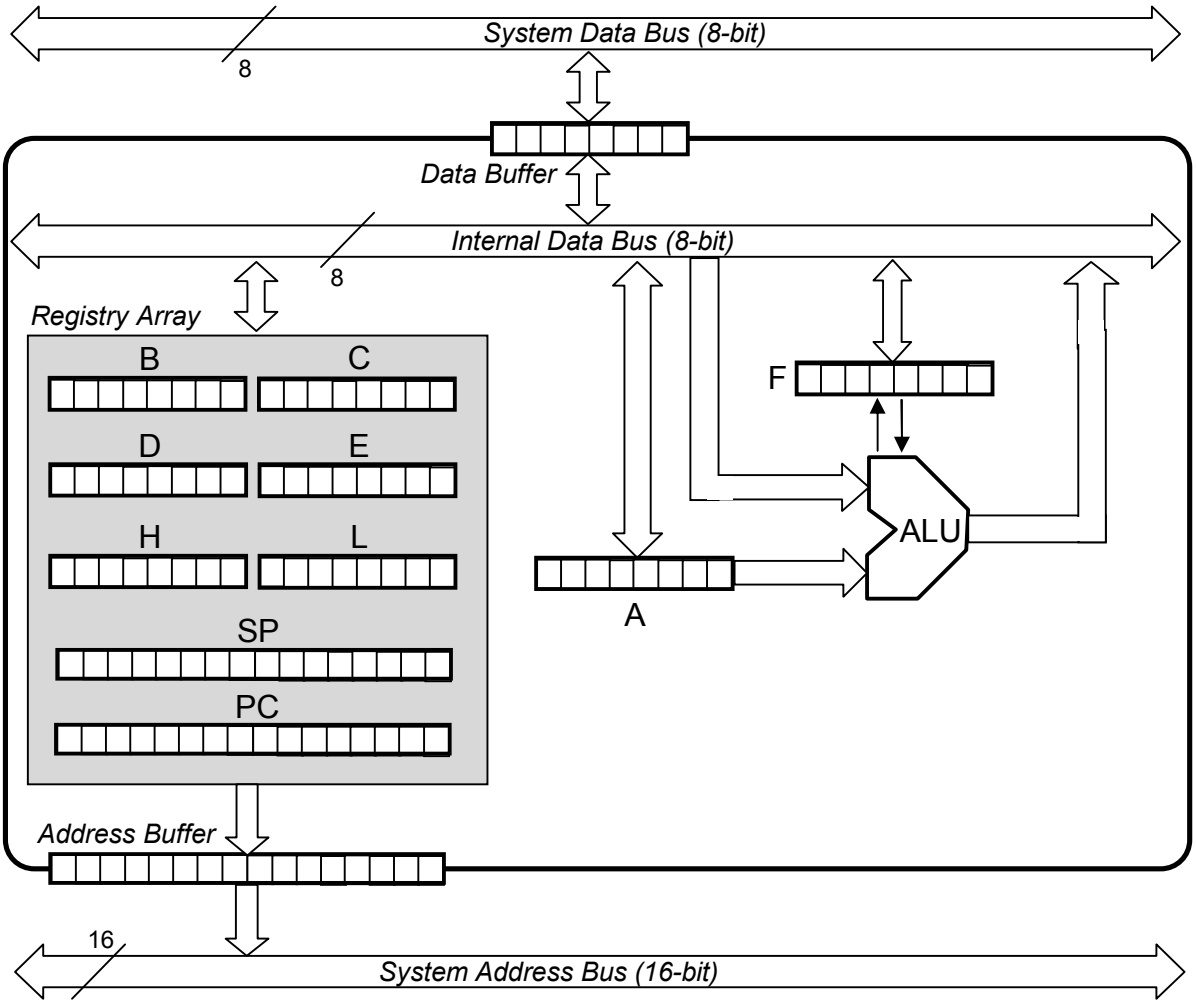
Intel 8080 CPU block diagram



Internal register addressing:

8-bit register (R)	3-bit address
A	111
B	000
C	001
D	010
E	011
H	100
L	101

16-bit register pair (P)	2-bit address
BC (B)	00
DE (D)	01
HL (H)	10
SP	11

Flag register (F) structure:

S	Z	0	AC	0	P	1	CY

Where:

- CY – *carry flag*, set to 1 if the result of arithmetical or logical operation excides the 8-bit A register (Accumulator) or operation needs to borrow one bit – in other words it's carry/borrow from/to the bit 7;
- P – *parity flag*, set to 1 if the result of arithmetical operation has even number of bits equal to 1, set to 0 if this number is odd (in Z80 CPU this flag is also the *overflow* indicator for TC arithmetical operations);
- AC – *auxiliary carry flag*, set to 1 if there was carry from bit 3 to 4 in the result of arithmetical operation (useful in programming operations with packed BCD numbers);
- Z – *zero flag*, set to 1 if the result of arithmetical operation is zero;
- S – *sign flag*, equal to the most significant (oldest) bit of the result of arithmetical operation.

Intel 8080 instruction set architecture

Instruction code formats:

– One byte instructions:

i_7	i_6	i_5	i_4	i_3	i_2	i_1	i_0

All bits used to encode instruction, no operands.

i_7	i_6	i_5	i_4	i_3	***r***	***r***	***r***

One operand in 8-bit internal register A to L.

i_7	i_6	***r***	***r***	***r***	i_2	i_1	i_0

One operand in 8-bit internal register A to L.

i_7	i_6	***r1***	***r1***	***r1***	***r2***	***r2***	***r2***

Two operands in two 8-bit internal registers A to L.

i_7	i_6	***p***	***p***	i_3	i_2	i_1	i_0

One operand in 16-bit register pair BC, DE, HL or in SP.

– Two byte instructions:

i_7	i_6	i_5	i_4	i_3	i_2	i_1	i_0
d_7	d_6	d_5	d_4	d_3	d_2	d_1	d_0

All bits of the first byte used to encode instruction,
the second byte is the *immediate* operand (argument).

i_7	i_6	***r***	***r***	***r***	i_2	i_1	i_0
d_7	d_6	d_5	d_4	d_3	d_2	d_1	d_0

First operand in 8-bit internal register A to L,
the second operand is *immediate*.

i_7	i_6	i_5	i_4	i_3	i_2	i_1	i_0
p_7	p_6	p_5	p_4	p_3	p_2	p_1	p_0

All bits of the first byte used to encode instruction,
the second byte is the 8-bit address of I/O port.

– Three byte instructions with immediate memory addressing:

i_7	i_6	***p***	***p***	i_3	i_2	i_1	i_0
l_7	l_6	l_5	l_4	l_3	l_2	l_1	l_0
h_7	h_6	h_5	h_4	h_3	h_2	h_1	h_0

First operand in 16-bit register pair BC, DE, HL or in SP,
the second byte is *lower* part of 16-bit second operand,
the third byte is *higher* part of the 16-bit second operand.

– Three byte instructions with direct memory addressing:

i_7	i_6	i_5	i_4	i_3	i_2	i_1	i_0
l_7	l_6	l_5	l_4	l_3	l_2	l_1	l_0
h_7	h_6	h_5	h_4	h_3	h_2	h_1	h_0

All bits of the first byte used to encode instruction,
the second byte is *lower* part of 16-bit address in memory,
the third byte is *higher* part of address of the operand.

i_7	i_6	i_5	i_4	i_3	i_2	i_1	i_0
l_7	l_6	l_5	l_4	l_3	l_2	l_1	l_0
h_7	h_6	h_5	h_4	h_3	h_2	h_1	h_0

All bits of the first byte used to encode instruction,
the second byte is *lower* part of 16-bit *address*,
the third byte is *higher* part of *address* of jump or call.

Internal registers A, B, C, D, E, H, L (***rrr***), pairs of registers BC, DE, HL and SP register (***pp***) are addressed according to rules shown on first page.

Intel's processors always store data longer than one byte in the *lower-to-higher* byte order – *little endian* convention.

Mnemonics used for instructions are copyrighted, so other processors which instruction lists are compatible with 8080 (Z80 for example) have different names and mnemonics for the same instructions.

Instruction list:

Data transfer instructions

MOV *R1, R2* (*Move register*)

0	1	***r1***	***r1***	***r1***	***r2***	***r2***	***r2***

R1 ← R2 *data from R2 is copied to R1*

MOV *R, M* (*Move from memory, address in HL*)

0	1	***r***	***r***	***r***	1	1	0

R ← [HL] *data from memory (address in HL) copied to R*

MOV *M, R* (*Move to memory, address in HL*)

0	1	1	1	0	***r***	***r***	***r***

[HL] ← R *data from R copied to memory (address in HL)*

MVI *R, data8* (*Move to register immediate*)

0	0	***r***	***r***	***r***	1	1	0
d_7	d_6	d_5	d_4	d_3	d_2	d_1	d_0

R ← data8 *1 byte (next to instruction) copied to R*

MVI *M, data8* (*Move to memory immediate*)

0	0	1	1	0	1	1	0
d_7	d_6	d_5	d_4	d_3	d_2	d_1	d_0

[HL] ← data8 *1 byte copied to memory (address in HL)*

LXI *P, data16* (*Load register pair immediate*)

0	0	***p***	***p***	0	0	0	1
l_7	l_6	l_5	l_4	l_3	l_2	l_1	l_0
h_7	h_6	h_5	h_4	h_3	h_2	h_1	h_0

P ← data16 *2 bytes copied to register pair*
"lower" register – 2nd byte
"higher" register – 3rd byte

LDA *addr16* (*Load accumulator direct*)

0	0	1	1	1	0	1	0
l_7	l_6	l_5	l_4	l_3	l_2	l_1	l_0
h_7	h_6	h_5	h_4	h_3	h_2	h_1	h_0

A ← [addr16] *1 byte copied to register A from memory*
lower byte of address – 2nd byte
higher byte of address – 3rd byte

STA *addr16* (*Store accumulator direct*)

0	0	1	1	0	0	1	0
l_7	l_6	l_5	l_4	l_3	l_2	l_1	l_0
h_7	h_6	h_5	h_4	h_3	h_2	h_1	h_0

[addr16] ← A *1 byte copied to memory from register A*
lower byte of address – 2nd byte
higher byte of address – 3rd byte

LHLD *addr16* (*Load H and L direct*)

0	0	1	1	0	0	1	0
l_7	l_6	l_5	l_4	l_3	l_2	l_1	l_0
h_7	h_6	h_5	h_4	h_3	h_2	h_1	h_0

L ← [addr16] *2 bytes copied from memory to HL*
H← [addr16 + 1] *lower byte of address – 2nd byte*
higher byte of address – 3rd byte

SHLD *addr16* (Store *H and L direct*)

0	0	1	0	0	0	1	0
l_7	l_6	l_5	l_4	l_3	l_2	l_1	l_0
h_7	h_6	h_5	h_4	h_3	h_2	h_1	h_0

[addr16] ← L *2 bytes copied from HL to memory*
[addr16 + 1] ← H *lower byte of address – 2nd byte*
higher byte of address – 3rd byte

LDAX *P* (*Load accumulator indirect, address in BC or DE*)

0	0	***p***	***p***	1	0	1	0

A ← [P] *data from memory (address in P) copied to A*

STAX *P* (*Store accumulator indirect, address in BC or DE*)

0	0	***p***	***p***	0	0	1	0

[P] ← A *data from A copied to memory (address in P)*

XCHG (*Exchange H and L with D and E*)

1	1	1	0	1	0	1	1

H ↔ D L ↔ E *data in HL and DE is switched*

Arithmetical instructions

ADD *R* (*Add register*)

1	0	0	0	0	***r***	***r***	***r***

A ← A + R *data from R is added to data in A*
flags affected: Z, S, P, CY, AC

ADD *M* (*Add memory, address in HL*)

1	0	0	0	0	1	1	0

A ← A + [HL] *data from memory is added to A*
flags affected: Z, S, P, CY, AC

ADI *data8* (*Add immediate*)

1	1	0	0	0	1	1	0
d_7	d_6	d_5	d_4	d_3	d_2	d_1	d_0

A ← A + data8 *one byte (next to instruction) added to A*
flags affected: Z, S, P, CY, AC

ADC *R* (*Add register with carry*)

1	0	0	0	1	***r***	***r***	***r***

A ← A + R + CY *data from R and CY flag are added to A*
flags affected: Z, S, P, CY, AC

ADC *M* (*Add memory with carry, address in HL*)

1	0	0	0	1	1	1	0

A ← A + [HL] + CY *data from memory and CY added to A*
flags affected: Z, S, P, CY, AC

ACI *data8* (*Add immediate with carry*)

1	1	0	0	1	1	1	0
d_7	d_6	d_5	d_4	d_3	d_2	d_1	d_0

A ← A + data8 + CY *one byte and CY added to A*
flags affected: Z, S, P, CY, AC

SUB *R* (*Subtract register*)

1	0	0	1	0	***r***	***r***	***r***

A ← A - R *data from R is subtracted from A*
flags affected: Z, S, P, CY, AC

SUB *M* (*Subtract memory, address in HL*)

1	0	0	1	0	1	1	0

A ← A - [HL] *data from memory is subtracted from A*
flags affected: Z, S, P, CY, AC

SUI *data8* (*Subtract immediate*)

1	1	0	1	0	1	1	0
d_7	d_6	d_5	d_4	d_3	d_2	d_1	d_0

A ← A - data8 *one byte subtracted from A*
flags affected: Z, S, P, CY, AC

SBB *R* (*Subtract register with borrow*)

1	0	0	1	1	***r***	***r***	***r***

A ← A - R - CY *R and CY are subtracted from A*
flags affected: Z, S, P, CY, AC

SBB *M* (*Subtract memory with borrow*)

1	0	0	1	1	1	1	0

A ← A - [HL] - CY *data from memory and CY subtracted from A, flags affected: Z, S, P, CY, AC*

SBI *data8* (*Subtract immediate with borrow*)

1	1	0	1	1	1	1	0
d_7	d_6	d_5	d_4	d_3	d_2	d_1	d_0

A ← A - data8 - CY *one byte and CY subtracted from A*
flags affected: Z, S, P, CY, AC

INR *R* (*Increment register*)

0	0	***r***	***r***	***r***	1	0	0

R ← R + 1 *Data in R is incremented by 1*
flags affected: Z, S, P, AC

INR *M* (*Increment memory, address in HL*)

0	0	1	1	0	1	0	0

[HL] ← [HL] + 1 *Data in memory is incremented by 1*
flags affected: Z, S, P, AC

DCR *R* (*Decrement register*)

0	0	***r***	***r***	***r***	1	0	1

R ← R - 1 *Data in R is decremented by 1*
flags affected: Z, S, P, AC

DCR *M* (*Decrement memory, address in HL*)

0	0	1	1	0	1	0	1

[HL] ← [HL] - 1 *Data in memory is decremented by 1*
flags affected: Z, S, P, AC

INX *P* (*Increment register pair*)

0	0	***p***	***p***	0	0	1	1

P ← P + 1 *Data in register pair P is incremented by 1*
flags affected: none

DCX *P* (*Decrement register pair*)

0	0	***p***	***p***	1	0	1	1

P ← P - 1 *Data in register pair P is decremented by 1*
flags affected: none

DAD *P* (*Decrement register pair*)

0	0	***p***	***p***	1	0	0	1

HL ← HL + P *Data in register pair P is added to HL*
flags affected: CY (from higher byte)

DAA (*Decimal adjust Accumulator*)

0	0	1	0	0	1	1	1

A ← adjust_{BCD}(A) *Data in A is adjusted as packed BCD:*
if $(a_3 \ldots a_0) > 9$ *or AC=1 then* $(a_3 \ldots a_0) \leftarrow (a_3 \ldots a_0) + 6$
if $(a_7 \ldots a_4) > 9$ *or CY=1 then* $(a_7 \ldots a_4) \leftarrow (a_7 \ldots a_4) + 6$
explanation: 6 is the 4-bit U2 code of -10
flags affected: Z, S, P, CY, AC

Logical instructions

ANA *R* (*AND with register*)

1	0	1	0	0	***r***	***r***	***r***

A ← A ∧ R *Bits in A logically multiplied with bits from R flags affected: Z, S, P, CY=0, AC=0*

ANA *M* (*AND with memory – address in HL*)

1	0	1	0	0	1	1	0

A ← A ∧ [HL] *Bits in A logically multiplied with bits from memory, flags affected: Z, S, P, CY=0, AC=0*

ANI *data8* (*AND immediate*)

1	1	1	0	0	1	1	0
d_7	d_6	d_5	d_4	d_3	d_2	d_1	d_0

A ← A ∧ data8 *Bits in A logically multiplied with bits from 2nd byte of instruction, flags affected: Z, S, P, CY=0, AC=0*

XRA *R* (*XOR with register*)

1	0	1	0	1	***r***	***r***	***r***

A ← A ⊗ R *Bits in A logically xor-ed with bits from R flags affected: Z, S, P, CY=0, AC=0*

XRA *M* (*XOR with memory – address in HL*)

1	0	1	0	1	1	1	0

A ← A ⊗ [HL] *Bits in A logically xor-ed with bits from memory, flags affected: Z, S, P, CY=0, AC=0*

XRI *data8* (*XOR immediate*)

1	1	1	0	1	1	1	0
d_7	d_6	d_5	d_4	d_3	d_2	d_1	d_0

A ← A ⊗ data8 *Bits in A logically xor-ed with bits from 2nd byte of instruction, flags affected: Z, S, P, CY=0, AC=0*

ORA *R* (*OR with register*)

1	0	1	1	0	***r***	***r***	***r***

A ← A ∨ R *Bits in A logically added with bits from R flags affected: Z, S, P, CY=0, AC=0*

ORA *M* (*OR with memory – address in HL*)

1	0	1	1	0	1	1	0

A ← A ∨ [HL] *Bits in A logically added with bits from memory, flags affected: Z, S, P, CY=0, AC=0*

ORI *data8* (*OR immediate*)

1	1	1	1	0	1	1	0
d_7	d_6	d_5	d_4	d_3	d_2	d_1	d_0

A ← A ∨ data8 *Bits in A logically added with bits from 2nd byte of instruction, flags affected: Z, S, P, CY=0, AC=0*

CMP *R* (*Compare with register*)

1	0	1	1	1	***r***	***r***	***r***

A - R *Data in R is subtracted from data in A, no result is stored, only flags are affected: Z, S, P, CY, AC*

CMP *M* (*Compare with memory – address in HL*)

1	0	1	1	1	1	1	0

A - [HL] *Data in memory is subtracted from data in A, no result is stored, only flags are affected: Z, S, P, CY, AC*

CPI *data8* (*Compare immediate*)

1	1	1	1	1	1	1	0
d_7	d_6	d_5	d_4	d_3	d_2	d_1	d_0

A - data8 *Data in 2nd byte of instruction is subtracted from data in A, only flags are affected: Z, S, P, CY, AC*

Comment:

Interpretation of "compare" operations is possible by checking Z and CY flags after execution:

if Z=1 then values compared are equal,
else (if Z=0)
if CY=0 then A > compared value,
else (if CY=1) A < compared value.

RLC (*Rotate left / rotate logically left*)

0	0	0	0	0	1	1	1

$A_{i+1} \leftarrow A_i$, $A_0 \leftarrow A_7$, $CY \leftarrow A_7$ *Bits in A shifted left, oldest bit copied to youngest bit and CY, flags affected: CY*

RRC (*Rotate right / rotate logically right*)

0	0	0	0	1	1	1	1

$A_i \leftarrow A_{i+1}$, $A_7 \leftarrow A_0$, $CY \leftarrow A_0$ *Bits in A shifted right, youngest bit copied to oldest bit and CY, flags affected: CY*

RAL (*Rotate left through carry / rotate arithmetically left*)

0	0	0	1	0	1	1	1

$A_{i+1} \leftarrow A_i$, $A_0 \leftarrow CY$, $CY \leftarrow A_7$ *Bits in A shifted left, CY copied to youngest bit, oldest bit copied to CY, flags affected: CY*

RAR (*Rotate right through carry / rotate arithmetically right*)

0	0	0	1	0	1	1	1

$A_i \leftarrow A_{i+1}$, $A_7 \leftarrow CY$, $CY \leftarrow A_0$ *Bits in A shifted right, CY copied to oldest bit, youngest bit copied to CY, flags affected: CY*

CMA (*Complement Accumulator*)

0	0	1	0	1	1	1	1

$A \leftarrow \neg A$ *Bitwise negation of A (one's complement) flags affected: none*

CMC (*Complement carry*)

0	0	1	1	1	1	1	1

$CY \leftarrow \neg CY$ *Negation (inversion) of CY flag flags affected: CY*

Branch instructions

Comment:

These instructions are passing control to the new address in program (not just to the address of next instruction). There are two basic types of branches:

- *unconditional* – just go to new address,
- *conditional* – check if particular condition (detected by status of one of the flags in F register) occurs and jump if so, continue with next instruction if not.

The conditions are encoded (inside the codes of instructions) according to this table:

Condition	**Mnemonic (*CND*)**	**Code (*ccc*)**
Not zero (Z = 0)	NZ	000
Zero (Z = 1)	Z	001
No carry (CY = 0)	NC	010
Carry (CY = 1)	C	011
Parity odd (P = 0)	PO	100
Parity even (P = 1)	PE	101
Plus (S = 0)	P	110
Minus (S = 1)	M	111

JMP *addr16* (*Jump*)

1	1	0	0	0	0	1	1
l_7	l_6	l_5	l_4	l_3	l_2	l_1	l_0
h_7	h_6	h_5	h_4	h_3	h_2	h_1	h_0

PC ← [addr16] *Unconditional jump to direct address*
lower byte of address – 2nd byte
higher byte of address – 3rd byte

J*CND addr16* (*Conditional jump*)

1	1	***c***	***c***	***c***	0	1	0
l_7	l_6	l_5	l_4	l_3	l_2	l_1	l_0
h_7	h_6	h_5	h_4	h_3	h_2	h_1	h_0

if (*CND*) then PC ← [addr16]
lower byte of address – 2nd byte
higher byte of address – 3rd byte

CALL *addr16* (*Call procedure*)

1	1	0	0	1	1	0	1
l_7	l_6	l_5	l_4	l_3	l_2	l_1	l_0
h_7	h_6	h_5	h_4	h_3	h_2	h_1	h_0

[SP-1] ← PC_H, [SP-2] ← PC_L, SP ← SP-2,
PC ← [addr16] *lower byte of address – 2nd byte*
higher byte of address – 3rd byte

C*CND addr16* (*Conditional call*)

1	1	***c***	***c***	***c***	0	1	0
l_7	l_6	l_5	l_4	l_3	l_2	l_1	l_0
h_7	h_6	h_5	h_4	h_3	h_2	h_1	h_0

if (*CND*) then [SP-1] ← PC_H, [SP-2] ← PC_L,
SP ← SP-2, *lower byte of address – 2nd byte*
PC ← [addr16] *higher byte of address – 3rd byte*

RET (*Return from procedure*)

1	1	0	0	1	0	0	1

PC_L, ← [SP], PC_H ← [SP+1], SP ← SP+2

R*CND* (*Conditional return from procedure*)

1	1	***c***	***c***	***c***	0	0	0

if (*CND*) then PC_L, ← [SP], PC_H ← [SP+1], SP ← SP+2

RST *N* (*Restart procedure / interrupt routine No. N*)

1	1	***n***	***n***	***n***	1	1	1

[SP-1] ← PC_H, [SP-2] ← PC_L, SP ← SP-2, PC ← N × 8

PCHL (*Move HL to PC*)

1	1	1	0	1	0	0	1

PC_H ← H, PC_L ← L

Stack manipulations

PUSH *P* (*Push register pair B, D or H on stack*)

1	1	***p***	***p***	0	1	0	1

$[SP-1] \leftarrow P_H$, $[SP-2] \leftarrow P_L$, $SP \leftarrow SP-2$

PUSH PSW (*Push Processor Status Word on stack*)

1	1	1	1	0	1	0	1

[SP-1] ← A, [SP-2] ← F, SP ← SP-2

POP *P* (*Pop register pair B, D or H from stack*)

1	1	***p***	***p***	0	0	0	1

$P_L \leftarrow [SP]$, $P_H \leftarrow [SP+1]$, $SP \leftarrow SP+2$

POP PSW (*Pop Processor Status Word from stack*)

1	1	1	1	0	0	0	1

F ← [SP], A ← [SP+1], SP ← SP+2
flags affected: Z, S, P, CY, AC

XTHL (*Exchange stack top with HL*)

1	1	1	0	0	0	1	1

L ↔ [SP], H ↔ [SP+1]

SPHL (*Move HL to SP*)

1	1	1	1	1	0	0	1

SP ← HL

Input / Output instructions

IN *adr8* (*Input from port*)

1	1	0	1	1	0	1	1
p_7	p_6	p_5	p_4	p_3	p_2	p_1	p_0

A ← Port[adr8] — *One byte of data from port stored in A*
Notice: port address is 8-bit long

OUT *adr8* (*Output to port*)

1	1	0	1	0	0	1	1
p_7	p_6	p_5	p_4	p_3	p_2	p_1	p_0

Port[adr8] ← A — *One byte of data from A stored in port*
Notice: port address is 8-bit long

Other instructions

EI (*Enable interrupt*)

1	1	1	1	1	0	1	1

INT input (hardware interrupt signal) is enabled

DI (*Disable interrupt*)

1	1	1	1	0	0	1	1

INT input (hardware interrupt signal) is blocked

HLT (*Halt*)

0	1	1	1	0	1	1	0

Processor is stopped

NOP (*No operation*)

0	0	0	0	0	0	0	0

Processor doesn't perform any operation