Introduction:

What is a microprocessor?

It is a multifunctional Digital Circuit having both combinational and sequential components implemented in a single silicon wafer or chip. Unlike simple digital circuits designed and optimized to perform specific tasks, a microprocessor is a generalized circuit that performs a variety of tasks. The operation of the microprocessor is selected by external command called instruction. Instructions are internally decoded to generate control signals enabling hardware resources required to complete the tasks. Let us take an example to understand the multifunction of a circuit.

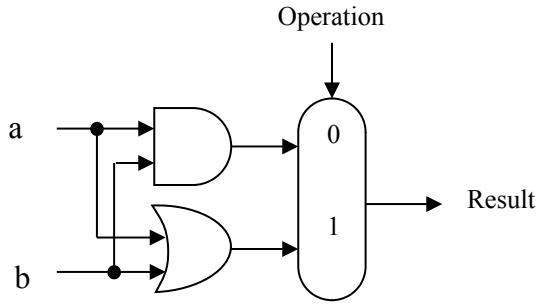


Fig.1: A circuit performing different operations on its inputs

The above is a combinational circuit that performs ANDing and ORing on its two inputs 'a' & 'b' depending upon the control input 'Operation'. For Operation = 0, Result = (a AND b) and for Operation = 1, Result = (a OR b). Further circuit blocks can be added for achieving more functions. Fig. 2 is such a circuit.

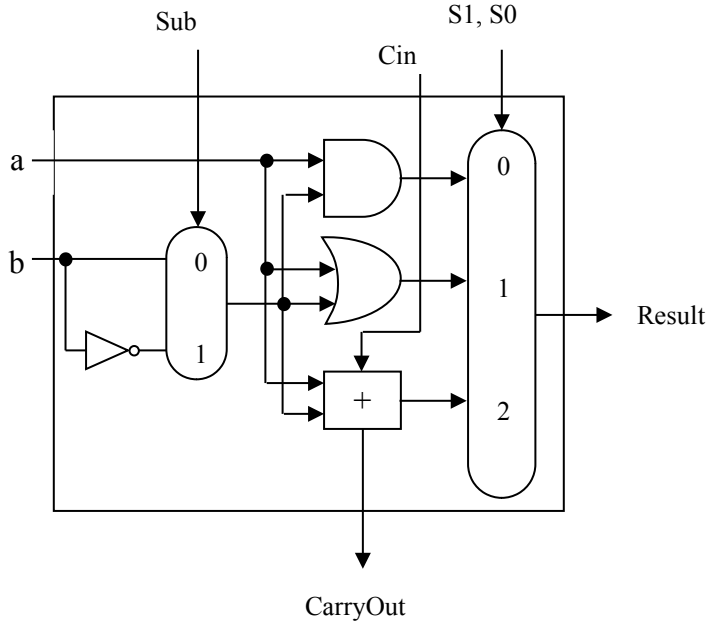


Fig.2 An one bit Arithmatic & Logic Unit

So the operations of the above circuit (can be called a simple microprocessor) can be summarized as:

Opeartion	Control Signals				Opcode
	SUB	Cin	S1	S0	
AND	0	X	0	0	0_H
OR	0	X	0	1	1_H
ADD	0	0	1	0	2_H
SUB	1	1	1	0	E_H

Instruction Set for the machine will be:

0000 a b
0001 a b
0010 a b
1110 a b

Since the microprocessor can perform only four operations, 2 bits are enough to represent opcodes. So an instruction decoder is to be used which takes 2-bit opcode and decodes to generate control signals required for the operations.

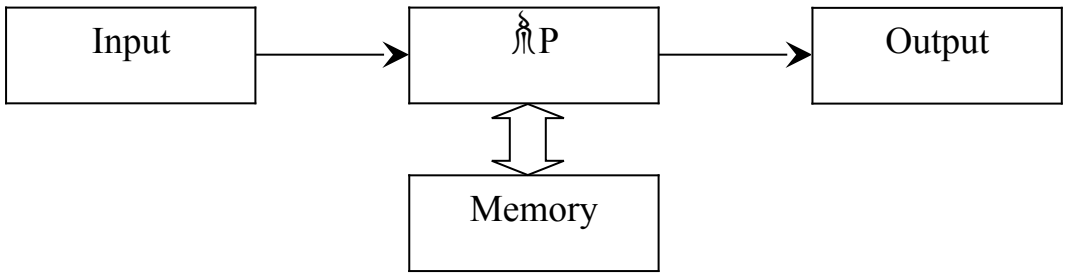
External memory is required for storing all the instructions of a task. Also the arrangement for reading instructions sequencilaly from the memory should be there inside the microprocessor for their execution.

So a microprocessor is:

- It is a digital circuit implemented in a single silicon wafer or chip
- It contains both combinational and sequential circuit components
- It is a general purpose device
- It can implement various types of functions
- Its operation can be controlled externally
- External commands are stored in memory
- Man machine interface through I/O ports

What does a microprocessor do?

- Any electronic system, whatever complex it may be, can be built around a microprocessor
- New feature can be implemented without changing the system hardware in most of the cases.
- Typical block diagram of a µP based system:



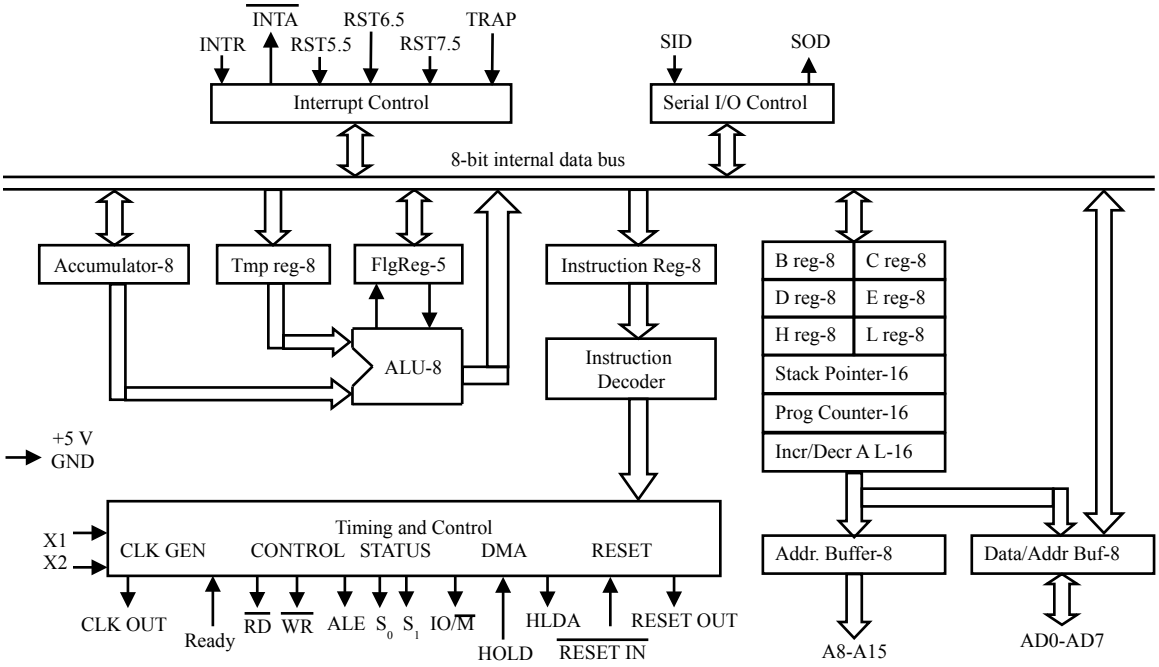
Basic three functions done by any μP:

- Data transfer
- Arithmetic & logic operations
- Decision making

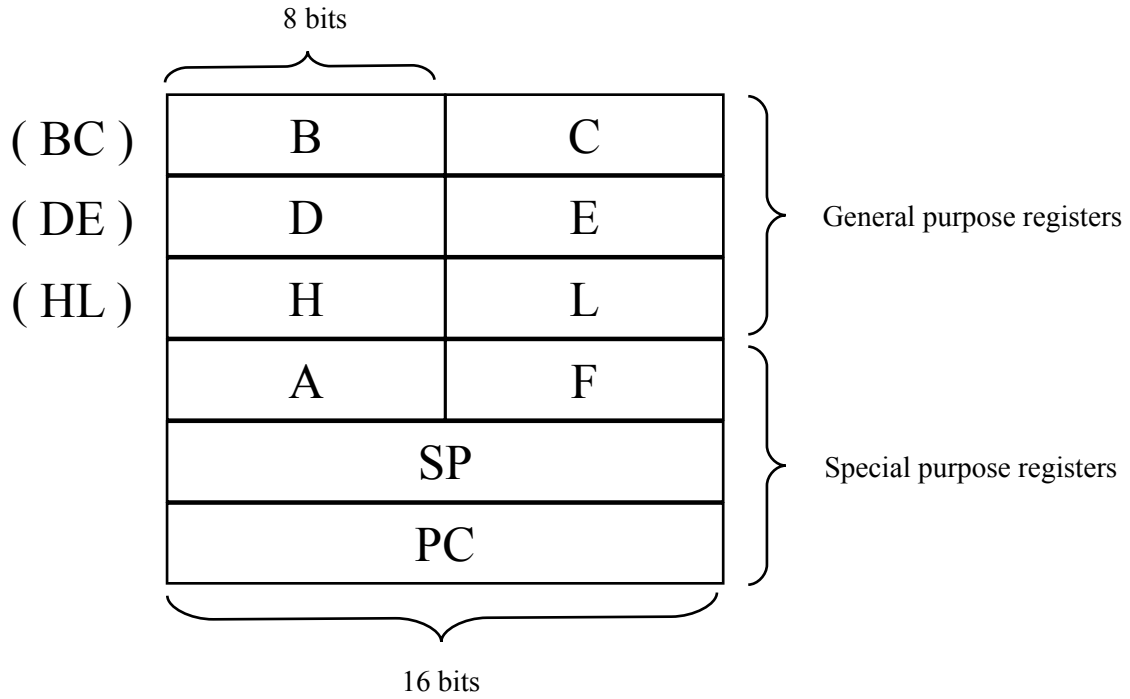
Classification of μP:

- Based on the width of data handled by the μP for data transfer, arithmetic and logic operations by a single command or instruction.
- For example: - 4-bit, 8-bit, 16-bit, 32-bit etc.
- Intel 4004 is a 4-bit microprocessor, Intel 8085 is an 8-bit μP, Intel 8088/86 is 16-bit μP etc.
- It should not be confused with the external data bus width. For example, though the external data bus width of 8088 is 8 bits, it is a 16-bit μP.

Internal Block Diagram of 8085A:



Programming Model of 8085A:



General Purpose Registers:

- B, C, D, E, H, L
- They can be used in any manner by the programmer, for house keeping, memory addressing, arithmetic operation.
- Flexible to use as six 8-bit registers or as three 16-bit register pairs
- Valid register pairs are BC, DE, HL

Special Purpose Registers:

- A, F, SP, PC
- They are used for accumulating results from arithmetic and logic instructions and also for housekeeping.
- AF register pair is known as Program status Word (PSW) as they contain the status of the result after execution of any instruction.

Flag Register:

- Contains 5 flag bits
- Most of the arithmetic and logic instructions modify them
- They reflect the condition of the outcome of the answer from ALU
- They are used for decision making

F7	F6	F5	F4	F3	F2	F1	F0
S	Z	x	AC	x	P	x	CY

S = 1, if -ve; Z=1, if zero; AC=1, if aux carry; P=1 for even parity (even no of 1's); CY=1, if carry/borrow

Example:

	1 81H +7FH	0 7FH - 81H	0 55H OR AAH
Result with Carry	1 | 00H	1 | FEH	FFH
Flags after the operation	S=0, Z=1, AC=1, P=1, CY=1	S=1, Z=0, AC=0, P=0, CY=1	S=1, Z=0, AC=0, P=1, CY=0

Problem:

What will be contents of the flags after executing the following operations?
XRA, A; ORA, A; SUB A; CMP A

Data word formats:

1. Unsigned Integers:

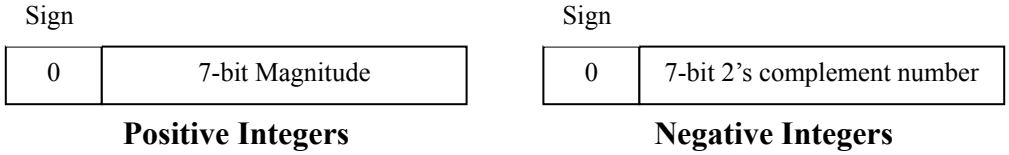
- They are 8 bits, 16 bits or any multiple of 8 bits in width
- 8-bit unsigned integers can be found in 8-bit registers
- They can also be stored in single memory locations
- 16-bit unsigned integers are found in register pairs and in two consecutive memory locations (lower byte in low numbered memory and higher byte in high numbered memory location)
- Binary weights of 8-bit unsigned integer are:

D7	D6	D5	D4	D3	D2	D1	D0
(128)	(64)	(32)	(16)	(8)	(4)	(2)	(1)

- Examples: $1000\ 1001_2$ (89_H) = 128 + 8 + 1 = 137_{10}
- Same logic may be extended for 16-bit or higher order integers

2. Signed Integers:

- Single-byte signed integers are 7-bit numbers plus a sign bit
- Left most bit is the sign bit, 0 for +ve & 1 for –ve
- Signed 8-bit integer formats are (in 2's complement):



- +ve integers range from 0 to 127
- -ve integers range from –1 to –128 (in 2's complement format)
- Binary weights of the bit positions for –ve numbers are:

D7	D6	D5	D4	D3	D2	D1	D0
(-128)	(+64)	(+32)	(+16)	(+8)	(+4)	(+2)	(+1)

- Examples: $0111\ 1011_2$ ($7B_H$) = 64+32+16+8+2+1 = 123_{10}
- $1111\ 1011_2$ (FB_H) = -128+32+16+8+2+1 = -5_{10}
- Same logic may be extended for 16-bit or higher integers.

3. ASCII data format:

- ASCII is the acronym for American Standard Code for Information Interchange
- It is used by all manufacturer of computer peripherals
- ASCII is a 7-bit code, the 8^{th} is used to hold the parity bit in a data communications system
- In computer systems this bit is often a logic 0.
- In some printers a 0 in the 8^{th} bit causes it to print ASCII characters and a 1 to print graphics characters

4. BCD data formats:

- BCD is the acronym for Binary Coded Decimal
- BCD is used in I/O devices for human to understand
- Expressed in two ways – Packed and Unpacked BCD
- Packed BCD is stored as two digits per byte
- Examples: 79_{10} in packed BCD is 0111 1001
- Unpacked BCD is stored as single digit per byte
- Example: 7_{10} in unpacked BCD is 0000 0111

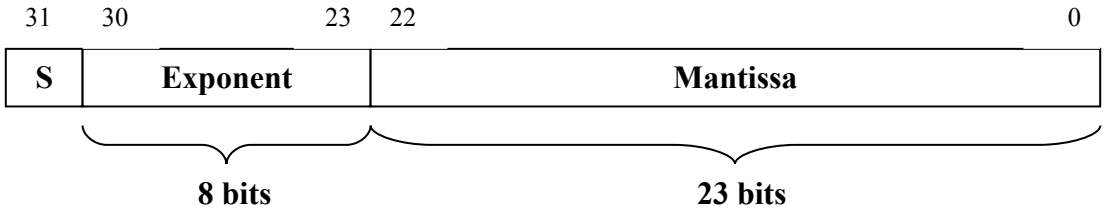
- Unpacked BCD codes are useful to refer look up table for code conversion etc.
- Microprocessor can also perform BCD operations but not preferred to avoid complication

5. Binary Fractions:

- Binary fractions can be stored in either byte or two-byte form
- Usually they are expressed in unsigned numbers
- Binary weights from left to right are: 2^{-1}, 2^{-2} , 2^{-3} , 2^{-4} , 2^{-5} and so on
- Example: 1010 0101 = $2^{-1} + 2^{-3} + 2^{-6} + 2^{-8}$ = 0.5+0.125+0.015625+0.00390625 = 0.64453125

6. Floating Point data:

- It is similar to scientific notation in base 10
- The floating point format is suitable to express large numbers
- It is used to store mixed as well as integer data
- Floating point numbers are often stored in four bytes
- Format of 4-byte (single precision) floating point number is:



- The left most bit indicates the sign of the mantissa
- Next 8 bits are for exponent stored in excess 127 notation
- Exponent in excess 127 is an unsigned integer that is equal to the actual exponent plus 127
- Mantissa is a normalized 23-bit number with a hidden or implied 1 in 24th bit position
- Examples:

$100_{10} = 1100100_2 = 1.1001 \times 2^6$

S	Exponent	Mantissa
0	1000 0101	10010000000000000000000

$-12.70_{10} = -1100.11_2 = 1.10011 \times 2^3$

S	Exponent	Mantissa
1	1000 0010	10011000000000000000000

Sample Questions and answers

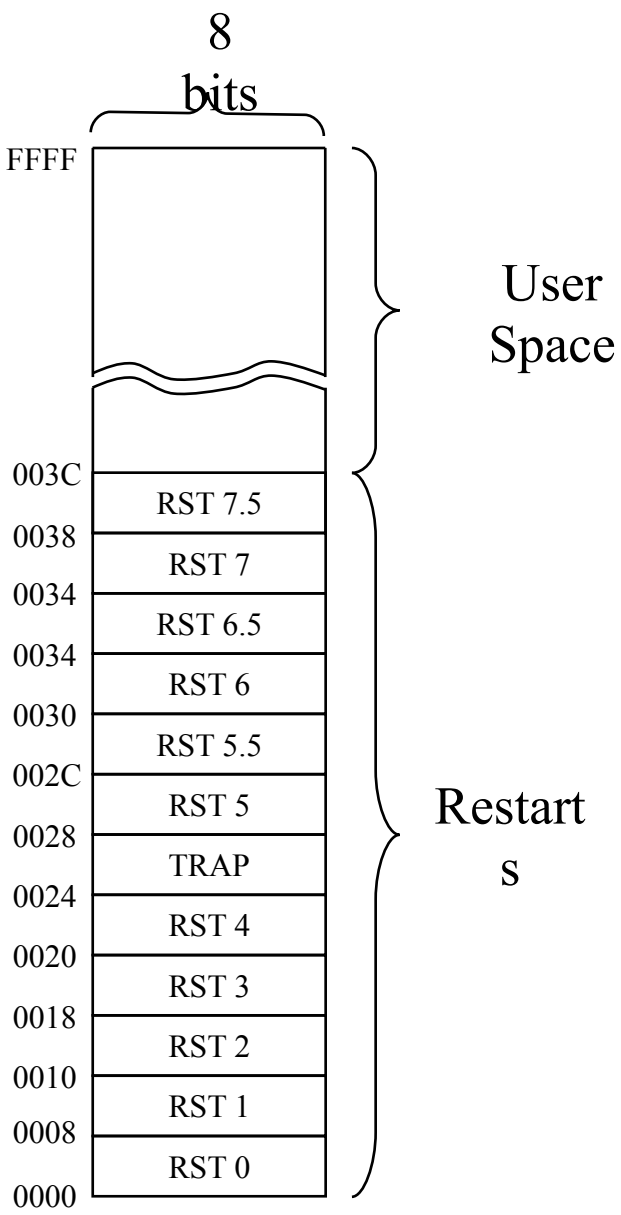
Q1	Convert the following decimal numbers to 8-bit unsigned integers: 12, 33, 55, 100, 155, 196 and 212
Q2	Convert the following decimal numbers to 16-bit unsigned integers: 156, 222, 1000, 2009, and 10,000
Q3	Convert the following 8-bit signed integers to decimal numbers: 1111 1111, 1000 0111, 0110 1000, and 0111 0000.
Q4	Convert the following decimal numbers to 8-bit signed integers: 12, -12, 32, -63, and –100
Q5	Write the following decimal numbers as both packed and unpacked BCD numbers: 12, 3, 10, 99, 13, and 712
Q6	Convert the following decimal numbers to four-byte binary floating point form: 12, -22, 10.5, 0.002, and -4.25.
Q7	Convert the following 4-byte floating point numbers into decimal numbers: 0100 0001 0100 0000 0000 0000 0000 0000 1011 1111 1000 0000 0000 0000 0000 0000 0100 1000 1110 0000 0000 0000 0000 0000

Answers:

Q1.	Decimal Number	Corresponding 8-bit Unsigned Binary Number	Corresponding Hexadecimal Number
	12	0000 1100	0C
	33	0010 0001	21
	55	0011 0111	37
	100	0110 0100	64
	155	1001 1011	9B
	196	1100 0100	C4
	212	1101 0100	D4
Q2.	Decimal Number	Corresponding 16-bit Unsigned Binary Number	Corresponding Hexadecimal Number
	156	0000 0000 1001 1100	009C
	222	0000 0000 1101 1110	00DE
	1000	0000 0011 1110 1000	03E8
	2009	0000 0111 1101 1001	07D9
	10000	0010 0111 0001 0000	2710
Q3.	8-bit Signed Integers	2's Complement if -ve	Corresponding Decimal Number
	1111 1111	0000 0001	-1
	1000 0111	0111 1001	-121
	0110 1000	-	+104
	0111 0000	-	+112
Q4.	Decimal Number	-/+ & 8-bit magnitude	8-bit Binary in 2's Complement
	12	+ 0000 1100	0000 1100
	-12	- 0000 1100	1111 0100
	32	+ 0010 0000	0010 0000
	-63	- 0011 1111	1100 0001
	-100	- 0110 0100	1001 1100
Q5.	Decimal Number	Packed BCD (in HEX)	Unpacked BCD (in HEX)
	12	0001 0010 (12H)	0000 0010 (02H) 0000 0001 (01H)
	3	0000 0011 (03H)	0000 0011 (03H)
	10	0001 0000 (10H)	0000 0000 (00H) 0000 0001 (01H)
	99	1001 1001 (99H)	0000 1001 (09H)

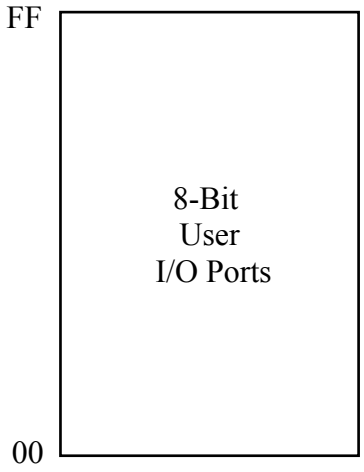
			0000 1001 (09H)
	13	0001 0011 (13H)	0000 0011 (03H) 0000 0001 (01H)
	712	0000 0111 0001 0010 (0712H)	0000 0010 (02H) 0000 0001 (01H) 0000 0111 (07H)
Q6.	Decimal Number	Binary with fraction and sign	4-byte floating point binary
	12	+1100.0	0 1000 0010 100……0000
	-22	-1 0110.0	1 1000 0011 01100……0000
	10.5	+1010.1	0 1000 0010 010100……0000
	0.002	+0.0000 0000 1	0 0111 0110 000000……0000
	-4.25	-0100.01	1 1000 0001 000100……0000
Q7.	4-byte floating point number	Binary with fraction and sign	Decimal number
	0 1000 0010 100……000	$+0.1 \times 2^{130-127}$	4
	1 0111 1111 000……000	$-0.0 \times 2^{127-127}$	0
	0 1001 0001 110……000	$+0.11 \times 2^{145-127}$	3x64x1024 = 196608

Memory Map of 8085A:



- It has 16-bit address lines and 8-bit data lines.
- It can therefore address 2^{16} or 2^6 x 2^{10} or 64K byte memory locations.
- Memory locations are numbered from 0000H to FFFFH.

I/O Space of 8085A:



- 8085A can address 256 input devices.
- It can also address 256 output devices.
- They are numbered from 00H to FFH.
- Input and Output Devices are identified by RD and WR signals respectively
- I/O devices are accessed by IN/OUT instructions.

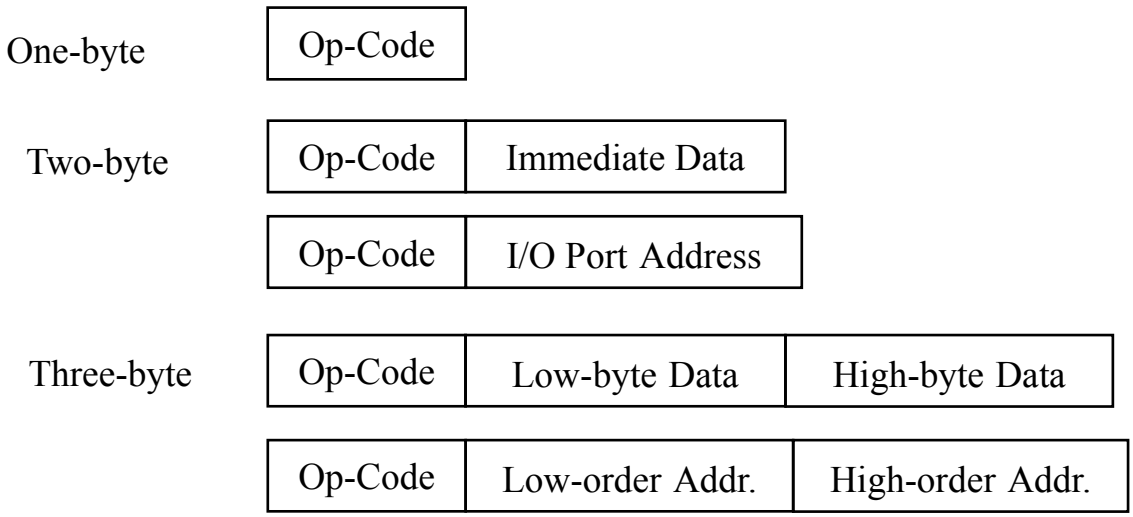
Instruction Set and Programming The 8085

Command Word Formats:
Three different Command Word Formats are used by 8085A

- One byte long - 204
- Two-byte long - 18
- Three-byte long - 24

Total - 246

- Command Word (Instruction) Formats are:



Instruction Set of 8085A:
It can be broken into three main categories:

1. Data Transfer Instructions,
2. Arithmetic & Logic Instructions and
3. Program Control Instructions.

Rules of writing instructions:
[Label:] op-code [des op], [source op] [; Comments]

Fields within 3rd brackets are optional.

Addressing Modes: The way the operands (internal/external resources like registers/meory/IO) are represented in an instruction is called addressing mode. Addressing modes may be of the following types:

- Register addressing
- Immediate addressing
- Direct addressing and
- Register indirect addressing

Register addressing
The instruction specifies the registers (B, C, D, E, H, L or A) or the register pairs (BC, DE, HL or SP) used with the instruction. These are all **one byte** instructions.
Examples:
MOV B, A; ADD D etc.

Immediate addressing
This mode of addressing is used when constant data are used in a program. The data are placed immediately following the op-code and stored in the program memory. 8085A has two forms immediate addressing: 8-bit and 16-bit immediate addressing.
Instruction format:

byte1 byte2 byte3

Op-code	8-bit Immediate Data

Op-code	Low byte of 16-bit Data	High byte of 16-bit Data

Examples:
MVI A, 12H; ADI 34H etc.

Direct Addressing
Instructions that directly address the memory always include the memory address of the data. This address is stored following the op-code in the program. Instruction format is:

byte1 byte2 byte3

Op-code	Low order address	High order address

Examples:
LDA 1234H; STA 9876H etc.

Register Indirect Addressing:
In some instructions register pairs BC, DE and HL are used to indicate the memory location containing the operand. This type of addressing the memory indirectly by the memory pointers is called register indirect addressing. These are all one byte instructions.

Examples:
LDAX B (or D); STAX B (or D) etc.

Instruction Naming Conventions:
The mnemonics assigned to the instructions are designed to indicate the function of the instruction. Instructions fall into the following functional categories:

SL	Mnemonic	Purpose	Syntax	Note
Data Transfer Instructions The data transfer instructions move data between registers or between memory and register.				
1	MVI	Move Immediate data	MVI rd, d8	rd may be one of 7 8-bit registers and M. M is the memory pointed to by HL register pair.
2	MOV	Move data between 8-bit register to register and between register to memory pointed to by HL.	MOV rd, rs	rd and rs may be any combination of 7 8-bit registers and M except M, M.
3	LDA	Load Accumulator Directly from Memory	LDA a16	A = [a16], a16 is 16-bit address of a memory location
4	STA	Store Accumulator Directly in Memory	STA a16	[a16] = A
5	LHLD	Load H & L Registers Directly from Memory	LHLD a16	L = [a16], H = [a16+1]
6	SHLD	Store H & L Registers Directly in Memory	SHLD a16	[a16] = L, [a16+1] = H
7	LXI	Load Register Pair with 16-bit Immediate data	LXI rp	rp may be one of HL, BC, DE, SP.
8	LDAX	Load Accumulator from Address in Register Pair	LDAX rp	rp may be one of BC and DE
9	STAX	Store Accumulator in Address in Register Pair	STAX rp	rp may be one of BC and DE
10	XCHG	Exchange H & L with D & E	XCHG	
11	XTHL	Exchange Top of Stack with H & L	XTHL	
Arithmetic Group The arithmetic instructions are used to perform arithmetic operations				
12	ADI	Add Immediate Data to Accumulator	ADI d8	d8 is the 8-bit immediate data
13	ADD	Add to Accumulator	ADD r	r may be one of 7 8-bit registers and M
14	ADC	Add to Accumulator Using Carry Flag	ADC r	r may be one of 7 8-bit registers and M
15	ACI	Add Immediate data to Accumulator Using Carry	ACI r	r may be one of 7 8-bit registers and M
16	SUI	Subtract Immediate Data from Accumulator	SUI d8	d8 is the 8-bit immediate data
17	SUB	Subtract from Accumulator	SUB r	r may be one of 7 8-bit registers and M
18	SBI	Subtract Immediate from	SBI d8	d8 is the 8-bit immediate data

		Accumulator Using Borrow (Carry) Flag		
19	SBB	Subtract from Accumulator Using Borrow (Carry) Flag	SBB r	r may be one of 7 8-bit registers and M
20	INR	Increment Specified Byte by One	INR r	r may be one of 7 8-bit registers and M
21	DCR	Decrement Specified Byte by One	DCR r	r may be one of 7 8-bit registers and M
22	INX	Increment Register Pair by One	INX rp	rp may be one of BC, DE, HL, SP
23	DCX	Decrement Register Pair by One	DCX rp	rp may be one of BC, DE, HL, SP
24	DAD	Double Register Add; Add Content of Register Pair to H & L Register Pair	DAD rp	rp may be one of BC, DE, HL, SP
25	DAA	Decimal adjust accumulator after BCD addition	DAA	
Logical Group This group performs logical (Boolean) operations on data in registers and memory and on condition flags. The logical AND, OR, and Exclusive OR instructions enable you to set specific bits in the accumulator ON or OFF.				
26	ANI	Logical AND with Accumulator Using Immediate Data	ANI d8	d8 is the 8-bit immediate data
27	ANA	Logical AND with Accumulator	ANA r	r may be one of 7 8-bit registers and M
28	ORA	Logical OR with Accumulator	ORA r	r may be one of 7 8-bit registers and M
29	ORI	Logical OR with Accumulator Using Immediate Data	ORI d8	d8 is the 8-bit immediate data
30	XRI	Exclusive OR Using Immediate Data	SRI d8	d8 is the 8-bit immediate data
31	XRA	Exclusive Logical OR with Accumulator	XRA r	r may be one of 7 8-bit registers and M
32	CPI	Compare Using Immediate Data	CPI d8	d8 is the 8-bit immediate data
33	CMP	Compare	CMP r	r may be one of 7 8-bit registers and M
34	RLC	Rotate Accumulator Left	RLC	Rotation by one bit position
35	RRC	Rotate Accumulator Right	RRC	Rotation by one bit position
36	RAL	Rotate Left Through Carry	RAL	Rotation by one bit position
37	RAR	Rotate Right Through Carry	RAR	Rotation by one bit position
38	CMA	Complement Accumulator	CMA	
39	CMC	Complement Carry Flag	CMC	
40	STC	Set Carry Flag	STC	

Branch Group:

The branching instructions alter normal sequence of program flow, either unconditionally or conditionally. The unconditional branching instructions are as follows:

JMP Jump
CALL Call
RET Return

Conditional branching instructions examine the status of one of four condition flags to determine whether the specified branch is to be executed. The conditions that may be specified are as follows:

NZ Not Zero (Z = 0)
Z Zero (Z = 1)
NC No Carry (C = 0)
C Carry (C = 1)
PO Parity Odd (P = 0)
PE Parity Even (P = 1)
P Plus (S = 0)
M Minus (S = 1)

Thus, the conditional branching instructions are specified as follows:

Jumps	Calls	Returns	
JC	CC	RC	(Carry)
JNC	CNC	RNC	(No Carry)
JZ	CZ	RZ	(Zero)
JNZ	CNZ	RNZ	(Not Zero)
JP	CP	RP	(Plus)
JM	CM	RM	(Minus)
JPE	CPE	RPE	(Parity Even)
JPO	CPO	RPO	(Parity Odd)

Two other instructions can affect a branch by replacing the contents or the program counter:

PCHL Move H & L to Program Counter
RST Special Restart Instruction Used with Interrupts

Stack, I/O and Machine Control Instructions:
The following instructions affect the Stack and/or Stack Pointer:

PUSH Push Two bytes of Data onto the Stack
POP Pop Two Bytes of Data off the Stack
XTHL Exchange Top of Stack with H & L
SPHL Move content of H & L to Stack Pointer

The I/0 instructions are as follows:

IN Initiate Input Operation
OUT Initiate Output Operation

The Machine Control instructions are as follows:
EI Enable Interrupt System
DI Disable Interrupt System
HLT Halt
NOP No Operation

8085A Instruction Set:

SL	Mnemonic	Op-code	Bytes	Clock	Function	Flag				
						Z	C	A	S	P
Immediate data transfer instructions:										
1	MVI B, d8	06-d8	2	7	B = d8	-	-	-	-	-
2	MVI C, d8	0E-d8	2	7	C = d8	-	-	-	-	-
3	MVI D, d8	16-d8	2	7	D = d8	-	-	-	-	-
4	MVI E, d8	1E-d8	2	7	E = d8	-	-	-	-	-
5	MVI H, d8	26-d8	2	7	H = d8	-	-	-	-	-
6	MVI L, d8	2E-d8	2	7	L = d8	-	-	-	-	-
7	MVI M, d8	36-d8	2	10	M = d8	-	-	-	-	-
8	MVI A, d8	3E-d8	2	7	A = d8	-	-	-	-	-
9	LXI B, d16	01-ll-hh	3	10	BC = d16	-	-	-	-	-
10	LXI D, d16	11-ll-hh	3	10	DE = d16	-	-	-	-	-
11	LXI H, d16	21-ll-hh	3	10	HL = d16	-	-	-	-	-
12	LXI SP, d16	31-ll-hh	3	10	SP = d16	-	-	-	-	-
Direct Data Transfer										
13	LDA a16	3A-ll-hh	3	13	A = [a16]	-	-	-	-	-
14	STA a16	32-ll-hh	3	13	[a16] = A	-	-	-	-	-
15	LHLD a16	2A-ll-hh	3	16	HL = [a16]	-	-	-	-	-
16	SHLD a16	22-ll-hh	3	16	[a16] = HL	-	-	-	-	-
Indirect Data Transfer Instructions										
17	LDAX B	0A	1	7	A = [BC]	-	-	-	-	-
18	LDAX D	1A	1	7	A = [DE]	-	-	-	-	-
19	STAX B	02	1	7	[BC] = A	-	-	-	-	-
20	STAX D	12	1	7	[DE] = A	-	-	-	-	-
Register Data Transfer Instructions										
21	MOV B, B	40	1	4	B = B	-	-	-	-	-
22	MOV B, C	41	1	4	B = C	-	-	-	-	-
23	MOV B, D	42	1	4	B = D	-	-	-	-	-
24	MOV B, E	43	1	4	B = E	-	-	-	-	-
25	MOV B, H	44	1	4	B = H	-	-	-	-	-
26	MOV B, L	45	1	4	B = L	-	-	-	-	-
27	MOV B, M	46	1	7	B = M	-	-	-	-	-

SL	Mnemonic	Op-code	Bytes	Clock	Function	Flag				
						Z	C	A	S	P
28	MOV B, A	47	1	4	B = A	-	-	-	-	-
29	MOV C, B	48	1	4	C = B	-	-	-	-	-
30	MOV C, C	49	1	4	C = C	-	-	-	-	-
31	MOV C, D	4A	1	4	C = D	-	-	-	-	-
32	MOV C, E	4B	1	4	C = E	-	-	-	-	-
33	MOV C, H	4C	1	4	C = H	-	-	-	-	-
34	MOV C, L	4D	1	4	C = L	-	-	-	-	-
35	MOV C, M	4E	1	7	C = M	-	-	-	-	-
36	MOV C, A	4F	1	4	C = A	-	-	-	-	-
37	MOV D, B	50	1	4	D = B	-	-	-	-	-
38	MOV D, C	51	1	4	D = C	-	-	-	-	-
39	MOV D, D	52	1	4	D = D	-	-	-	-	-
40	MOV D, E	53	1	4	D = E	-	-	-	-	-
41	MOV D, H	54	1	4	D = H	-	-	-	-	-
42	MOV D, L	55	1	4	D = L	-	-	-	-	-
43	MOV D, M	56	1	7	D = M	-	-	-	-	-
44	MOV D, A	57	1	4	D = A	-	-	-	-	-
45	MOV E, B	58	1	4	E = B	-	-	-	-	-
46	MOV E, C	59	1	4	E = C	-	-	-	-	-
47	MOV E, D	5A	1	4	E = D	-	-	-	-	-
48	MOV E, E	5B	1	4	E = E	-	-	-	-	-
49	MOV E, H	5C	1	4	E = H	-	-	-	-	-
50	MOV E, L	5D	1	4	E = L	-	-	-	-	-
51	MOV E, M	5E	1	7	E = M	-	-	-	-	-
52	MOV E, A	5F	1	4	E = A	-	-	-	-	-
53	MOV H, B	60	1	4	H = B	-	-	-	-	-
54	MOV H, C	61	1	4	H = C	-	-	-	-	-
55	MOV H, D	62	1	4	H = D	-	-	-	-	-
56	MOV H, E	63	1	4	H = E	-	-	-	-	-
57	MOV H, H	64	1	4	H = H	-	-	-	-	-
58	MOV H, L	65	1	4	H = L	-	-	-	-	-
59	MOV H, M	66	1	7	H = M	-	-	-	-	-
60	MOV H, A	67	1	4	H = A	-	-	-	-	-
61	MOV L, B	68	1	4	L = B	-	-	-	-	-
62	MOV L, C	69	1	4	L = C	-	-	-	-	-
63	MOV L, D	6A	1	4	L = D	-	-	-	-	-
64	MOV L, E	6B	1	4	L = E	-	-	-	-	-
65	MOV L, H	6C	1	4	L = H	-	-	-	-	-
66	MOV L, L	6D	1	4	L = L	-	-	-	-	-
67	MOV L, M	6E	1	7	L = M	-	-	-	-	-
68	MOV L, A	6F	1	4	L = A	-	-	-	-	-
69	MOV M, B	70	1	7	M = B	-	-	-	-	-
70	MOV M, C	71	1	7	M = C	-	-	-	-	-
71	MOV M, D	72	1	7	M = D	-	-	-	-	-
72	MOV M, E	73	1	7	M = E	-	-	-	-	-
73	MOV M, H	74	1	7	M = H	-	-	-	-	-
74	MOV M, L	75	1	7	M = L	-	-	-	-	-

SL	Mnemonic	Op-code	Bytes	Clock	Function	Flag				
						Z	C	A	S	P
-	MOV M, M	-		-						
75	MOV M, A	77	1	7	M = A	-	-	-	-	-
76	MOV A, B	78	1	4	A = B	-	-	-	-	-
77	MOV A, C	79	1	4	A = C	-	-	-	-	-
78	MOV A, D	7A	1	4	A = D	-	-	-	-	-
79	MOV A, E	7B	1	4	A = E	-	-	-	-	-
80	MOV A, H	7C	1	4	A = H	-	-	-	-	-
81	MOV A, L	7D	1	4	A = L	-	-	-	-	-
82	MOV A, M	7E	1	7	A = M	-	-	-	-	-
83	MOV A, A	7F	1	4	A = A	-	-	-	-	-
Stack Data Transfer Instructions										
84	POP B	C1	1	10	C = [SP], B = [SP+1]	-	-	-	-	-
85	POP D	D1	1	10	E = [SP], D = [SP+1]	-	-	-	-	-
86	POP H	E1	1	10	L = [SP], H = [SP+1]	-	-	-	-	-
87	POP PSW	F1	1	10	A = [SP], F = [SP+1]	*	*	*	*	*
88	PUSH B	C5	1	10	[SP-1] = B, [SP-2] = C	-	-	-	-	-
89	PUSH D	D5	1	10	[SP-1] = D, [SP-2] = E	-	-	-	-	-
90	PUSH H	E5	1	10	[SP-1] = H, [SP-2] = L	-	-	-	-	-
91	PUSH PSW	F5	1	10	[SP-1] = F, [SP-2] = A	-	-	-	-	-
92	XTHL	E3	1	16	HL ↔ stack data	-	-	-	-	-
Miscellaneous Data Transfer Instructions										
93	IN p8	DB-p8	2	10	Inputs data to A	-	-	-	-	-
94	OUT p8	D3-p8	2	10	Outputs data from A	-	-	-	-	-
95	SPHL	F9	1	6	SP = HL	-	-	-	-	-
96	XCHG	EB	1	4	HL ↔ DE	-	-	-	-	-
Arithmetic and Logic Instructions										
8-bit Binary Addition										
97	ADI d8	C6-d8	2	7	A = A + d8	*	*	*	*	*
98	ADD B	80	1	4	A = A + B	*	*	*	*	*
99	ADD C	81	1	4	A = A + C	*	*	*	*	*
100	ADD D	82	1	4	A = A + D	*	*	*	*	*
101	ADD E	83	1	4	A = A + E	*	*	*	*	*
102	ADD H	84	1	4	A = A + H	*	*	*	*	*
103	ADD L	85	1	4	A = A + L	*	*	*	*	*
104	ADD M	86	1	7	A = A + M	*	*	*	*	*
105	ADD A	87	1	4	A = A + A	*	*	*	*	*
Addition with Carry										
106	ACI d8	CE-d8	2	7	A = A + d8 + CY	*	*	*	*	*
107	ADC B	88	1	4	A = A + B + CY	*	*	*	*	*
108	ADC C	89	1	4	A = A + C + CY	*	*	*	*	*
109	ADC D	8A	1	4	A = A + D + CY	*	*	*	*	*
110	ADC E	8B	1	4	A = A + E + CY	*	*	*	*	*
111	ADC H	8C	1	4	A = A + H + CY	*	*	*	*	*
112	ADC L	8D	1	4	A = A + L + CY	*	*	*	*	*
113	ADC M	8E	1	7	A = A + M + CY	*	*	*	*	*
114	ADC A	8F	1	4	A = A + A + CY	*	*	*	*	*
16-bit Addition										

SL	Mnemonic	Op-code	Bytes	Clock	Function	Flag				
						Z	C	A	S	P
115	DAD B	09	1	10	HL = HL + BC	-	*	-	-	-
116	DAD D	19	1	10	HL = HL + DE	-	*	-	-	-
117	DAD H	29	1	10	HL = HL + HL	-	*	-	-	-
118	DAD SP	39	1	10	HL = HL + SP	-	*	-	-	-
BCD Addition										
119	DAA	27	1	4	Decimal adjust accumulator after BCD addition	*	*	*	*	*
Increment										
120	INR B	04	1	4	B = B + 1	*	-	*	*	*
121	INR C	0C	1	4	C = C + 1	*	-	*	*	*
122	INR D	14	1	4	D = D + 1	*	-	*	*	*
123	INR E	1C	1	4	E = E + 1	*	-	*	*	*
124	INR H	24	1	4	H = H + 1	*	-	*	*	*
125	INR L	2C	1	4	L = L + 1	*	-	*	*	*
126	INR M	34	1	10	M = M + 1	*	-	*	*	*
127	INR A	3C	1	4	A = A + 1	*	-	*	*	*
128	INX B	03	1	6	BC = BC + 1	-	-	-	-	-
129	INX D	13	1	6	DE = DE + 1	-	-	-	-	-
130	INX H	23	1	6	HL = HL + 1	-	-	-	-	-
131	INX SP	33	1	6	SP = SP + 1	-	-	-	-	-
Subtraction										
8-bit Subtraction										
132	SUI d8	D6-d8	1	7	A = A – d8	*	*	*	*	*
133	SUB B	90	1	4	A = A – B	*	*	*	*	*
134	SUB C	91	1	4	A = A – C	*	*	*	*	*
135	SUB D	92	1	4	A = A – D	*	*	*	*	*
136	SUB E	93	1	4	A = A – E	*	*	*	*	*
137	SUB H	94	1	4	A = A – H	*	*	*	*	*
138	SUB L	95	1	4	A = A – L	*	*	*	*	*
139	SUB M	96	1	7	A = A – M	*	*	*	*	*
140	SUB A	97	1	4	A = A – A	1	0	0	0	1
Subtract with Borrow										
141	SBI d8	DE-d8	2	7	A = A – d8 – CY	*	*	*	*	*
142	SBB B	98	1	4	A = A – B – CY	*	*	*	*	*
143	SBB C	99	1	4	A = A – C – CY	*	*	*	*	*
144	SBB D	9A	1	4	A = A – D – CY	*	*	*	*	*
145	SBB E	9B	1	4	A = A – E – CY	*	*	*	*	*
146	SBB H	9C	1	4	A = A – H – CY	*	*	*	*	*
147	SBB L	9D	1	4	A = A – L – CY	*	*	*	*	*
148	SBB M	9E	1	7	A = A – M – CY	*	*	*	*	*
149	SBB A	9F	1	4	A = A – A – CY	*	*	*	*	*
Decrement										
150	DCR B	05	1	4	B = B – 1	*	-	*	*	*
151	DCR C	0D	1	4	C = C – 1	*	-	*	*	*
152	DCR D	15	1	4	D = D – 1	*	-	*	*	*
153	DCR E	1D	1	4	E = E – 1	*	-	*	*	*

SL	Mnemonic	Op-code	Bytes	Clock	Function	Flag				
						Z	C	A	S	P
154	DCR H	25	1	4	H = H – 1	*	-	*	*	*
155	DCR L	2D	1	4	L = L – 1	*	-	*	*	*
156	DCR M	35	1	10	M = M – 1	*	-	*	*	*
157	DCR A	3D	1	4	A = A – 1	*	-	*	*	*
158	DCX B	0B	1	6	BC = BC – 1	-	-	-	-	-
159	DCX D	1B	1	6	DE = DE – 1	-	-	-	-	-
160	DCX H	2B	1	6	HL = HL – 1	-	-	-	-	-
161	DCX SP	33	1	6	SP = SP – 1	-	-	-	-	-
Compares										
162	CPI d8	FE-d8	2	7	Flags = A – d8	*	*	*	*	*
163	CMP B	B8	1	4	Flags = A – B	*	*	*	*	*
164	CMP C	B9	1	4	Flags = A – C	*	*	*	*	*
165	CMP D	BA	1	4	Flags = A – D	*	*	*	*	*
166	CMP E	BB	1	4	Flags = A – E	*	*	*	*	*
167	CMP H	BC	1	4	Flags = A – H	*	*	*	*	*
168	CMP L	BD	1	4	Flags = A – L	*	*	*	*	*
169	CMP M	BE	1	7	Flags = A – M	*	*	*	*	*
170	CMP A	BF	1	4	Flags = A – A	1	0	0	0	1
Logic Instructions										
Inversion										
171	CMA	2F	1	4	A = $\bar{A}$	-	-	-	-	-
The AND Operation										
172	ANI d8	E6-d8	2	7	A = A * d8	*	0	0	*	*
173	ANA B	A0	1	4	A = A * B	*	0	*	*	*
174	ANA C	A1	1	4	A = A * C	*	0	*	*	*
175	ANA D	A2	1	4	A = A * D	*	0	*	*	*
176	ANA E	A3	1	4	A = A * E	*	0	*	*	*
177	ANA H	A4	1	4	A = A * H	*	0	*	*	*
178	ANA L	A5	1	4	A = A * L	*	0	*	*	*
179	ANA M	A6	1	7	A = A * M	*	0	*	*	*
180	ANA A	A7	1	4	A = A * A	*	0	*	*	*
The OR Operation										
181	ORI d8	F6-d8	2	7	A = A ˅ d8	*	0	0	*	*
182	ORA B	B0	1	4	A = A ˅ B	*	0	0	*	*
183	ORA C	B1	1	4	A = A ˅ C	*	0	0	*	*
184	ORA D	B2	1	4	A = A ˅ D	*	0	0	*	*
185	ORA E	B3	1	4	A = A ˅ E	*	0	0	*	*
186	ORA H	B4	1	4	A = A ˅ H	*	0	0	*	*
187	ORA L	B5	1	4	A = A ˅ L	*	0	0	*	*
188	ORA M	B6	1	7	A = A ˅ M	*	0	0	*	*
189	ORA A	B7	1	4	A = A ˅ A	*	0	0	*	*
The Exclusive-OR Operation										
190	XRI d8	EE-d8	2	7	A = A XOR d8	*	0	0	*	*
191	XRA B	A8	1	4	A = A XOR B	*	0	0	*	*
192	XRA C	A9	1	4	A = A XOR C	*	0	0	*	*
193	XRA D	AA	1	4	A = A XOR D	*	0	0	*	*
194	XRA E	AB	1	4	A = A XOR E	*	0	0	*	*

SL	Mnemonic	Op-code	Bytes	Clock	Function	Flag				
						Z	C	A	S	P
195	XRA H	AC	1	4	A = A XOR H	*	0	0	*	*
196	XRA L	AD	1	4	A = A XOR L	*	0	0	*	*
197	XRA M	AE	1	7	A = A XOR M	*	0	0	*	*
198	XRA A	AF	1	4	A = A XOR A	1	0	0	0	1
Rotate Instructions										
199	RLC	07	1	4	Rotate A left	-	*	-	-	-
200	RRC	0F	1	4	Rotate A right	-	*	-	-	-
201	RAL	17	1	4	Rotate A left thru Cary	-	*	-	-	-
202	RAR	1F	1	4	Rotate A right thru Carry	-	*	-	-	-
Program Control Instructions										
Unconditional Jump Instructions										
203	JMP a16	C3-ll-hh	3	10	Program continues at a16	-	-	-	-	-
204	PCHL	E9	1	6	Program continues at address HL	-	-	-	-	-
Conditional Jump Instructions										
205	JZ a16	CA-ll-hh	3	7/10	Jump if zero	-	-	-	-	-
206	JNZ a16	C2-ll-hh	3	7/10	Jump if not zero	-	-	-	-	-
207	JC a16	DA-ll-hh	3	7/10	Jump if carry set	-	-	-	-	-
208	JNC a16	D2-ll-hh	3	7/10	Jump if carry cleared	-	-	-	-	-
209	JM a16	FA-ll-hh	3	7/10	Jump if minus	-	-	-	-	-
210	JP a16	F2-ll-hh	3	7/10	Jump if positive	-	-	-	-	-
211	JPE a16	EA-ll-hh	3	7/10	Jump if parity even	-	-	-	-	-
212	JPO a16	E2-ll-hh	3	7/10	Jump if parity odd	-	-	-	-	-
Linking to a Subroutine										
213	CALL a16	CE-ll-hh	3	18	Call subroutine at a16	-	-	-	-	-
214	CC a16	DC-ll-hh	3	9/18	Call subroutine on carry	-	-	-	-	-
215	CNC a16	D4-ll-hh	3	9/18	Call subroutine on no carry	-	-	-	-	-
216	CZ a16	CC-ll-hh	3	9/18	Call subroutine on zero	-	-	-	-	-
217	CNZ a16	C4-ll-hh	3	9/18	Call subroutine on not zero	-	-	-	-	-
218	CM a16	FC-ll-hh	3	9/18	Call subroutine on minus	-	-	-	-	-
219	CP a16	F4-ll-hh	3	9/18	Call subroutine on positive	-	-	-	-	-
220	CPE a16	EC-ll-hh	3	9/18	Call sub on parity even	-	-	-	-	-
221	CPO a16	E4-ll-hh	3	9/18	Call sub on parity odd	-	-	-	-	-
Returning from a subroutine										
222	RET	C9	1	10	Return from subroutine	-	-	-	-	-
223	RC	D8	1	6/12	Return if carry set	-	-	-	-	-
224	RNC	D0	1	6/12	Return if carry cleared	-	-	-	-	-
225	RZ	C8	1	6/12	Return if zero	-	-	-	-	-
226	RNZ	C0	1	6/12	Return if not zero	-	-	-	-	-
227	RM	F8	1	6/12	Return if minus	-	-	-	-	-
228	RP	F0	1	6/12	Return if positive	-	-	-	-	-
229	RPE	E8	1	6/12	Return if parity even	-	-	-	-	-
230	RPO	E0	1	6/12	Return if parity odd	-	-	-	-	-
Restart Instructions										
231	RST 0	C7	1	12	Call subroutine at 0000H	-	-	-	-	-
232	RST 1	CF	1	12	Call subroutine at 0008H	-	-	-	-	-
233	RST 2	D7	1	12	Call subroutine at 0010H	-	-	-	-	-

SL	Mnemonic	Op-code	Bytes	Clock	Function	Flag				
						Z	C	A	S	P
234	RST 3	DF	1	12	Call subroutine at 0018H	-	-	-	-	-
235	RST 4	E7	1	12	Call subroutine at 0020H	-	-	-	-	-
236	RST 5	EF	1	12	Call subroutine at 0028H	-	-	-	-	-
237	RST 6	F7	1	12	Call subroutine at 0030H	-	-	-	-	-
238	RST 7	FF	1	12	Call subroutine at 0038H	-	-	-	-	-
Miscellaneous Instructions										
Microprocessor Control Instructions										
239	NOP	00	1	4	Performs no operation	-	-	-	-	-
240	STC	37	1	4	Set carry flag	-	1	-	-	-
241	CMC	3F	1	4	Complement carry flag	-	*	-	-	-
242	HLT	76	1	4	Halt until reset or interrupt	-	-	-	-	-
243	EI	FB	1	4	Enable interrupts	-	-	-	-	-
244	DI	F3	1	4	Disable interrupts	-	-	-	-	-
245	RIM	20	1	4	Read interrupt mask	-	-	-	-	-
246	SIM	30	1	4	Set interrupt mask	-	-	-	-	-

Note: -, no change; *, changes;

Sample Questions and Solutions:

1. Write a sequence of immediate instructions that will place a 0000 in BC and a 12H in Accumulator?
2. Write a sequence of immediate instructions that will store 16H in memory location 1200H and a 17H in memory location 1202H?
3. Explain how does the LDA 1000H instruction function?
4. Explain what answer is found in memory locations 1200H and 1201H after the execution of the following instructions

```
MVI H, 22H
MVI L, 44H
SHLD 1200H
```

5. Explain what answer is found in memory locations 1200H in the following sequence of instructions.

```
MVI B, 12H
MVI C, 00H
MVI A, 77H
STAX B
```

6. Write a sequence of instructions that will use register indirect addressing to transfer the number stored in memory location 1300H into memory location 1301H.
7. Explain what does the MOV M, C instruction do if HL=1234H and C=34H.
8. Write a sequence of instructions that use MOV instructions to swap the contents of the BC to DE register pairs.
9. Write a sequence of instructions that will store a zero in memory location 1000H through 10003H.
10. If a 1000H is pushed into the stack followed by a 2000H, which number is the first to come off the stack?
11. What number appears in BC register pair after the following sequence of instructions.

```
LXI H, 3000H
LXI D, 2500H
PUSH H
PUSH D
POP H
POP B
```

12. If a PUSH PSW is immediately followed by a POP B, in which register do the flag data appear?
13. Write a sequence of instructions that will add a 56H to the number in the B register.
14. Write a sequence of instructions that will add the content of H to that of L register.
15. Write a sequence of instructions that will add the content of HL to that of DE register pair (not using DAD D)
16. Write a sequence of instructions that will add the BCD numbers placed in B and L registers.
17. Write e sequence of instructions that will one's complement the contents of DE register pair.

Solutions:

1. Write a sequence of immediate instructions that will place a 0000 in BC and a 12H in Accumulator?

LXI B, 0000H		*MVI B, 00H*
	or	*MVI C, 00H*
MVI A, 12H		*MVI A, 12H*

2. Write a sequence of immediate instructions that will store 16H in memory location 1200H and a 17H in memory location 1202H?

```
LXI H, 1200H
MVI M, 16H
INX H
INX H
MVI M, 17H
```

3. Explain how does the LDA 1000H instruction function?

After the execution of the instruction, the content of memory having address 1000H is loaded into accumulator.

4. Explain what answer is found in memory locations 1200H and 1201H after the execution of the following instructions

```
MVI H, 22H
MVI L, 44H
SHLD 1200H
```

After the execution of the first two instructions, the content of HL register pair will be 2214H. SHLD instruction is a direct mode of instruction after execution of which the content of L and H registers will be stored in memory location1200H and 1201H respectively.

5. Explain what answer is found in memory locations 1200H in the following sequence of instructions.

```
MVI B, 12H
MVI C, 00H
MVI A, 77H
STAX B
```

After the execution of first two instructions, BC register pair will initialized with 1200H. The third instruction stores 77H in accumulator. The final instruction is a register indirect mode of instruction which stores the content of accumulator in memory location pointed out by BC register pair. Therefore, the accumulator value (77H) will be stored in memory location 1200H.

6. Write a sequence of instructions that will use register indirect addressing to transfer the number stored in memory location 1300H into memory location 1301H.

```
LXI B, 1300H
LDAX B
INX B
STAX B
```

7. Explain what does the MOV M, C instruction do if HL=1234H and C=34H.

   The instruction MOV M, C stores the content of register C in memory location pointed out by HL register pair. In the present case, C=34H and HL=1234H, thus the data 34H will be stored in memory location 1234H.

8. Write a sequence of instructions that use MOV instructions to swap the contents of the BC to DE register pairs.

```
MOV H, B        ; content of BC is temporarily stored in HL
MOV L, C

MOV B, D        ; content of DE is stored in BC
MOV C, E

MOV D, H        ; finally the content of BC kept aside in HL is stored in DE
MOV E, L
```

9. Write a sequence of instructions that will store a zero in memory location 1000H through 10003H.

```
LXI H, 1000H    ; HL =1000H
MOV A, L
MOV M, A
INX H           ; HL =1001H
MOV M, A
INX H           ; HL =1002H
MOV M, A
INX H
MOV M, A        ; HL = 1003H
HLT
```

   This is an example of iteration i.e. repetitive work. The above piece of program stores values in four memory locations by using four separate instructions which is not efficient in case of large number of iteration. Large iterations are implemented by repeating a group of instructions in a number of times called looping and is demonstrated below with the same example. A counter is used to control looping:

```
        LXI H, 1000H    ; memory pointer
        MVI C, 04H      ; four memory locations to be loaded
LOOP:   MOV M, A        ; data stored in memory
        INX H           ; pointer updated for next memory
```

```
DCR C          ; counter decremented
JNZ LOOP       ; the process is repeated until counter is zero
HLT
```

10. If a 1000H is pushed into the stack followed by a 2000H, which number is the first to come off the stack?

    In the present case, 1000H is PUSHed first and then 2000H. As stack is a LIFO memory, the last data PUSHed (2000H) in stack will be retrieved first.

11. What number appears in BC register pair after the following sequence of instructions.

```
LXI H, 3000H
LXI D, 2500H
PUSH H
PUSH D
POP H
POP B
```

    Content of HL is pushed in stack first and then the content of DE. Poping of HL is done first and then BC. Therefore, POP H stores the value of DE (as pushed last) in HL (2500H) while POP B retrieves the original value of HL in BC which is 3000H.

12. If a PUSH PSW is immediately followed by a POP B, in which register do the flag data appear?

    PSW, the program status word, is the combination of Accumulator (msb) and flag register (lsb). Thus execution of POP B immediately after PUSH PSW will retrieve the value of Accumulator plus flag register in BC register pair. The value of accumulator will appear in B and that of flag register in C.

13. Write a sequence of instructions that will add a 56H to the number in the B register.

```
MVI A, 56H
ADD B
```

14. Write a sequence of instructions that will add the content of H to that of L register.

```
MOV A, H
ADD L
```

15. Write a sequence of instructions that will add the content of HL to that of DE register pair (not using DAD D)

```
MOV A, E
ADD L
MOV L, A       ; low byte of result is placed in L
MOV A, D
ADC H
MOV H, A       ; high byte  result is placed in H and carry, if any, is available in CY flag
```

16. Write a sequence of instructions that will add the BCD numbers placed in B and L registers.

```
MOV A, B
ADD L
DAA             ; result is in Accumulator and carry, if any, is available in CY flag.
```

17. Write a sequence of instructions that will do one's complement to the contents of DE register pair.

```
MOV A, E
CMA
MOV E, A
MOV A, D
CMA
MOV D, A
```

Assembly Language Programming

Lecture-10

Assembly Language Programming:

- The sequence of commands used to tell a microprocessor what to do is called a program. The commands are called instructions.
- Some part of it is called Monitor Program or Operating System and the other is called User Program or Application Program
- Operating Program basically organizes the inputs and the outputs with the system
- User Program supplies the variables and their formats
- Microprocessor can only understand the instructions coded in binary called machine language.
- Machine language is difficult, if not impossible, for human to handle.
- Assembly language was developed to provide *mnemonics* plus other features to make the programming easy, faster and less prone to error.
- Instructions abbreviated in English letters to represent the operation to be performed by the microprocessor are called *mnemonics*.
- Assembly language programs must be translated into machine code by a program called *assembler*.
- Assembly language is referred to as a low-level language because it deals directly with the internal structure of the microprocessor.
- High level language like C, BASIC, Java etc. can also be used for programming.
- High level language is converted into machine code by a program called *compiler*.

Structure of Assembly Language:

- An assembly language program consists of a number of assembly language instructions used to tell the CPU what to do.
- It also contains instructions giving direction to the assembler called *directives*.
- For example, MOV, ADA instructions are commands to the CPU whereas ORG, END are directives to the assembler. ORG followed by an address tells the assembler to place the op-code at that memory location while END indicates the end of the source code.
- An assembly language instruction consists of five fields:

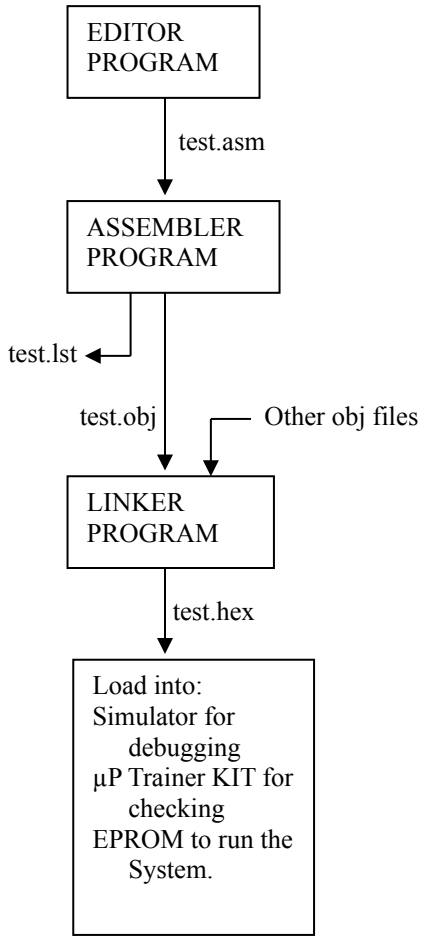
  [label:] mnemonic [destination operand] [source operand] [;comment]

- Brackets indicate that a field is optional and not all lines have them. Brackets should not be typed in.
- The label field allows the program to refer to a line of code by name. There are rules for writing the label like type & maximum number of characters, starting characters etc. which is assembler specific.
- The assembly language mnemonic together with the operands forms the command for the CPU. For example MOV A, B. MOV is the mnemonic, which is the abbreviation of data movement. The operands are supplied by A and B registers. The data from source register B moves to destination register A.
- The comment field begins with a semicolon. Comments may be at the end of a line or on a line itself. The assembler ignores comments, but they should be present to make the program understandable to others and at a later time.

Assembling and running of an 8085A program:

- A machine can only understand machine language. So assembly language program is to be translated back into machine language. Human for convenience uses assembly language.
- Assembly language program is time consuming and difficult, if not impossible, to translate into machine language manually.
- A PC based program called an assembler can do the same instantaneously. It takes an assembly language program as input, and produces an object file having extension .obj for machine codes.
- The assembly language file can be written in any EDITOR program like DOS EDIT, WINDOWS NOTEPAD etc. which saves the file in ASCII format having extension .asm.

- Program can also be written in high level languages like 'C', BASIC, PASCAL. An interpreter program or a compiler is used to translate the same into machine codes.
- All the object files created by assembler or compiler can then be combined together to form a single machine language program by another program called linker.
- The total flow diagram of machine language program development is depicted below.
- An Assembler produces a listing file having extension .lst containing the original assembly language instructions and the corresponding binary codes. It also reports for any error encountered during conversion.



Steps to create a program

- Programs can be developed faster in high level language than assembly language as the high-level language uses higher building blocks. However a program written in high level language usually occupies more space in memory and takes more time to execute than that developed by assembly language.
- Programs that involve a lot of hardware-control are normally written in assembly language.

Program Development Steps:

Defining the problem

The first step in writing a program is to write down the operations to be done by the program and the order of executing them. An example of a simple problem may be

1. Read temperature from thermocouple sensor
2. Read ambient temperature from an ambient sensor
3. Add correction for ambient temperature
4. Save result in memory

For a program as simple as this the four actions desired are very close to the assembly statements. However, for more complex problem we need to develop more extensive outline of the problem so that the actions can be replaced by assembly language statements.

Representing Program Operations

The formula or sequence of operations used to solve a problem is called ***algorithm***. An algorithm can be written using graphic shapes called flowcharts. Algorithm can also be written by pseudocodes using standard program structures.

FLOWCHARTS

Different graphic shapes are used to represent different types of operations. The figure below shows some of the commonly used graphic shapes:

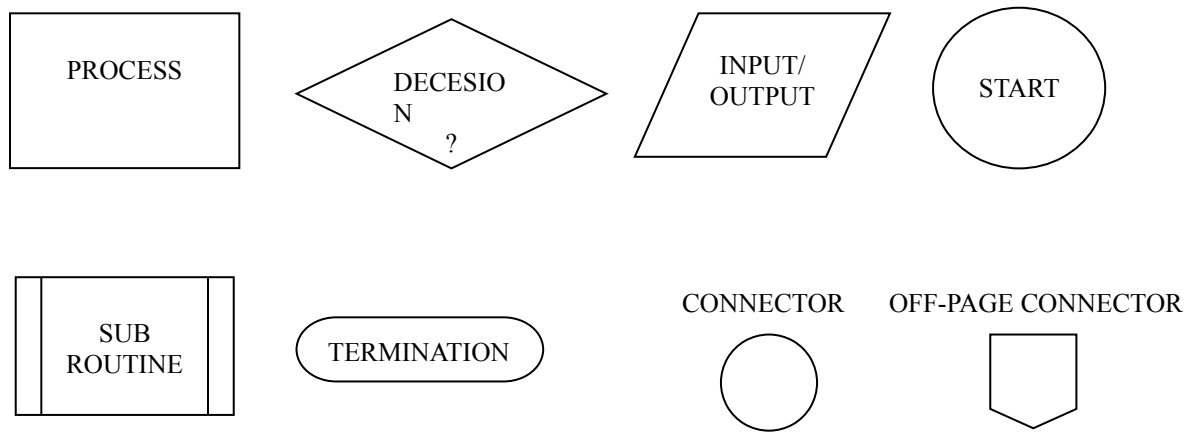
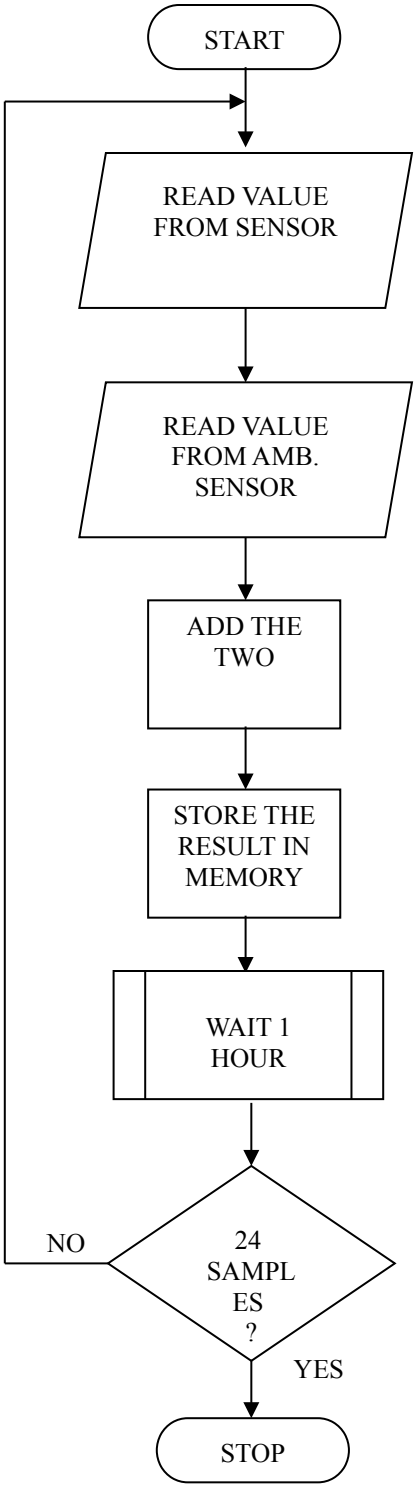


Figure: Flowchart symbols

Figure below shows a flowchart for a program to read 24 data samples from a thermocouple sensor at an interval of 1 hour.



PSEUDOCODES

- Flowchart symbols are space consuming and are normally not used for large programs. Instead English like statements called pseudo codes are used to represent the algorithm of the program.
- Three basic operations viz. *Sequence, Decision, and Iteration* can represent the operations of any desired problem.
- Sequence represents a series of actions
- Decision means choosing between two alternative actions
- Repetition means repeating a series of actions for a number of time
- Three to seven standard structures can represent all the operations in a typical program
- These standard structures are:

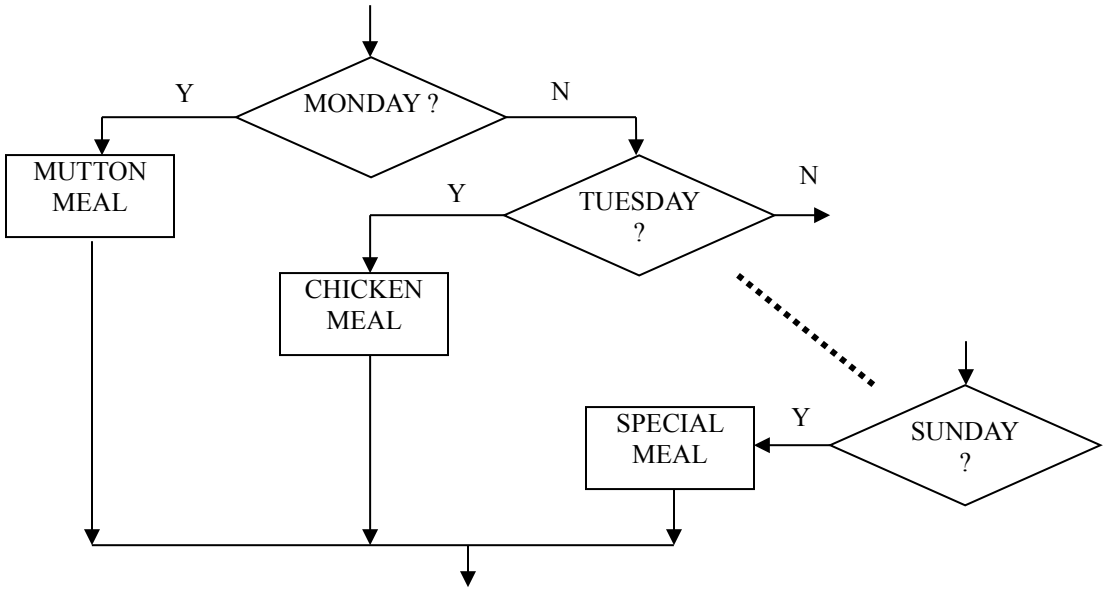
1. SIMPLE SEQUENCE
2. IF-THEN-ELSE
3. IF-THEN
4. CASE expressed as nested IF-THEN-ELSE
5. CASE
6. WHILE-DO LOOP
7. REPEAT UNTIL

Example of different cooking in different days of the week in the students' Hostel using Flow Chart and Pseudo Codes:

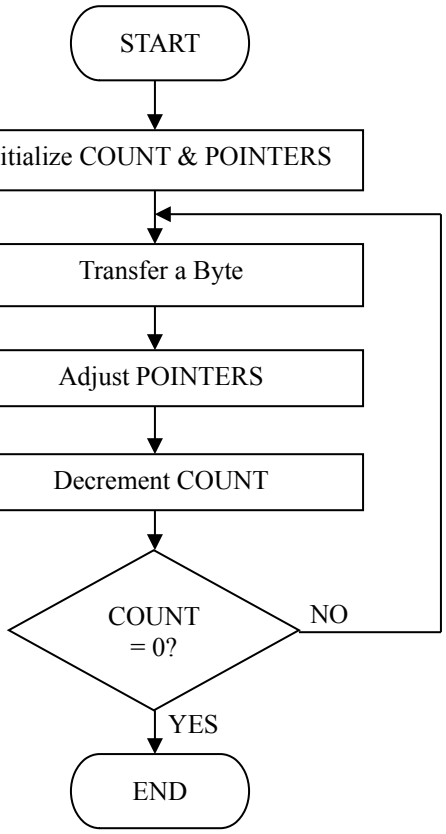
Pseudo Codes

```
IF MONDAY THEN
        MAKE MUTTON MEAL
ELSE IF TUESDAY THEN
        MAKE VEG MEAL
ELSE IF WEDNESDAY THEN
        MAKE CHICKEN MEAL
                :
                :
                :
ELSE IF SUNDAY THEN
        MAKE SPECIAL MEAL
```

Flow Chart:


MONDAY ?
Y
N
MUTTON MEAL
TUESDAY ?
Y
N
CHICKEN MEAL
SUNDAY ?
Y
SPECIAL MEAL

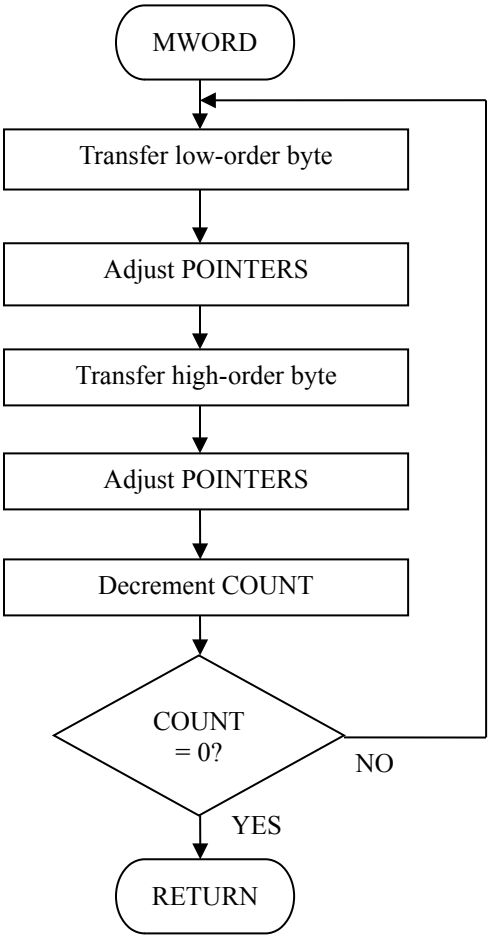
Assembly Language Programming:
Flow Chart for transferring block of bytes from one area of memory to another




```
;---------------------------------------------------------------------
;8085 Program to transfer a number of bytes from one place to another
;in memory (small string having 255 or less elements)
;---------------------------------------------------------------------
                ORG 8000H
     START:     LXI H,SOUR       ;SOURCE MEMORY POINTER
                LXI D,DEST       ;DESTINATION MEMORY POINTER
                MVI B,COUNT      ;STRING ELIMENT COUNTER

     LOOP:      MOV A,M          ;MOVE SOURCE ELEMENT INTO ACC
                STAX D           ;STORE ACC IN DESTINATION

                INX H            ;SOURCE MEM POINTER INCREMENTED
                INX D            ;DESTINATION MEM POINTER INCREMENTED

                DCR B
                JNZ LOOP
     ENDP:      JMP ENDP
     COUNT:     EQU 100D
     SOUR:      EQU 9000H
     DEST:      EQU 9100H
                END
```

```
;---------------------------------------------------------------------
;8085 Program to transfer a number of bytes from one place to another
;in memory (large string having 256 or greater no. of elements)
;---------------------------------------------------------------------

              ORG 8000H
START:        LXI H,SOUR       ;SOURCE MEMORY POINTER
              LXI D,DEST       ;DESTINATION MEMORY POINTER
              LXI B,COUNT      ;STRING ELIMENT COUNTER

LOOP:         MOV A,M          ;MOVE SOURCE ELEMENT INTO ACC
              STAX D           ;STORE ACC IN DESTINATION

              INX H            ;SOURCE MEM POINTER INCREMENTED
              INX D            ;DESTINATION MEM POINTER INCREMENTED

              DCX B            ;CHECK IF BC=0, IF ZERO
                               STOP ;EXECUTION.
              MOV A,B          ;IF NOT ZERO, CONTINUE.
              ORA C            ;
              JNZ LOOP         ;

ENDP:         JMP ENDP

COUNT:        EQU 1000D
SOUR:         EQU 9000H
DEST:         EQU 9400H
              END
```

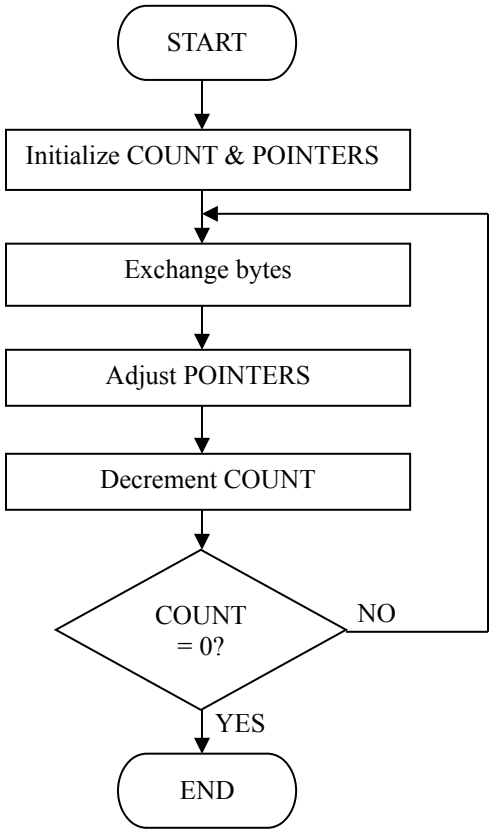
Flow Chart of a sub routine that moves a block of words from one area of memory to another




```
;-------------------------------------------------------------------
;8085 Program to transfer a number of words from one place to another
;in memory (large string having 256 or greater no. of elements)
;-------------------------------------------------------------------

                ORG 8000H
     START:     LXI SP,8500H     ;SP INITIALIZED
                LXI H,SOUR       ;SOURCE MEMORY POINTER
                LXI D,DEST       ;DESTINATION MEMORY POINTER
                LXI B,COUNT      ;STRING ELIMENT COUNTER

                CALL MWORD

     ENDP:      JMP ENDP

                ORG 8200H
     MWORD:     MOV A,M          ;MOVE LOW-ORDER BYTE
                STAX D           ;
                INX H
                INX D
```

```
                MOV A,M          ;MOVE HIGH-ORDER BYTE
                STAX D           ;
                INX H
                INX D
                DCX B            ;CONTINUE MOVING IF NOT EXHUASTED
                MOV A,B          ;OTHERWISE STOP.
                ORA C            ;
                JNZ MWORD        ;
                RET


COUNT:      EQU 1000D
SOUR:       EQU 9000H
DEST:       EQU 9400H
            END
```

Byte Block Exchanges:



Flow Chart for exchanging a block bytes from one area of memory with another

```
;-----------------------------------------------------------------
```

```
;8085 Program to exchange two string stored in memory
;(large string having 256 or greater no. of elements)
;-----------------------------------------------------------------


              ORG 8000H
     START:   LXI H,SOUR      ;SOURCE MEMORY POINTER
              LXI D,DEST      ;DESTINATION MEMORY POINTER
              LXI B,COUNT     ;STRING ELIMENT COUNTER

     LOOP:    MOV C,M         ;SOURCE ELEMENT IN REG-C
              LDAX D          ;DESTINATION ELEMENT IN ACC
              MOV M,A         ;
              MOV A,C         ;EXCHANGE ELEMENTS
              STAX D          ;

              INX H
              INX D

              DCX B           ;CONTINUE MOVING IF NOT EXHUASTED
              MOV A,B         ;OTHERWISE STOP.
              ORA C           ;
              JNZ LOOP        ;

     ENDP:    JMP ENDP

     COUNT:   EQU 1000D
     SOUR:    EQU 9000H
     DEST:    EQU 9400H
              END
```

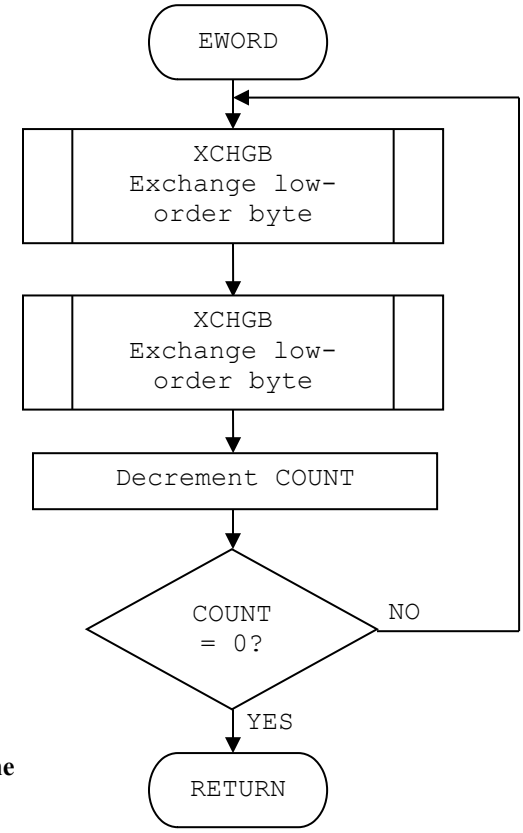
Word Block Exchanges:

Like word transfer word exchanges are accomplished in a very similar manner. The only difference between their flow charts is the exchange word instead of exchange byte.

```
;
;Equates for calling sequence EWORD
;
     DEST:          EQU 2800H
     SOUR:          EQU 0000H
     COUNT:    EQU 0200H
     STACK:    EQU 7FFFH

               LXI D,DEST
               LXI H,SOUR
               LXI B,COUNT
               LXI SP,STACK
               CALL EWORD
     ENDP:          JMP ENDP

;-------------------------------------------------------------
```

```
;Subroutine EWORD
;-----------------------------------------------------------------
                ORG 3000H
     EWORD:     PUSH B    ;SAVE COUNT
                MOV C,M   ;EXCHANGE LOW-ORDER BYTE
                LDAX D
                MOV M,A
                MOV A,C
                STAX D
                INX D          ;ADJUST POINTERS
                INX H
                MOV C,M   ;EXCHANGE HIGH-ORDER BYTE
                LDAX D
                MOV M,A
                MOV A,C
                STAX D
                INX D          ;ADJUST POINTERS
                INX H
                POP B          ;RESTORE COUNTER
                DCX B     ;DECREMENT COUNTER
                MOV A,B   ;TEST COUNTER
                ORA C
                JNZ EWORD ;IF COUNTER NOT ZERO
                RET       ;END SUBROUTINE
```

The subroutine EWORD contains two identical sequences of instructions, one to exchange low-order bytes and one to exchange high-order bytes. This sequence can be written as a subroutine improving the readability of the subroutine.

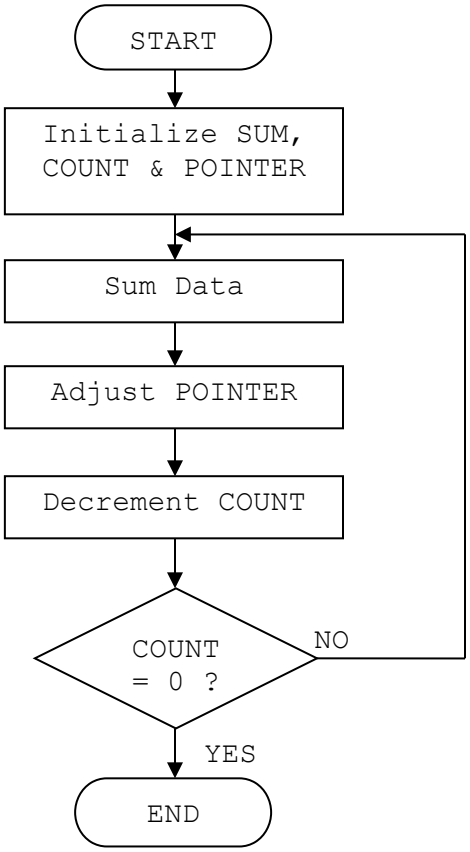


Flow Chart of a subroutine that exchanges a block words from one area of memory with another

```
EWORD:    CALL XCHGB
          CALL XCHGB
          DCX B
          MOV A,B
          ORA C
          JNZ EWORD
          RET

XCHGB:    PUSH B
          MOV C,M
          LDAX D
          MOV M,A
          MOV A,C
          STAX D
          INX D
          INX H
          POP B
          RET
```

8-bit Binary addition and subtraction:

Flow chart of a program that sums a no. of 8-bit data stored in memory locations.



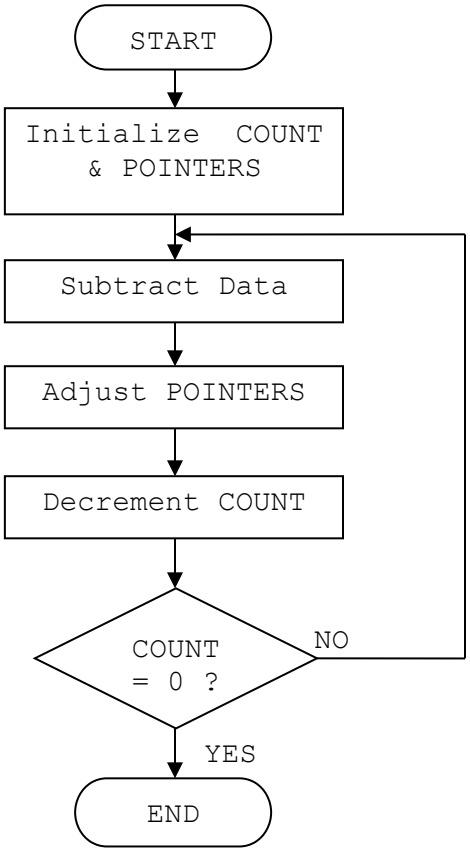
Assembly Language Programming:

```
;---------------------------------------------------------------------
;                  ADD 100 BYTES OF DATA STORED IN MEMORY AND
;                        LEAVES THE RESULT IN C & ACC
;
;              9:31 PM 8/19/2004   WRITTEN BY SAIBAL PRADHAN
;---------------------------------------------------------------------

            ORG 8000H

DATA1:      EQU 8500H          ;DATA BYTES IN MEMORY STARTING AT 8500H

MADD:       LXI H,DATA1        ;MEMORY POINTER TO PICK UP DATA
            MVI B,100          ;BYTE COUNTER
            XRA A              ;CLEAR THE SUM & MSB SUM
            MOV C,A            ;

LOOP:       ADD M              ;SUM DATA
            JNC SKIP
            INR C              ;ADD CARRY TO MSB SUM
SKIP:       DCR B
            JNZ LOOP           ;REPEAT UNTILL 100 DATA ARE ADDED.
```

```
ENDP:       JMP ENDP
            END
```

Flow Chart:

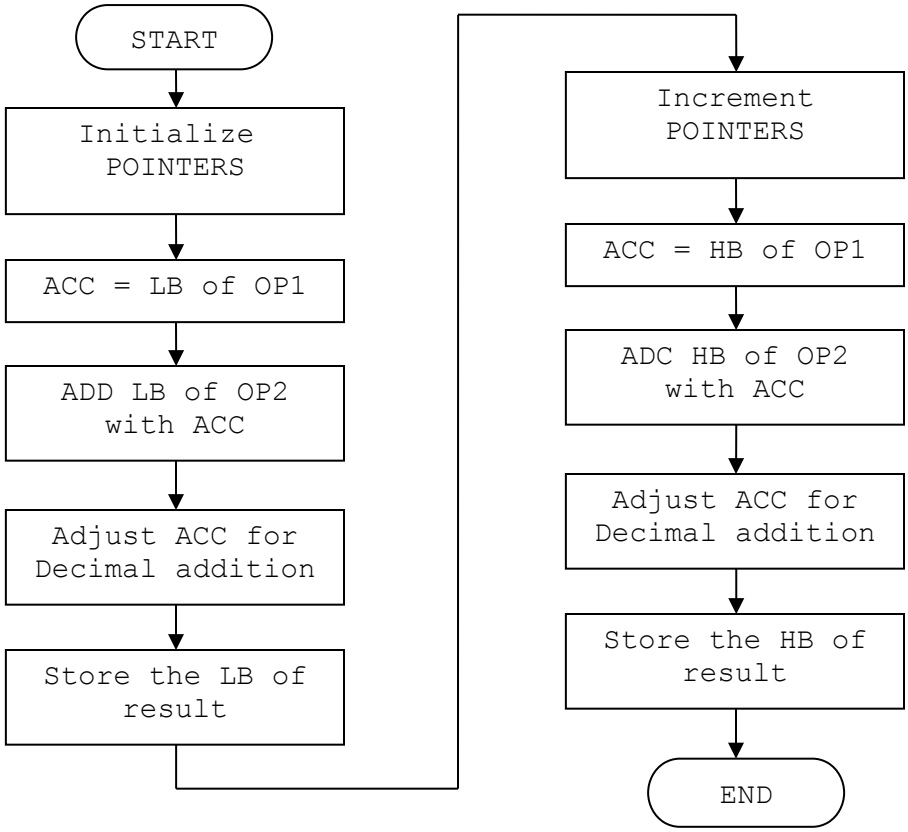
Flow Chart of a program that subtracts bytes of an array from bytes of another array and store the results in second array.




```
;-------------------------------------------------------------------------
;   SUBTARCT LIST2 DATA FROM LIST1 DATA AND STORE THE RESULT IN LIST2
;
;             10:33 PM 8/19/2004   WRITTEN BY SAIBAL PRADHAN
;-------------------------------------------------------------------------

            ORG 8000H

LIST1:      EQU 8500H         ;ADDRESS OF LIST1
LIST2:      EQU 9000H         ;ADDRESS OF LIST2

ASUB:       LXI H,LIST2       ;LIST2 POINTER
            LXI D,LIST1       ;LIST1 POINTER
            MVI B,100         ;LOAD COUNTER

LOOP:       LDAX D
            SUB M
            MOV M,A
```

```
            INX H
            INX D
            DCR B
            JNZ LOOP
ENDP:       JMP ENDP
```

Flow Chart of a program that adds two 4-digit BCD numbers stored in memory. Store the result also in memory.




```
;-------------------------------------------------------------------
;           ADD TWO 4-DIGIT BCD NUMBERS STORED IN MEMORY.
;                 ALSO STORE THE RESULT IN MEMORY
;
;             8:18 PM 8/19/2004 WRITTEN BY SAIBAL PRADHAN
;-------------------------------------------------------------------

            ORG 8000H

DATA1:      EQU 8500H         ;OP1 IN MEMORY 8500-8501H
DATA2:      EQU 8502H         ;OP2 IN MEMORY 8502-8503H
RESULT:     EQU 8504H         ;RESULT IN MEM 8504-8506H

ADDBCD:     LXI H,DATA1       ;MEMORY POINTER TO PICK UP OP1
            LXI B,DATA2       ;MEMORY POINTER TO PICK UP OP2
            LXI D,RESULT      ;MEMORY POINTER TO STORE RESULT
```

```
            LDAX B          ;LOAD 1ST BYTE OF OP2 IN ACC
            ADD M           ;1ST TWO DIGITS OF OP1 & OP2 ARE ADDED
            DAA             ;ADJUST FOR BCD ADDTION
            STAX D          ;STORE THE 1ST TWO DIGITS OF THE RESULT

            INX H           ;
            INX B           ;INCREMENT POINTERS TO ADD 2ND BYTES
            INX D           ;

            LDAX B          ;
            ADC M           ;2RD & 4TH DIGITS OF OP1 & OP2 ARE ADDED
            DAA             ;AND ADJUSTED FOR BCD
            STAX D          ;STORE THE 2ND BYTE OF THE RESULT

            INX D           ;
            MVI A,00H       ;5TH DIGIT OF THE RESULT IS PROCESSED
            ACI 00H         ;AND STORED IN MEMORY.
            STAX D          ;

PEND:       JMP PEND
```

Problems:

Develop flow charts and 8085 Assembly Language Programs (ALP) for the following problems

---

1. Write an ALP to add two 16-bit numbers already stored in memory locations 2000H and 2002H respectively. Justify the length of the result and store it at memory starting at 2004H.
2. Write an ALP to subtract two 16-bit numbers already stored in memory locations 2010H and 2012H respectively. Justify the length of the result and store it in memory starting at 2014H.
3. Write an ALP to multiply two unsigned numbers.
4. Write an alp to multiply two signed numbers.
5. Repeat problems 3 and 4 for division.
6. A byte is stored at memory location 2007H. Write an ALP to separate the two nibbles and store ls-nibble and ms-nibble in 2008H and 2009H respectively.
7. Write an ALP to clear the memory ranging 2010H to 204FH.
8. Write an ALP to add the contents of memory locations starting at 2051H. The no. of data is given in memory location 2050H. Store the result in BC register pair.
9. Write an ALP to add the first 10 prime numbers.

Solutions:

1. Write an ALP to add two 16-bit numbers already stored in memory locations 2000H and 2002H respectively. Justify the length of the result and store it at memory starting at 2004H.

Addition of two 16-bit data (unsigned say) may result a 17-bit answer which requires three memory locations for its storage.
8085 can add two 8-bit data by using ADD instruction. ADD requires one of the operands to be kept in ACC and other may be in any 8-bit register or it may be an immediate number also. 9-bit result will be stored in ZF plus ACC. So whole addition will be in two steps, lower two bytes of the operands will be added first and then the upper bytes taking the carry of the first addition, if any. Carry of the second addition will be the 17th bit of the result.
It can also add two 16-bit data using DAD instruction which requires one operand to be stored in HL and the other in any of the four 16-bit registers (HL, BC, DE and SP). 17-bit result will be available in CF plus HL.

A. Using ADD instruction

```
LXI H, 2002H    ;HL points 2nd operand
LXI B, 2004H    ;BC points result
LDA 2000H       ;ls-byte of 1st operand is taken in ACC
ADD M           ;ls-byte of 2nd operand is added with ACC
STAX B          ;ls-byte of the result is stored
```

```
        INX H           ;Pointers are incremented for next byte
        INX B           ;

        LDA 2001H       ;ms-byte of the 1st operand is taken in ACC
        ADC M           ;ms-bytes of the operands are
                        added     ;considering the previous carry
        STAX B          ;2nd byte of the result byte is stored

        INX B

        MVI A, 00H      ;Carry of the 2nd addition is stored as 3rd
        ADC A           ;byte of the result.
        STAX B          ;
        HLT
```

B. Using DAD instruction

```
        LHLD 2000H      ;1st operand is taken in HL
        XCHG            ;1st operand is shifted in DE from HL
        LHLD 2002H      ;2nd operand is taken in HL
        DAD D           ;They are added and HL stores the first 16
        SHLD 2004H      ;bit of the result which is stored in 2004H
                        ;and 2005H
        MVI A, 00H      ;
        ADC A           ;17th bit of the result is taken in ACC and
        STA 2006H       ;finally stored in 2006H
        HLT
```

Important!!

For 16-bit signed addition, final carry to be considered as sign bit; accordingly the 3rd byte to be made either FFH for CF=1 or 00H for CF=0.

2. Write an ALP to subtract two 16-bit numbers already stored in memory locations 2010H and 2012H respectively. Justify the length of the result and store it in memory starting at 2014H.

As 8085 can only subtract two 8-bit data, we have to do it in two steps. The borrow (CF), if any, after first subtraction to be considered during the second subtraction. The final borrow to be considered as the sign bit and to be properly stored in the 3rd byte.

```
   LXI H, 2012H     ; HL points 2nd operand
   LDA 2010H        ; ls-byte of 1st operand is in Acc
   SUB M            ; Acc <- ls-byte of op1 - ls-byte of op2
   STA 2014H        ; RESULT STORED

   INX H            ; ms-bytes are subtracted and stored
   LDA 2011H        ;
   SBB M            ;
   STA 2015H        ;
```

```
   MVI A, 00H       ; If CF=0, 3rd byte of the result is 00H
   JNC SKIP         ; and if CF=1, it is FFH
   MVI A, FFH       ;
   SKIP: STA 2016H  ;
   HLT
```

3. Write an ALP to multiply two unsigned numbers.

We will write this program in a subroutine form as it may be used in different parts of a main routine. Let us consider the numbers to be multiplied are passed to the subroutine through registers B and C and the subroutine will return the 16-bit result through BC register pair.

```
         .ORG 2000H
MAIN:    LXI SP, 8000H    ; STACK POINTER INITIALIZED

         LDA 3000H        ; ORIGINALLY OPERANDS WERE IN MEMORY AT
         MOV B, A         ; 3000H AND 3001H. THEY ARE PASSED
         LDA 3001H        ; THROUGH REG B AND REG C BEFORE THE
         MOV C, A         ; SUBROUTINE IS CALLED.
         CALL MUL8
         .
         .
         .
         .


MUL8:    PUSH PSW
         PUSH D

         MVI A, 00H       ; Check for zero operands. If one or both
         ORA B            ; operands are zero, result will be zero.
         JZ ZERO          ;
         MVI A, 00H       ;
         ORA C            ;
         JNZ NEXT         ;
ZERO:    LXI B, 0000H     ;
         JMP STOP         ;


NEXT:    MVI A, 00H
         MOV D, A
AGAIN:   ADD B
         JNC SKIP
         INR D
SKIP:    DCR C
         JNZ AGAIN
         MOV C, A
         MOV B, D

STOP:    POP D
```

```
        POP PSW
        RET
```

4. Write an alp to multiply two signed numbers.

For signed multiplication, basically we have to multiply the magnitudes of two numbers and the sign of the result will be positive if operands are of same signs and negative if they are of opposite signs.
As the range of 8-bit sign number system is -128 to +127, the range of the result will be -128 x +127 (-16256D or C080H)to -128 x -128 (+16384D or 4000H) which can be expressed in 2's complement 16-bit format.

```
        .ORG 2000H
MAIN:   LXI SP, 8000H    ; STACK POINTER INITIALIZED

        LDA 3000H        ; ORIGINALLY OPERANDS WERE IN MEMORY AT
        MOV B, A         ; 3000H AND 3001H. THEY ARE PASSED
        LDA 3001H        ; THROUGH REG B AND REG C BEFORE THE
        MOV C, A         ; SUBROUTINE IS CALLED.
        CALL SMUL8
        MOV A, C
        STA 3002H
        MOV A, B
        STA 3003H
        .
        .
        .

SMUL8:  PUSH PSW
        PUSH D

        MVI D, 00H       ; This block determines sign of the result
        MOV A, B         ; D=0 for result to be positive
        XRA C            ;
        ANI 80H
        JZ POS
        MVI D, 01H       ; D=1 for result to be negative

POS:    MOV A, B         ; Checking the sign of reg B. If it is
        ORI 00H          ; negative it is replaced by its magnitude
        JP BPOS
        CMA
        INR A
        MOV B, A

BPOS:   MOV A, C
        ORI 00H
        JP CPOS
        CMA
        INR A
```

```
        MOV C, A

CPOS:   CALL MUL8
        MOV A, B        ; if result is zero, BC = BC
        ORA C
        JZ RESPOS

        MOV A, D        ; If result is positive BC = BC
        ORI 00H
        JZ RESPOS

; If result is negative, BC is replaced  by its 2's complement value

        MOV A, B        ;
        CMA             ;
        MOV B, A        ; 1's complement of B

        MOV A, C
        CMA
        INR A
        MOV C, A        ; 2's complement of C

        MVI A, 00H      ; Carry, if any, be added with B's
        ADC B           ; complement
        MOV B, A

RSEPOS: POP D
        POP PSW
        RET

MUL8:   PUSH PSW
        PUSH D

        MOV A, B        ; Check for zero operands. If one or both
        ANA C           ; operands are zero, result will be zero.
        JNZ NEXT

ZERO:   LXI B, 0000H    ;
        JMP STOP        ;


NEXT:   MVI A, 00H      ; Unsigned multiplication between two
        MOV D, A        ; non-zero 8-bit data by the method of
AGAIN:  ADD B           ; repeated addition. Carry generated, if
        JNC SKIP        ; any, is taken care of in the next byte
        INR D
SKIP:   DCR C
        JNZ AGAIN
        MOV C, A
        MOV B, D

STOP:   POP D
```

```
        POP PSW
        RET
```

5. Repeat problems 3 and 4 for division.

Try of your own to solve following the solutions of multiplications.

6. A byte is stored at memory location 2007H. Write an ALP to separate the two nibbles and store ls-nibble and ms-nibble in 2008H and 2009H respectively.

```
        .ORG 2000H
        LDA 2007H       ; data is taken in Acc
        MOV B, A        ; a copy is placed in Reg B
        ANA 0FH         ; ms-nibble is masked to have the ls-nibble
        STA 2008H       ; ls-nibble is stored in 2008H

        MOV A, B        ; data is again taken in Acc
        RRC             ; Acc is rotated right 4 times to have
        RRC             ; ms-nibble in place of ls-nibble
        RRC
        RRC
        ANA 0FH         ; ms-nibble is masked
        STA 2009H       ; original ms-nibble is stored in 2009H
        HLT
```

7. Write an ALP to clear the memory ranging 2010H to 204FH.

In this problem 64 memory locations to be cleared starting from 2010H to 204FH.

```
        .ORG 2000H
        LXI H, 2010H
        MVI C, 64H
        XRA A
LOOP:   MOV M, A
        INX H
        DCR C
        JNZ LOOP
        HLT
```

8. Write an ALP to add the contents of memory locations starting at 2051H. The no. of data is given in memory location 2050H. Store the result in BC register pair.

```
        .ORG 3000H
        LXI H, 2050H
        MOV C, M        ; no of data to be added is taken in Reg C
        XRA A           ; Acc is cleared for addition
        MOV B, A        ; Reg B is also cleared to have carry
REPEAT: INX H
```

```
            ADD M
            JNC SKIP
            INR B
SKIP:       DCR C
            JNZ PEPEAT
            MOV C, A        ; BC pair holds the result
            HLT
```

9. Write an ALP to add the first 10 prime numbers.

The problem can be solved in two steps. The first routine will find the first 10 prime numbers and stored them in memory and the second routine will add them up and store the result.
The second routine is exactly similar to that of problem 8. Try to write the first routine of your own.

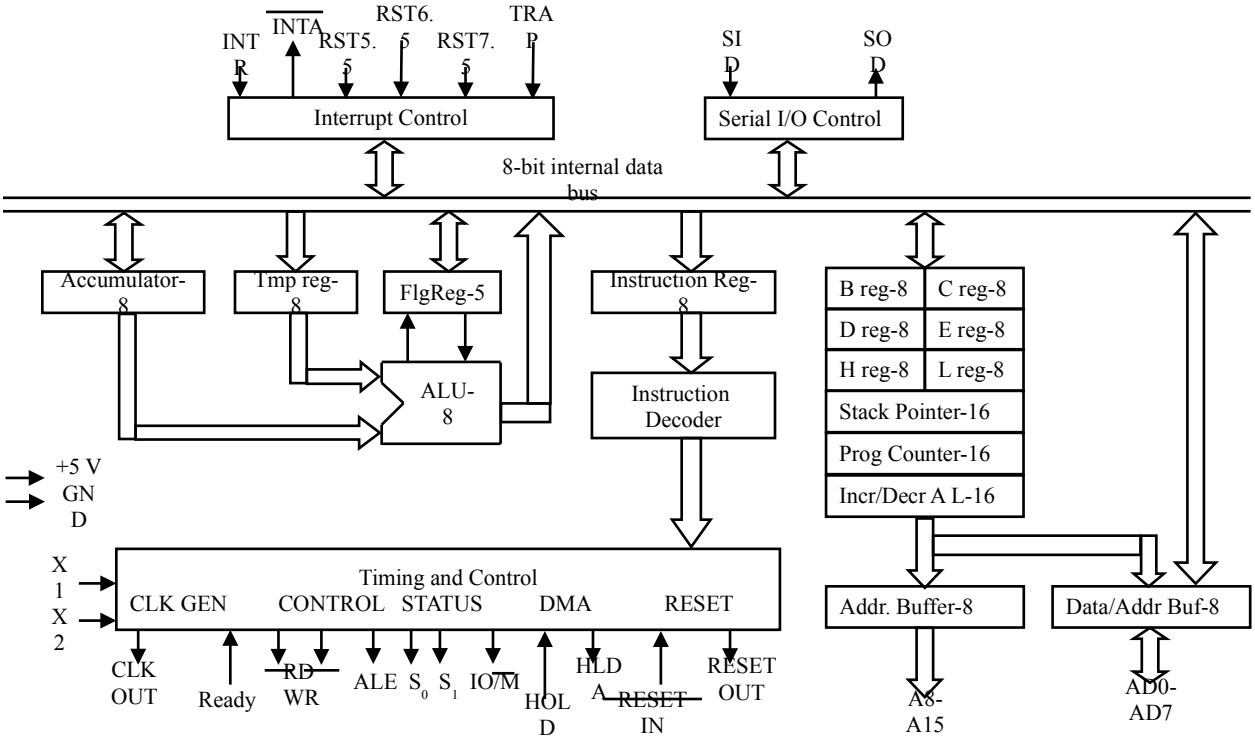
Problems:

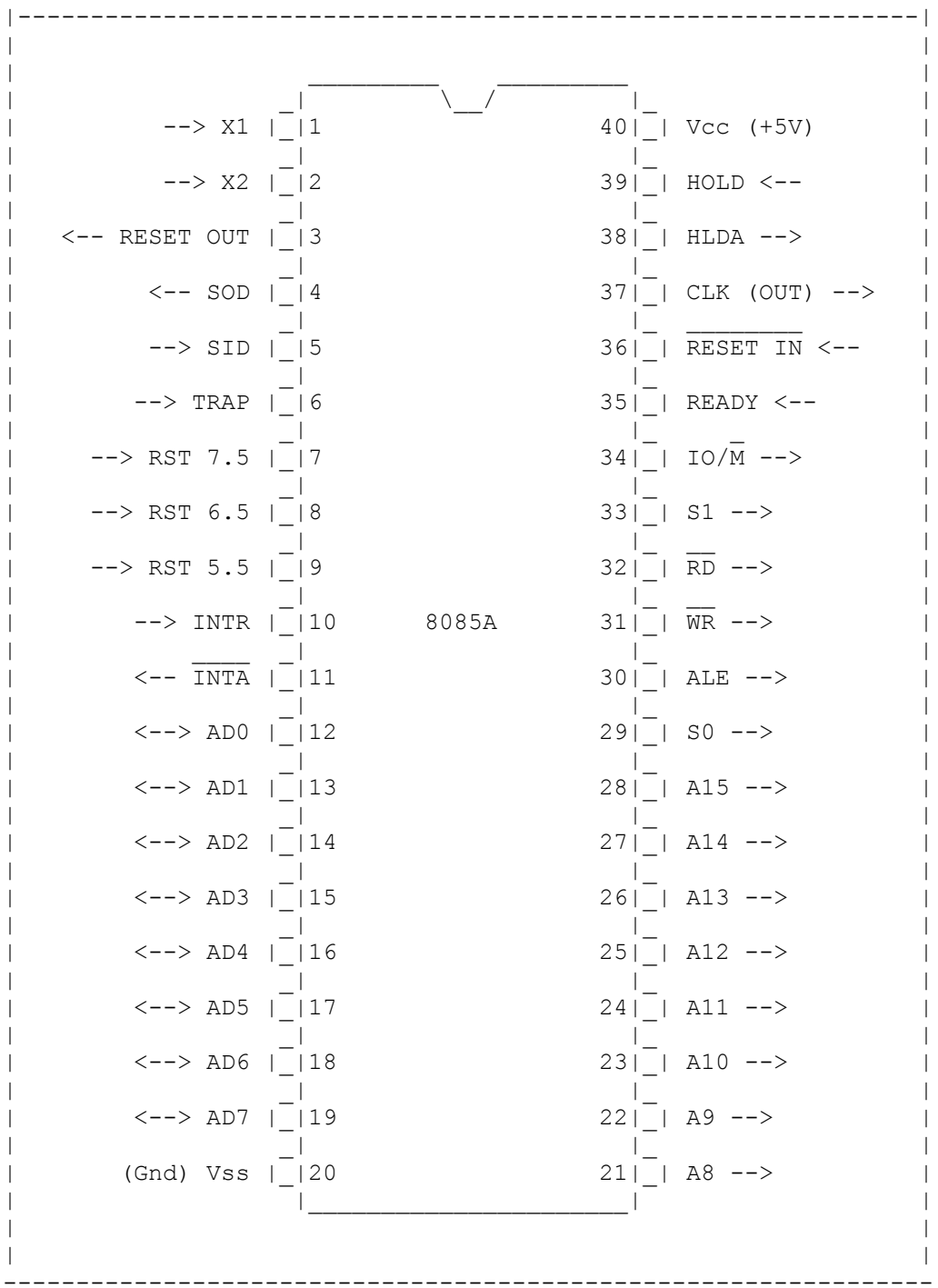
Develop flow charts and 8085 Assembly Language Programs (ALP) for the following problems

---

1. Write an ALP to add the odd numbers between 50 to 70.
2. Write an ALP to find the maximum in a given series of numbers starting from 2051H. The length is given in 2050H.
3. Write an ALP to copy a block of 100 data bytes starting from 2050H at 2100H.
4. Write an ALP to shift a block of data between 2600H - 261FH to the block starting at 2600H.
5. Write an ALP to find the no. of negative numbers in an array stored in memory between 2060H and 207FH.
6. Write an ALP to find the no. of odd numbers in an array stored in memory between 2060H and 207FH.
7. Write an ALP to find the square of a given number supplied through Register B.
8. Write an ALP to find the square root of a given number present in Register B using loop-up table method.
9. Write an ALP to convert hexadecimal digits (0-F) supplied through register B into 7-segment data using look-up table method. Assume that the data lines (d0-d7) are connected with the segments as: a,b,c,d,e,f,g,dp : d0.....d7 and the 7_segment display is common anode type.
10. Repeat problem 15 considering common cathode display.

Internal Block Diagram of 8085A




--> X1
--> X2
<-- RESET OUT
<-- SOD
--> SID
--> TRAP
--> RST 7.5
--> RST 6.5
--> RST 5.5
--> INTR
<-- INTA
<--> AD0
<--> AD1
<--> AD2
<--> AD3
<--> AD4
<--> AD5
<--> AD6
<--> AD7
(Gnd) Vss
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
8085A
40
39
38
37
36
35
34
33
32
31
30
29
28
27
26
25
24
23
22
21
Vcc (+5V)
HOLD <--
HLDA -->
CLK (OUT) -->
RESET IN <--
READY <--
IO/M -->
S1 -->
RD -->
WR -->
ALE -->
S0 -->
A15 -->
A14 -->
A13 -->
A12 -->
A11 -->
A10 -->
A9 -->
A8 -->

Timming Diagram (8085)

Timing Diagram is a graphical representation. It represents the execution time taken by each instruction in a graphical format. The execution time is represented in T-states.

Instruction Cycle:

The time required to execute an instruction is called instruction cycle.

Machine Cycle:

The time required to access the memory or input/output devices is called machine cycle.

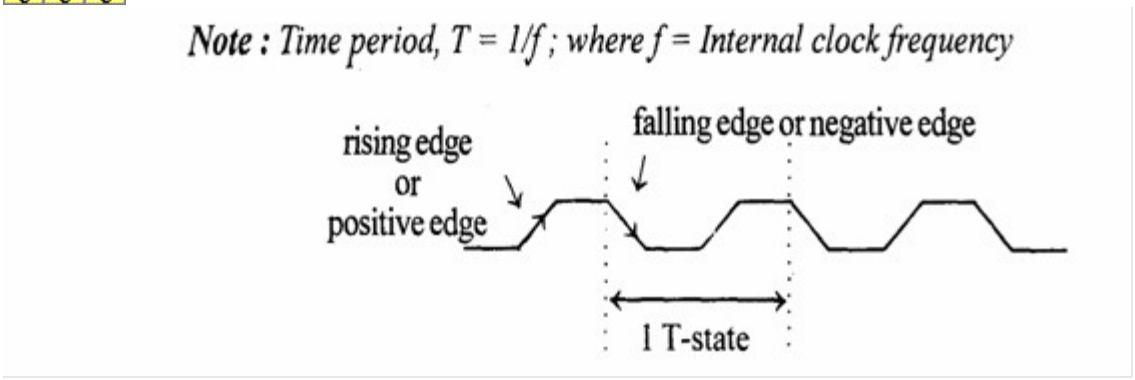
T-State:

- **The machine cycle and instruction cycle takes multiple clock periods.**
- **A portion of an operation carried out in one system clock period is called as T-state.**

MACHINE CYCLES OF 8085:

The 8085 microprocessor has 5 (seven) basic machine cycles. They are

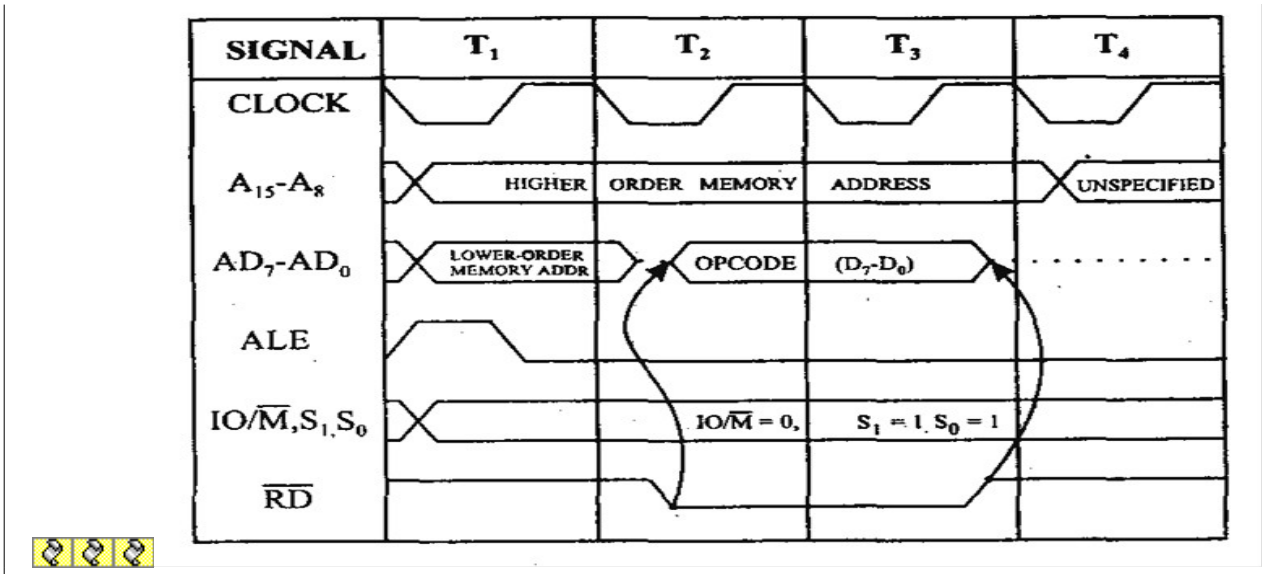
1. **Opcode fetch cycle (4T)**
2. **Memory read cycle (3 T)**
3. **Memory write cycle (3 T)**
4. **I/O read cycle (3 T)**
5. **I/O write cycle (3 T)**



- **Each instruction of the 8085 processor consists of one to five machine cycles, i.e., when the 8085 processor executes an instruction, it will execute some of the machine cycles in a specific order.**
- **The processor takes a definite time to execute the machine cycles. The time taken by the processor to execute a machine cycle is expressed in T-states.**
- **One T-state is equal to the time period of the internal clock signal of the processor.**
- **The T-state starts at the falling edge of a clock.**

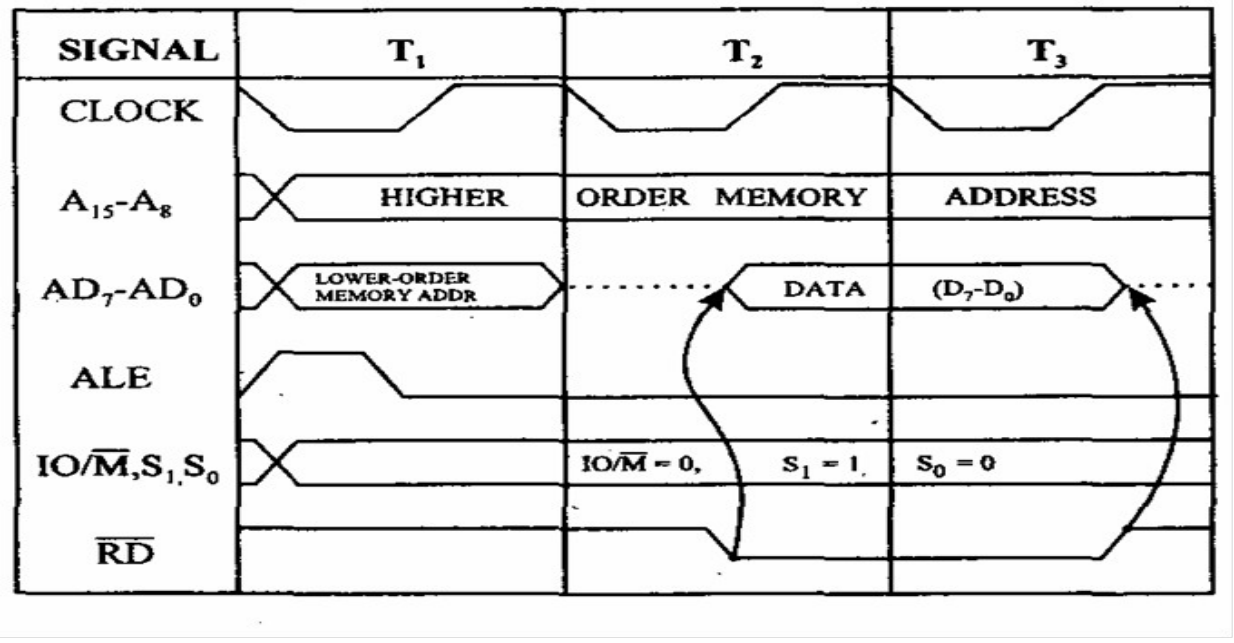
Opcode fetch machine cycle of 8085 :

- Each instruction of the processor has one byte opcode.
- The opcodes are stored in memory. So, the processor executes the opcode fetch machine cycle to fetch the opcode from memory.
- Hence, every instruction starts with opcode fetch machine cycle.
- The time taken by the processor to execute the opcode fetch cycle is 4T.
- In this time, the first, 3 T-states are used for fetching the opcode from memory and the remaining T-states are used for internal operations by the processor.



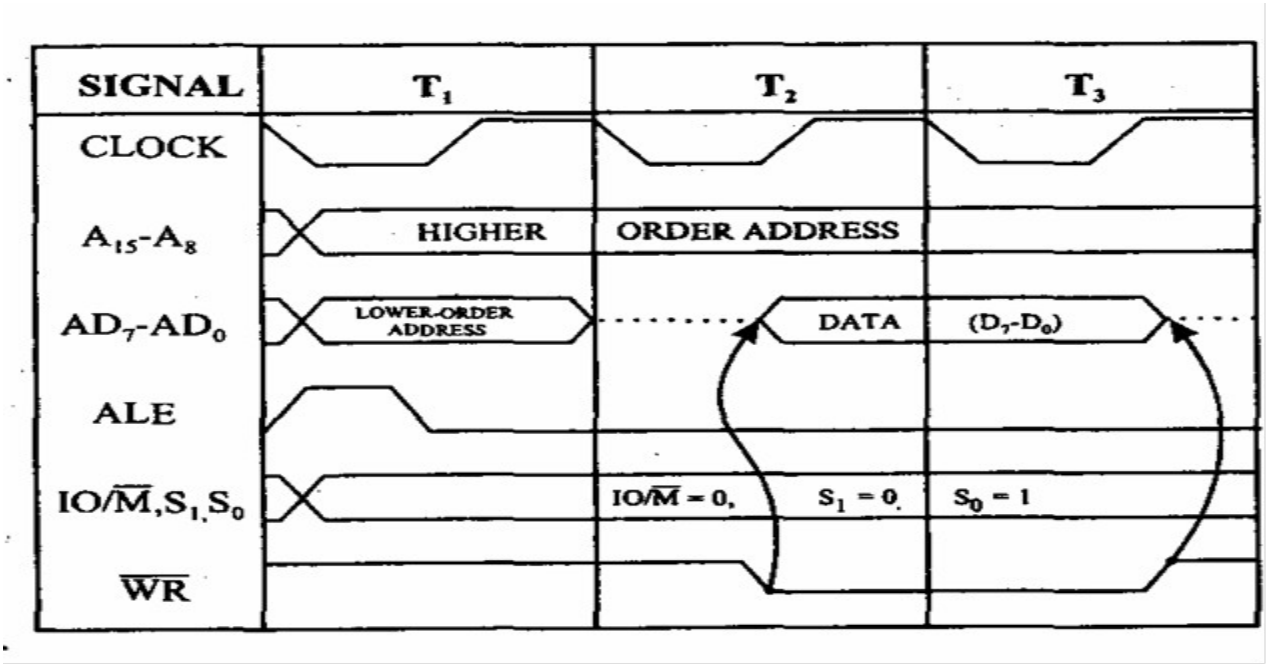
Memory Read Machine Cycle of 8085:

- The memory read machine cycle is executed by the processor to read a data byte from memory.
- The processor takes 3T states to execute this cycle.
- The instructions which have more than one byte word size will use the machine cycle after the opcode fetch machine cycle.



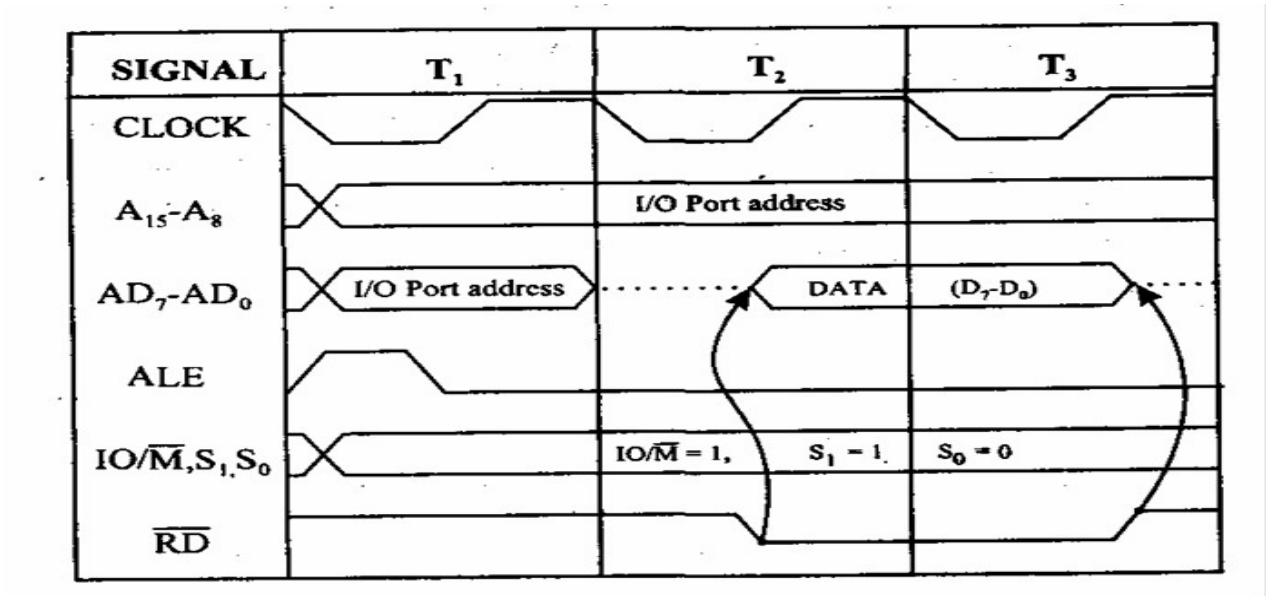
Memory Write Machine Cycle of 8085:

- The memory write machine cycle is executed by the processor to write a data byte in a memory location.
- The processor takes, 3T states to execute this machine cycle.



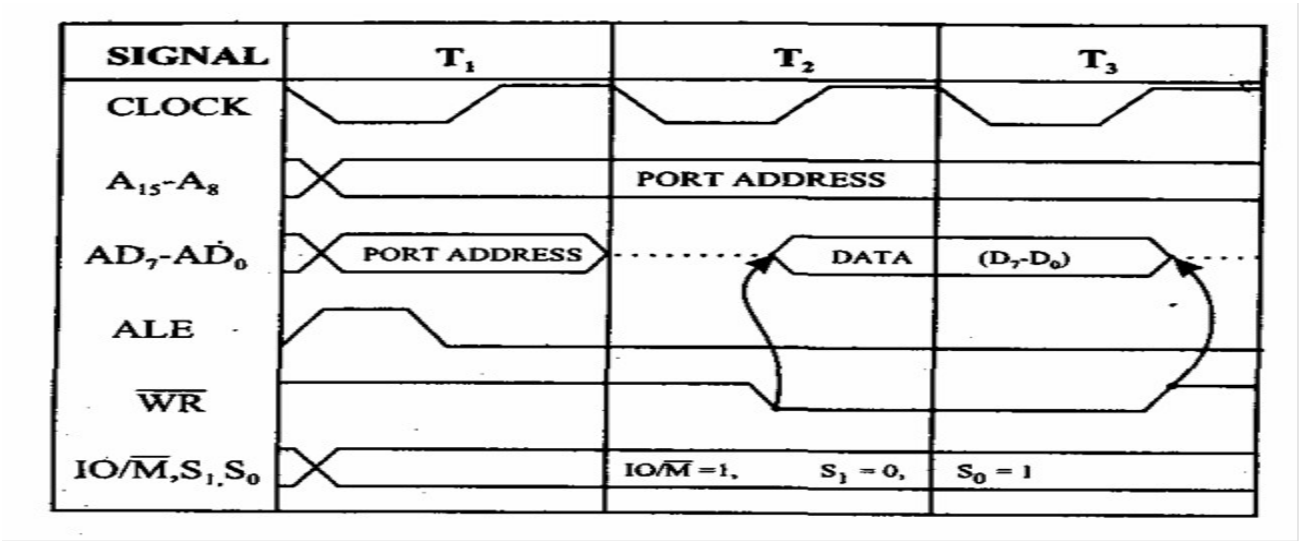
I/O Read Cycle of 8085:

- The I/O Read cycle is executed by the processor to read a data byte from I/O port or from the peripheral, which is I/O, mapped in the system.
- The processor takes 3T states to execute this machine cycle.
- The IN instruction uses this machine cycle during the execution.



I/O Write Cycle of 8085:

- The I/O write machine cycle is executed by the processor to write a data byte in the I/O port or to a peripheral, which is I/O, mapped in the system.
- The processor takes, 3T states to execute this machine cycle.



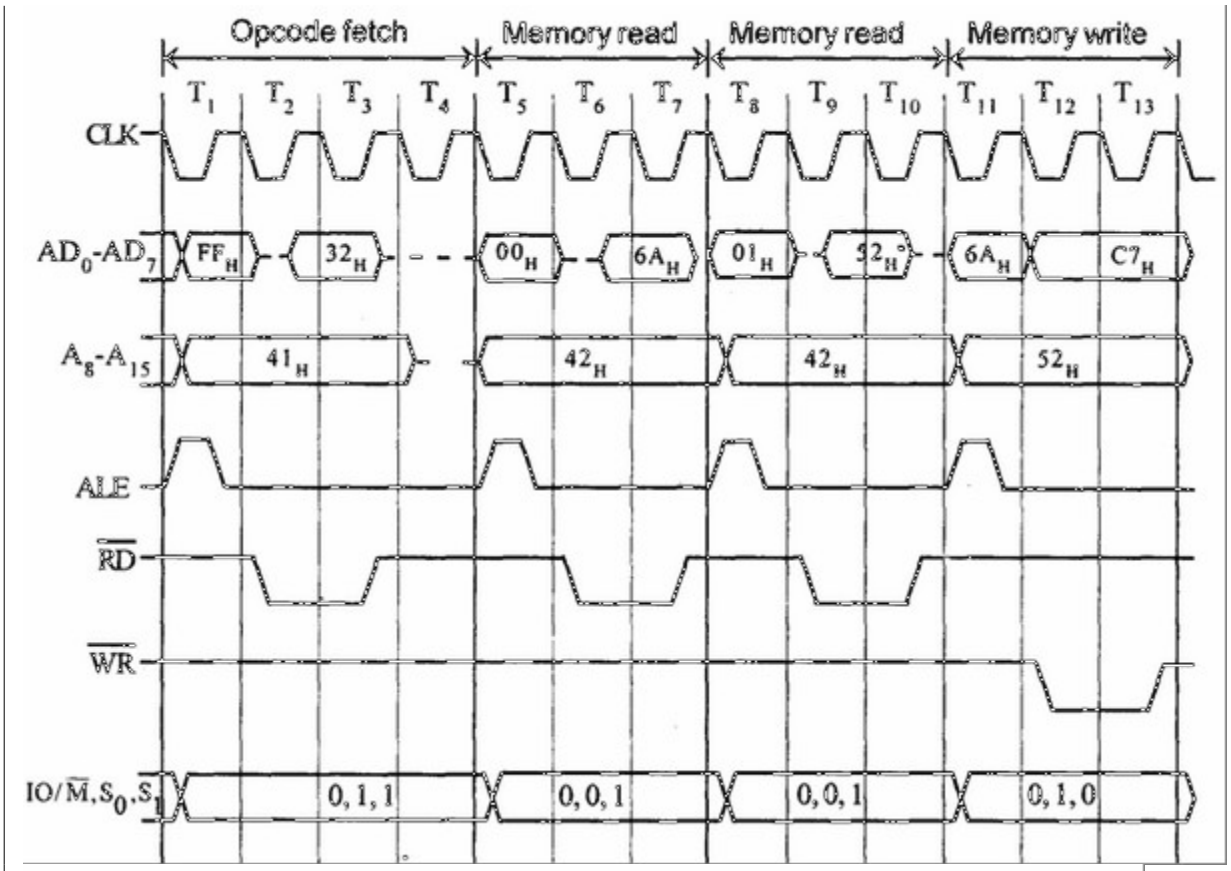
Timming Diagram of 8085 Instructions

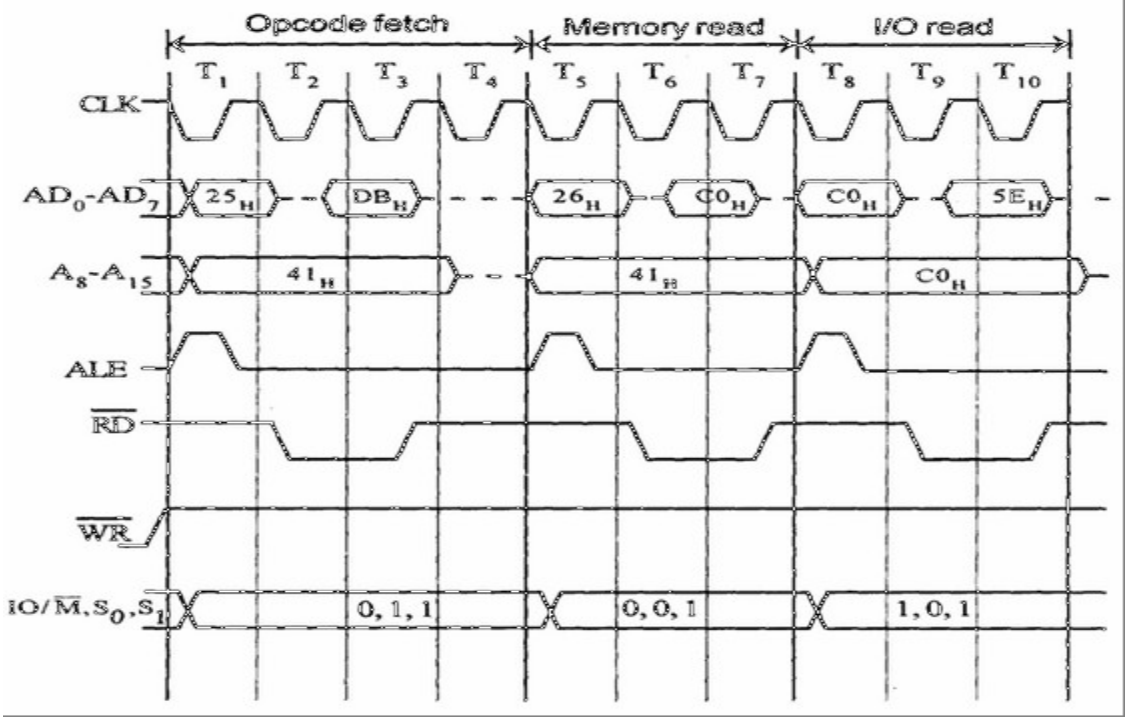
- The 8085 instructions consist of one to five machine cycles.
- Actually the execution of an instruction is the execution of the machine cycles of that instruction in the predefined order.
- The timing diagram of an instruction ate obtained by drawing the timing diagrams of the machine cycles of that instruction, one by one in the order of execution.

TIMING DIAGRAM OF 8085 INSTRUCTIONS

Timing diagram for STA 526AH.

- STA means Store Accumulator -The contents of the accumulator is stored in the specified address(526A).
- The opcode of the STA instruction is said to be 32H. It is fetched from the memory 41FFH(see fig). - *OF machine cycle*
- Then the lower order memory address is read(6A). - *Memory Read Machine Cycle*
- Read the higher order memory address (52).- *Memory Read Machine Cycle*
- The combination of both the addresses are considered and the content from accumulator is written in 526A. - *Memory Write Machine Cycle*
- Assume the memory address for the instruction and let the content of accumulator is C7H. So, C7H from accumulator is now stored in 526A.

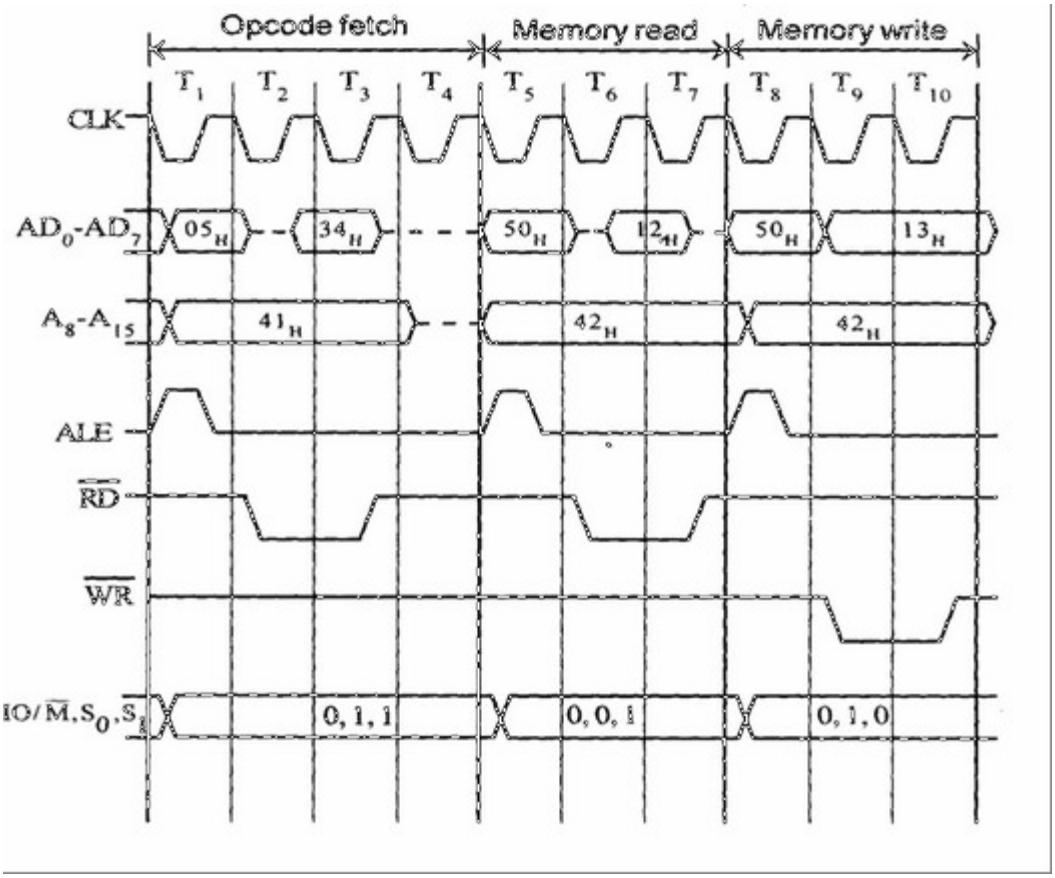

Opcode fetch
Memory read
Memory read
Memory write
T1 T2 T3 T4 T5 T6 T7 T8 T9 T10 T11 T12 T13
CLK
AD0-AD7
FFH 32H 00H 6AH 01H 52H 6AH C7H
A8-A15
41H 42H 42H 52H
ALE
RD
WR
IO/M, S0, S1
0, 1, 1 0, 0, 1 0, 0, 1 0, 1, 0


Opcode fetch
Memory read
I/O read
T1 T2 T3 T4 T5 T6 T7 T8 T9 T10
CLK
AD0-AD7
25H DBH 26H C0H C0H 5EH
A8-A15
41H 41H C0H
ALE
RD
WR
IO/M, S0, S1
0, 1, 1 0, 0, 1 1, 0, 1

Timing diagram for INR M

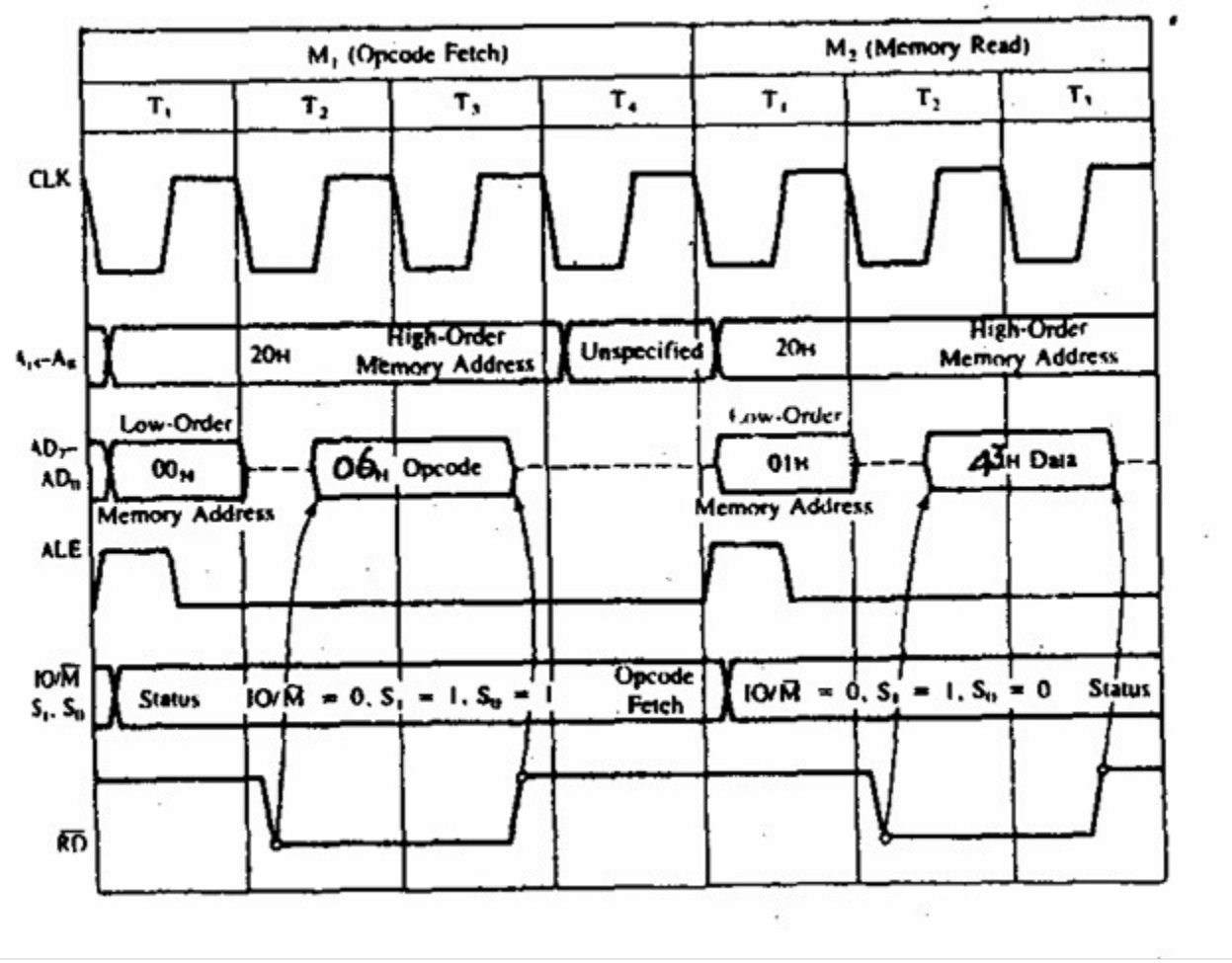
- **Fetching the Opcode 34H from the memory 4105H. (OF cycle)**
- **Let the memory address (M) be 4250H. (MR cycle -To read Memory address and data)**
- **Let the content of that memory is 12H.**
- **Increment the memory content from 12H to 13H. (MW machine cycle)**

Address	Mnemonics	Opcode
4105	INR M	34_H



Timing diagram for MVI B, 43H.

- **Fetching the Opcode 06H from the memory 2000H. (OF machine cycle)**
- **Read (move) the data 43H from memory 2001H. (memory read)**



Interrupts

Diverting the μP from its normal program flow is called interrupt. External I/O devices may require the attention of the μP at any point of time. This may be achieved in two ways:

1. to **scan or poll** them and
2. to use **interrupts**.

Scanning is just what it sounds like. Each possible event is scanned in a sequence, one at a time. This is ok for things that don't require immediate action. Interrupts, on the other hand, cause the current process to be suspended temporarily and the event that caused the interrupt is serviced, or handled, immediately. The routine that is executed as a result of an interrupt is called the interrupt service routine (ISR), or recently, the interrupt handler routine.

In the 8085, as with any CPU that has interrupt capability, there is a method by which the interrupt gets serviced in a timely manner. When the interrupt occurs, and the current instruction that is being processed is finished, the address of the next instruction to be executed is pushed onto the Stack. Then a jump is made to a dedicated location where the ISR is located. Some interrupts have their own vector or unique location where its service routine starts. These are hard coded into the 8085 and can't be changed (see below).

TRAP - has highest priority and **cannot** be masked or disabled. A rising-edge pulse will cause a jump to location 0024H.

RST 7.5- 2nd priority and can be masked or disabled. Rising-edge pulse will cause a jump to location 7.5 * 8 = 003CH.

This interrupt is latched internally and must be reset before it can be used again.

RST 6.5 – 3rd priority and can be masked or disabled. A high logic level will cause a jump to location 6.5 * 8 = 0034H.

RST 5.5 – 4th priority and can be masked or disabled. A high logic level will cause a jump to location 5.5 * 8 = 002CH.

INTR – 5th priority and can be masked or disabled. A high logic level will cause a jump to specific location as follows:

When the interrupt request (INTR) is made, the CPU first completes its current execution. Provided no other interrupts are pending, the CPU will take the INTA pin low thereby acknowledging the interrupt. It is up to the hardware device that first triggered the interrupt, to now place the op-code of a RST n (n=0 to 7) instruction to the CPU to the service routine after the PC contains to be pushed in the stack.

You will notice that there are not many locations between vector addresses. What is normally done is that at the start of each vector address, a jump instruction (3 bytes) is placed, that jumps to the actual start of the service routine which may be in RAM. This way the service routines can be anywhere in program memory. The vector address jumps to the service routine. There is more than enough room between each vector address to put a jump instruction. Looking at the table above, there are at least 8 locations for each of the vectors except RST 5.5, 6.5, and 7.5. When actually writing the software, address 0000h will have a jump instruction that jumps around the other vector locations.

Besides being able to disable/enable all of the interrupts at once (DI / EI) ie: except **TRAP**, there is a way to enable or disable them individually using the SIM instruction and also, check their status using RIM.

There are other things about interrupts that we will cover as they come up, but this lesson was to get you used to the idea of interrupts and what they're used for in a typical system. It's similar to the scene where one is standing at a busy intersection waiting for the traffic light to change, when a person came up and tapped us on the shoulder and asked what time it was. It didn't stop us from going across the street, it just temporarily interrupted us long enough to tell them what time it was. This is the essence of interrupts. They *interrupt* normal program execution long enough to handle some event that has occurred in the system.

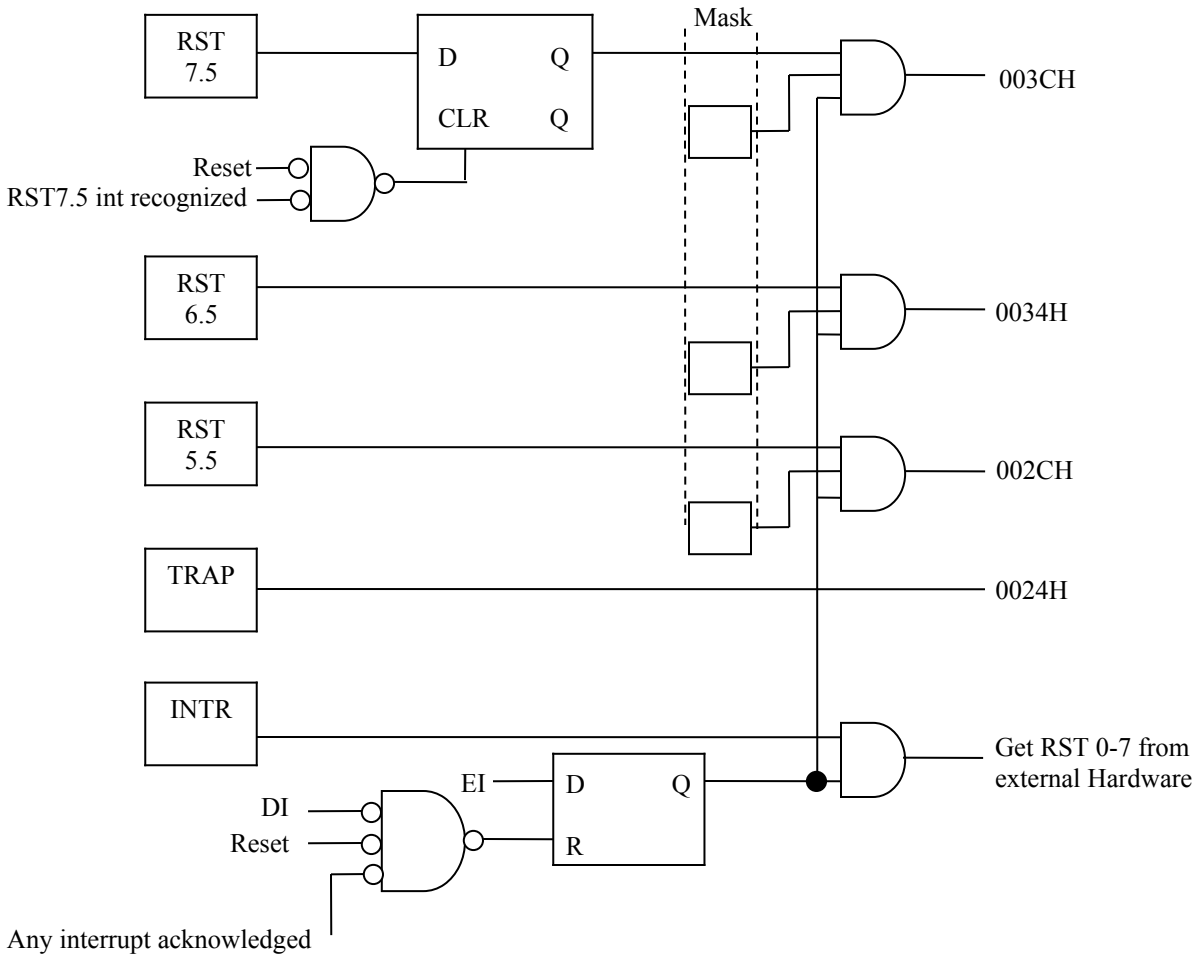
Polling, or scanning, is the other method used to handle events in the system. It is much slower than interrupts because the servicing of any single event has to wait its turn in line while other events are checked to see if they have occurred. There can be any number of polled events but a limited number of interrupt driven events. The choice of which method to use is determined by the speed at which the event must be handled.

The software interrupts are the instructions RST n, where n = 0 – 7. The value n is multiplied by 8 and the result forms an address that the program jumps to as it vector address ie: RST 4 would jump to location 4*8 = 32 (20H).

Interrupts and their starting addresses:

SL	Name	Address	Type	Priority	Maskable
1	RST 0	00H	Software		-
2	RST 1	08H	Software		-
3	RST 2	10H	Software		-
4	RST 3	18H	Software		-
5	RST 4	20H	Software		-
6	TRAP	24H	Hardware	Highest	No
7	RTS 5	28H	Software		-
8	RTS 5.5	2CH	Hardware	2nd highest	Yes
9	RST 6	30H	Software		-
10	RST 6.5	34H	Hardware	3rd highest	Yes
11	RST 7	38H	Software		-
12	RST 7.5	3CH	Hardware	4th highest	Yes
13	INTR	RTS 0 - 7	Hardware	Lowest	Yes

Internal organization of the interrupt system:



SIM – Set interrupt Mask

7	6	5	4	3	2	1	0
SOD	SDE	XXX	R7.5	MSE	M7.5	M6.5	M5.5

Set Mask RST 7.5, RST 6.5 and RST 5.5

Mask Set Enable

Reset RST 7.5

Ignored

Serial data enable, if 1, bit 7 is output serial output data latch

Serial output data; ignored if bit 6 is 0

This is a one byte instruction and can be used for three different purposes

1. To reset RST 7.5 flip-flop. If D4=1 then RST 7.5 is reset.
2. To set mask for RST 7.5, RST 6.5 and RST 5.5 interrupts. Bit D3 is a control bit and should be made 1 for D2, D1 and D0 to be effective. Logic 0 will enable and 1 will disable the corresponding interrupts
3. To output serial data. Bit D7 of the accumulator is sent out to the SOD pin if D6 is 1

RIM – Read Interrupt Mask

7	6	5	4	3	2	1	0
SID	I 7.5	I 6.5	I 5.5	IE	M7.5	M6.5	M5.5

Read Mask RST 7.5, RST 6.5 and RST 5.5: 1 = masked

Interrupt Enable Flag: 1 = enable

Pending Interrupt: 1 = pending

Serial output data

This is a single byte instruction and is used to check for pending interrupts. It performs the three following functions

1. To read interrupt masks
2. To receive Serial data
3. To identify pending interrupts.

History of Intel Microprocessors

The Intel introduced 4004, world's first microprocessor, in early 1970's. It was a 4-bit microprocessor and could address 4096 4-bit memory locations. 4004 instruction set contained only 45 instructions. It was fabricated by the then current state-of-the-art P-channel MOSFET technology. It could execute at the rate of 50 KIPs.

The evolution of 4-bit microprocessor ended with 4040 an upgraded version of 4004. 4040 operated at higher speed but lacked improvement in word size and addressing capability.

In 1971, Intel introduced 8008, the first 8-bit microprocessor. It could address up to 16K bytes and had 48 instructions. Its small memory size, low speed and instruction set limited its usefulness. Intel appreciated these limitations and came up with 8080 in 1973 – the first of the modern 8-bit microprocessors. 8080 could not only address 64K bytes memory and execute more instructions but also it executed them 10 times faster than 8008. In 1978, Intel introduced an updated version of 8080 – the 8085. The 8085 was the last 8-bit general purpose microprocessor developed by Intel. 8085 executed software even in higher speed. The main advantages of 8085 are its internal clock generator, internal system controller and higher clock frequency.

Intel 8086/88 Microprocessor

In 1978, Intel released the 8086 microprocessor; a year later, it released the 8088. Both devices were 16-bit microprocessors, which executed instructions in as little as 400 ns (2.5 MIPs). This was a major improvement in operating speed with respect to 8085. In addition, the 8086/88 addressed 1M bytes memory. The number of instructions increased from 246 in 8085 to well over 20,000 variations in 8086/88. Note that these microprocessors are called CISC (complex instruction set computers) because of the number and complexity of instructions. The other distinct feature found in the 8086/88 was the introduction of a small 6 or 4 bytes instruction cache or queue that pre-fetched a few instructions before they were executed. The queue sped the operation of many sequences of instructions and proved to be the basis for the much larger instruction caches found in modern computers.

The Programming Model

Before a low-level program (assembly and machine language) is written, the internal architecture of the microprocessor must be known. Program visible internal architecture of 8086/88 microprocessors is shown in Fig.1 below:

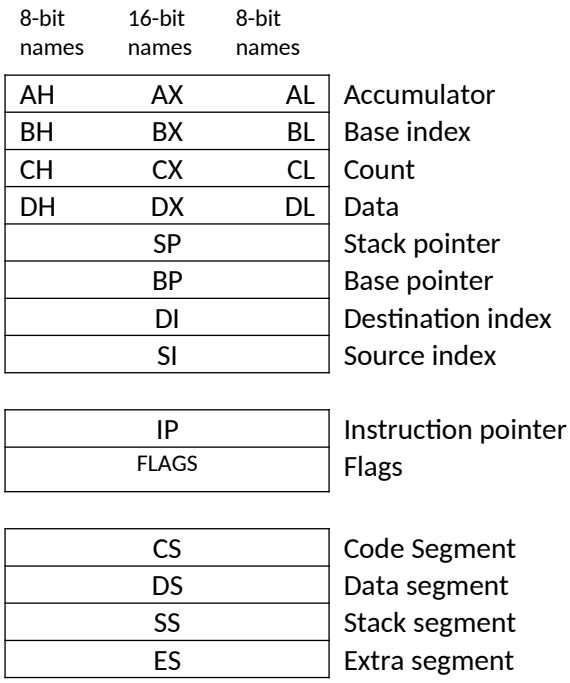


Fig.1: The Programming Model of the Intel 8086 through Pentium 4.

The Programming model contains 8 and 16-bit registers. The 8-bit registers are AH, AL, BH, BL, CH, CL, DH and DL and are referred by these two letter names in instructions. The 16-bit registers are AX, BX, CX, DX, SP, BP, SI, DI, IP, FLAGS, CS, DS, SS and ES.

Some registers are general purpose or multi-purpose, and some are dedicated and have special purposes. The multi-purpose registers are AX, BX, CX, DX, BP, DI and SI. These registers hold various data and are used for almost any purpose as indicated by the program.

Multipurpose Registers

AX (Accumulator): AX can be used as a 16-bit register or AL/AH can be used as 8-bit registers. The accumulator is used for instructions such as multiplication, division, input and output and some of the adjustment instructions. For these instructions, the accumulator has a special purpose. AL and AX act as accumulator for 8-bit and 16-bit operations respectively.

BX (Base index): This register can be addressed as BL, BH or BX. The BX register sometimes holds the offset address of a memory data.

CX (Count): CX is a general-purpose register and can be referred as CL, CH or CX. In some instructions, CL or CX holds the count for repetitive/iterative operations. Instructions that use a count are the repeated string instructions (REP/REPE/REPNE); and shift, rotate and LOOP instructions. The shift and rotate instructions use CL and LOOP and repeated string instructions use CX as count.

DX (Data): DX is also a general-purpose register and can be used as DL, DH or DX. For multiplication and division instructions, DX is used to hold a part of operand/result. DX is also used as I/O pointer.

BP (Base pointer): BP is used to hold the offset address of a memory location in Stack Segment.

DI (Destination index): DI can be used to hold the offset address of a general memory (Data Segment). For string instructions, it holds the offset address of the destination string in Extra Segment.

SI (Source index): SI can be used to hold the offset address of a general memory (Data Segment). For string instructions, it holds the offset address of the source string in Data Segment.

Special Purpose Registers

Special purpose registers include Instruction Pointer (IP), Stack Pointer (SP), Flag Register (FLAGS), and segment registers CS, DS, SS and ES.

IP (Instruction Pointer): IP holds the offset address (in Code Segment) of the next instruction to be executed. It can be modified by a call or a jump instruction.

SP (Stack Pointer): SP holds the offset address of the stack memory (Stack Segment).

FLAGS: Flags indicate the condition of the microprocessor and control its operations. 8086/88 have a 16-bit flag register. Different flags for 8086/8088 microprocessor are as follows:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
--	--	--	--	O	D	I	T	S	Z	--	AC	--	P	--	C

Fig.2: Flags of 8086/88 Microprocessor

Flag register has 9 active flags. The first five viz. C (carry), P (Parity), AC (auxiliary carry), Z (zero) and S (sign) are 8085 like flags and the remaining four are new in 8086/8088. They are T (trap), I (interrupt), D (direction) and O (overflow).

C (Carry): This flag holds the carry after addition and borrow after subtraction.

P (Parity): Parity is logic 0 for odd parity and logic 1 for even parity. Parity is the count of ones in a number expressed as even or odd. For example, a binary number has four 1s, so the parity is even. A number having no 1s, has the even parity.

AC (auxiliary carry): The auxiliary carry holds the carry after addition or borrow after subtraction between bit position 3 and 4 of the results. This is used for adjustment of the result for BCD operations.

Z (zero): This flag indicates whether the result is zero or non-zero. It will be set if the result is zero and reset if non-zero.

S (sign): Sign flag indicates polarity of the result. Set if negative and reset if positive. So, it is same as the most significant bit (msb) of the result.

T (trap): The trap flag enables trapping (debugging) through an on-chip debugging feature. If T=1, microprocessor interrupts after execution of every instruction and goes to a service routine so that the programmer can verify the contents of registers, memory etc.

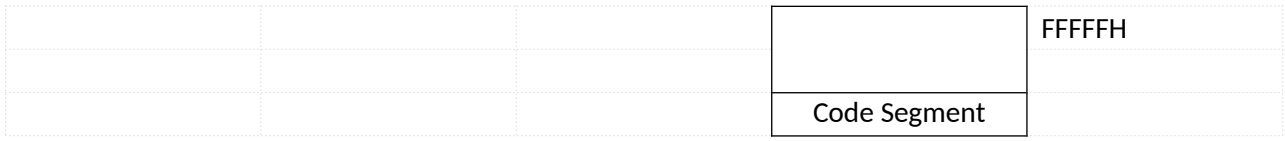
I (interrupt): The interrupt flag enables or disables the INTR input pin. If I=1, INTR pin is enabled. The state of the Interrupt flag can be set or reset by STI (set interrupt flag) and CLI (clear interrupt flag) instructions.

D (direction): The direction flag determines auto increment or auto decrement of SI and DI registers for string operations. If D=1, the registers are automatically decremented and if D=0, they will automatically be incremented. STD (set direction) and CLD (clear direction) instructions are used to set and reset this flag.

O (overflow): Overflows occur when sign numbers are added or subtracted. An overflow indicates that the result has exceeded the capacity of destination register/memory. For example, if 7FH (+127) is added, using an 8-bit addition, with 01H (+1), the result will be 80H (-128) - this is an overflow condition and indicated by the overflow flag becoming set. For unsigned operation the overflow flag is ignored.

Segment Registers: Though the 8086/88 has 1 MB memory, all are not active at any point of time. The whole memory is partitioned into 16 segments of 64 KB each. Out of these 16 segments, only 4 segments are active at a time. The four active memory segments are: (i) Code Segment, an area to keep aside for program codes, (ii) Data Segment, to store the data generated/to be used by the program, (iii) Stack Segment to keep aside for stack area and (iv) Extra Segment to store data again. Starting addresses, sometimes called base address, of these four segments are pointed out by four segment registers CS, DS, SS and ES. Addressing within a segment is done by supplying the displacement/offset of the memory with respect to this base address through index and pointer registers viz. SI, DI, BX, SP, BP and IP.

Fig.3 illustrates the segmentation and active segments of memory for 8086/88 microprocessor.



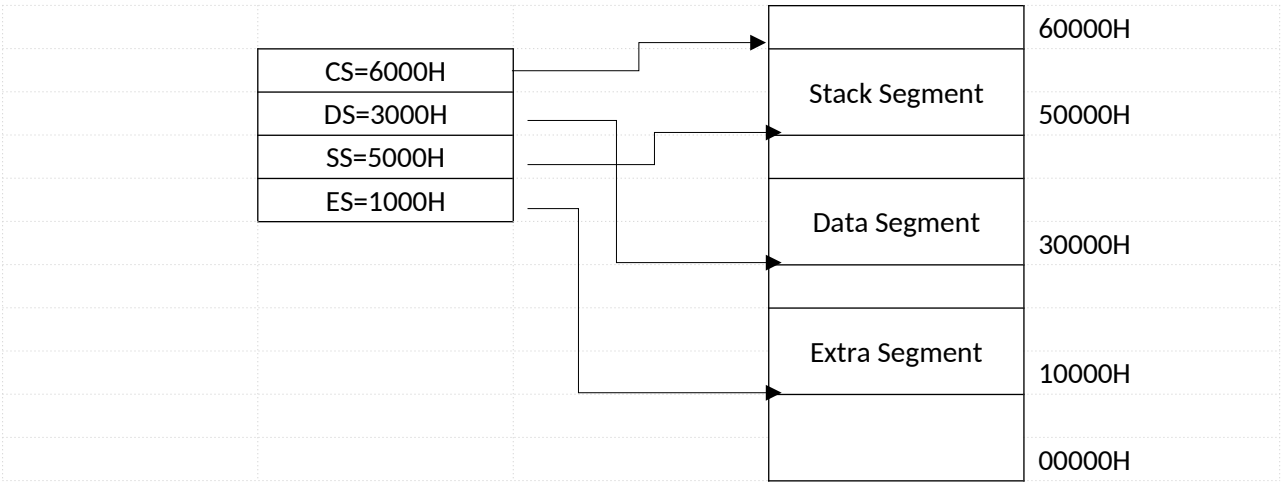


Fig.3: Memory Segmentation and active segments for 8086/88 microprocessor

Actual address of a memory location is 20-bit wide. But the segment registers can store only 16-bit data. So, a default 0H is placed at the right of a segment register to indicate the starting address of the corresponding segment. The 16-bit displacement to point any memory location within a segment is then supplied through pointer registers or index registers or 8-bit/ 16-bit displacements or any suitable combination of these three. Thus, the actual 20-bit address can be formed as follows:

20-bit actual address = Segment register x 10H + 16-bit displacement/offset

The 20-bit actual address is called *Physical Address (PA)*, the 16-bit displacement is called *Offset address* and the content of the segment register is called the *segment address*.
An alternative way of representing physical address is the *segment base:offset*. A segment base and an offset describe a *logical address* in 8086/88 microprocessor system.

Default segment and offset combinations: The source of the offset depends on which segment is used for addressing. For code segment, IP is the default source of the offset address. Thus, CS:IP gives the physical address of the current instruction to be fetched for execution. For string operations, SI is the source of default offset for data segment and DI is the source of default offset for extra segment. Other than string operations, SI, DI and BX are used to supply the offset address for data segment. SP and BP are the default sources of offset address for stack segment. A provision called the *segment override prefix* is used to change the segment from which the variable is accessed. A data is always accessed from the data segment. If one wants to access data from other segments one must specify the new segment in the instruction itself through segment override prefix. For example,

```
MOV AX, [BX]          ; Physical Address, PA = DS:BX, default segment is DS
ADD AX, [SI]          ; PA = DS:SI, default segment is DS
ADD AX, CS:[BX]               ; Example of Segment override prefix, PA = CS:BX
```

Internal Architecture of 8086/88

As shown in Fig.4, the internal architecture of 8086/88 is divided into two independent functional units called *Bus Interface Unit (BIU)* and *Execution Unit (EU)*. BIU organizes and controls the operations of address and data buses which include address generation, instruction fetching, data read from memory and IO and data write to memory and IO devices. Thus, BIU handles all transfer of data and addresses on the bus for Execution Unit. On the other hand, the Execution Unit tells the BIU where to fetch instructions and data from, decodes instructions and executes them.

The EU contains the control units which direct the internal operations. A decoder in EU translates the instruction into a series of actions which EU carries out. EU has a 16-bit *Arithmetic and Logic Unit (ALU)* which can add, subtract, AND, OR, XOR, increment, decrement, complement or shift binary numbers.

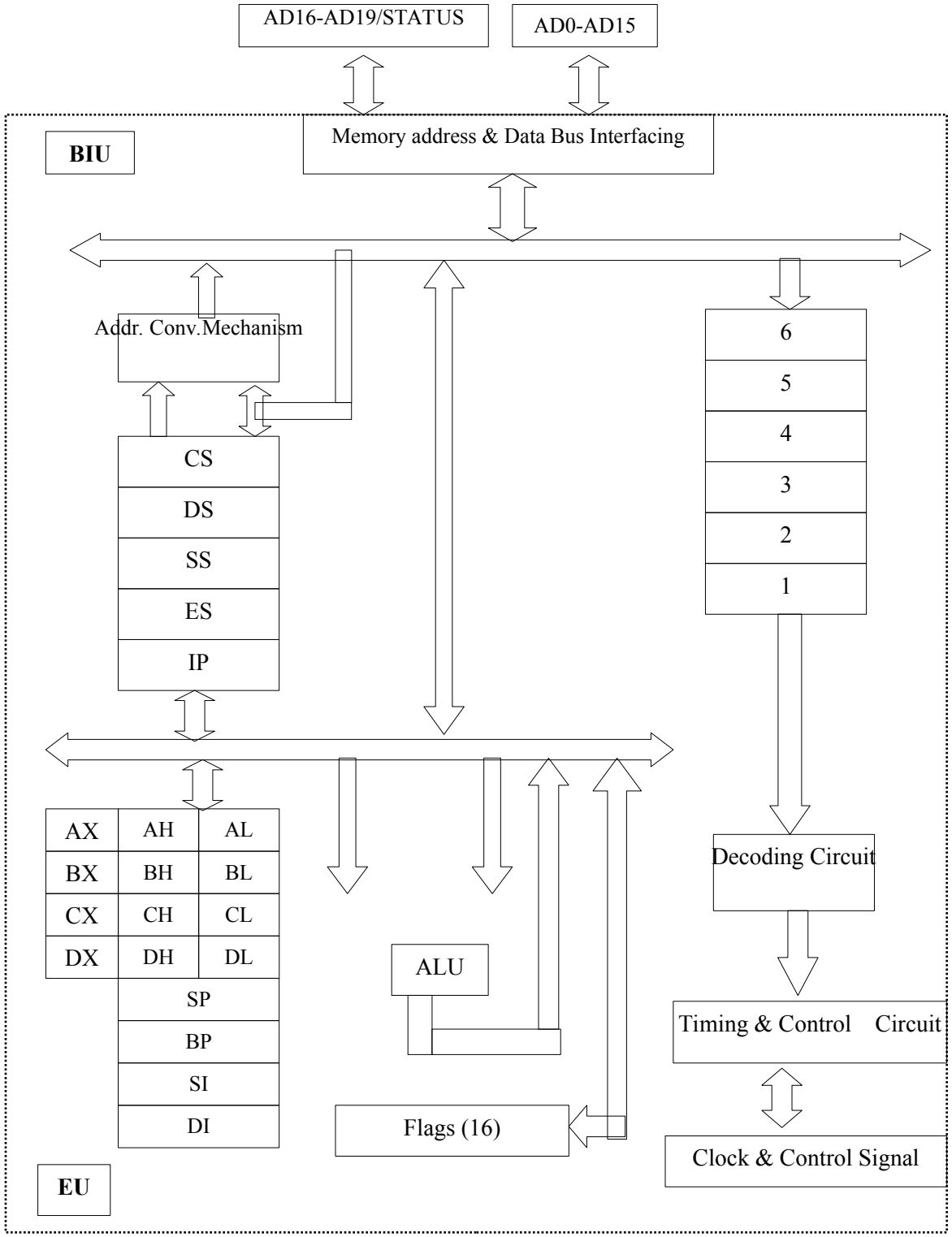


Fig.4: Internal Architecture of 8086/88 Microprocessor

While EU decodes or executes an instruction which does not require the use of the buses, the BIU fetches up to 6 bytes for 8086 and 4 bytes for 8088 for the following instructions and stores them in an internal *first-in-first-*

out (FIFO) register called *queue* or *cache*. After the execution of the current instruction, the EU does not fetch the next instruction from memory; rather it simply reads it from the internal queue. This is much faster than instruction fetching from memory. This *pre-fetch and queue scheme* greatly speeds up the operation of the 8086/88. This fetching of next instruction while the current instruction is executed is called *pipelining*. The architecture supporting pipelining is called *pipelined architecture*.

Microprocessor without Pipelined Architecture									
	Fetch1	Decode1	Execute1	Fetch2	Decode2	Execute2	Fetch3	Decode3	Execute3
Microprocessor with Pipelined Architecture									
BIU	Fetch1	Fetch2	Fetch2	Fetch3	Fetch3				
EU		Decode1	Execute1	Decode2	Execute2	Decode3	Execute3		
							Save in time	◄———	——►

Fig. 5: A comparison of operations of microprocessors having simple and pipelined architecture showing faster instruction execution in case of pipelined architecture

Instruction Templates

Number of instructions is huge in 8086/88 and thus it is not possible to provide any concise list of instructions like 8085 for programmer's reference. Ways an operand can be expressed in an 8085 instruction is limited, but it is much larger in case of 8086/88. This leads to a wide variation of each 8086/88 instruction. For example, there are 32 ways to specify the source operand in an instruction such as MOV CX, source. The source can be any one of eight 16-bit registers, or a memory location specified by any one of 24 memory addressing modes. If CX is made source, also there are 32 ways one can express the destination. So MOV instruction, having CX as one of the operands, can be written in 64 different ways. Likewise, another 64 codes are required for MOV using CL as source or destination and 64 more codes for MOV with CH as source or destination. Thus, MOV instruction with CX (including CL and CH) has 192 variants and it is impracticable to provide a list of all possible instructions of 8086/88. Instead, a template for each basic instruction can be used to generate the codes of the instructions of 8086/88. Fig. 6 below describes such a template of MOV instruction:

Byte 1								Byte 2								Byte 3	Byte 4
1	0	0	0	1	0											LOW DISPLACEMENT	HIGH DISPLACEMENT
OPCODE						D	W	MOD		REG			R/M				
																OR	
																DIRECT ADDRESS LOW BYTE	DIRECT ADDRESS HIGH BYTE

OPCODE	– Operation Code
D	– Direction TO/FROM REG; 0 = FROM, 1 = TO
W	– Byte/Word Data; 0 = Byte, 1 = Word
MOD & R/M (5 bits)	– ADDRESSING MODE
REG	– REGISTER SELECT

Fig. 6: Coding Template of MOV instruction

MOD and R/M bit patterns for 8086/88 instructions are shown in Fig. 7. Fig. 8 shows the bit pattern for REG field.

MOD / R/M	00	01	10	11	
				W=0	W=1
000	[BX]+[SI]	[BX]+[SI]+d8	[BX]+[SI]+d16	AL	AX
001	[BX]+[DI]	[BX]+[DI] +d8	[BX]+[DI] +d16	CL	CX
010	[BP]+[SI]	[BP]+[SI] +d8	[BP]+[SI] +d16	DL	DX
011	[BP]+[DI]	[BP]+[DI] +d8	[BP]+[DI] +d16	BL	BX
100	[SI]	[SI] +d8	[SI] +d16	AH	SP
101	[DI]	[DI] +d8	[DI] +d16	CH	BP
110	d16 (direct address)	[BP] +d8	[BP] +d16	DH	SI
111	[BX]	[BX]+d8	[BX]+d16	BH	DI
	MEMORY ADDRESSING			REGISTER ADDRESSING	

Fig. 7: MOD and R/M bit patterns in instruction template of 8086/88 microprocessor

General Purpose Registers			**Segment Registers**	
REG bit pattern	8-bit Registers (W=0)	16-bit Registers (W=16)	CODE	SEGMENT REGISTER
000	AL	AX	00	ES
001	CL	CX	01	CS
010	DL	DX	10	SS
011	BL	BX	11	DS
100	AH	SP		
101	CH	BP		
110	DH	SI		
111	BH	DI		

Fig. 8: Registers codes

Example-1:

MOV AX, BX => 100010 1 1 || 11 000 011 => 8B C3H ; “TO” REG

-OR- => 100010 0 1 || 11 011 000 => 89 D8H ; “FROM” REG

Example-2:

MOV AX, [BX][SI]1234H =>100010 1 1 || 10 000 000 || 0011 0100 || 0001 0010 => 8B 80 34 12H

The coding template of MOV instruction for segment override prefix is shown in Fig. 9 below. It is a 3-byte code and an extra byte will be added for new segment before byte1 and byte2 of normal MOV instruction.

Byte 1 Segment Override Prefix								Byte 2								Byte 3							
0	0	1			1	1	0	1	0	0	0	1	0	D	w								

SEG REG	OPCODE			MOD	REG	R/M

Fig. 9: MOV instruction coding template for segment override prefix

Example-3:

MOV CS:[BX],DL => 001 01 110 || 100010 0 0 || 00 010 111 => 2E 88 17H

Addressing Modes

Microprocessors operate on data which are commonly known as operands. Operand in an instruction may be the content of a register, a memory or it can be a constant. There are numerous ways to express them in an instruction and are known *as data-addressing modes*. Similarly, program address, in case of control transfer instructions, may be expressed in different ways giving rise to *program memory-addressing modes*.

Data-Addressing Modes: The five fields of an assembly language instruction are as shown in Fig.10.

[Label:] OPCODE [Destination] [Source] [; Comments]

Fig. 10: General format of an assembly language instruction.

The fields within brackets are optional, so an instruction may not have all the fields. "*Label*" is used to refer an instruction from other parts of an assembly language program and is ended with a colon. It is case sensitive. No other fields are case sensitive. "*Opcode*" is the short form of the operation to be performed by the microprocessor and sometimes called "*Mnemonic*" meaning memory-aid as it helps us to remember the operation. "*Destination*" and "*Source*" are the two operands and indicate the data flow direction. Data flows from source to destination. "*Comment*" always starts with a semicolon. Assembler always ignores anything written after a semicolon. Comment is used for documentation purpose only which helps us to understand the purpose of use an instruction or a group of instructions or the whole program in future. A comment may appear either at the end of an instruction or even in a new line.

Various types of Data-addressing modes are as follows:

1. Register Addressing — It refers different ways of expressing registers, byte or word, in an instruction. For example,
   MOV AX, BX ; This instruction copies the content of BX register to AX register
   ADD AL, CL ; This instruction adds the contents of AL and CL registers and puts the result in AL

2. Immediate Addressing — It refers that the source is a constant data often called an immediate data as it appears in memory immediately after the opcode. For example,
   MOV AL, 12H ; This instruction places 12H in AL register
   ADD AX, 1234H ; This instruction adds 1234H with AX and accumulates result in AX

3. Memory Addressing — It refers different ways of indicating a memory location storing an operand in an instruction. Memory location may be indicated directly along with the opcode. Then it

is called *direct addressing*. It may also be expressed indirectly through a pointer register BX, BP, SP, SI, DI, DX or their suitable combinations as indicated in Fig.7 giving rise to different *indirect addressing*.
Note: Both the operands in an instruction can never be memory except string instructions.

3A. Direct Addressing

Offset address of the memory operand is present in the instruction itself. For example,
MOV AL, [1234H] ; Here a byte data is copied from memory in data segment having offset address 1234H

3B. Register-Indirect Addressing

In this case, offset address of a memory operand is supplied through one of index or base registers i.e. SI, DI, BX. For example,
MOV CL, [BX], this instruction loads the content of a memory location in data segment whose offset address is kept BX into CL. Other examples are OR AX, [SI]; ADD [DI], DX; INC WORD PTR [BX] etc.

3C. Base-plus-index addressing

Base-plus-index addressing refers a memory operand addressed by a base register (BP or BX) plus an index register (SI or DI). For example,
MOV AL, [BX][SI], this instruction copies the content of a memory location in data segment whose offset address is the sum of BX and SI. Other examples are ADD AX, [BP][DI], INC BYTE PTR [BX][DI] etc.

3D. Register relative Addressing

Register relative addressing refers a memory operand whose offset address is supplied through an index or a base register plus a displacement, 8-bit or 16-bit. For example,
MOV AH, [BX]12H ; ADD AX, [SI]1000H etc.

3E. Base relative-plus-index addressing

In this case, the offset of the memory operand is supplied through a base register plus an index register plus an 8-bit or a 16-bit displacement. For example,
MOV AX, [BX][SI]12H ; ADD BX, [BP][SI]1234H etc.

Program Memory Addressing: It is used with JMP (jump) and CALL instructions and is of three distinct forms – *direct, relative and indirect*.

Direct Program Memory Addressing

In case of *direct program memory addressing*, JMP and CALL instructions take the execution control beyond the current code segment called inter-segment CALL or JMP. In this case, the address is stored with the opcode by two 16-bit values, one for code segment and the other for offset. The direct jump or call are often called ***far*** jump or call. For example,
HERE: JMP FAR BEGIN, in this case, the labels HERE and BEGIN are in different code segments, the machine code of this instruction considering HERE as 2000:100H and BEGIN as 5000:1234H is:

	Opcode	Offset Low	Offset High	Segment Low	Segment High
2000:100H	EA	34	12	00	50
2000:105H					

Relative Program Memory Addressing

The term *relative* means "relative to the instruction pointer (IP)". Therefore, the displacement (8-bit or 16-bit signed numbers) given in the instruction will be added with IP to determine the new value of IP where from the next instruction will be taken for execution. Relative jump and call are always intra-segment. 8-bit displacement can take the execution backward by 128 bytes and forward by 127 bytes and are known as ***short*** jump and call. On the other hand, 16-bit displacement can take the control 32768 bytes backward and 32767 bytes forward with respect to current IP position and are called ***near*** jump and call. For example,

```
1000:0100   B8 00 20          BEGIN:  MOV AX, 2000H
1000:0103   BB 00 03                  MOV BX, 300H
1000:0106   B2 00                     MOV DL,00H
1000:0108   05 23 F1                  ADD AX, F123H
1000:010B   73 02                     JNC SKIP          ; Short Jump, 2 bytes forward
1000:010D   FE C2                     INC DL
1000:010F   89 07             SKIP:   MOV [BX],AX
1000:0111   88 57 02                  MOV [BX]02H,DL
.
.
1000:0200   E9 FD FE                  JMP BEGIN         ; Near Jump, 258B backward
1000:0203   EA 00 01 00 10            JMP FAR BEGIN     ; Inter-segment Jump
```

Note: Conditional jumps are short jumps.

Indirect Program Memory Addressing

In case of indirect program memory addressing, jump and call locations are supplied indirectly through registers or memory pointed out by registers. For example,

```
1000:0100   B8 00 02          BEGIN:  MOV AX, 200H
1000:0103   BB 34 12                  MOV BX, 1234H
1000:0106   FF E0                     JMP AX            ; IP modified by AX
1000:0108   FF 27                     JMP [BX]          ; IP modified by content of
                                                        ; memory DS:BX
1000:010A   FF 67 12                  JMP [BX]12H       ; DS:BX+12H
```

Finding the Right Instruction

For finding the right instruction in right place we must know the instructions in groups of functional operations like data transfer operations, logical operations, arithmetic operations, bit manipulation operations, string operations, program execution transfer operations, processor control operations etc. The important instructions under these groups are as follows:

MNEMONIC WITH OPs	DESCRIPTIONS	OP1 (DES OP)	OP2 (SOU OP)	OPERATION/EXAMPLE
DATA TRANSFER GROUP:				
General-purpose byte or word transfer instructions:				
MOV	Copy bytes/words	m8, r8, m16, r16	m8, r8, m16, r16, d8, d16	OP1←OP2
PUSH	Save a word on stack	r16, m16	-	(SP) ←OP1
POP	Copy word from stack	r16, m16	-	OP1 ← (SP)
XCHG	Exchange	m8, r8, m16, r16	m8, r8, m16, r16	OP1←→ OP2
XLAT	Translate using look-up table	-	-	AL ← (BX+AL); BX will not be modified.
Simple input and output port transfer instructions:				
IN	Read data from a port	AL, AX	8p8, 8p16, DX	AL ← 8p8/DX AX ← 8p16/DX
OUT	Write data to a port	8p8, 8p16, DX	AL, AX	8p8/DX← AL 8p16/DX ←AX
Special Address transfer instructions:				
LEA	Load effective address (offset addr.) in register	R16	m16	
Example of LEA: 1000:0100 3412 DATA1: DW 1234H 1000:0102 CDAB DATA2: DW ABCDH 1000:0104 2E 8D 36 00 01 BEGIN: LEA SI, DATA1 ; SI = 0100H 1000:0109 BE 02 01 MOV SI, OFFSET DATA2 ; SI = 0102H				
LDS/LES	Loads DS/ES and mem ptr with the 32-bit content of data segment memory at m16	BX, BP, SI, DI	m16	
Example of LDS/LES DATA1: DW 1234H, 2000H DATA2: DW ABCDH, 3000H BEGIN: LES SI, DATA1 ; ES=2000H, SI=1234H LDS DI, DATA2 ; DS=3000H, DI=ABCDH				
Flag transfer instructions:				
LAHF	Load lower byte of Flag register to AH register			
SAHF	Store content of AH register to lower byte of Flag Register.			
PUSHF	Push Flag Register in Stack.			
POPF	Retrieve Flag Register from Stack.			
ARITHMETIC INSTRUCTIONS				

Addition instructions:				
ADD	Arithmetic Addition of two byte/word operands	m8, r8, m16, r16	m8, r8, m16, r16, d8, d16	OP1←OP1+OP2 (Both the operands cannot be memory)
ADC	Arithmetic Addition of two byte/word operands plus previous carry	m8, r8, m16, r16	m8, r8, m16, r16, d8, d16	OP1←OP1+OP2+CY (Both the operands cannot be memory)
INC	Increment operand by 1	m8, r8, m16, r16		OP1 ← OP1+1
AAA	Adjust Accumulator after ASCII addition			After adding ASCII data, AAA gives correct unpacked BCD result.
Example of AAA: Assume AL=0011 1000, ASCII 8 BL=0011 0111, ASCII 7 ADD AL, BL ; AL=0110 1111 = 6EH which is incorrect BCD, Correct one should be 15 AAA ; AL=0000 0101, Unpacked BCD 5, CF=1 indicates the answer is 15 decimal.				
DAA	Adjust Accumulator after BCD addition			After adding packed BCD data (one in AL), DAA gives correct packed BCD result in CF & AL
Subtraction instructions:				
SUB	Subtraction between operands	m8, r8, m16, r16	m8, r8, m16, r16, d8, d16	OP1←OP1-OP2 (Both the operands cannot be memory)
SBB	Subtraction between operands minus borrow (Carry flag).	m8, r8, m16, r16	m8, r8, m16, r16, d8, d16	OP1←OP1-OP2-CY (Both the operands cannot be memory)
DEC	Decrement operand by 1	m8, r8, m16, r16		OP1 ← OP1-1
NEG	Finds 2's complement	m8, r8, m16, r16		OP1←2's Complement of OP1
CMP	Compares between two operands by subtracting op2 from op1. Result is not stored, only flags are modified.	m8, r8, m16, r16	m8, r8, m16, r16, d8, d16	FLAGS ←OP1-OP2
AAS	Adjust Accumulator after ASCII subtraction			
DAS	Adjust Accumulator after BCD subtraction			
Multiplication Instructions:				
MUL	Multiplication between two unsigned 8-bit/16-bit data, one stored in AL/AX and other in register/memory.	m8, r8, m16, r16		AX ← AL*OP1 or DX-AX ← AX*OP1
IMUL	Multiplication between two signed 8-bit/16-bit data, one stored in AL/AX and other in register/memory.	m8, r8, m16, r16		AX ← AL*OP1 or DX-AX ← AX*OP1
AAM	Adjust result after multiplying two ASCII digits (one must in AL) and gives unpacked BCD results in AX.			
Division Instructions:				
DIV	Divides an unsigned 16-bit/32-bit number by an unsigned 8-bit/16-bit number. Dividend is at AX/DX-AX and divisor in r8, m8/r16,	r8, r16, m8, m16		AX ÷ OP1 (r8/m8), AL=Quotient, AH=remainder or DX-AX ÷ OP1 (r16/m16), AX=Quotient, DX=remainder

	m16. Remainder will be in AH/DX.			
IDIV	-do- dealing with signed integers			
AAD	Unlike other adjustment instructions, this instruction appears before a division. It converts an unpacked 2-digit BCD number in AX into a 16-bit binary number which is then divided by an unpacked number to generate a single digit quotient in AL and remainder in AH.			
CBW	It converts a signed byte data in AL into a signed word data in AX (2's complement form). This is useful when data lengths are not as per the requirements of IDIV and IMUL instructions.			
CWD	It converts a signed word data in AX into a signed double word data in DX-AX (2's complement form). This is useful when data lengths are not as per the requirements of IDIV and instruction.			
BIT MANIPULATION INSTRUCTIONS				
Logical Instructions:				
NOT	Finds 1's complement	r8, r16, m8, m16		
AND	Finds bit-by-bit logical AND operation between two operands	m8, r8, m16, r16	m8, r8, m16, r16, d8, d16	OP1 ← OP1 . OP2
OR	Finds bit-by-bit logical OR operation between two operands	m8, r8, m16, r16	m8, r8, m16, r16, d8, d16	OP1 ← OP1 + OP2
XOR	Finds bit-by-bit logical XOR operation between two operands	m8, r8, m16, r16	m8, r8, m16, r16, d8, d16	OP1 ← OP1 ⊕ OP2
TEST	It performs logical AND operation between two operands, updates flags but result not stored in any of the operands.	m8, r8, m16, r16	m8, r8, m16, r16, d8, d16	FLAGS ← OP1 . OP2
Shift Instructions:				
SHL/SAL	Shift left operand one bit position/no. of bits position in CL. A 0 is inserted at lsb and msb goes to CY.	m8, r8, m16, r16	1, CL	
SHR	Shift (logical) operand right by CL times or once. A 0 is inserted at msb and lsb goes to CY.	m8, r8, m16, r16	1, CL	
SAR	Shift (arithmetic) operand right by CL times or once. Sign bit is inserted at msb and lsb goes to CY.	m8, r8, m16, r16	1, CL	
Rotate Instructions:				
ROL	Rotate the operand left once or by CL times. msb goes to CY and lsb.	m8, r8, m16, r16	1, CL	
ROR	Rotate the operand right once or by CL times. lsb goes to CY and msb.	m8, r8, m16, r16	1, CL	
RCL	Rotate through CY the operand left once or by CL times. msb goes to CY and CY to lsb.	m8, r8, m16, r16	1, CL	
RCR	Rotate through CY the operand right once or by CL times. lsb goes to	m8, r8, m16, r16	1, CL	

	CY and CY to msb.			
STRING ISTRUCTIONS				
A string is a series of bytes or words stored in successive memory locations. In the list a "/" is used to separate different mnemonics for the same instruction. A "B" is used to identify the byte string and a "W" is used to identify the word string. For string instructions, SI and DI will automatically be incremented if DF=0 and decremented if DF=1. SI/DI will be incremented or decremented by 1 for byte string and by 2 for word string.				
REP	It is an instruction prefix. The string instruction next to REP will be repeated CX times.			
REPE/REPZ	This is a variant of REP. The string instruction next to REPE/REPZ is repeated until CX=0 or ZF=1.			
REPNE/REPNZ	This is the other variant of REP. The string instruction next to REPNE/REPNZ is repeated until CX=0 or ZF=0.			
MOVS/MOVSB/MOVSW	Copies a string with byte/word data from data segment memory pointed out by SI (DS:SI) to extra segment pointed out by DI (ES:DI). The size of the string is indicated by CX.			
COMPS/COMPSB/COMPSW	Compares two strings with byte/word data stored at DS:SI and ES:DI. The instruction is normally used with REPE/REPNE, this causes the search to continue as long as an equal/not equal condition exists and CX becomes not zero.			
INS/INSB/INSW	Reads string from an Input port.			
OUTS/OUTSB/OUTSW	Writes string to an output port.			
SCAS/SCASB/SCASW	It scans a string in extra segment addressed by DI to find a match with AL/AX			
LODS/LODSB/LODSW	This instruction loads AL/AX with data stored in Data Segment pointed out by SI.			
STOS/STOSB/STOSW	This instruction stores AL/AX to memory at extra segment pointed out by DI.			
PROGRAM EXECUTION TRANSFER INSTRUCTIONS:				
These instructions are used to instruct the 8086/88 to start from a new location, rather than continuing in sequence.				
Unconditional transfer instructions:				
CALL	This instruction transfers the program execution to a subroutine. The subroutine may be within the current code segment or may be outside. Within segment, subroutine is called through CALL NEAR and outside the current code segment, subroutine is called through CALL FAR. When 8086/88 executes a near call instruction, it pushes the offset address of the instruction next to CALL in stack and when executes far call, it pushes segment as well as offset addresses of the next instruction in stack. Near call is also called intra-segment call and far call is called as inter-segment call.			
RET	Return instruction appears as the last instruction of the subroutine. This instruction retrieves back the value of IP or IP & CS from stack depending upon near and far call.			
JMP	This instruction transfers the execution to some other address. Like call, jump can also be near jump or far jump.			
Conditional transfer instructions:				
These instructions are often used after a compare instruction. The terms ***below*** and ***above*** refer to unsigned binary numbers. ***Above*** means larger in magnitude. The terms ***greater than*** or ***less than*** refer to signed binary numbers. ***Greater than*** means more positive. All the conditional jumps are short jumps meaning the destination address will be within 128 bytes backward and 127 forward from the current value of IP.				
JA/JNBE	Jump if above/not below equal			
JAE/JNB	Jump if above equal/not below			
JB/JNAE	Jump if below/not above equal			
JC	Jump if carry (set)			
JE/JZ	Jump if equal/zero			
JG/JNLE	Jump if greater/not less equal			
JGE/JNL	Jump if greater equal/not less			
JL/JNGE	Jump if less/not greater equal			
JLE/JNG	Jump if less equal/not greater			
JNC	Jump if no carry (reset)			
JNE/JNZ	Jump if not equal/not zero			
JNO	Jump if no overflow			

JNP/JPO	Jump if no parity (parity bit is reset)/jump if parity odd	
JNS	Jump if no sign (positive)	
JO	Jump if overflow (overflow flag set)	
JP/JPE	Jump if parity (parity bit set)/even parity	
JS	Jump on sign (negative)	
Iteration Control instructions:		
These instructions are used to repeat a set of instructions for a few times. CX must be loaded with the number of iterations beforehand. After execution of each LOOP instruction, CX is decremented by 1 and execution will jump to the destination specified after LOOP instruction. When CX=0, execution goes to the instruction next to LOOP. LOOP is an unconditional loop. Other variants of loop i.e. LOOPE or LOOPNE are conditional loops where loop may be stopped depending upon other flag condition even before CX=0.		
LOOP	Loop CX times	
LOOPE/LOOPZ	Loop until CX=0 or ZF=1	
LOOPNE/LOOPNZ	Loop until CX=0 or ZF=0	
JCXZ	Jump if CX=0	
Interrupt instructions:		
INT		
INTO		
IRET		
PROCESSOR CONTROL INSTRUCTIONS		
Flag set/clear instructions:		
STC	Set carry flag	
CLC	Clear carry flag	
CMC	Complement carry flag	
STD	Set direction flag	
CLD	Clear direction flag	
STI	Set interrupt flag	
CLI	Clear interrupt flag	
External Hardware Synchronization Instructions:		
HLT	Halt	
WAIT	Wait	
ESC		
LOCK		
No operation instruction:		
NOP	No operation	

Assembly Language Programming

- The sequence of commands used to tell a microprocessor what to do is called a program. The commands are called instructions.
- Some part of it is called Monitor Program or Operating System and the other is called User Program or Application Program.
- Operating Program basically organizes the inputs and the outputs with the system.
- User Program supplies the variables and their formats.
- Microprocessor can only understand the instructions coded in binary called machine language.
- Machine language is difficult, if not impossible, for human to handle.
- Assembly language was developed to provide *mnemonics* plus other features to make the programming easy, faster and less prone to error.
- Instructions abbreviated in English letters to represent the operation to be performed by the microprocessor are called *mnemonics* meaning memory-aid.
- Assembly language programs must be translated into machine code by a program called *assembler*.
- Assembly language is referred to as a low-level language because it deals directly with the internal structure of the microprocessor.
- High level language like C, BASIC, Java etc. can also be used for programming.
- High level language is converted into machine code by a program called *compiler*.

Structure of Assembly Language:

- An assembly language program consists of a number of assembly language instructions used to tell the CPU what to do.
- It also contains instructions giving direction to the assembler called *directives*.
- For example, MOV, ADD instructions are commands to the CPU whereas ORG, END are directives to the assembler. ORG followed by an address tells the assembler to place the op-code at that memory location while END indicates the end of the source code.
- An assembly language instruction consists of five fields:

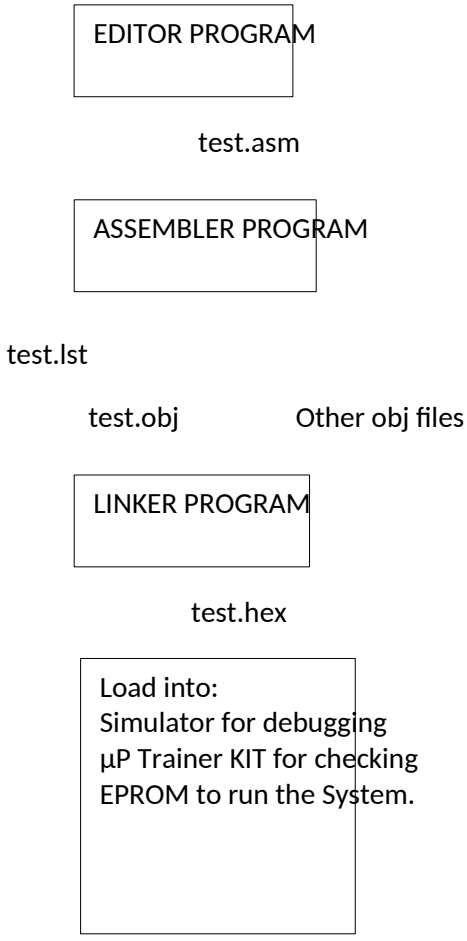
[label:] **mnemonic** **[destination operand]** **[source operand]** **[; comment]**

- Brackets indicate that a field is optional and not all lines have them. Brackets should not be typed in while typing an instruction.
- The label field allows the program to refer to a line of code by name. There are rules for writing the label like type and maximum number of characters, starting characters etc. which is assembler specific.
- The assembly language mnemonic together with the operands forms the command for the CPU. For example MOV AL, BL. MOV is the mnemonic, which is the abbreviation of data movement. The operands are supplied by AL and BL registers. The data from source register BL moves to destination register AL.
- The comment field begins with a semicolon. Comments may be at the end of a line or on a new line itself. The assembler ignores comments, but they should be present to make the program understandable to others and at a later time.

Assembling and running of an 8088/86 program:

- A machine can only understand machine language. So assembly language program is to be translated back into machine language. Human for convenience uses assembly language.

- Assembly language program is time consuming and difficult, if not impossible, to translate into machine language manually.
- A PC based program called an assembler can do the same instantaneously. It takes an assembly language program as input and produces an object file having extension .obj for machine codes.
- The assembly language file can be written in any EDITOR program like DOS EDIT, WINDOWS NOTEPAD etc. which saves the file in ASCII format having extension .asm.
- Program can also be written in high level languages like 'C', BASIC, PASCAL. An interpreter program or a compiler is used to translate the same into machine codes.
- All the object files created by assembler or compiler can then be combined to form a single machine language program by another program called linker.
- The total flow diagram of machine language program development is depicted below.
- An Assembler produces a listing file having extension .lst containing the original assembly language instructions and the corresponding binary codes. It also reports for any error encountered during conversion.



Steps to create a Program

- Programs can be developed faster in high level language than assembly language as the high-level language uses higher building blocks. However, a program written in high level language usually occupies more space in memory and takes more time to execute than that developed by assembly language.
- Programs that involve a lot of hardware-control are normally written in assembly language.

Program Development Steps:

Defining the problem

The first step in writing a program is to write down the operations to be done by the program and the order of executing them. An example of a simple problem may be:

1. Read temperature from thermocouple sensor
2. Read ambient temperature from an ambient sensor
3. Add correction for ambient temperature
4. Save result in memory

For a program as simple as this, the four actions desired are very close to the assembly statements. However, for more complex problem we need to develop more extensive outline of the problem so that the actions can be replaced by assembly language statements.

Representing Program Operations

The formula or sequence of operations used to solve a problem is called ***algorithm***. An algorithm can be written using graphic shapes called flowcharts. Algorithm can also be written by pseudo codes using standard program structures.

FLOWCHARTS

Different graphic shapes are used to represent different types of operations. The figure below shows some of the commonly used graphic shapes:

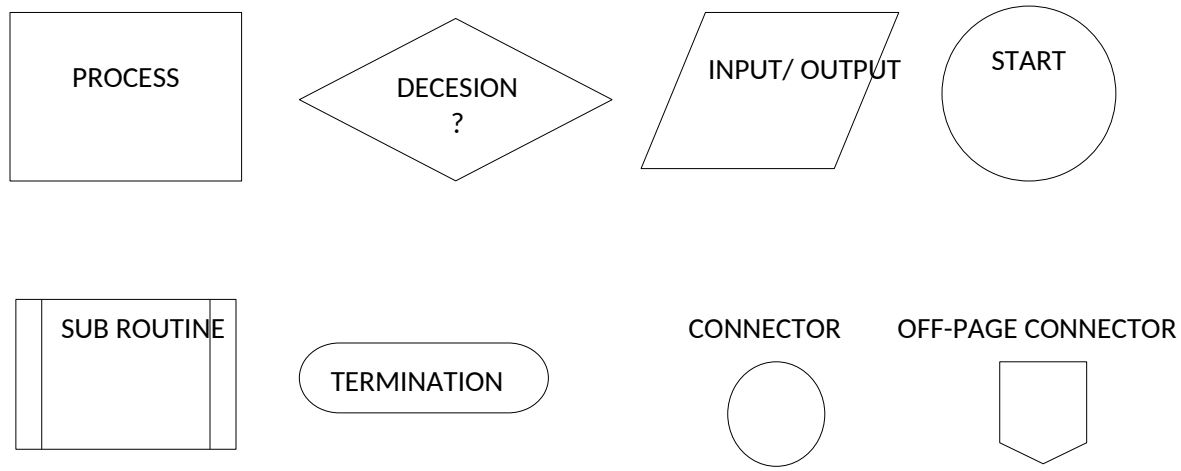
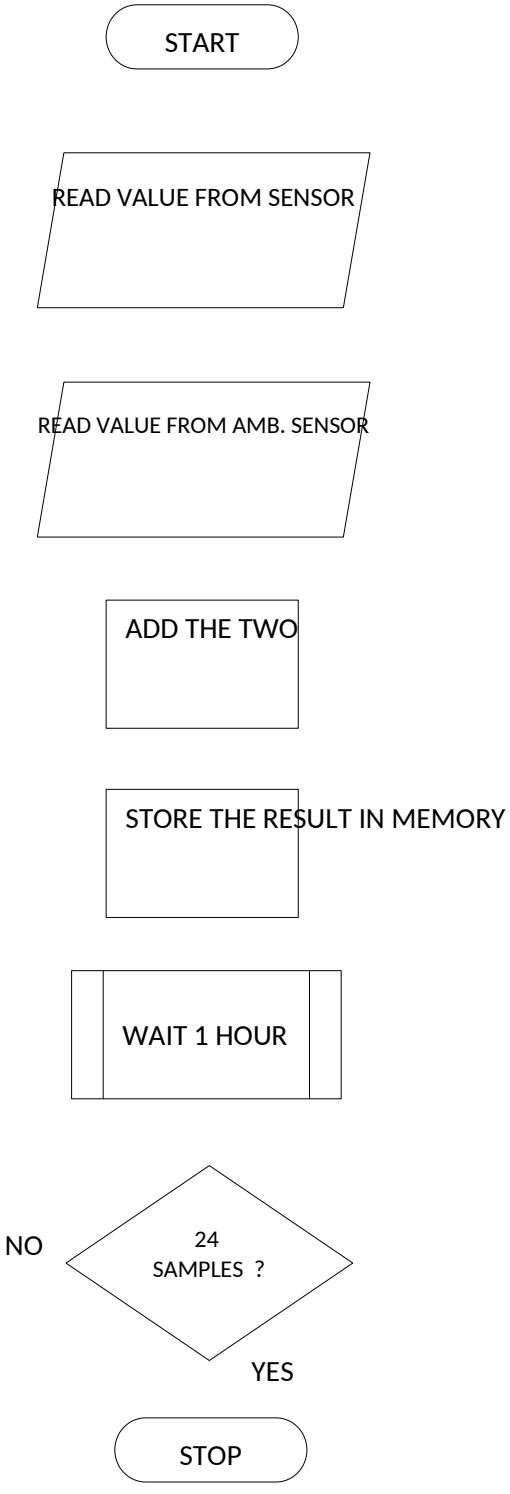


Figure: Flowchart symbols

Figure below shows a flowchart for a program to read 24 data samples from a thermocouple sensor at an interval of 1 hour.



PSEUDOCODES

- Flowchart symbols are space consuming and are normally not used for large programs. Instead English like statements called ***pseudo codes*** are used to represent the algorithm of the program.

- Three basic operations viz. *Sequence*, *Decision*, *and Iteration* can represent the operations of any desired problem.
- Sequence represents a series of actions
- Decision means choosing between two alternative actions
- Repetition means repeating a series of actions for a number of time
- Three to seven standard structures can represent all the operations in a typical program
- These standard structures are:

1. SIMPLE SEQUENCE
2. IF-THEN-ELSE
3. IF-THEN
4. CASE expressed as nested IF-THEN-ELSE
5. CASE
6. WHILE-DO LOOP
7. REPEAT UNTIL

Example of different cooking in different days of the week in the students' Hostel using Flow Chart and Pseudo Codes:

Pseudo Codes

IF MONDAY THEN

MAKE MUTTON MEAL

ELSE IF TUESDAY THEN

MAKE VEG MEAL

ELSE IF WEDNESDAY THEN

MAKE CHICKEN MEAL

:

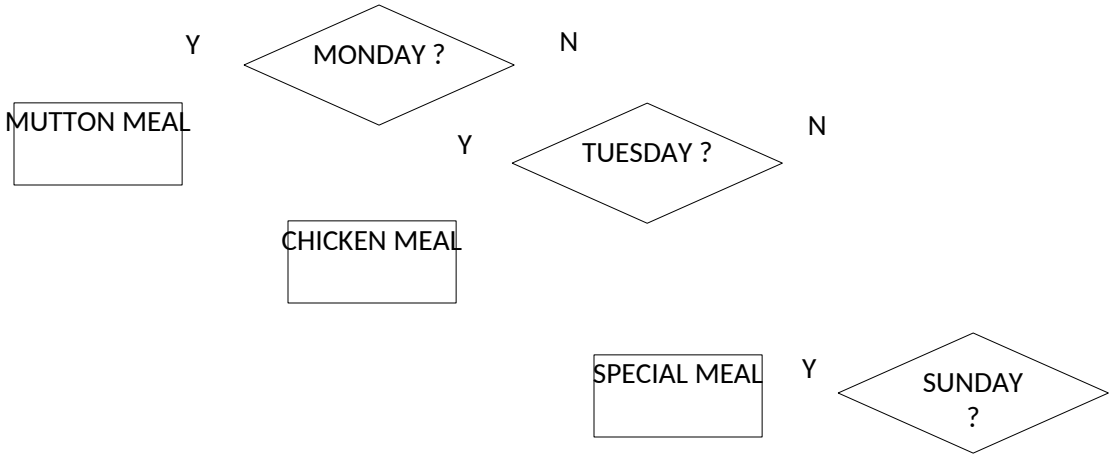
:

:

ELSE IF SUNDAY THEN

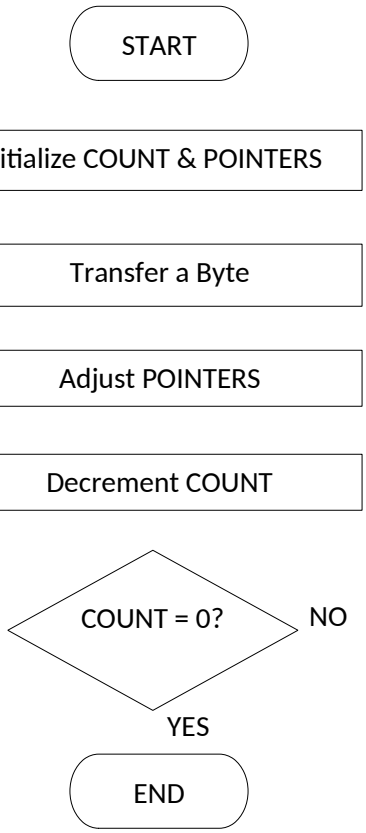
MAKE SPECIAL MEAL

Flow Chart:


Y
MONDAY ?
N
MUTTON MEAL
Y
TUESDAY ?
N
CHICKEN MEAL
SPECIAL MEAL
Y
SUNDAY
?

Programming Examples:

Examples-1: Flow Chart for transferring block of bytes data from one area of memory to another




```
;---------------------------------------------------------------------
;8086 Program to transfer a number of bytes from one place to another
;in memory. Let us also consider that code segment is in 8000H and
;data segment is at 2000H
;---------------------------------------------------------------------
                ORG 8000:100H
     START:     MOV AX, 2000H
                MOV DS, AX
                MOV SI, SOUR     ;SOURCE MEMORY POINTER
                MOV DI, DEST     ;DESTINATION MEMORY POINTER
                MOV CX, COUNT    ;STRING ELEMENT COUNTER

     RPT:       MOV AL, [SI]     ;MOVE SOURCE ELEMENT INTO AL REG
                MOV [DI], AL     ;STORE AL IN DESTINATION

                INC SI           ;SOURCE MEM POINTER INCREMENTED
                INC DI           ;DESTINATION MEM POINTER INCREMENTED

                LOOP RPT
```

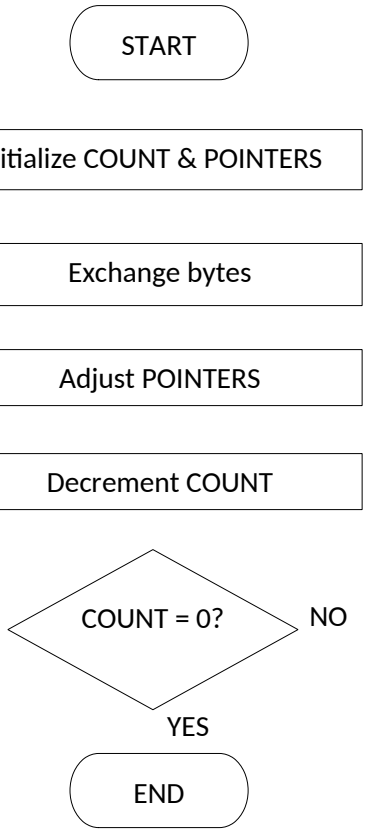
```
STAY:      JMP STAY

COUNT:     EQU 100D
SOUR:      EQU 100H
DEST:      EQU 200H
           END
```

Note:

In case of word (16-bit) data transfer, word data transfer occurs through a 16-bit register and memory pointers will be incremented by two instead of one as shown below:

```
           ORG 8000:100H
START:     MOV AX, 2000H
           MOV DS, AX
           MOV SI, SOUR     ;SOURCE MEMORY POINTER
           MOV DI, DEST     ;DESTINATION MEMORY POINTER
           MOV CX, COUNT    ;STRING ELEMENT COUNTER

RPT:       MOV AX, [SI]     ;MOVE SOURCE ELEMENT INTO AX REG
           MOV [DI], AX     ;STORE AX IN DESTINATION

           ADD SI, 02H      ;SOURCE MEM POINTER INCREMENTED BY TWO

           ADD DI, 02H      ;DEST MEM POINTER INCREMENTED BY TWO

           LOOP RPT
STAY:      JMP STAY

COUNT:     EQU 100D
SOUR:      EQU 100H
DEST:      EQU 200H
           END
```

Examples-2: Exchange of two blocks of byte data



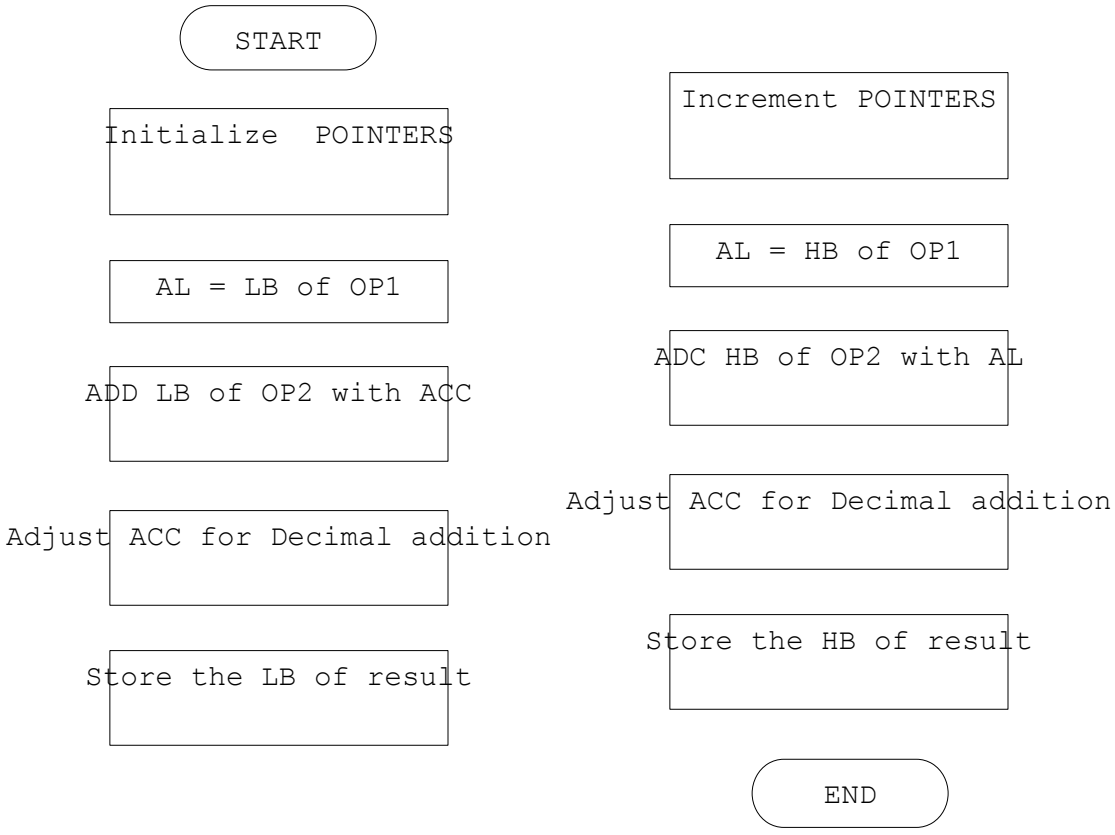
Flow Chart for exchanging two blocks of byte data from one area of memory with another

```
            ORG 8000:100H
START:      MOV AX, 2000H
            MOV DS, AX

            MOV SI, SOUR     ;SOURCE MEMORY POINTER
            MOV DI, DEST     ;DESTINATION MEMORY POINTER
            MOV CX, COUNT    ;STRING ELEMENT COUNTER

RPT:        MOV AL, [SI]     ;
            XCHG [DI], AL    ;EXCHANGE SOU DATA WITH DEST DATA
            MOV [SI], AL     ;

            INC SI           ;SOURCE MEM POINTER INCREMENTED BY TWO

            INC DI           ;DEST MEM POINTER INCREMENTED BY TWO

            LOOP RPT
STAY:       JMP STAY
```

```
COUNT:      EQU 100D
SOUR:       EQU 100H
DEST:       EQU 200H
            END
```

Example-3: Flow Chart of a program that adds two 4-digit BCD numbers stored in memory. Store the result also in memory.

Preamble: Let us consider that BCD numbers are stored in memory as packed BCD i.e. stored as two digits per byte and they will be added two-digits at a time in AL register as Decimal adjustment after addition takes place in AL only.



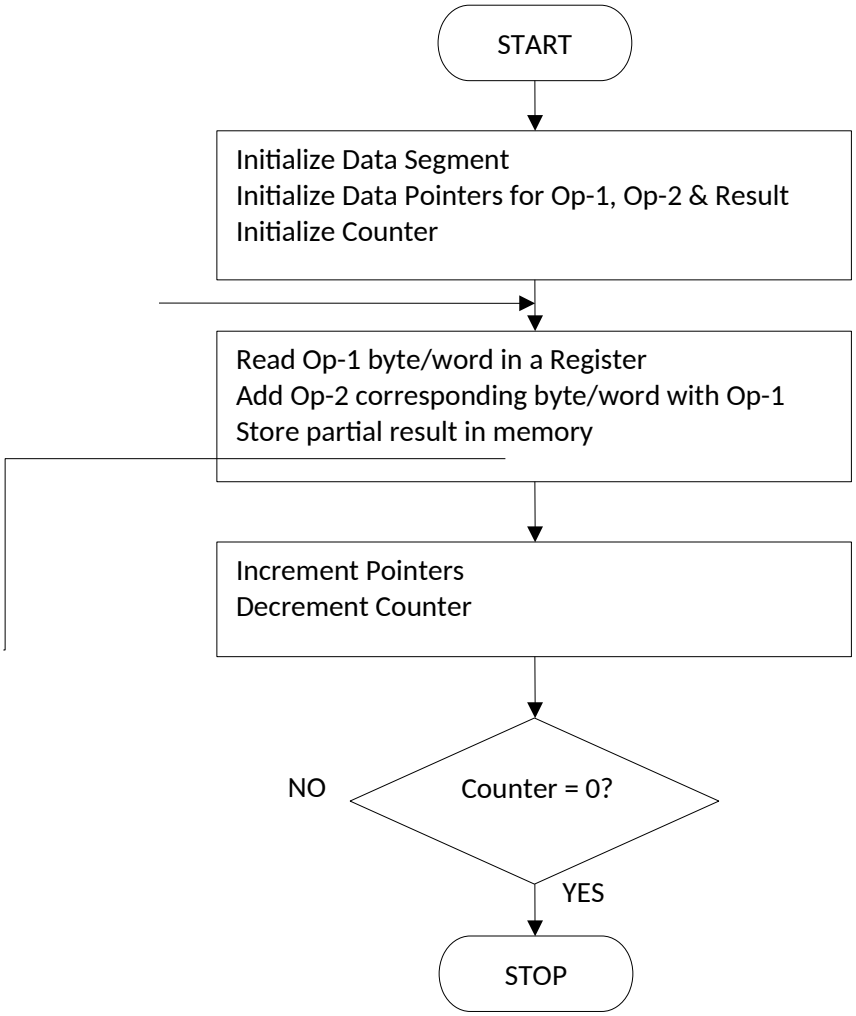
```
;-------------------------------------------------------------------
;          ADD TWO 4-DIGIT BCD NUMBERS STORED IN MEMORY.
;               ALSO STORE THE RESULT IN MEMORY
;
;             8:18 PM 8/19/2004 WRITTEN BY SAIBAL PRADHAN
;-------------------------------------------------------------------

           ORG 8000:100H

DATA1:     EQU 500H         ;OP1 IN MEMORY 500-501H
DATA2:     EQU 502H         ;OP2 IN MEMORY 502-503H
RESULT:    EQU 504H         ;RESULT IN MEM 504-506H

ADDBCD:    MOV SI, DATA1    ;MEMORY POINTER TO PICK UP OP1, OP2 AND TO
                            ;STORE RESULT

           MOV AX, [SI]     ;LOAD OP1 IN AX
           MOV BX, [SI]02H  ;LOAD OP2 IN BX
           ADD AL, BL       ;LOWER TWO DIGITS ARE ADDED
           DAA              ;ADJUST FOR BCD ADDTION
           XCHG AL, AH

           ADC AL, BH       ;HIGHER TWO DIGITS ARE ADDED TAKING CARE OF
                            ;PREVIOUS CARRY, IF ANY
           DAA
           XCHG AL, AH
           MOV [SI]04H, AX  ;STORE THE 1ST TWO DIGITS OF THE RESULT

STAY:      JMP STAY
```

More Examples:

Ex-4: Add two 64-bit data stored in memory and also store the result in memory.

Solution: Two 64-bit data cannot be added at a time. As 8086/88 can add either two 8-bit or two 16-bit data, we have to add multi-byte data either byte-wise or word-wise.

Flow Chart:




```
;-------------------------------------------------------------------------
;     TO ADD TWO 64-BIT DATA RESIDING IN MEMORY. RESULT IS ALSO STORED
;     IN MEMORY.
;     9:45 PM 29/01/2015 WRITTEN BY DR. SAIBAL KUMAR PRADHAN
;-------------------------------------------------------------------------

           .ORG 2000:100H
; INITIALIZATION OF POINTERS AND COUNTER
           MOV AX, 1000H
           MOV DS, AX       ; DATA SEGMENT STARTS AT 10000H
           MOV SI, 100H     ; OP-1 POINTER
           MOV BX, 110H     ; OP-2 POINTER
           MOV DI, 120H     ; RESULT POINTER
```

```
            MOV CX, 04H      ; 64-BIT DATA WILL BE ADDED WORD-WISE
            CLC              ; CLEAR CARRY FLAG
; ADDING CORRESPONDING WORDS OF OP-1 & OP-2
AGAIN:      MOV AX, [SI]
            ADC AX, [BX]
            MOV [DI], AX

; PROCESSING OF POINTERS & COUNTER
            INC SI
            INC SI
            INC BX
            INC BX
            INC DI
            INC DI

            LOOP AGAIN

; STORING THE FINAL CARRY
            MOV AL, 00
            ADC AL, 00
            MOV [DI], AL
STOP:       JMP STOP
```

Ex-5: Add one hundred bytes stored in memory. Justify the length of the result and also store it in memory.

Solution: One hundred bytes can be added with 99 ADD instructions and in worst case all the ADD instructions may generate carries which when added can give 99D. This can easily be stored in a byte and so the result will be a 16-bit data.

```
            .ORG 2000:100H
START:      MOV AX, 1000H
            MOV DS, AX
            MOV SI, 100H
            MOV CX, 100D     ; DATA COUNT
            XOR AH, AH       ; TAKE CARE CARRY, IF ANY

            XOR AL, AL       ; INITIAL SUM = 0

REPEAT:     ADD AL, [SI]
            JNC SKIP
            INC AH
SKIP:       INC SI
            LOOP REPEAT
            MOV DS:200H, AX  ; STORE RESULT
STOP:       JMP STOP
```

Ex-6: Find the N^{th} term and sum of N terms of an Arithmetic Progression whose first term and common difference are stored in 1000:100H and 1000:101H locations respectively. Store T_N and S_N in suitable memory.

Solution: For an AP, we know that

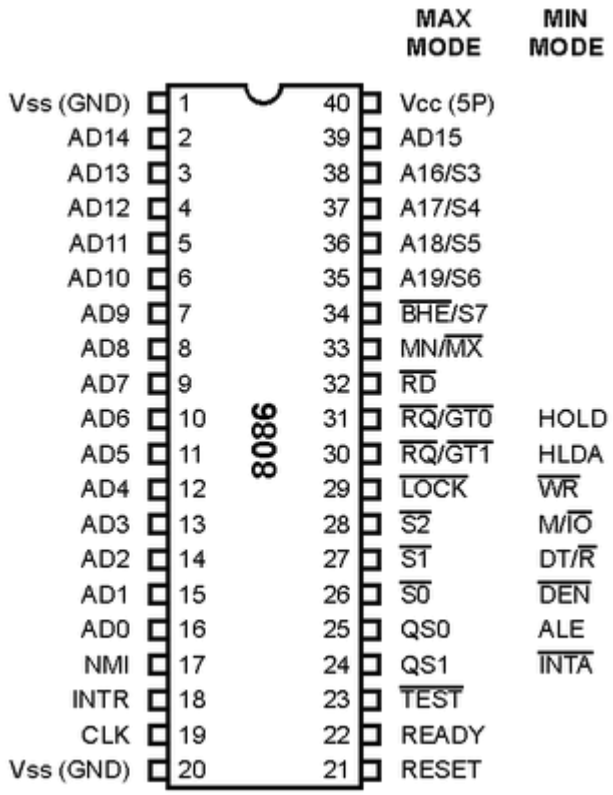
$T_N = a + (N-1).d$

and, $S_N = (N/2).[(2a+(N-1).d]$

Let us consider a, d and N are supplied through memory locations 1000:100H, 101H and 102H respectively. T_N and S_N are considered to be word and double word data respectively and will be stored at 103H and 105H respectively.

```
            .ORG 2000:200H
START:      MOV AX, 1000H
            MOV DS, AX

; CALCULATION OF TN
            MOV AL, DS:100H         ; AL=a (8-BIT FORMAT)
            CBW
            MOV CX, AX              ; CX=a (16-BIT FORMAT)
            MOV BL, DS:101H         ; BL=d
            MOV AL, DS:102H         ; AL=N
            DEC AL                  ; AL=N-1
            IMUL BL                 ; AX=(N-1).d
            ADD AX, CX              ; AX= a + (N-1).d = TN
            MOV DS:103H, AX         ; STORE TN

; CALCULATION OF SN


            ADD AX, CX              ; AX= a + (N-1).d +a = 2a +(N-1).d
            MOV BL, DS:102H         ; BL=N
            SHR BL,1                ; BL=N/2
            XCHG AX, BX
            CBW
            IMUL BX                 ; DX-AX=(N/2).[(2a+(N-1).d]=SN

            MOV DS:105H, AX         ; STORE SN
            MOV DS:107H, DX         ;
STOP:       JMP STOP
```

Pin Diagram of 8086/88



AD0 - AD15 (I/O): Address Data Bus

These lines constitute the time multiplexed memory/IO address during the first clock cycle (T1) and data during T2, T3 and T4 clock cycles. A0 is analogous to BHE' for the lower byte of the data bus, pins D0-D7. A0 bit is Low during T1 state when a byte is to be transferred on the lower portion of the bus in memory or I/O operations. 8-bit oriented devices tied to the lower half would normally use A0 to condition chip select functions. These lines are active high and float to tri-state during interrupt acknowledge and local bus "Hold acknowledge".

A19/S6, A18/S5, A17/S4, A16/S3 (0): Address/Status

During T1 state these lines are the four most significant address lines for memory operations. During I/O operations these lines are low. During memory and I/O operations, status information is available on these lines during T2, T3, and T4 states.

S6:

When Low, it indicates that 8086 is in control of the bus. During a "Hold acknowledge" machine cycle, the 8086 tri-states the S6 pin and thus allows another bus master to take control of the bus.

S5:

The status of the interrupt enable flag bit is updated at the beginning of each cycle. The status of the flag is indicated through this bit.

S4 & S3:

Lines are decoded as follows:

S4	S3	Function
0	0	Extra segment access
0	1	Stack segment access
1	0	Code segment access
1	1	Data segment access

After the first clock cycle of an instruction execution, the A17/S4 and A16/S3 pins specify which segment register generates the segment portion of the 8086 address. Thus by decoding these lines and using the decoder outputs as chip selects for memory chips, up to 4 Megabytes (one Mega per segment) of memory can be accesses. This feature also provides a degree of protection by preventing write operations to one segment from erroneously overlapping into another segment and destroying information in that segment.

BHE /S7 (O): Bus High Enable/Status

During T1 state the BHE should be used to enable data onto the most significant half of the data bus, pins D15 - D8. Eight-bit oriented devices tied to the upper half of the bus would normally use BHE to control chip select functions. BHE is Low during T1 state of read, write and interrupt acknowledge cycles when a byte is to be transferred on the high portion of the bus.
The S7 status information is available during T2, T3 and T4 states. The signal is active Low and floats to tri-state during "hold" state. This pin is Low during T1 state for the first interrupt acknowledge cycle.

RD (O): READ

The Read strobe indicates that the processor is performing a memory or I/O read cycle. This signal is active low during T2 and T3 states and the Tw states of any read cycle. This signal floats to tri-state in "hold acknowledge cycle".

TEST (I)

TEST pin is examined by the "WAIT" instruction. If the TEST pin is Low, execution continues. Otherwise, the processor waits in an "idle" state. This input is synchronized internally during each clock cycle on the leading edge of CLK.

INTR (I): Interrupt Request

It is a level triggered input which is sampled during the last clock cycle of each instruction to determine if the processor should enter into an interrupt acknowledge operation. A subroutine is vectored to via an interrupt vector look up table located in system memory. It can be internally masked by software resetting the interrupt enable bit INTR is internally synchronized. This signal is active HIGH.

NMI (I): Non-Muskable Interrupt

An edge triggered input, causes a type-2 interrupt. A subroutine is vectored to via the interrupt vector look up table located in system memory. NMI is not maskable internally by software. A

transition from a LOW to HIGH on this pin initiates the interrupt at the end of the current instruction. This input is internally synchronized.

Reset (I)

Reset causes the processor to immediately terminate its present activity. To be recognized, the signal must be active high for at least four clock cycles, except after power-on which requires a 50 Micro Sec. pulse. It causes the 8086 to initialize registers DS, SS, ES, IP and flags to all zeros. It also initializes CS to FFFF H. Upon removal of the RESET signal from the RESET pin, the 8086 will fetch its next instruction from the 20 bit physical address FFFF0H. The reset signal to 8086 can be generated by the 8284. (Clock generation chip). To guarantee reset from power-up, the reset input must remain below 1.5 volts for 50 Micro sec. after Vcc has reached the minimum supply voltage of 4.5V.

Ready (I)

Ready is the acknowledgement from the addressed memory or I/O device that it will complete the data transfer. The READY signal from memory or I/O is synchronized by the 8284 clock generator to form READY. This signal is active HIGH. The 8086 READY input is not synchronized. Correct operation is not guaranteed if the setup and hold times are not met.

CLK (I): Clock

Clock provides the basic timing for the processor and bus controller. It is asymmetric with 33% duty cycle to provide optimized internal timing. Minimum frequency of 2 MHz is required, since the design of 8086 processors incorporates dynamic cells. The maximum clock frequencies of the 8086-4, 8086 and 8086-2 are 4MHz, 5MHz and 8MHz respectively.
Since the 8086 does not have on-chip clock generation circuitry, and 8284 clock generator chip must be connected to the 8086 clock pin. The crystal connected to 8284 must have a frequency 3 times the 8086 internal frequency. The 8284 clock generation chip is used to generate READY, RESET and CLK.

MN/MX (I): Maximum / Minimum

This pin indicates what mode the processor is to operate in. In minimum mode, the 8086 itself generates all bus control signals. In maximum mode the three status signals are to be decoded to generate all the bus control signals.
Minimum Mode Pins The following 8 pins function descriptions are for the 8086 in minimum mode; MN/ MX = 1. The corresponding 8 pins function descriptions for maximum mode is explained later.

M/IO (O): Status line

This pin is used to distinguish a memory access or an I/O accesses. When this pin is Low, it accesses I/O and when high it access memory. M / IO becomes valid in the T4 state preceding a bus cycle and remains valid until the final T4 of the cycle. M/IO floats to 3 - state OFF during local bus "hold acknowledge".

WR (O): Write

Indicates that the processor is performing a write memory or write IO cycle, depending on the state of the M /IOsignal. WR is active for T2, T3 and Tw of any write cycle. It is active LOW, and floats to 3-state OFF during local bus "hold acknowledge ".

INTA (O): Interrupt Acknowledge

It is used as a read strobe for interrupt acknowledge cycles. It is active LOW during T2, T3, and T4 of each interrupt acknowledge cycle.

DT/ R (O): DATA Transmit/Receive

In minimum mode, 8286/8287 transceiver is used for the data bus. DT/ R is used to control the direction of data flow through the transceiver. This signal floats to tri-state off during local bus "hold acknowledge".

DEN (O): Data Enable

It is provided as an output enable for the 8286/8287 in a minimum system which uses the transceiver. DEN is active LOW during each memory and IO access. It will be low beginning with T2 until the middle of T4, while for a write cycle, it is active from the beginning of T2 until the middle of T4. It floats to tri-state off during local bus "hold acknowledge".

HOLD & HLDA (I/O): Hold and Hold Acknowledge

Hold indicates that another master is requesting a local bus "HOLD". To be acknowledged, HOLD must be active HIGH. The processor receiving the "HOLD " request will issue HLDA (HIGH) as an acknowledgement in the middle of the T1-clock cycle. Simultaneous with the issue of HLDA, the processor will float the local bus and control lines. After "HOLD" is detected as being Low, the processor will lower the HLDA and when the processor needs to run another cycle, it will again drive the local bus and control lines.

Maximum Mode:

The following pins function descriptions are for the 8086/8088 systems in maximum mode (i.e.. MN/MX = 0). Only the pins which are unique to maximum mode are described below.

S2, S1, S0 (O): Status Pins

These pins are active during T4, T1 and T2 states and is returned to passive state (1,1,1 during T3 or Tw (when ready is inactive). These are used by the 8288 bus controller to generate all memory and I/O operation) access control signals. Any change by S2, S1, S0 during T4 is used to indicate the beginning of a bus cycle. These status lines are encoded as shown in table 3.

S2	S1	S0	Characteristics
0	0	0	Interrupt acknowledge
0	0	1	Read I/O port
0	1	0	Write I/O port
0	1	1	Halt
1	0	0	Code access
1	0	1	Read Memory
1	1	0	Write memory
1	1	1	Passive State

QS0, QS1 (O): Queue – Status

Queue Status is valid during the clock cycle after which the queue operation is performed. QS0, QS1 provide status to allow external tracking of the internal 8086 instruction queue. The condition of queue status is shown in table 4.

Queue status allows external devices like In-circuit Emulators or special instruction set extension co-processors to track the CPU instruction execution. Since instructions are executed from the 8086 internal queue, the queue status is presented each CPU clock cycle and is not related to the bus cycle activity. This mechanism allows (1) A processor to detect execution of a ESCAPE instruction which directs the co- processor to perform a specific task and (2) An in-circuit Emulator to trap execution of a specific memory location.

QS1	QS1	Characteristics
0	0	No operation
0	1	First byte of opcode from queue
1	0	Empty the queue
1	1	Subsequent byte from queue

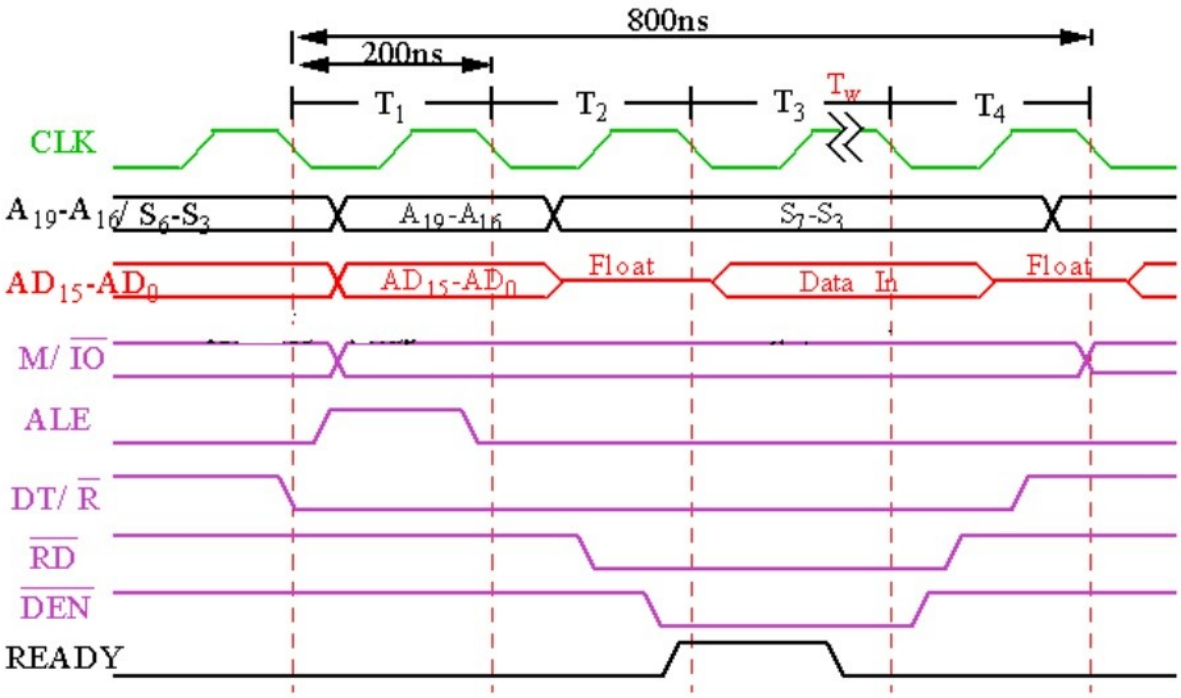
LOCK (O)

It indicates to another system bus master, not to gain control of the system bus while LOCK is active Low. The LOCK signal is activated by the "LOCK" prefix instruction and remains active until the completion of the instruction. This signal is active Low and floats to tri-state OFF during 'hold acknowledge'. Example:

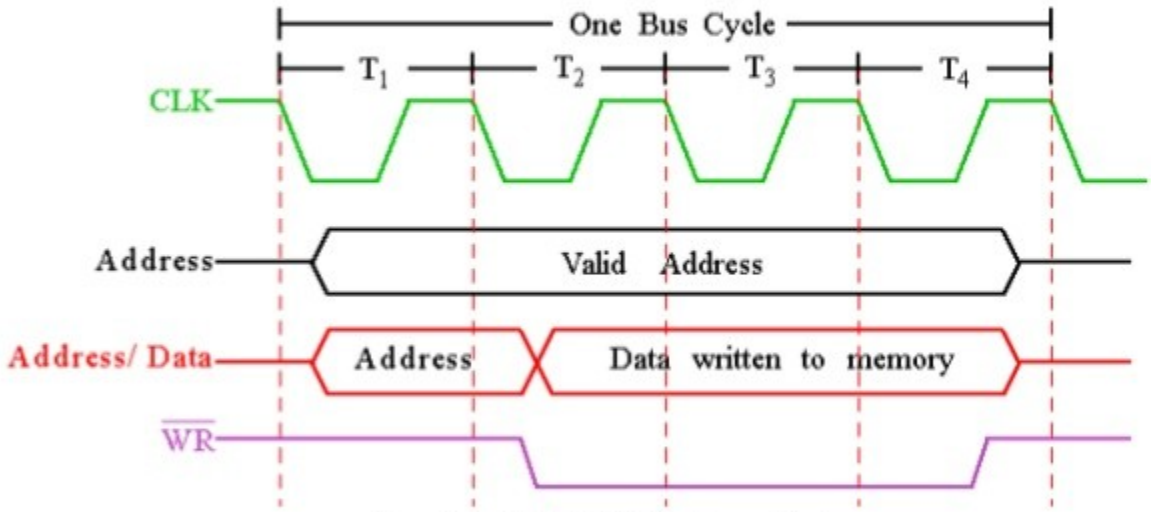
```
LOCK XCHG reg., Memory  ; Register is any register and memory GT0
                        ; is the address of the semaphore.
```

RQ/GT0 and RQ/GT1 (I/O): Request/Grant

These pins are used by other processors in a multi processor organization. Local bus masters of other processors force the processor to release the local bus at the end of the processors current bus cycle. Each pin is bi-directional and has an internal pull up resistors. Hence they may be left un-connected.



Bus Timing for a Read Operation



Simplified 8086 Write Bus Cycle

Clock Generator

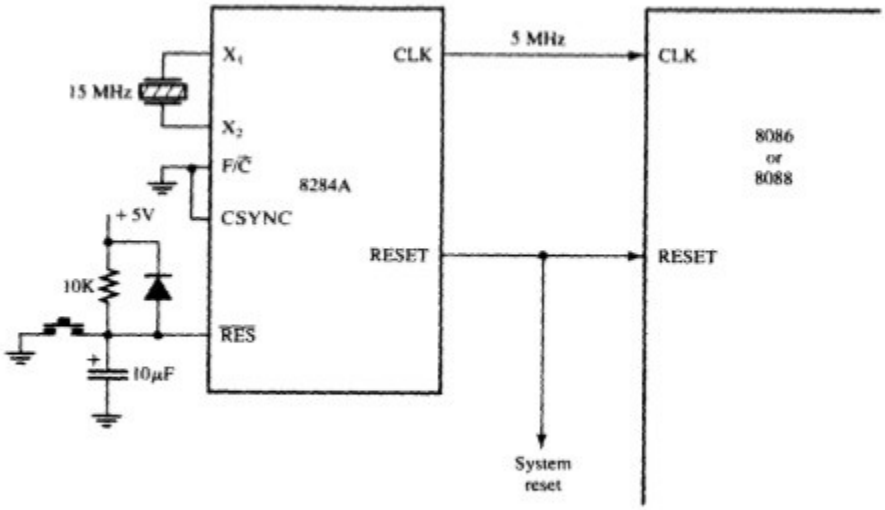


FIGURE 9–4 The clock generator (8284A) and the 8086 and 8088 microprocessor illustrating the connection for the clock and reset signals. A 15 MHz crystal provides the 5 MHz clock for the microprocessor.

8051 Microcontroller

The 8051 is an 8 bit microcontroller originally developed by Intel in 1980. It is the world's most popular microcontroller core, made by many independent manufacturers (truly multi-sourced). There were 126 million 8051s (and variants) shipped in 1993!!

A typical 8051 contains:

- CPU with boolean processor
- 5 or 6 interrupts: 2 are external, 2 priority levels
- 2 or 3 16-bit timer/counters
- programmable full-duplex serial port (baud rate provided by one of the timers)
- 32 I/O lines (four 8-bit ports)
- RAM
- ROM/EPROM in some models

The 8051 instruction set is optimized for the one-bit operations so often desired in real-world, real-time control applications. The boolean processor provides direct support for bit manipulation. This leads to more efficient programs that need to deal with binary input and output conditions inherent in digital-control problems. Bit addressing can be used for test pin monitoring or program control flags.

8051 Flavors

The 8051 has the widest range of variants of any embedded controller on the market. The smallest device is the Atmel 89c1051, a 20 Pin FLASH variant with 2 timers, UART, 20mA. The fastest parts are from Dallas, with performance close to 10 MIPS! The most powerful chip is the Siemens 80C517A, with 32 Bit ALU, 2 UARTS, 2K RAM, PLCC84 package, 8 x 16 Bit PWMs, and other features.

Among the major manufacturers are:

AMD	Enhanced 8051 parts (no longer producing 80x51 parts)
Atmel	FLASH and semi-custom parts
Dallas	Battery backed, program download, and fastest variants
Intel	8051 through 80c51gb / 80c51sl
ISSI	IS80C51/31 runs up to 40MHz
Matra	80c154, low voltage static variants
OKI	80c154, mask parts

Philips 87c748 thru 89c588 - more variants than anyone else
Siemens 80c501 through 80c517a, and SIECO cores
SMC COM20051 with ARCNET token bus network engine
SSI 80x52, 2 x HDLC variant for MODEM use

Internal Architecture of 8051:

The registers and memory map is very important to develop assembly language program. 8051 has 128 bytes RAM in the address range 00 – 7FH. It has also some special function registers (SFRs) like A, B, PSW, DPH, DPL, SP etc. These registers are also having some addresses in the range between 80H – FFH. Some of the variants of 8051 family like 8052 has 256 bytes of internal RAM. So apparently, the SFRs and the higher 128 bytes RAM overlap each other. This is taken care of by using different addressing modes for different space.

8-bit Registers supporting register addressing mode are:

A
R0
R1
R2
R2
R3
R4
R5
R6
R7

16-bit register supporting register addressing is DPTR

Other important registers not referred by their names but by their addresses are listed in the given table below:

Special Function Registers and their addresses:

SL	Symbol	Name	Addr	Bit Address							
1	A*	Accumulator	E0H	E7	E6	E5	E4	E3	E2	E1	E0
2	B*	B register	F0H	F7	F6	F5	F4	F3	F2	F1	F0
3	PSW*	Program Status Word	D0H	D7	D6	D5	D4	D3	D2	D1	D0
4	SP	Stack Pointer	81H								
5	DPTR	Data Pointer (16-bit)									
6	DPL	DPTR Low Byte	82H								
7	DPH	DPTR High Byte	83H								
8	P0*	Port 0	80H	87	86	85	84	83	82	81	80
9	P1*	Port 1	90H	97	96	95	94	93	92	91	90
10	P2*	Port 2	A0H	A7	A6	A5	A4	A3	A2	A1	A0
11	P3*	Port 3	B0H	B7	B6	B5	B4	B3	B2	B1	B0
12	IP*	Interrupt Priority Control	B8H	BF	BE	BD	BC	BB	BA	B9	B8
13	IE*	Interrupt Enable Control	A8H	AF	AE	AD	AC	AB	AA	A9	A8
14	TMOD	Timer/Counter Mode Control	89H								
15	TCON*	Timer/Counter Control	88H	8F	8E	8D	8C	8B	8A	89	88
16	T2CON*	Timer/Counter 2 Control	C8H	CF	CE	CD	CC	CB	CA	C9	C8
17	T2MOD	Timer/Counter Mode Control	C9H								
18	TH0	Timer/Counter 0 High Byte	8CH								
19	TL0	Timer/Counter 0 Low Byte	8AH								
20	TH1	Timer/Counter 1 High Byte	8DH								
21	TL1	Timer/Counter 1 Low Byte	8BH								
22	TH2	Timer/Counter 2 High Byte	CDH								
23	TL2	Timer/Counter 2 Low Byte	CCH								

24	RCAP2H	T/C 2 Capture Reg. High Byte	CBH								
25	RCAP2L	T/C 2 Capture Register Low Byte	CAH								
26	SCON*	Serial Control	98H	9F	9E	9D	9C	9B	9A	99	98
27	SBUF	Serial Data Buffer	99H								
28	PCON	Power Control.	87H								

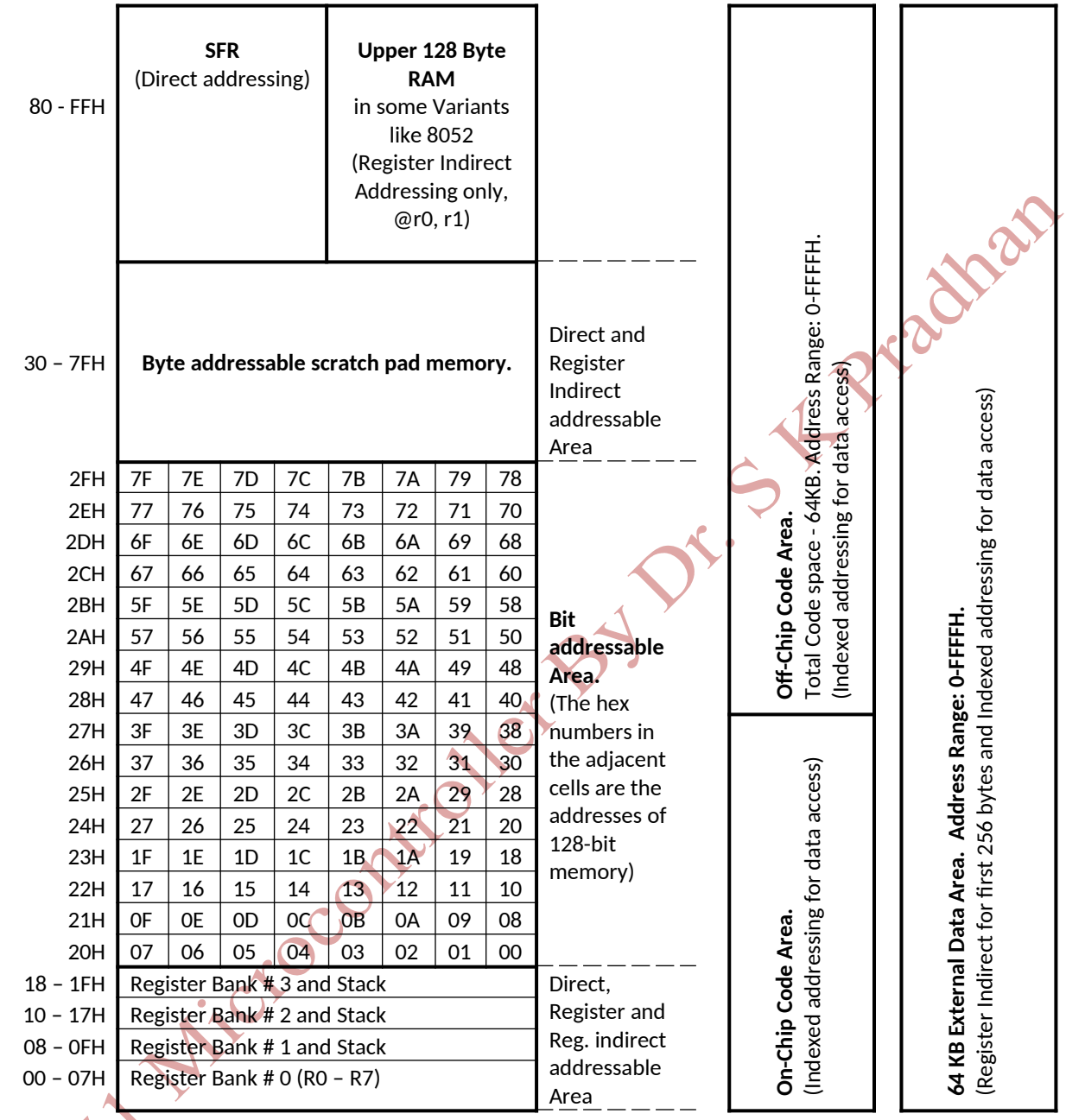
Register Bank Selection Through PSW.4 and PSW.3

CY	AC	F0	RS1	RS0	OV	--	P

RS1	RS0	Register Bank	Address
0	0	Bank 0	00H – 07H
0	1	Bank 1	08H – 0FH
1	0	Bank 2	10H – 17H
1	1	Bank 3	18H - 1FH

The 8051 can address 64KB program memory as well as 64KB data memory. Out of 64KB total program memory, some are on-chip. Different variants of 8051 have different amounts of on-chip program memory. For instance, 8051 has 4KB on-chip ROM. External Program and data memory (ROM and RAM both) can be interfaced further to cater the need of the system.

The register and memory map is shown in figure below:



Assembly Language Programming

8051 Registers Structure
Registers are mostly 8-bits viz. A, B, PSW, DPL, DPH, R0-R7, SP etc. DPTR is a 16-bit register composed of DPH and DPL.

Addressing Modes supported by 8051:

1. Immediate
2. Register Direct
3. Direct
4. Register Indirect and
5. Indexed

General Format of Assembly Language Instruction

```
Lebel:      Mnemonic                    ; Comments

Here:       MOV   Destination, Source   ; This is the format of MOV command
```

Immediate Addressing:
This addressing mode is used when a constant data to be loaded in a destination operand like internal RAM/ External Data RAM/ Internal Registers etc. "#" sign is used to represent a constant/immediate data

```
MOV A, #12H          ; Place a value 12H in Accumulator
MOV B, #55           ; Place 55 in B Register
MOV 12H, #78H        ; Place 78H in internal RAM at address 12H
MOV DPTR, #1234H     ; Place 1234H in DPTR Register
```

Register Direct Addressing:
This mode is used when registers are used to hold the data to be manipulated.

```
MOV A, Rn            ; where n is 0 – 7
```

Direct Addressing:
The amount of internal RAM in 8051 is 128 bytes having address range 00H – 7FH. This entire RAM can be accessed by mentioning their addresses directly in the instructions. However, this addressing mode is normally used for address range 30H – 7FH. RAM locations 00H – 1FH are assigned to Register banks and stack. Stack uses direct addressing; however, register banks can also be addressed by their names. RAM locations 20H – 2FH, 16 bytes are bit addressable area. These memory locations are used to save bit information and can be set or reset using SETB or CLR instructions respectively. These 128 Bit addressable memories have address range 00H – 7FH.

```
MOV 70H, #12H
MOV A, 70H
MOV R6, 56H
```

Register Indirect Addressing:
In this addressing mode, memory addresses are supplied indirectly through R0 and R1. This mode is supported by internal RAM and first 256 bytes in external data memory.

```
MOV A, @Ri           ; i = 0 and 1 only
MOV A, @r0
MOV @r1, A
MOV 12H, @r1
MOVX A, @r1
MOVX @r0, A
```

Indexed Addressing:

While accessing data from code and external data areas, the memory addresses are supplied through a 16-bit register DPTR. This mode is very useful in accessing look up table stored in code memory.

MOVC A, @A+DPTR
MOVX A, @DPTR
MOVX @DPTR, A

8051 Instruction Set Summary

SL	Mnemonic	Operands	Description	Byte	T States	Flags affected		
						C	OV	AC
A. Data Transfer Group								
1	MOV	A, R_n	Move register to Acc	1	12			
2	MOV	A, direct	Move direct byte to Acc	2	12			
3	MOV	A, @R_i	Move indirect to Acc	1	12			
4	MOV	A, # data 8	Move immediate data to Acc	2	12			
5	MOV	R_n, A	Move Acc to register	1	12			
6	MOV	R_n, direct	Move direct byte to register	2	24			
7	MOV	R_n, # data 8	Move immediate data to register	2	12			
8	MOV	direct, A	Move Acc to direct byte	2	12			
9	MOV	direct, R_n	Move register to direct byte	2	24			
10	MOV	direct, direct	Move direct to direct in internal RAM	3	24			
11	MOV	direct, @R_i	Move indirect RAM to direct byte	2	24			
12	MOV	direct, # data 8	Move immediate data to direct byte	3	24			
13	MOV	@R_i, A	Move Acc to indirect RAM	1	12			
14	MOV	@R_i, direct	Move direct byte to indirect RAM	2	24			
15	MOV	@R_i, # data 8	Move immediate data to indirect RAM	2	12			
16	MOV	DPTR, # data 16	Load data pointer with 16 bit constant	3	24			
17	MOVC	A, @A+DPTR	Move code byte relative to DPTR to Acc	1	24			
18	MOVC	A, @A+PC	Move code byte relative to PC to Acc	1	24			
19	MOVX	A, @R_i	Move external RAM (8-bit addr) data to Acc	1	24			
20	MOVX	A, @DPTR	Move external RAM (16-bit addr) data to Acc	1	24			
21	MOVX	@R_i, A	Move Acc to external RAM	1	24			

22	MOVX	@DPTR, A		1	24			
23	PUSH	direct	Push direct byte onto Stack	2	24			
24	POP	direct	Pop direct byte from Stack	2	24			
25	XCH	A, R_n	Exchange Acc with register	1	12			
26	XCH	A, direct	Exchange Acc with direct	2	12			
27	XCH	A, @R_i	Exchange Acc with indirect	1	12			
28	XCHD	A, @R_i	Exchange low order digit indirect with Acc	1	12			
B. Arithmetic Operations								
30	ADD	A, R_n	Add register to Acc	1	12	x	x	x
31	ADD	A, direct	Add direct to Acc	2	12	x	x	x
32	ADD	A, @R_i	Add indirect to Acc	1	12	x	x	x
33	ADD	A, # data 8	Add immediate data to Acc	2	12	x	x	x
34	ADC	A, R_n	Add with Carry reg to Acc	1	12	x	x	x
35	ADC	A, direct	Add with Carry direct to Acc	2	12	x	x	x
36	ADC	A, @R_i	Add with Carry indirect to Acc	1	12	x	x	x
37	ADC	A, # data 8	Add with Carry immediate to Acc	2	12	x	x	x
38	SUBB	A, R_n	Sub with borrow reg from Acc	1	12	x	x	x
39	SUBB	A, direct	Sub with borrow direct from Acc	2	12	x	x	x
40	SUBB	A, @R_i	Sub with borrow indirect from Acc	1	12	x	x	x
41	SUBB	A, # data 8	Sub with borrow immediate from Acc	2	12	x	x	x
42	INC	A	Increment Acc by one	1	12			
43	INC	R_n	Increment Reg by one	1	12			
44	INC	direct	Increment direct by one	2	12			
45	INC	@R_0	Increment indirect by one	1	12			
46	INC	@R_1		1	12			
47	DEC	A	Decrement Acc by one	1	12			
48	DEC	R_n	Decrement reg by one	1	12			
49	DEC	direct	Decrement direct by one	2	12			

50	DEC	@R_i	Decrement indirect by one	1	12			
51	INC	DPTR	Increment data pointer by one	1	24			
52	MUL	AB	Multiply A & B, Result in BA	1	48	0	x	
53	DIV	AB	Divide A by B, A	1	48	0	x	
54	DA	A	Decimal adjust Acc	1	12	x		
C. Logical Operations								
55	ANL	A, R_n	AND register to Acc	1	12			
56	ANL	A, direct	AND direct byte to Acc	2	12			
57	ANL	A, @R_i	AND indirect RAM to Acc	1	12			
58	ANL	A, # data 8	AND immediate data to Acc	2	12			
59	ANL	direct, A	AND Acc to direct byte	2	12			
60	ANL	direct, # data 8	AND immediate data to direct byte	3	24			
61	ORL	A, R_n	OR reg to Acc	1	12			
62	ORL	A, direct	OR direct to Acc	2	12			
63	ORL	A, @R_i	OR indirect to Acc	1	12			
64	ORL	A, # data 8	OR immediate data to Acc	2	12			
65	ORL	direct, A	OR Acc to direct	2	12			
66	ORL	direct, # data 8	OR immediate data to Direct	3	24			
67	XRL	A, R_n	Ex-OR reg to Acc	1	12			
68	XRL	A, direct	Ex-OR direct to Acc	2	12			
69	XRL	A, @R_i	Ex-OR indirect to Acc	1	12			
70	XRL	A, # data 8	Ex-OR immediate data to Acc	2	12			
71	XRL	direct, A	Ex-OR Acc to direct	2	12			
72	XRL	direct, # data 8	Ex-OR immediate data to Direct	3	24			
73	CLR	A	Clear Acc	1	12			
74	CPL	A	Compliment Acc	1	12			
75	RL	A	Rotate Acc left	1	12			
76	RLC	A	Rotate Acc left through carry	1	12	x		

77	RR	A	Rotate Acc right	1	12			
78	RRC	A	Rotate Acc right through carry	1	12	x		
79	SWAP	A	Swap nibbles within Acc	1	12			
D. Boolean Variable Manipulation								
80	CLR	C	Clear carry flag	1	12	0		
81	CLR	bit	Clear direct bit	2	12			
82	SETB	C	Set carry	1	12	1		
83	SETB	bit	Set direct bit	2	12			
84	CPL	C	Complement carry	1	12	x		
85	CPL	bit	Complement direct bit	2	12			
86	ANL	C, bit	AND direct bit to carry	2	24	x		
87	ANL	C, /bit	AND complement of direct bit to carry	2	24	x		
88	ORL	C, bit	OR bit to Carry Flag	2	24	x		
89	ORL	C, /bit	OR complement of bit to Carry Flag	2	24	x		
90	MOV	C, bit	Move direct bit to carry	2	12	x		
91	MOV	bit, C	Move carry to direct bit	2	24			
92	JC	rel	Jump if carry is set	2	24			
93	JNC	rel	Jump if carry is not set	2	24			
94	JB	bit, rel	Jump if bit is set	3	24			
95	JNB	bit, rel	Jump if bit is not set	3	24			
96	JBC	bit, rel	Jump if direct bit set and clear bit	3	24			
E. Program Branching								
97	ACALL	addr 11	Absolute subroutine call	2	24			
98	LCALL	addr 16	Long Subroutine call	3	24			
99	RET		Return from subroutine	1	24			
100	RETI		Return from interrupt subroutine	1	24			
101	AJMP	addr 11	Absolute jump	2	24			
102	LJMP	addr 16	Long jump	3	24			

103	SJMP	rel	Short jump (relative addr)	2	24			
104	JMP	@A+DPTR	Jump indirect relative to DPTR	1	24			
105	JZ	rel	Jump if Acc is zero	2	24			
106	JNZ	rel	Jump if Acc is not zero	2	24			
107	CJNE	A, direct, rel	Compare direct with Acc and jump if not equal	3	24	x		
108	CJNE	A, # data 8, rel	Compare data with Acc and jump if not equal	3	24	x		
109	CJNE	R_n, # data 8, rel	Compare data with reg and jump if not equal	3	24	x		
110	CJNE	@R_i, # data 8, rel	Compare data with indirect and jump if not equal	3	24	x		
111	DJNZ	R_n, rel	Decrement reg by one and jump if not zero	2	24			
112	DJNZ	direct, rel	Decrement direct by one and jump if not zero	3	24			
113	NOP	No operation	No operation	1	12			

Program examples

1. Write an 8051 Assembly Language Program to transfer a number of bytes from one place to another in memory (small string having 255 or less elements)

Both the blocks of data may be in internal/external RAM or one in internal RAM and the other in external RAM. Internal RAM and first 256 bytes of external RAM can be addressed indirectly by R0 and R1. However, data in the external RAM can be addressed indirectly by dptr anywhere in the entire 64KB space.

Case – I: Both the blocks are in internal RAM

```
        .ORG 100H
        MOV R0, 30H          ; R0 points the source block
        MOV R1, 40H          ; R1 points the destination block
        MOV R2, 10D          ; No of bytes present in the block is 10

LOOP:   MOV A, @R0
        MOV @R1, A
        INC R0
        INC R1
        DJNZ R2, LOOP
STOP:   SJMP STOP
```

Case-II: One block in internal RAM (destination) and other in external RAM within 256 bytes boundary.

```
        .ORG 100H
        MOV R0, 30H          ; R0 points the source block in external RAM
        MOV R1, 40H          ; R1 points the destination block in internal RAM
        MOV R2, 10D          ; No of bytes present in the block is 10

LOOP:   MOVX A, @R0
        MOV @R1, A
        INC R0
        INC R1
        DJNZ R2, LOOP
STOP:   SJMP STOP
```

Case-III: One block in internal RAM (destination) and other in external RAM inside/outside 256 bytes boundary.

```
        .ORG 100H
        MOV DPTR, 0030H      ; DPTR points the source block in external RAM
        MOV R1, 40H          ; R1 points the destination block in internal RAM
        MOV R2, 10D          ; No of bytes present in the block is 10

LOOP:   MOVX A, @DPTR
        MOV @R1, A
        INC DPTR
        INC R1
        DJNZ R2, LOOP

STOP:   SJMP STOP
```

Case-III: Both the blocks are in external RAM

```
        .ORG 100H
        MOV DPTR, 1234H     ; DPTR points the source block in external RAM
        MOV R1, 40H         ; R1 points the destination block also in external RAM
        MOV R2, 10D         ; No of bytes present in the block is 10

LOOP:   MOVX A, @DPTR
        MOVX @R1, A
        INC DPTR
        INC R1
        DJNZ R2, LOOP

STOP:   SJMP STOP
```

2. Write an 8051 Assembly Language Program to interchange two blocks of data residing in internal RAM

```
        .ORG 2000H
        MOV R0, #30H        ; R0 points one block
        MOV R1, #40H        ; R1 points another block
        MOV R2, #10H        ; No of data in both the blocks is 16

RPT:    MOV A, @R0          ; Byte exchanged between two blocks
        MOV R3, A
        MOV A, @R1
        MOV @R0, A
        MOV A, R3
        MOV @R1, A

        INC R0
        INC R1
        DJNZ R2, RPT
STOP:   SJMP STOP
```

3. Write an 8051 Assembly Language Program to add to 32-bit data stored in internal RAM and store the result in internal RAM only.

```
        .ORG 1000H
        MOV R0, #30H        ; OP1 pointer
        MOV R1, #34H        ; OP2 pointer
        MOV R2, #38H        ; Result pointer
        MOV R3, #04H        ; Length of the operands (in byte)

        CLR C
AGN:    MOV A, @R0
        ADC A, @R1          ; Partial result in Acc
        PUSH 00H            ; R0 is stored in stack
        MOV 00, R2          ; R0 is modified by the address of destination block storing result
        MOV @R0, A          ; Partial result is stored in memory pointed out by @R0
        POP 00H             ; R0 is retrieved from stack
```

```
        INC R0
        INC R1
        INC R2

        DJNZ R3, AGN            ; Repeat process until partial additions are completed

        CLR A                   ; Carry stored, if any.
        ADC A, #00H
        MOV 00, R2
        MOV @R0, A

STOP:   SJMP STOP
```

4. Write an 8051 Assembly Language Program to count number of positive and negative data present in a list residing in internal RAM.

```
        .ORG 2000H
        MOV R0, #30H            ; R0 points the list having both positive and negative data
        MOV R1, #10D            ; R1 counts number of data not yet counted
        MOV R2, #00H            ; R2 counts number of positive data
        MOV R3, #00H            ; R3 counts number of negative data
LOOP:   MOV A, @R0
        JB E7H, NEG             ; E7 is the address of msb of the accumulator indicating sign of the data
POS:    INC R2
        SJMP SKIP
NEG:    INC R3
SKIP:   INC R0
        DJNZ R1, LOOP
STOP:   SJMP STOP
```

5. Write an 8051 Assembly Language Program to add two 4-digit BCD numbers residing in internal RAM in packed form and also store the result (5-digit) in internal RAM.

```
        .ORG 1000H
        MOV R0, #30H            ; Starting address of the first 4-digit packed BCD number
        MOV R1, #32H
        MOV R2, #34H
        MOV R3, #02H            ; R3 counting partial addition still left
        CLR C

RPT:    MOV A, @R0
        ADC A, @R1
        DA A
        PUSH 00H
        MOV R0, 02H
        MOV @R0, A
        POP 00H

        INC R0
        INC R1
        INC R2
```

```
        DJNZ R3, RPT            ; Repeat addition, if left

        MOV A, #00H             ; Store the carry, if any, as the last byte of the result
        ADC A, #00H

        MOV R0, 02H
        MOV @R0, A

STOP:   SJMP STOP
```

6. Write an 8051 Assembly Language Program to add N (<256) 8-bit data residing in external RAM. Store the result (2-byte) in internal RAM.

```
        .ORG 1000H
        MOV DPTR, #1234H        ; Starting address of the N data
        MOV R0, #100D           ; R0 counts the number of data to be still added
        CLR A
        MOV R1, A               ; R1 counts number of times the carry is generated giving the ms byte of result

RPT:    PUSH E0H                ; Accumulator stored in stack before being modified
        MOVX A, @DPTR
        MOV R2, A
        POP E0H                 ; Accumulator is restored
        ADD A, R2
        JNC SKIP
        INC R1
SKIP:   INC DPTR
        DJNZ R0, RPT

        MOV 30H, A              ; 16-bit result is stored in 30H (lsb) and 31H (msb)
        MOV 31H, R1

STOP:   SJMP STOP
```

7. Write an 8051 Assembly Language Program to multiply two 8-bit numbers residing in internal RAM. Also store the result (2-byte) in internal RAM.

```
        .ORG 200H
        MOV A, 40H              ; OP1 is taken in Acc
        MOV F0H, 41H            ; OP2 is taken in Register B
        MUL AB

        MOV 42H, A              ; lsb of the result in Acc is stored in 42H memory
        MOV 43H, F0H            ; msb of the result in Register B is stored in 43H memory

STOP:   SJMP STOP
```

8. Write an 8051 Assembly Language Program to divide an 8-bit number by an 8-bit number residing in internal RAM. Also store the result in quotient and remainder form in internal RAM.

```
        .ORG 300H
        MOV A, 30H          ; dividend is taken in Accumulator
        MOV B, 31H          ; divisor is taken in register B
        DIV AB              ; quotient in A and remainder in B

        MOV 32H, A          ; quotient stored in 32H memory
        MOV 33H, B          ; remainder stored in 33H memory

STOP:   SJMP STOP
```