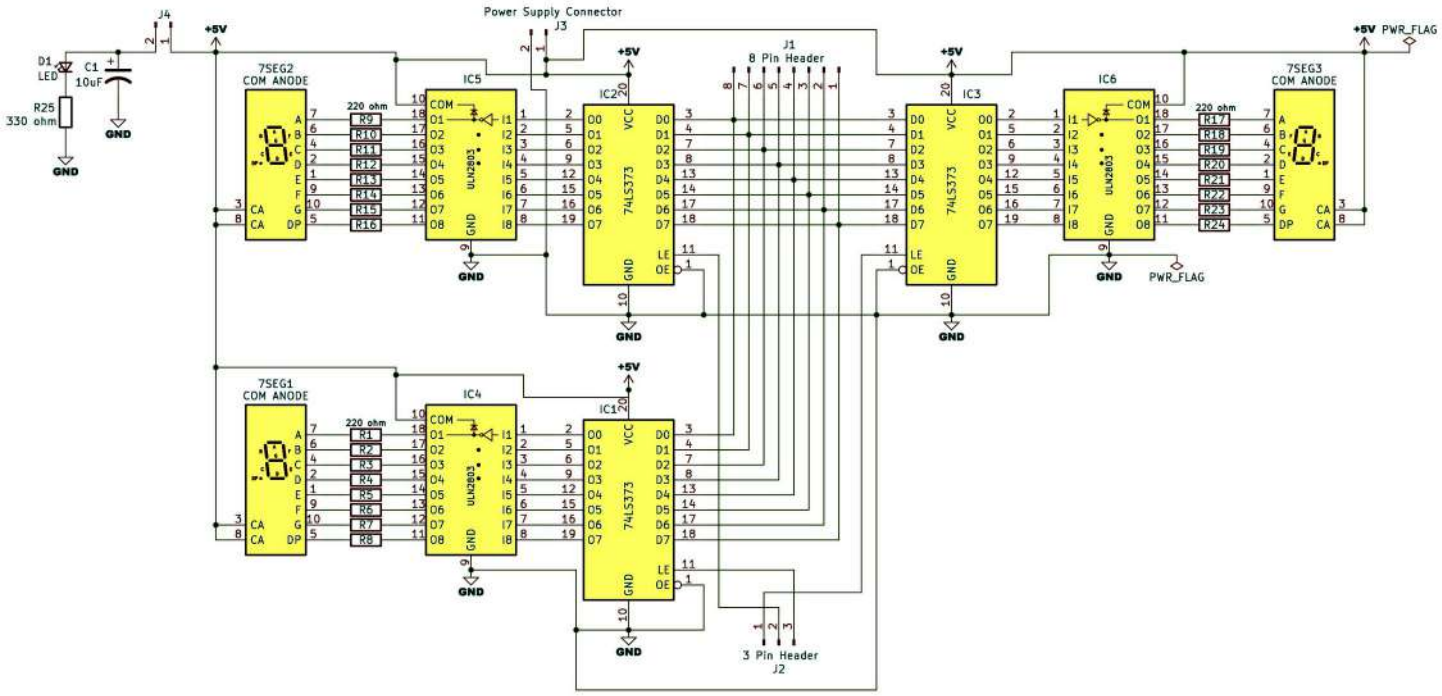


Fig-20.4: Circuit diagram of 3 digit 7 segment display driver

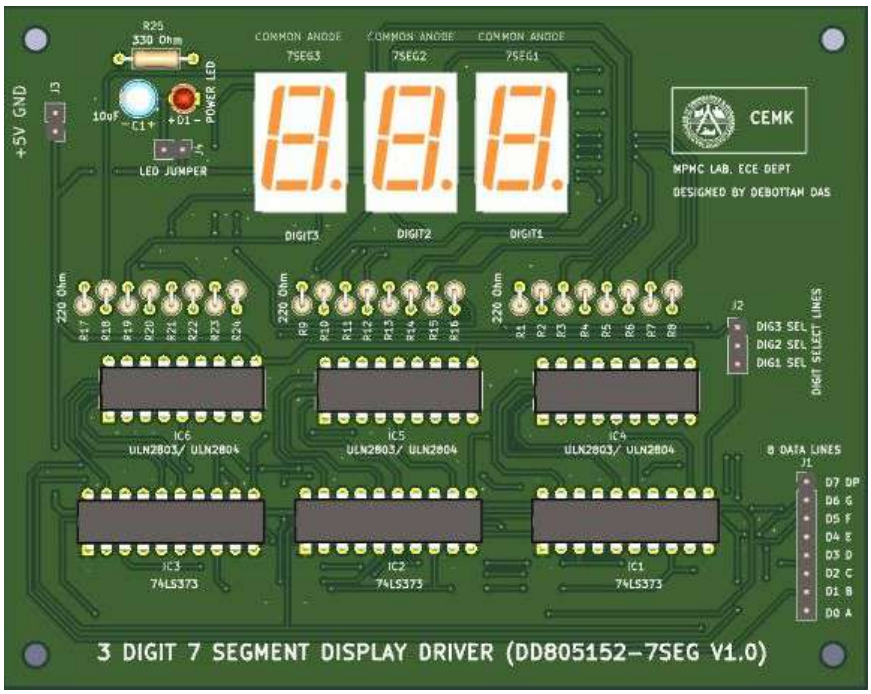


Fig-20.5: Front view of 3 digit 7 segment display driver board

Now this 7 segment display driver can be used for various purposes to implement different programs of 8051.

20.1: Write a program for 8051 microcontroller to implement a Mod-N counter with the help of 7 segment display, where the maximum value of N can be 256.

In this case a Mod-N counter is implemented using 8051 microcontroller and a 3 digit 7 segment display driver where the value of N may be up to 256. We know that a Mod-N counter has N no. of states starting from 0 to (N-1). Therefore the Mod-N counter starts counting from 0 and ends to (N-1) and returns to 0 again to repeat the same counting sequence. Here register R5 is used to count from 0 to (N-1). After every counting 3 digits (Digit1, Digit2 and Digit3) of counting value are extracted from the counting value (packed BCD) stored in the register R5 and these digits (unpacked BCD) are converted to its equivalent 7 segment codes using a lookup table. Finally these 7 segment codes are sent to the corresponding 7 segment displays one by one to glow the entire 3 digits for representing a counting value. The selection of these 7 segment displays is done using Latch Enable (LE) pin of 74LS373 IC which are connected to P3.0, P3.1 and P3.2 of Port3. Therefore it is clear that Port2 is connected to the common 8-bit data bus of the display driver and 3 pins of Port3 (P3.0, P3.1, P3.2) are connected to LE pins of three 74LS373 ICs. All the 7 segment displays used here is common anode type which are driven by another three driver ICs namely ULN2803. The circuit diagram of 3 digit 7 segment display driver board is shown in Fig.20.6 for visualization.

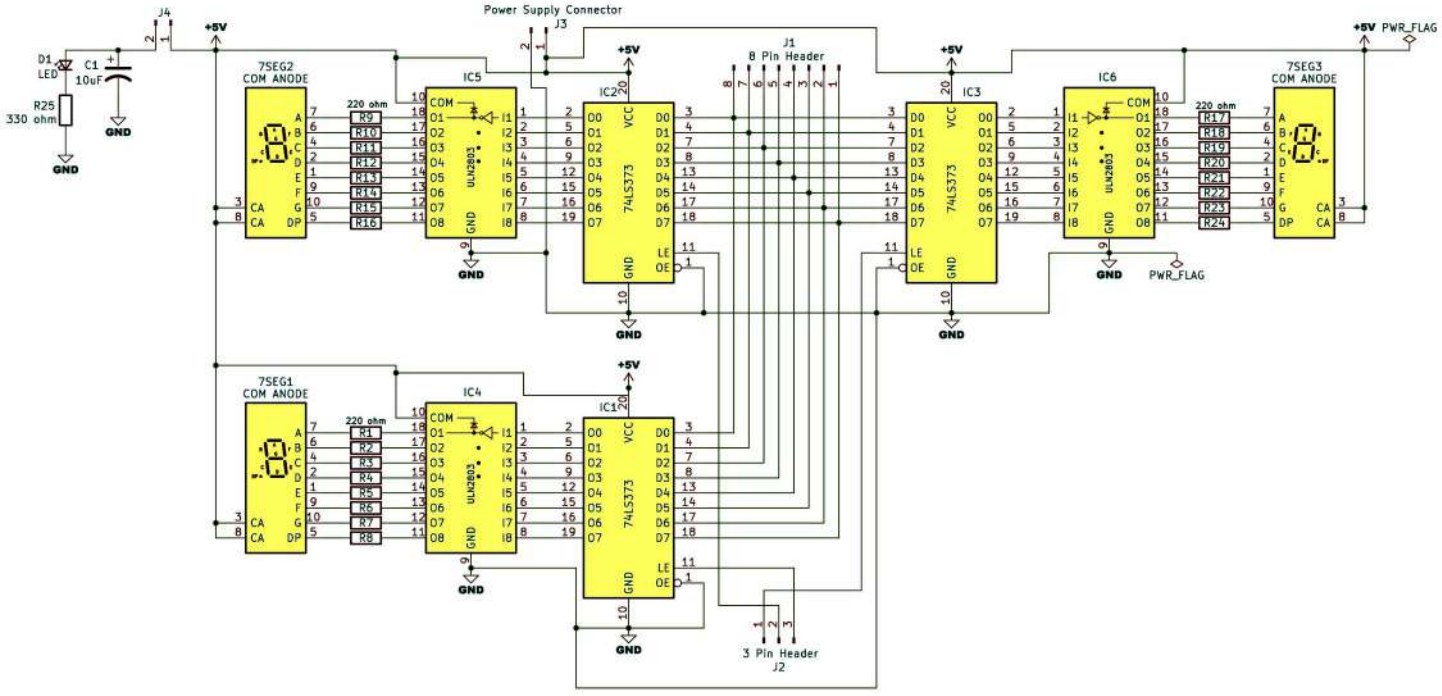


Fig-20.4: Circuit diagram of 3 digit 7 segment display driver

Now 7 segments (a, b, c, d, e, f, g) and DP of every 7 segment display are controlled by the pins of Port2 as follows.

P2.0 → a P2.1 → b P2.2 → c P2.3 → d

P2.4 → e P2.5 → f P2.6 → g P2.7 → DP

Depending upon the above configuration the following table gives the 7 segment codes corresponding each digits and these 7 segment codes are stored in a lookup table starting from the memory address 200 in the code memory of 8051.

P2.7	P2.6	P2.5	P2.4	P2.3	P2.2	P2.1	P2.0	Code in Hex	Memory Addresses to store the 7 segment codes	Displayed Single Digit Number
DP	g	f	e	d	c	b	a			
0	0	1	1	1	1	1	1	3F	200	0
0	0	0	0	0	1	1	0	06	201	1
0	1	0	1	1	0	1	1	5B	202	2
0	1	0	0	1	1	1	1	4F	203	3
0	1	1	0	0	1	1	0	66	204	4
0	1	1	0	1	1	0	1	6D	205	5
0	1	1	1	1	1	0	1	7D	206	6
0	0	0	0	0	1	1	1	07	207	7
0	1	1	1	1	1	1	1	7F	208	8
0	1	1	0	1	1	1	1	6F	209	9

If 254 is to be displayed on the 7 segment display driver, the 7 segment code of 4 (LSD) i.e. 66H will be sent to the data bus via Port2 first, then P3.0 is made high keeping P3.1 and P3.2 low to select Digit1 so that the 7 segment code 66H reaches only to Digit1 to glow 4 on it. Other two digits 5 and 2 (MSD) are also shown on Digit2 and Digit3 respectively maintaining the above mentioned sequences and enabling P3.1 and P3.2 respectively. Thus the entire counting value 254 will be shown on 3 7 segment displays finally. In this program a delay of 1 sec approximately is maintained between two consecutive counting values. The assembly language program for Mod-N counter is given below.

Assembly Language Program 20.1:

SL.	Label	Instructions of 8051
1		ORG 0000H
2		N EQU 256
3		T EQU 10
4		MOV P3, #00H
5		MOV DPTR, #200
6	START:	MOV R1, #N
7		MOV R5, #00H
8	BACK:	ACALL CONVT
9		ACALL DELAY
10		INC R5
11		DJNZ R1, BACK
12		SJMP START
13		
14	CONVT:	MOV B, #100
15		MOV A, R5
16		DIV AB
17		MOVC A, @A+DPTR
18		MOV P2, A
19		NOP
20		NOP
21		NOP
22		NOP
23		SETB P3.1
24		NOP
25		NOP
26		NOP
27		NOP
28		CLR P3.1

SL.	Label	Instructions of 8051
29		MOV A, B
30		MOV B, #10
31		DIV AB
32		MOVC A, @A+DPTR
33		MOV P2, A
34		NOP
35		NOP
36		NOP
37		NOP
38		SETB P3.2
39		NOP
40		NOP
41		NOP
42		NOP
43		CLR P3.2
44		MOV A, B
45		MOVC A, @A+DPTR
46		MOV P2, A
47		NOP
48		NOP
49		NOP
50		NOP
51		SETB P3.0
52		NOP
53		NOP
54		NOP
55		NOP
56		CLR P3.0
57		RET

SL.	Label	Instructions of 8051
58	DELAY:	MOV R2, #T
59	L2:	MOV R3, #255
60	L3:	MOV R4, #255
61	L4:	DJNZ R4, L4
62		DJNZ R3, L3
63		DJNZ R2, L2
64		RET
65		
66		ORG 200 *//Lookup table starts from 200 memory address*
67		DB 3FH, 06H, 5BH, 4FH, 66H, 6DH, 7DH, 07H, 7FH, 6FH
68		END

APPENDIX – A

INDEX SHEET

LABORATORY NAME: ……………………………………………………………………………..……..

NAME: ……………………………………… COLLEGE ROLL NO. ……………………

Index Sheet

SL	Title of the Experiment	Exp. No.	Date of Experiment	Date of Submission	Page No.	Grade	Teacher's Signature
						☐ E ☐ G ☐ F ☐ P ☐ NS	
						☐ E ☐ G ☐ F ☐ P ☐ NS	
						☐ E ☐ G ☐ F ☐ P ☐ NS	
						☐ E ☐ G ☐ F ☐ P ☐ NS	
						☐ E ☐ G ☐ F ☐ P ☐ NS	
						☐ E ☐ G ☐ F ☐ P ☐ NS	
						☐ E ☐ G ☐ F ☐ P ☐ NS	
						☐ E ☐ G ☐ F ☐ P ☐ NS	
						☐ E ☐ G ☐ F ☐ P ☐ NS	

** E: Excellent G: Good F: Fair P: Poor NS: Not Submitted

APPENDIX – B

CODING SHEET

EC493: Microprocessor & Microcontroller Laboratory	Exp. No.	Page No.

Title: **Date:**

Program Description:

Assembly Language Program		Machine Language Program				Comments
Label	Mnemonics	Memory Address	Hex Code			
			Opcode	Operand		

Student Roll No: **Teacher Signature:**

APPENDIX – C

SAMPLE LAB ASSESSMENT SHEET

Electronics and Communication Engineering Department
Continuous Lab assessment for 4ECE
Microprocessor and Microcontroller Lab [EC 493]

EN: ***Experiment No.***
FA: ***File Assessment***
PRFM:** ***Performance-
E = Excellent (10), G = Good (8),
F = Fair (6), P = Poor (4),
AB = Absent (4), NS = Not Submitted (2)

SL.	Roll No.	Name	UNIV Roll No.	Week-			Week-			Week-			Week-			Week-		
				EN	FA	*PRFM	EN	FA	*PRFM	EN	FA	*PRFM	EN	FA	*PRFM	EN	FA	*PRFM
1	ECE/24/001	DURBADAL ROUTH	10700324010															
2	ECE/24/002	ANIKET PANDA	10700324009															
3	ECE/24/003	SHUBHANKAR BOS	10700324008															
4	ECE/24/004	RATUL ROY	10700324007															
5	ECE/24/005	DEVDIPTA MONDA	10700324006															
6	ECE/24/006	SRIJIT DAS	10700324005															
7	ECE/24/007	RIYA HEMBRAM	10700324004															
8	ECE/24/008	SUSMITA SENA	10700324003															
9	ECE/24/009	SOUMEN MANNA	10700324014															
10	ECE/24/010	SWARNAVA DAS	10700324001															
11	ECE/24/011	SAYAN SAMANTA	10700324002															
12	ECE/24/012	ANKAN MAITI	10700324013															
13	ECE/24/013	ARUP MAITY	10700324012															
14	ECE/24/015	NIBEDITA DAS	10700324011															
		Name of the Teacher																
		Signature with date																

Exp. No.	Experiment Details	Performed in