Book: Digital Computer Electronics
3rd Edition.

Lec 7.

CSE-304

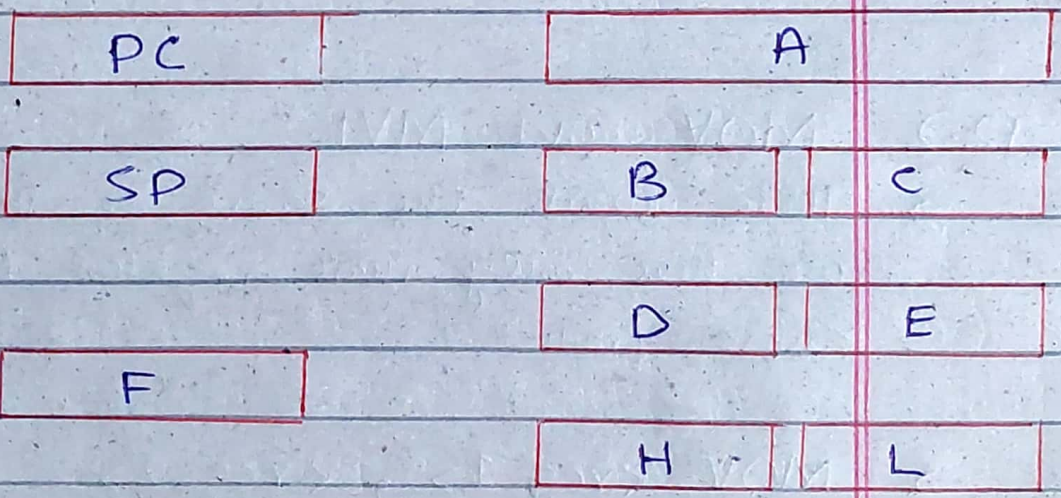
COA NOTES We-17, Jan 2024

Chapter # 12: SAP-3

→ SAP-3 is an 8-bit computer that is upward-compatible with the 8085 microprocessor.

→ In this chapter emphasis is on SAP-3 instruction set (includes all instructions of SAP-2 plus new).

12-1 Programming Model:



→ This is a diagram showing the CPU registers needed by a programer

→ Some register are same as SAP-2 Like, the PC (16-bit counter), the register A, B and C (used in arthemitic and logic operations).

→ As accumulator is only 8 bit wide

the range of unsigned number is 0 to 255 and for signed 2's complement is -128 to +127.

→ SAP-3 has additional 8-bit CPU register (D, E, H & L) which can be loaded with MOV and MVI.
→ The F-register stores flag bits of S, Z and others.
→ The stack pointer (SP), a 16-bit register control portion of memory known as stack.

12.2 MOV and MVI.

→ These work the same as in SAP-2. we have only more register to chose from.

→ MOV <reg_1>, <reg_2>
reg1 = A, B, C, D, E, H or L
reg2 = A, B, C, D, E, H or L
→ This instruction sends data in reg-2 to reg-1.
reg-1 ← reg-2.
→ The data is copied nondestructively into reg-1.

→ MVI loads the register with the byte that follows imidiatly

→ MVI <reg>, <byte>
reg ← byte.
where reg = A, B, C, D, E, H, or L.

→ More register make program writing simpler. It also make SAP-3 faster because we can use more MOV and MVI instruction and few MRIs.

12.3 Arithematic Instructions:

12.3.1 Carry-flag: ~~Instructions~~:

→ we need to detect overflows, sum or difference that lie outside the normal range of the accumulator. This is where the carry flag comes in.

→ ~~Two~~ The adder/subtract produces a sum and a carry. In simple computers the carry is disregarded, SAP-3 however takes the carry into account.

→ During an add operation the CY is called a carry. During a

a subtract operation the CY is reffered to as a borrow.

CY	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0

→ The carry/borrow is stored in a special flip-flop called the carry flag. designated CY.

12.3.2 Carry Flag Instructions

→ There are two instruction to controll the carry bit. i.e

1. STC (set carry) used to set the CY flag. It produces CY = 1.
2. CMC (complement the carry) it produce the complement of CY. i.e $CY = CY'$

→ If we want to reset carry flag we have to execute STC then CMC.

12.3.3 ADD. Instruction

→ ADD <reg>

where reg = A, B, C, D, E, H or L.

→ CY, A ← A + reg

→ It also set/reset carry flag.

12.3.4 ADC Instruction

→ Add with carry.

→ add the content of specified register plus carry flag with content of accumulator.

→ ADC <reg>

where reg = A, B, C, D, E, H or L

→ CY, A ← A + reg + CY

12.3.5 SUB Instruction

→ SUB <reg>

where reg = A, B, C, D, E, H or L

→ CY, A ← A − Reg

→ It set or rest the CY flag depends on borrow produced.

12.3.6 SBB Instruction

→ subtract with borrow.

→ It subtract the content of a specified register and the CY flag from accumulator contents.

→ SBB <reg>

where reg = A, B, C, D, E, H or L

→ CY, A ← A − [reg + CY]

12.4 Increments, decrements and rotates:

→ Increment and decrement are similar to SAP-2, but rotates are different because of carry flag.

12.4.1 Increment

→ INR <reg>

where reg = A, B, C, D, E, H or L.

→ It has no effect on carry flag but can affect zero/sign flags.

→ reg ← reg + 1.

12.4.2 Decrement.

→ DCR <reg>

where reg = A, B, C, D, E, H or L

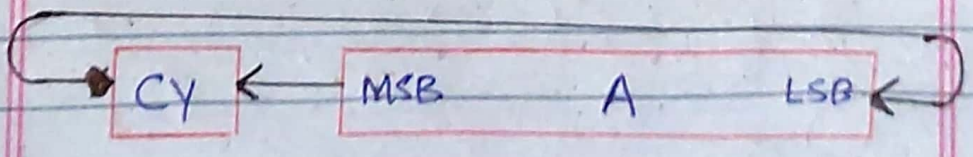
→ It also doesn't affect carry flag, but as before, it does affect sign or zero flags.

→ reg ← reg − 1

12.4.3 Rotate all left:

→ RAL (rotate all Left).

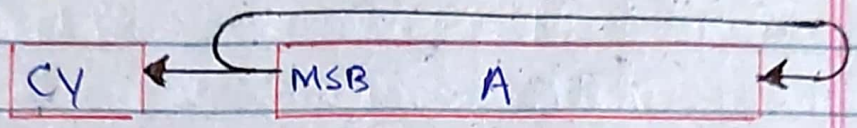
→ All bits including the CY flag are rotated to left.



12.4.4 Rotate Left with carry:

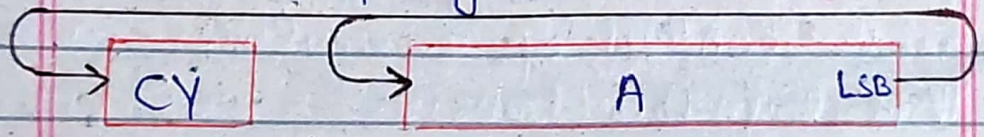
→ RLC (rotate left with carry).

→ The accumulator bits are rotated left and the MSB is saved in CY flag.



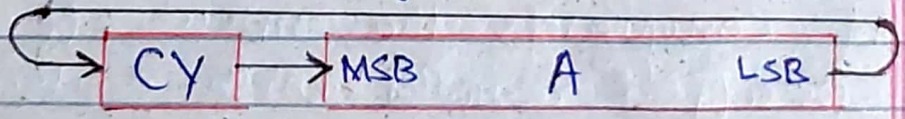
12.4.5 Rotate right with carry (RRC):

→ The accumulator bits are rotated ~~left~~ right and the LSB is saved in CY flag.



12.4.6 Rotate all right (RAR):

→ This rotates all the bits including the CY flag to the right.



12.4.7 Multiply and Divide by 2:

→ Rotating has the effect of multiplying or dividing the accumulator contents by a factor of 2.

→ RAL has the effect of multiplying by 2.

→ RAR has the effect of dividing by 2.

12.5 Logic Instructions:

→ SAP-3 Logic instructions are almost the same as in SAP-2.

i.e ANA <reg>
ORA <reg>
XRA <reg>

where reg = A, B, C, D, E, H or L.

These instruction will AND, OR or XOR the content of specified register with content of accumulator on a bit-by-bit basis.

→ The only new logic instruction is the CMP, formatted as

CMP <reg>

It compares the content of specified register with content of accumulator.

the zero flag indicates the output of these comparison as:

$$Z = \begin{cases} 1 & ; \text{If } A = reg. \\ 0 & ; \text{If } A \neq reg. \end{cases}$$

→ The comparison is done by subtracting the content of specified

register are subtracted from content of temporary register. Thus zero flag is affected and we get the result.

12.6 Arithemetics and Logi immediates:

→ The inidiate instructions from SAP-2 that carry over to SAP-3 are

ANI <byte>
ORI <byte>
XRI <byte>

where 'byte' is anded, ored or xored with accumulator respectively.

→ SAP-3 has the followin immediate instructions.

ADI <byte>
ACI <byte>
SUI <byte>
SBI <byte>
CPI <byte>

where the immediate 'byte' is added, ~~with~~ added with CY, subtracted, subtracted with CY, compared with the accumulator byte immediatly.

12.7 Jump Instructions:

→ here are SAP-2 jump instruction that become part of SAP-3.

JMP	<address>	unconditional
JM	<address>	if minus (conditional)
JZ	<address>	if zero (conditional)
JNZ	<address>	if not zero (conditional)

→ The JP instruction has the opposite effect of JM. It jumps to an address if positive. (sign flag = 0).

JP <address>

→ The JC instruction means to jump to the specified address if the carry flag is set (CY = 1).

JC <address>

→ The JNC is jump to an address if carry flag is not set (CY=0).

JNC <address>.

→ Beside sign, zero and carry flags; SAP-3 has parity flag designated P. If result of Acc has even no. of 1's the parity is set; If an odd number of 1's the parity is reset.

S	Z	0	0	0	P	0	CY

F-register stores flags

→ The JPE <address> produce a jump to specified address if parity flag is set (even parity).

→ The JPO <address> results in a jump when parity flag is reset (odd parity).

12.8 Extended-Register Instructions:

→ In SAP-3 during execution of certain instructions, the CPU registers are cascaded, as shown.

Register Pairs-			
B	+	C	B
D	+	E	D
H	+	L	H

→ The pairs are B with C, D with E and H with L.

→ Through thes instruction, the letter X stands for "extended register".

12.8.1 LXI (Load Extended Inidiate):

→ LXI <B|D|H>, dble

wher B = BC, D = DE and H = HL

the dble stands for double byte.

→ This instruction loads the specified register pair with the inidiate double byte: eg LXI B, 90FFH.

with upper and lower bytes

Here the B and C registers are loaded to get

B = 90H and C = FF H.

hence BC = 90FF H.

12.8.2 DAD (double-add):

→ DAD <B|D|H>

where B = BC, D = DE & H = HL.

→ This instruction adds the content of the specified register pair to the content of "HL" register pair. the result is then stored in the HL register pair.

HL ← HL + <B|D|H>.

→ This instruction can affect the carry flag CY.

12.8.3 INX (Increment the extended register)
DCX (Decrement the extended register):

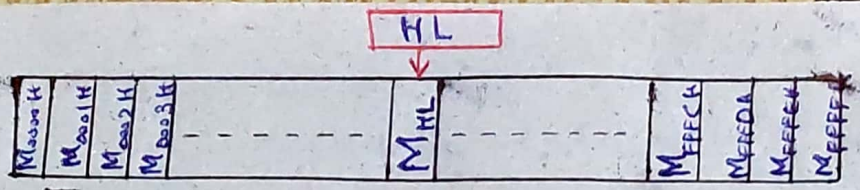
→ INX <B|D|H>,

→ DCX <B|D|H>,

where B = BC, D = DE and H = HL.

→ These instructions are used to increment/decrement a given extended register with value 1.

→ INX and DCX has no effect on the flags



The HL pointer to a data location.

12.9 Indirect Instructions:

→ As PC points to the memory location where the next instruction is stored, hence also known as "instruction pointer".

→ The HL register pair points to the location where the data is stored, hence also known as "data pointer". as shown in fig above.

→ In some SAP-3 instructions, the contents of the HL register pairs are used as the address for data in memory. The content are send to MAR and data read/write is performed.

→ Whenever an instruction uses HL pointer, the addressing is called "indirect addressing".

→ An indirect read is performed if we execute the instruction

MOV <reg>, M

where reg = A, B, C, D, E, H, or L

and M = M_{HL}

This instruction says to load the specified register with the data

addressed by HL. eg
If HL = 3000H and M_{3000H} = 87H
the after MOV C, M
produces C = 87 H.

→ An Indirect write is performed if we execute the instruction:

MOV M, <reg>

when M = M_{HL}
and reg = A, B, C, D, E, H or L.
This says to load the memory location addressed by HL with the content of the specified register. eg:
if HL = E300H and B = F2H
then MOV M, B
produce M_{E300H} = F2H.

→ An indirect-immediate write is performed if we execute the instruction:

MVI M, <byte>

This is instruction says to store the immediat byte in memor location ~~perform~~ pointed by HL.

→ Here are some more indirect instructions that uses HL pointer. In each of these, M. is the memory location addressed by HL. Think of M as another register where data is stored. Each instruction operates on this data as previously described.

ADD M; ADC M,
SUB M, SBB M,
INR M, DCR M,
ANA M, ORA M,
XRA M, CMP M.

12.10 Stack instructions:-

→ A "stack" is a portion of memory set aside primarily for saving return addresses.

→ In SAP-2 the addresses FFFEH and FFFFH are used exclusively for saving the return addresses of subroutine call.

→ In SAP-3, the programmer decides where to locate the stack and how large to make it. The stack can be located anywhere in memory, but once they have setup the Stack

they no longer use that portion of memory for program and data. The stack becomes a special space in memory where return addresses of subroutine calls are stored.

→ The instruction that read or write into the stack are called stack instructions.

→ Stack instructions use indirect addressing because a 16-bit register called the stack pointer (SP) holds the address of the desired memory location.

→ To initialize the stack pointer we can use the extended immediate instruction

LXI SP, dble

12-10-1 PUSH instructions:

→ The content of accumulator and the flag register are known as "program status word" (PSW).

i.e PSW = AF

where A = content of accumulator are high byte and F = content of flag register is the lower byte.

→ When calling subroutines we have to save PSW and may save other register content to resume the program after subroutine is executed.

→ PUSH instruction allow us to save data in a stack. i.e

PUSH <B|D|H|PSW>

where B = BC, D = DE, H = HL or PSW = AF.

→ When PUSH is executed the following things happen

1. SP is decremented ∴SP--
2. High byte is stored in M_{SP-1}
3. SP is decremented ∴SP--
4. lower byte is stored in M_{SP-2}

→ For example; If SP = 2100 H, AF = 1234 H DE = 5678 H, HL = 9A25 H.

the executing

PUSH PSW, PUSH D, PUSH H

produce

$M_{20FFH} = 12H$, $M_{20FEH} = 34H$

$M_{20FD} = 56H$, $M_{20FCH} = 78H$
$M_{20FBH} = 9AH$, $M_{20FAH} = 25H$.

12.10.2 POP Instructions:

→ This instruction is used to read the data from Stack. ie

POP <B|D|H|PSW>

where B=BC, D=DE, H=HL or PSW=AF.

→ When pop is executed, the following happens:

1. The low byte is read from the memory location addressed by the stack pointer in lower half of the specified register
2. The stack pointer is incremented.
3. The high byte is read and stored in upper half of specified register
4. The stack pointer is incremented

→ for example (above) if SP=20FAH then by executing POP H we get

L = M_{20FA} = 25H and
H = M_{20FB} = 9AH then
H=9A25H and SP=20FCH.

12-10-3 CALL and RET:

→ Main purpose of stack is to save return addresses automatically when using CALLs.

CALL <address>.

→ During execution of CALL.

1. Content of PC are pushed onto stack
2. Starting address of subroutine is loaded into the PC. In this way the subroutine is completed
3. On completion, a RET instruction pops the return address off the stack into the PC.

→ The stack operation is automatic during CALL and RET instructions.

12-10-4 Conditional calls and returns:

→ Here is a list of SAP-3 conditional calls;

CNZ <address>	;	If Z = 0	} zero flag
CZ < " >	;	If Z = 1	
CNC < " >	;	If CY =	} carry flag
CC < " >	;	If CY =	
CPO < " >	;	If P = 0	} parity flag
CPE < " >	;	If P = 1	
CP < " >	;	If S = 0	} sign flag
CM < " >	;	If S = 1	

positive

→ The return for sub routine may also be conditional, here is List of all conditional returns.

```
RNZ     ; return if not zero
RZ      ; return if zero
RNC     ; return if not carry
RC      ; return if carry
RPO     ; return if odd parity
RPE     ; return if even parity
RP      ; return if positive
RM      ; return if minus.
```

→ These are similar to the conditional jumps discussed earlier.

Instruction	Opcode	Bytes	Flags	T States	Description
INR A	3C	1	SZP-	4	A = A + 1
INR B	04	1	SZP-	6	B = B + 1
INR C	0C	1	SZP-	6	C = C + 1
INR D	14	1	SZP-	6	D = D + 1
INR E	1C	1	SZP-	6	E = E + 1
INR H	24	1	SZP-	6	H = H + 1
INR L	2C	1	SZP-	6	L = L + 1
INR M	34	1	SZP-	7	[HL] = [HL] + 1
DCR A	3D	1	SZP-	4	A = A - 1
DCR B	05	1	SZP-	6	B = B - 1
DCR C	0D	1	SZP-	6	C = C - 1
DCR D	15	1	SZP-	6	D = D - 1
DCR E	1D	1	SZP-	6	E = E - 1
DCR H	25	1	SZP-	6	H = H - 1
DCR L	2D	1	SZP-	6	L = L - 1
DCR M	35	1	SZP-	7	[HL] = [HL] - 1
INX B	03	1	----	4	BC = BC + 1
INX D	13	1	----	4	DE = DE + 1
INX H	23	1	----	4	HL = HL + 1
INX SP	33	1	----	4	SP = SP + 1
DCX B	0B	1	----	4	BC = BC - 1
DCX D	1B	1	----	4	DE = DE - 1
DCX H	2B	1	----	4	HL = HL - 1
DCX SP	3B	1	----	4	SP = SP - 1
DAD B	09	1	---C	12	HL = HL + BC
DAD D	19	1	---C	12	HL = HL + DE
DAD H	29	1	---C	12	HL = HL + HL
DAD SP	39	1	---C	12	HL = HL + SP
ADD A	87	1	SZPC	4	A = A + A
ADD B	80	1	SZPC	5	A = A + B
ADD C	81	1	SZPC	5	A = A + C
ADD D	82	1	SZPC	5	A = A + D
ADD E	83	1	SZPC	5	A = A + E
ADD H	84	1	SZPC	5	A = A + H
ADD L	85	1	SZPC	5	A = A + L
ADD M	86	1	SZPC	6	A = A + [HL]
ADI byte	C6	2	SZPC	6	A = A + *byte*
ADC A	8F	1	SZPC	4	A = A + A + FlagC
ADC B	88	1	SZPC	5	A = A + B + FlagC
ADC C	89	1	SZPC	5	A = A + C + FlagC
ADC D	8A	1	SZPC	5	A = A + D + FlagC
ADC E	8B	1	SZPC	5	A = A + E + FlagC
ADC H	8C	1	SZPC	5	A = A + H + FlagC
ADC L	8D	1	SZPC	5	A = A + L + FlagC
ADC M	8E	1	SZPC	6	A = A + [HL] + FlagC
ACI byte	CE	2	SZPC	6	A = A + *byte* + FlagC
SUB A	97	1	SZPC	4	A = A - A
SUB B	90	1	SZPC	5	A = A - B
SUB C	91	1	SZPC	5	A = A - C
SUB D	92	1	SZPC	5	A = A - D
SUB E	93	1	SZPC	5	A = A - E
SUB H	94	1	SZPC	5	A = A - H
SUB L	95	1	SZPC	5	A = A - L
SUB M	96	1	SZPC	6	A = A - [HL]
SUI byte	D6	2	SZPC	6	A = A - *byte*
SBB A	9F	1	SZPC	4	A = A - *byte* - FlagC
SBB B	98	1	SZPC	5	A = A - *byte* - FlagC
SBB C	99	1	SZPC	5	A = A - *byte* - FlagC
SBB D	9A	1	SZPC	5	A = A - *byte* - FlagC
SBB E	9B	1	SZPC	5	A = A - *byte* - FlagC
SBB H	9C	1	SZPC	5	A = A - *byte* - FlagC
SBB L	9D	1	SZPC	5	A = A - *byte* - FlagC
SBB M	9E	1	SZPC	6	A = A - *byte* - FlagC
SBI byte	DE	2	SZPC	6	A = A - *byte* - FlagC
ANA A	A7	1	SZPC	4	A = A and A
ANA B	A0	1	SZPC	5	A = A and B
ANA C	A1	1	SZPC	5	A = A and C
ANA D	A2	1	SZPC	5	A = A and D
ANA E	A3	1	SZPC	5	A = A and E
ANA H	A4	1	SZPC	5	A = A and H
ANA L	A5	1	SZPC	5	A = A and L
ANA M	A6	1	SZPC	6	A = A and [HL]
ANI byte	E6	2	SZPC	6	A = A and *byte*
ORA A	B7	1	SZPC	4	A = A or A
ORA B	B0	1	SZPC	5	A = A or B
ORA C	B1	1	SZPC	5	A = A or C
ORA D	B2	1	SZPC	5	A = A or D
ORA E	B3	1	SZPC	5	A = A or E
ORA H	B4	1	SZPC	5	A = A or H
ORA L	B5	1	SZPC	5	A = A or L
ORA M	B6	1	SZPC	6	A = A or [HL]
ORI byte	F6	2	SZPC	6	A = A or *byte*
XRA A	AF	1	SZPC	4	A = A xor A
XRA B	A8	1	SZPC	5	A = A xor B
XRA C	A9	1	SZPC	5	A = A xor C
XRA D	AA	1	SZPC	5	A = A xor D
XRA E	AB	1	SZPC	5	A = A xor E
XRA H	AC	1	SZPC	5	A = A xor H
XRA L	AD	1	SZPC	5	A = A xor L
XRA M	AE	1	SZPC	6	A = A xor [HL]
XRI byte	EE	2	SZPC	6	A = A xor *byte*
RLC	07	1	---C	4	Shift A left and FlagC = A[7]
RAL	17	1	---C	4	Shift A left and shift FlagC into A[(
RAR	1F	1	---C	4	Shift A right and shift FlagC into A[
RRC	0F	1	---C	4	Shift A right and FlagC = A[0]
CMA	2F	1	----	4	A = ~A
STC	37	1	---C	4	FlagC = 1
CMC	3F	1	---C	4	FlagC = ~FlagC
CMP A	BF	1	SZPC	4	FlagZ = 1 if A == A
CMP B	B8	1	SZPC	5	FlagZ = 1 if A == B
CMP C	B9	1	SZPC	5	FlagZ = 1 if A == C
CMP D	BA	1	SZPC	5	FlagZ = 1 if A == D
CMP E	BB	1	SZPC	5	FlagZ = 1 if A == E
CMP H	BC	1	SZPC	5	FlagZ = 1 if A == H
CMP L	BD	1	SZPC	5	FlagZ = 1 if A == L

CMP H	BC	1	SZPC	5	FlagZ = 1 if A == H
CMP L	BD	1	SZPC	5	FlagZ = 1 if A == L
CMP M	BE	1	SZPC	6	FlagZ = 1 if A == [HL]
CPI byte	FE	2	----	6	FlagZ = 1 if A == *byte*
LDA addr	3A	3	----	11	Load A with [*addr*]
LDAX B	0A	1	----	8	Load A with [BC]
LDAX D	1A	1	----	8	Load A with [DE]
LXI B, dble	01	3	----	10	Load BC with *dble*
LXI D, dble	11	3	----	10	Load DE with *dble*
LXI H, dble	21	3	----	10	Load HL with *dble*
LXI SP, dble	31	3	----	10	Load SP with *dble*
STA addr	32	3	----	11	Store A at [*addr*]
STAX B	02	1	----	8	Store A at [BC]
STAX D	12	1	----	8	Store A at [DE]
LHLD addr	2A	3	----	14	Load HL with [*addr*]
SHLD addr	22	3	----	14	Store HL at [*addr*]
MOV A, A	7F	1	----	4	A = A
MOV A, B	78	1	----	4	A = B
MOV A, C	79	1	----	4	A = C
MOV A, D	7A	1	----	4	A = D
MOV A, E	7B	1	----	4	A = E
MOV A, H	7C	1	----	4	A = H
MOV A, L	7D	1	----	4	A = L
MOV A, M	7E	1	----	5	A = [HL]
MOV B, A	47	1	----	4	B = A
MOV B, B	40	1	----	4	B = B
MOV B, C	41	1	----	4	B = C
MOV B, D	42	1	----	4	B = D
MOV B, E	43	1	----	4	B = E
MOV B, H	44	1	----	4	B = H
MOV B, L	45	1	----	4	B = L
MOV B, M	46	1	----	5	B = [HL]
MOV C, A	4F	1	----	4	C = A
MOV C, B	48	1	----	4	C = B
MOV C, C	49	1	----	4	C = C
MOV C, D	4A	1	----	4	C = D
MOV C, E	4B	1	----	4	C = E
MOV C, H	4C	1	----	4	C = H
MOV C, L	4D	1	----	4	C = L
MOV C, M	4E	1	----	5	C = [HL]
MOV D, A	57	1	----	4	D = A
MOV D, B	50	1	----	4	D = B
MOV D, C	51	1	----	4	D = C
MOV D, D	52	1	----	4	D = D
MOV D, E	53	1	----	4	D = E
MOV D, H	54	1	----	4	D = H
MOV D, L	55	1	----	4	D = L
MOV D, M	56	1	----	5	D = [HL]
MOV E, A	5F	1	----	4	E = A
MOV E, B	58	1	----	4	E = B
MOV E, C	59	1	----	4	E = C
MOV E, D	5A	1	----	4	E = D
MOV E, E	5B	1	----	4	E = E
MOV E, H	5C	1	----	4	E = H
MOV E, L	5D	1	----	4	E = L
MOV E, M	5E	1	----	5	E = [HL]
MOV H, A	67	1	----	4	H = A
MOV H, B	60	1	----	4	H = B
MOV H, C	61	1	----	4	H = C
MOV H, D	62	1	----	4	H = D
MOV H, E	63	1	----	4	H = E
MOV H, H	64	1	----	4	H = H
MOV H, L	65	1	----	4	H = L
MOV H, M	66	1	----	5	H = [HL]
MOV L, A	6F	1	----	4	L = A
MOV L, B	68	1	----	4	L = B
MOV L, C	69	1	----	4	L = C
MOV L, D	6A	1	----	4	L = D
MOV L, E	6B	1	----	4	L = E
MOV L, H	6C	1	----	4	L = H
MOV L, L	6D	1	----	4	L = L
MOV L, M	6E	1	----	5	L = [HL]
MOV M, A	77	1	----	5	[HL] = A
MOV M, B	70	1	----	5	[HL] = B
MOV M, C	71	1	----	5	[HL] = C
MOV M, D	72	1	----	5	[HL] = D
MOV M, E	73	1	----	5	[HL] = E
MOV M, H	74	1	----	5	[HL] = H
MOV M, L	75	1	----	5	[HL] = L
MVI A, byte	3E	2	----	6	A = *byte*
MVI B, byte	06	2	----	6	B = *byte*
MVI C, byte	0E	2	----	6	C = *byte*
MVI D, byte	16	2	----	6	D = *byte*
MVI E, byte	1E	2	----	6	E = *byte*
MVI H, byte	26	2	----	6	H = *byte*
MVI L, byte	2E	2	----	6	L = *byte*
MVI M, byte	36	2	----	8	[HL] = *byte*
PUSH B	C5	1	SZPC	9	Push value in BC onto the stack
PUSH D	D5	1	SZPC	9	Push value in DE onto the stack
PUSH H	E5	1	SZPC	9	Push value in HL onto the stack
PUSH PSW	F5	1	SZPC	9	Push value in AF onto the stack
POP B	C1	1	SZPC	9	Pop value on stack into BC
POP D	D1	1	SZPC	9	Pop value on stack into DE
POP H	E1	1	SZPC	9	Pop value on stack into HL
POP PSW	F1	1	SZPC	9	Pop value on stack into AF
CALL addr	CD	3	----	16	Call function at *addr*
CP addr	F4	3	----	4/16	Call function at *addr* if FlagS == 0
CM addr	FC	3	----	4/16	Call function at *addr* if FlagS == 1
CNZ addr	C4	3	----	4/16	Call function at *addr* if FlagZ == 0
CZ addr	CC	3	----	4/16	Call function at *addr* if FlagZ == 1
CPO addr	E4	3	----	4/16	Call function at *addr* if FlagP == 0
CPE addr	EC	3	----	4/16	Call function at *addr* if FlagP == 1
CNC addr	D4	3	----	4/16	Call function at *addr* if FlagC == 0
CC addr	DC	3	----	4/16	Call function at *addr* if FlagC == 1
RET	C9	1	----	4	Return from function
RP	F0	1	----	4/10	Return from function if FlagS == 0
RM	F8	1	----	4/10	Return from function if FlagS == 1
RNZ	C0	1	----	4/10	Return from function if FlagZ == 0
RZ	C8	1	----	4/10	Return from function if FlagZ == 1
RPO	E0	1	----	4/10	Return from function if FlagP == 0
RPE	E8	1	----	4/10	Return from function if FlagP == 1
RNC	D0	1	----	4/10	Return from function if FlagC == 0
RC	D8	1	----	4/10	Return from function if FlagC == 1
JMP addr	C3	3	----	4	Jump to *addr*
JP addr	F2	3	----	4/9	Jump to *addr* if FlagS == 0
JM addr	FA	3	----	4/9	Jump to *addr* if FlagS == 1
JNZ addr	C2	3	----	4/9	Jump to *addr* if FlagZ == 0
JZ addr	CA	3	----	4/9	Jump to *addr* if FlagZ == 1
JPO addr	E2	3	----	4/9	Jump to *addr* if FlagP == 0
JPE addr	EA	3	----	4/9	Jump to *addr* if FlagP == 1
JNC addr	D2	3	----	4/9	Jump to *addr* if FlagC == 0
JC addr	DA	3	----	4/9	Jump to *addr* if FlagC == 1
NOP	00	1	----	4	Do nothing
HLT	76	1	----	4	Halt execution