EPROMINT CPU

Quick User Manual

Majsterkowanie i nie tylko (MINT)

Table Of Contents

1. Overview

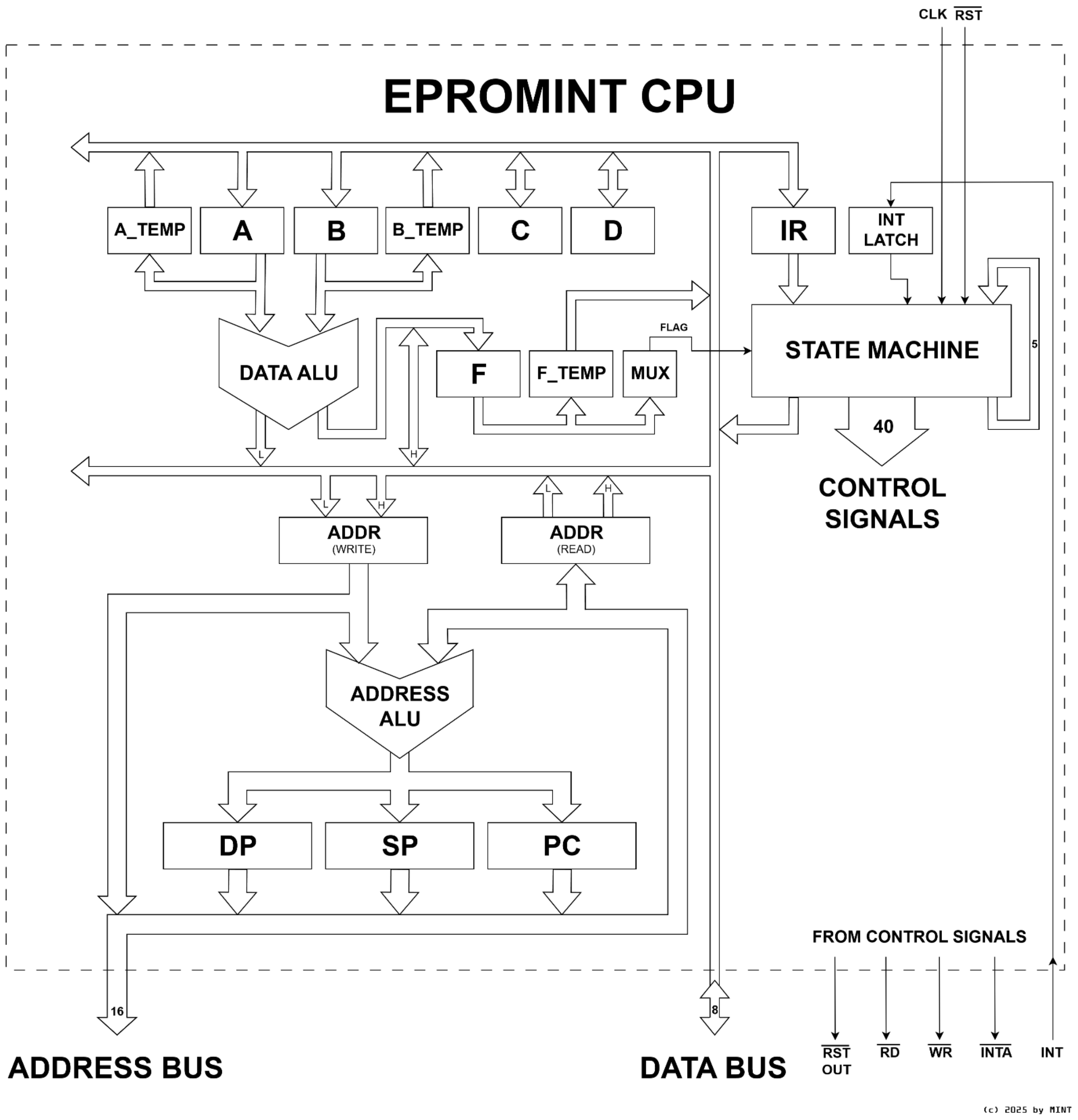
EPROMINT is a completely custom 8-bit CISC CPU based on Von Neumann architecture. The CPU is little-endian and has 16-bit address bus (64K memory space). It has four 8-bit working registers named **A** (**A**ccumulator), **B**, **C** and **D**. There is an additional **F** (**F**lags) register providing eight flags out of which all can be used to determine conditional branching. **EPROMINT** has three 16-bit address registers named **PC** (**P**rogram **C**ounter), **SP** (**S**tack **P**ointer) and **DP** (**D**ata **P**ointer). When mentioned names are present in the instruction mnemonic (case-sensitive), they specify a register and they are not used for anything else within this document. 8-bit registers can be used as 16-bit pairs, **AB** and **CD**. In such case, **A** and **C** are the low bytes of the resulting registers. When only a part of 16-bit register is used in the operation, either low or high byte, it is denoted as **reg_L** and **reg_H**, respectively. When it comes to math operators, C programming language conventions are used.

Presented CPU provides significant advantages over 8051, Z80, 6502 and other CPUs from 8-bit era. Main advantages are:

1. Very fast branching. Conditional relative jump takes 3 clock cycles, absolute jump takes 4 clocks. Indirect jump with displacement takes only 3 clocks, branch instructions are one of the fastest available in the **EPROMINT** instruction set. Instructions in general hardly ever exceed 10 clock cycles, and there are no instructions with variable execution time depending on certain conditions.
2. Less register-tied operations. Arithmetic and logical operations results can be directly stored in registers other than **A**, including memory locations.
3. Powerful ALU. Puny ***ADD*** and ***SUB*** is no longer the limit, **EPROMINT** supports multiplication and division with the speed of addition. Special instruction allows the user to access even more sophisticated operations like bitstream formation / slicing, square / cube roots, exponents, logarithms and trigonometric functions, both forward and inverse.
4. More memory-oriented addressing modes. Value from one memory location can be transferred into another without touching any register, the same applies to loading an immediate value into specified location. Any memory location can be pushed onto / popped from the stack with a single instruction, allowing to use memory as a large register bank. Read-modify-write operations included.
5. Dedicated I/O instructions with post increment / decrement and read-modify-write.
6. Built-in initialization routine after power-on that loads all registers with default values. Can be also executed by issuing a ***RESET*** instruction.

There are, however, some disadvantages: **EPROMINT** weights 0.5 kg, requires dozens of logic chips and weeks of soldering. **But it works and even has a C compiler development in progress!**

2. CPU Block Diagram



3. Instruction Set Summary

Opcode	Instruction	Operands	Flags	Clocks	Performed operation description
0x00	NOP	None	No	2	This instruction does literally nothing.
0x01	LD A, B	None	No	4	**A** gets loaded with **B** (short form is **A ← B**).
0x02	LD A, C	None	No	4	A ← C
0x03	LD A, D	None	No	4	A ← D
0x04	LD B, A	None	No	4	B ← A
0x05	LD B, C	None	No	4	B ← C
0x06	LD B, D	None	No	4	B ← D
0x07	LD C, A	None	No	4	C ← A
0x08	LD C, B	None	No	4	C ← B
0x09	LD C, D	None	No	4	C ← D
0x0A	LD D, A	None	No	4	D ← A
0x0B	LD D, B	None	No	4	D ← B
0x0C	LD D, C	None	No	4	D ← C
0x0D	LD A, n	n	No	3	A ← n
0x0E	LD B, n	n	No	3	B ← n
0x0F	LD C, n	n	No	3	C ← n
0x10	LD D, n	n	No	3	D ← n
0x11	LD A, (DP)	None	No	4	**A** gets loaded with value from memory location **DP** (**A ← mem(DP)**).
0x12	LD B, (DP)	None	No	4	B ← mem(DP)
0x13	LD C, (DP)	None	No	4	C ← mem(DP)
0x14	LD D, (DP)	None	No	4	D ← mem(DP)
0x15	LD A, (DP + d)	d	No	7	A ← mem(DP + d)
0x16	LD B, (DP + d)	d	No	7	B ← mem(DP + d)
0x17	LD A, (DP + D + d)	d	No	8	A ← mem(DP + D + d)
0x18	LD A, (SP + D + d)	d	No	8	A ← mem(SP + D + d)
0x19	LD B, (SP + d)	d	No	7	B ← mem(SP + d)
0x1A	LD C, (SP + d)	d	No	7	C ← mem(SP + d)
0x1B	LD D, (SP + d)	d	No	7	D ← mem(SP + d)
0x1C	LD A, (nn)	nn	No	5	A ← mem(nn)
0x1D	LD B, (nn)	nn	No	5	B ← mem(nn)
0x1E	LD C, (nn)	nn	No	5	C ← mem(nn)
0x1F	LD D, (nn)	nn	No	5	D ← mem(nn)
0x20	LD (DP), A	None	No	4	mem(DP) ← A
0x21	LD (DP), B	None	No	4	mem(DP) ← B
0x22	LD (DP), C	None	No	4	mem(DP) ← C
0x23	LD (DP), D	None	No	4	mem(DP) ← D
0x24	LD (DP + d), A	d	No	8	mem(DP + d) ← A
0x25	LD (DP + d), B	d	No	8	mem(DP + d) ← B
0x26	LD (DP + D + d), A	d	No	9	mem(DP + D + d) ← A
0x27	LD (SP + D + d), A	d	No	9	mem(SP + D + d) ← A
0x28	LD (SP + d), B	d	No	8	mem(SP + d) ← B
0x29	LD (SP + d), C	d	No	7	mem(SP + d) ← C

Opcode	Instruction	Operands	Flags	Clocks	Performed operation description
0x2A	LD (SP + d), D	d	No	7	mem(SP + d) ← D
0x2B	LD (nn), A	nn	No	6	mem(nn) ← A
0x2C	LD (nn), B	nn	No	6	mem(nn) ← B
0x2D	LD (nn), C	nn	No	5	mem(nn) ← C
0x2E	LD (nn), D	nn	No	5	mem(nn) ← D
0x2F	LD (nn), n	n, nn	No	9	mem(nn) ← n
0x30	LD (dd), (ss)	ss, dd	No	10	mem(dd) ← mem(ss)
0x31	LD A, (CD)	None	No	6	A ← mem(CD)
0x32	LD (CD), A	None	No	6	mem(CD) ← A
0x33	LD CD, (DP)	None	No	5	C ← mem(DP), D ← mem(DP + 1)
0x34	LD (DP), CD	None	No	6	mem(DP) ← C, mem(DP + 1) ← D
0x35	LD DP, AB	None	No	6	DP ← AB
0x36	LD DP, nn	nn	No	4	DP ← nn
0x37	LD SP, nn	nn	No	4	SP ← nn
0x38	LD DP, SP + d	d	No	3	DP ← SP + d
0x39	EX AB, CD	None	No	8	Exchanges **AB** with **CD** (**AB ↔ CD**)
0x3A	EX CD, DP	None	No	7	CD ↔ DP
0x3B	EX DP, (nn)	nn	No	11	DP_L ↔ mem(nn), DP_H ↔ mem(nn + 1)
0x3C	PUSH A	None	No	4	SP ← SP - 1; mem(SP) ← A (**SP** is decremented, then **A** is pushed)
0x3D	PUSH B	None	No	4	SP ← SP - 1; mem(SP) ← B
0x3E	PUSH F	None	No	4	SP ← SP - 1; mem(SP) ← F
0x3F	PUSH CD	None	No	6	SP ← SP - 1; mem(SP) ← D; SP ← SP - 1; mem(SP) ← C
0x40	PUSH DP	None	No	7	SP ← SP - 1; mem(SP) ← DP_H; SP ← SP - 1; mem(SP) ← DP_L
0x41	PUSH ALL	None	No	17	Equivalent to ***PUSH*** **DP**, **CD**, **F**, **B**, **A**, but faster
0x42	PUSH n	n	No	6	SP ← SP - 1; mem(SP) ← n
0x43	PUSH (nn)	nn	No	8	SP ← SP - 1; mem(SP) ← mem(nn)
0x44	POP A	None	No	4	A ← mem(SP); SP ← SP + 1 (**A** is restored, then **SP** is incremented)
0x45	POP B	None	No	4	B ← mem(SP); SP ← SP + 1
0x46	POP F	None	No	4	F ← mem(SP); SP ← SP + 1
0x47	POP CD	None	No	5	C ← mem(SP); SP ← SP + 1; D ← mem(SP); SP ← SP + 1
0x48	POP DP	None	No	5	DP_L ← mem(SP); SP ← SP + 1; DP_H ← mem(SP); SP ← SP + 1
0x49	POP ALL	None	No	10	Equivalent to ***POP*** **A, B, F, CD, DP**, but faster
0x4A	POP (nn)	nn	No	8	mem(nn) ← mem(SP); SP ← SP + 1
0x4B	ADD A, B	None	Yes	4	A ← A + B (**arg1** is **A**, **arg2** is **B** (for correct interpretation of flags))
0x4C	ADD A, C	None	Yes	7	A ← A + C (**arg1** is **A**, **arg2** is **C**)
0x4D	ADD A, D	None	Yes	7	A ← A + D (**arg1** is **A**, **arg2** is **D**)
0x4E	ADD A, n	n	Yes	7	A ← A + n (**arg1** is **A**, **arg2** is **n**)
0x4F	ADD B, n	n	Yes	9	B ← B + n (**arg1** is **B**, **arg2** is **n**)
0x50	ADD C, n	n	Yes	10	C ← C + n (**arg1** is **C**, **arg2** is **n**)
0x51	ADD D, n	n	Yes	10	D ← D + n (**arg1** is **D**, **arg2** is **n**)
0x52	ADDC A, A	None	Yes	10	A ← A + A + carry (**arg1** is **A**, **arg2** is **A + carry**)
0x53	ADDC A, B	None	Yes	7	A ← A + B + carry (**arg1** is **A**, **arg2** is **B + carry**)
0x54	ADDC A, C	None	Yes	9	A ← A + C + carry (**arg1** is **A**, **arg2** is **C + carry**)
0x55	ADDC A, D	None	Yes	9	A ← A + D + carry (**arg1** is **A**, **arg2** is **D + carry**)

Opcode	Instruction	Operands	Flags	Clocks	Performed operation description
0x56	ADDC A, n	n	Yes	9	A ← A + n + carry (**arg1** is **A**, **arg2** is **n + carry**)
0x57	ADDC B, n	n	Yes	11	B ← B + n + carry (**arg1** is **B**, **arg2** is **n + carry**)
0x58	ADDC C, n	n	Yes	10	C ← C + n + carry (**arg1** is **C**, **arg2** is **n + carry**)
0x59	ADDC D, n	n	Yes	10	D ← D + n + carry (**arg1** is **D**, **arg2** is **n + carry**)
0x5A	SUB A, B	None	Yes	5	A ← A - B (**arg1** is **A**, **arg2** is **B**)
0x5B	SUB A, C	None	Yes	7	A ← A - C (**arg1** is **A**, **arg2** is **C**)
0x5C	SUB A, D	None	Yes	7	A ← A - D (**arg1** is **A**, **arg2** is **D**)
0x5D	SUB A, n	n	Yes	7	A ← A - n (**arg1** is **A**, **arg2** is **n**)
0x5E	SUB B, n	n	Yes	9	B ← B - n (**arg1** is **B**, **arg2** is **n**)
0x5F	SUB C, n	n	Yes	10	C ← C - n (**arg1** is **C**, **arg2** is **n**)
0x60	SUB D, n	n	Yes	10	D ← D - n (**arg1** is **D**, **arg2** is **n**)
0x61	SUBB A, B	None	Yes	7	A ← A - B - carry (**arg1** is **A**, **arg2** is **B + carry**)
0x62	SUBB A, C	None	Yes	9	A ← A - C - carry (**arg1** is **A**, **arg2** is **C + carry**)
0x63	SUBB A, D	None	Yes	10	A ← A - D - carry (**arg1** is **A**, **arg2** is **D + carry**)
0x64	SUBB A, n	n	Yes	9	A ← A - n - carry (**arg1** is **A**, **arg2** is **n + carry**)
0x65	SUBB B, n	n	Yes	11	B ← B - n - carry (**arg1** is **B**, **arg2** is **n + carry**)
0x66	SUBB C, n	n	Yes	10	C ← C - n - carry (**arg1** is **C**, **arg2** is **n + carry**)
0x67	SUBB D, n	n	Yes	10	D ← D - n - carry (**arg1** is **D**, **arg2** is **n + carry**)
0x68	ADD A, (DP)	None	Yes	7	A ← A + mem(DP) (**arg1** is **A**, **arg2** is **mem(DP)**)
0x69	ADDC A, (DP)	None	Yes	9	A ← A + mem(DP) + carry (**arg1** is **A**, **arg2** is **mem(DP) + carry**)
0x6A	SUB A, (DP)	None	Yes	7	A ← A - mem(DP) (**arg1** is **A**, **arg2** is **mem(DP)**)
0x6B	SUBB A, (DP)	None	Yes	9	A ← A - mem(DP) - carry (**arg1** is **A**, **arg2** is **mem(DP) + carry**)
0x6C	ADD (DP), A	None	Yes	10	mem(DP) ← mem(DP) + A (**arg1** is **mem(DP)**, **arg2** is **A**)
0x6D	ADDC (DP), A	None	Yes	10	mem(DP) ← mem(DP) + A + carry (**arg1** is **mem(DP)**, **arg2** is **A + carry**)
0x6E	SUB (DP), A	None	Yes	10	mem(DP) ← mem(DP) - A (**arg1** is **mem(DP)**, **arg2** is **A**)
0x6F	SUBB (DP), A	None	Yes	10	mem(DP) ← mem(DP) - A - carry (**arg1** is **mem(DP)**, **arg2** is **A + carry**)
0x70	INC A	None	Yes	7	A ← A + 1 (**arg1** is **A**, **arg2** is **1**)
0x71	DEC A	None	Yes	7	A ← A - 1 (**arg1** is **A**, **arg2** is **1**)
0x72	INC (DP)	None	Yes	10	mem(DP) ← mem(DP) + 1 (**arg1** is **mem(DP)**, **arg2** is **1**)
0x73	DEC (DP)	None	Yes	10	mem(DP) ← mem(DP) - 1 (**arg1** is **mem(DP)**, **arg2** is **1**)
0x74	INC (nn)	nn	Yes	12	mem(nn) ← mem(nn) + 1 (**arg1** is **mem(nn)**, **arg2** is **1**)
0x75	DEC (nn)	nn	Yes	12	mem(nn) ← mem(nn) - 1 (**arg1** is **mem(nn)**, **arg2** is **1**)
0x76	CLR A	None	No	4	A ← 0
0x77	CPL A	None	Yes	7	A ← ~A (**arg1** is **A**, **arg2** is **0xFF**)
0x78	AND A, B	None	Yes	5	A ← A & B (**arg1** is **A**, **arg2** is **B**)
0x79	AND A, C	None	Yes	7	A ← A & C (**arg1** is **A**, **arg2** is **C**)
0x7A	AND A, D	None	Yes	7	A ← A & D (**arg1** is **A**, **arg2** is **D**)
0x7B	AND A, n	n	Yes	7	A ← A & n (**arg1** is **A**, **arg2** is **n**)
0x7C	AND B, n	n	Yes	9	B ← B & n (**arg1** is **B**, **arg2** is **n**)
0x7D	AND C, n	n	Yes	10	C ← C & n (**arg1** is **C**, **arg2** is **n**)
0x7E	AND D, n	n	Yes	10	D ← D & n (**arg1** is **D**, **arg2** is **n**)
0x7F	OR A, B	None	Yes	5	A ← A | B (**arg1** is **A**, **arg2** is **B**)
0x80	OR A, C	None	Yes	7	A ← A | C (**arg1** is **A**, **arg2** is **C**)
0x81	OR A, D	None	Yes	7	A ← A | D (**arg1** is **A**, **arg2** is **D**)

Opcode	Instruction	Operands	Flags	Clocks	Performed operation description
0x82	OR A, n	n	Yes	7	A ← A | n (**arg1** is **A**, **arg2** is **n**)
0x83	OR B, n	n	Yes	9	B ← B | n (**arg1** is **B**, **arg2** is **n**)
0x84	OR C, n	n	Yes	10	C ← C | n (**arg1** is **C**, **arg2** is **n**)
0x85	OR D, n	n	Yes	10	D ← D | n (**arg1** is **D**, **arg2** is **n**)
0x86	XOR A, B	None	Yes	5	A ← A ^ B (**arg1** is **A**, **arg2** is **B**)
0x87	XOR A, C	None	Yes	7	A ← A ^ C (**arg1** is **A**, **arg2** is **C**)
0x88	XOR A, D	None	Yes	7	A ← A ^ D (**arg1** is **A**, **arg2** is **D**)
0x89	XOR A, n	n	Yes	7	A ← A ^ n (**arg1** is **A**, **arg2** is **n**)
0x8A	XOR B, n	n	Yes	9	B ← B ^ n (**arg1** is **B**, **arg2** is **n**)
0x8B	XOR C, n	n	Yes	10	C ← C ^ n (**arg1** is **C**, **arg2** is **n**)
0x8C	XOR D, n	n	Yes	10	D ← D ^ n (**arg1** is **D**, **arg2** is **n**)
0x8D	CP A, C	None	Yes	6	**arg1** is **A**, **arg2** is **C**, operation is ***SUB***. Flags are updated.
0x8E	CP A, D	None	Yes	6	**arg1** is **A**, **arg2** is **D**, operation is ***SUB***. Flags are updated.
0x8F	MUL AB	None	No	7	A ← (A * B)_low, B ← (A * B)_high
0x90	MUL A, n	n	No	7	A ← (A * n)_low, B ← (A * n)_high
0x91	MUL (DP), AB	None	No	5	mem(DP) ← (A * B)_low, mem(DP + 1) ← (A * B)_high
0x92	MUL CD, AB	None	No	5	C ← (A * B)_low, D ← (A * B)_high
0x93	MUL16 AB, C	None	No	21	A ← (AB * C)_low, B ← (AB * C)_mid, C ← (AB * C)_high
0x94	DIV A, B	None	No	7	A ← (A / B), B ← (A % B)
0x95	DIV A, n	n	No	7	A ← (A / n), B ← (A % n)
0x96	DIV (DP), AB	None	No	5	mem(DP) ← (A / B), mem(DP + 1) ← (A % B)
0x97	DIV CD, AB	None	No	5	C ← (A / B), D ← (A % B)
0x98	DIVNR A, B	None	No	4	A ← (A / B)
0x99	EXT A	None	No	7	A ← EXT(A, B)_low, B ← EXT(A, B)_high
0x9A	EXTDIR A, oper	oper	No	7	A ← EXT(A, oper)_low, B ← EXT(A, oper)_high
0x9B	EXT (DP), A	None	No	5	mem(DP) ← EXT(A, B)_low, mem(DP + 1) ← EXT(A, B)_high
0x9C	EXT CD, A	None	No	5	C ← EXT(A, B)_low, D ← EXT(A, B)_high
0x9D	EXTDIRL A, oper	oper	No	6	A ← EXT(A, oper)_low
0x9E	INC DP	None	No	3	DP ← DP + 1
0x9F	DEC DP	None	No	3	DP ← DP - 1
0xA0	ADD DP, A	None	No	4	DP ← DP + A
0xA1	ADD DP, d	d	No	3	DP ← DP + d
0xA2	ADD CD, AB	None	Yes[1]	13	CD ← CD + AB (**arg1** is **D**, **arg2** is **B + carry** [**carry** is taken from **C + A**])
0xA3	SUB CD, AB	None	Yes[1]	13	CD ← CD - AB (**arg1** is **D**, **arg2** is **B + carry** [**carry** is **borrow** from **C - A**])
0xA4	ADD SP, A	None	No	4	SP ← SP + A
0xA5	ADD SP, d	d	No	3	SP ← SP + d
0xA6	CP A, B	None	Yes	4	**arg1** is **A**, **arg2** is **B**, operation is ***SUB***. Flags are updated.
0xA7	CP A, n	n	Yes	6	**arg1** is **A**, **arg2** is **n**, operation is ***SUB***. Flags are updated.
0xA8	CP B, n	n	Yes	9	**arg1** is **B**, **arg2** is **n**, operation is ***SUB***. Flags are updated.
0xA9	CP C, n	n	Yes	9	**arg1** is **C**, **arg2** is **n**, operation is ***SUB***. Flags are updated.
0xAA	CP D, n	n	Yes	9	**arg1** is **D**, **arg2** is **n**, operation is ***SUB***. Flags are updated.
0xAB	CP A, (DP)	None	Yes	6	**arg1** is **A**, **arg2** is **mem(DP)**, operation is ***SUB***. Flags are updated.

[1] Only **carry**, **overflow** and **negative** flags are valid for 16-bit result. The rest refers only to high byte of the result / high bytes of input operands.

Opcode	Instruction	Operands	Flags	Clocks	Performed operation description
0xAC	CP (DP), n	n	Yes	9	**arg1** is **mem(DP)**, **arg2** is **n**, operation is ***SUB***. Flags are updated.
0xAD	TEST A, n	n	Yes	6	**arg1** is **A**, **arg2** is **n,** operation is ***AND.*** Flags are updated.
0xAE	LD C, (DP + d)	d	No	7	C ← mem(DP + d)
0xAF	LD D, (DP + d)	d	No	7	D ← mem(DP + d)
0xB0	LD (DP + d), C	d	No	7	mem(DP + d) ← C
0xB1	LD (DP + d), D	d	No	7	mem(DP + d) ← D
0xB2	LD A, (SP + d)	d	No	7	A ← mem(SP + d)
0xB3	LD (SP + d), A	d	No	8	mem(SP + d) ← A
0xB4	LD CD, SP	None	No	5	CD ← SP
0xB5	LD DP, CD + d	d	No	7	DP ← CD; DP ← DP + d
0xB6	ADD CD, nn	nn	Yes[2]	14	CD ← CD + nn (**arg1** is **D**, **arg2** is **n_H + carry** [**carry** is taken from **C + n_L**])
0xB7	SUB CD, nn	nn	Yes[2]	14	CD ← CD - nn (**arg1** is **D**, **arg2** is **n_H + carry** [**carry** is **borrow** from **C - n_L**])
0xB8	ADD (DP), n	n	Yes	10	mem(DP) ← mem(DP) + n (**arg1** is **mem(DP)**, **arg2** is **n**)
0xB9	ADDC (DP), n	n	Yes	10	mem(DP) ← mem(DP) + n + carry (**arg1** is **mem(DP)**, **arg2** is **n + carry**)
0xBA	SUB (DP), n	n	Yes	10	mem(DP) ← mem(DP) - n (**arg1** is **mem(DP)**, **arg2** is **n**)
0xBB	SUBB (DP), n	n	Yes	10	mem(DP) ← mem(DP) - n - carry (**arg1** is **mem(DP)**, **arg2** is **n + carry**)
0xBC	AND A, (DP)	None	Yes	7	A ← A & mem(DP) (**arg1** is **A**, **arg2** is **mem(DP)**)
0xBD	AND (DP), A	None	Yes	10	mem(DP) ← mem(DP) & A (**arg1** is **mem(DP)**, **arg2** is **A**)
0xBE	AND (DP), n	n	Yes	10	mem(DP) ← mem(DP) & n (**arg1** is **mem(DP)**, **arg2** is **n**)
0xBF	OR A, (DP)	None	Yes	7	A ← A | mem(DP) (**arg1** is **A**, **arg2** is **mem(DP)**)
0xC0	OR (DP), A	None	Yes	10	mem(DP) ← mem(DP) | A (**arg1** is **mem(DP)**, **arg2** is **A**)
0xC1	OR (DP), n	n	Yes	10	mem(DP) ← mem(DP) | n (**arg1** is **mem(DP)**, **arg2** is **n**)
0xC2	XOR A, (DP)	None	Yes	7	A ← A ^ mem(DP) (**arg1** is **A**, **arg2** is **mem(DP)**)
0xC3	XOR (DP), A	None	Yes	10	mem(DP) ← mem(DP) ^ A (**arg1** is **mem(DP)**, **arg2** is **A**)
0xC4	XOR (DP), n	n	Yes	10	mem(DP) ← mem(DP) ^ n (**arg1** is **mem(DP)**, **arg2** is **n**)
0xC5	JP nn	nn	No	4	PC ← nn
0xC6	JP C, nn	nn	No	4	If **carry** flag **set**, then PC ← nn
0xC7	JP OV, nn	nn	No	4	If **overflow** flag **set**, then PC ← nn
0xC8	JP Z, nn	nn	No	4	If **zero** flag **set**, then PC ← nn
0xC9	JP NEG, nn	nn	No	4	If **negative** flag **set**, then PC ← nn
0xCA	JP POS, nn	nn	No	4	If **positive** flag **set**, then PC ← nn
0xCB	JP EQ, nn	nn	No	4	If **arg1 == arg2** flag **set**, then PC ← nn
0xCC	JP LT, nn	nn	No	4	If **arg1 < arg2** flag **set**, then PC ← nn
0xCD	JP GT, nn	nn	No	4	If **arg1 > arg2** flag **set**, then PC ← nn
0xCE	JP DP + d	d	No	3	PC ← DP + d
0xCF	JR d	d	No	3	PC ← PC + d
0xD0	JR C, d	d	No	3	If **carry** flag **set**, then PC ← PC + d
0xD1	DI	None	No	4	Disables interrupts.
0xD2	JR OV, d	d	No	3	If **overflow** flag **set**, then PC ← PC + d
0xD3	JR Z, d	d	No	3	If **zero** flag **set**, then PC ← PC + d
0xD4	JR NEG, d	d	No	3	If **negative** flag **set**, then PC ← PC + d
0xD5	JR POS, d	d	No	3	If **positive** flag **set**, then PC ← PC + d

[2] Only **carry**, **overflow** and **negative** flags are valid for 16-bit result. The rest refers only to high byte of the result / high bytes of input operands.

Opcode	Instruction	Operands	Flags	Clocks	Performed operation description
0xD6	JR EQ, d	d	No	3	If **arg1 == arg2** flag **set**, then PC ← PC + d
0xD7	JR LT, d	d	No	3	If **arg1 < arg2** flag **set**, then PC ← PC + d
0xD8	JR GT, d	d	No	3	If **arg1 > arg2** flag **set**, then PC ← PC + d
0xD9	JR NC, d	d	No	3	If **carry** flag **not set**, then PC ← PC + d
0xDA	JR NOV, d	d	No	3	If **overflow** flag **not set**, then PC ← PC + d
0xDB	JR NZ, d	d	No	3	If **zero** flag **not set**, then PC ← PC + d
0xDC	JR NNEG, d	d	No	3	If **negative** flag **not set**, then PC ← PC + d
0xDD	JR NPOS, d	d	No	3	If **positive** flag **not set**, then PC ← PC + d
0xDE	JR NEQ, d	d	No	3	If **arg1 == arg2** flag **not set**, then PC ← PC + d
0xDF	JR NLT, d	d	No	3	If **arg1 < arg2** flag **not set**, then PC ← PC + d (jumps if arg1 >= arg2)
0xE0	JR NGT, d	d	No	3	If **arg1 > arg2** flag **not set**, then PC ← PC + d (jumps if arg1 <= arg2)
0xE1	EI	None	No	4	Enables interrupts.
0xE2	DJNZ d	d	No	10	B ← B - 1; If B > 0, then PC ← PC + d
0xE3	CALL nn	nn	No	9	SP ← SP - 1; mem(SP) ← PC_H; SP ← SP - 1; mem(SP) ← PC_L; PC ← nn
0xE4	CALLR d	d	No	8	SP←SP-1; mem(SP)←PC_H; SP←SP-1; mem(SP)←PC_L; PC ← PC + d
0xE5	CALL DP + d	d	No	9	SP ← SP-1; mem(SP) ← PC_H; SP ← SP-1; mem(SP) ← PC_L; PC ← DP + d
0xE6	RET	None	No	5	PC_L ← mem(SP); SP ← SP + 1; PC_H ← mem(SP); SP ← SP + 1
0xE7	RETI	None	No	5	PC_L←mem(SP); SP←SP+1; PC_H←mem(SP); SP←SP+1; Enables interrupts.
0xE8	OUT (s), A	s	No	8	mem(0x8000 + s) ← A
0xE9	OUT (s), B	s	No	8	mem(0x8000 + s) ← B
0xEA	OUT (s), C	s	No	8	mem(0x8000 + s) ← C
0xEB	OUT (s), D	s	No	8	mem(0x8000 + s) ← D
0xEC	OUT (s), n	n, s	No	11	mem(0x8000 + s) ← n (note that n goes first after 0xEC opcode)
0xED	OUT (s), (DP)	s	No	12	mem(0x8000 + s) ← mem(DP)
0xEE	OUTI (s), (DP)	s	No	12	mem(0x8000 + s) ← mem(DP); DP ← DP + 1
0xEF	OUTD (s), (DP)	s	No	12	mem(0x8000 + s) ← mem(DP); DP ← DP - 1
0xF0	IN A, (s)	s	No	7	A ← mem(0x8000 + s)
0xF1	IN B, (s)	s	No	7	B ← mem(0x8000 + s)
0xF2	IN C, (s)	s	No	7	C ← mem(0x8000 + s)
0xF3	IN D, (s)	s	No	7	D ← mem(0x8000 + s)
0xF4	IN (DP), (s)	s	No	9	mem(DP) ← mem(0x8000 + s)
0xF5	INI (DP), (s)	s	No	10	mem(DP) ← mem(0x8000 + s); DP ← DP + 1
0xF6	IND (DP), (s)	s	No	10	mem(DP) ← mem(0x8000 + s); DP ← DP - 1
0xF7	INCIO (s)	s	Yes	13	mem(0x8000 + s) ← mem(0x8000 + s) + 1 (**arg1** is **IO_data**, **arg2** is **1**)
0xF8	DECIO (s)	s	Yes	13	mem(0x8000 + s) ← mem(0x8000 + s) - 1 (**arg1** is **IO_data**, **arg2** is **1**)
0xF9	ANDIO (s), A	s	Yes	11	mem(0x8000 + s) ← mem(0x8000 + s) & A (**arg1** is **A**, **arg2** is **IO_data**)
0xFA	ANDIO (s), n	s, n	Yes	14	mem(0x8000 + s) ← mem(0x8000 + s) & n (**arg1** is **IO_data**, **arg2** is **n**)
0xFB	ORIO (s), A	s	Yes	11	mem(0x8000 + s) ← mem(0x8000 + s) | A (**arg1** is **A**, **arg2** is **IO_data**)
0xFC	ORIO (s), n	s, n	Yes	14	mem(0x8000 + s) ← mem(0x8000 + s) | n (**arg1** is **IO_data**, **arg2** is **n**)
0xFD	XORIO (s), A	s	Yes	11	mem(0x8000 + s) ← mem(0x8000 + s) ^ A (**arg1** is **A**, **arg2** is **IO_data**)
0xFE	XORIO (s), n	s, n	Yes	14	mem(0x8000 + s) ← mem(0x8000 + s) ^ n (**arg1** is **IO_data**, **arg2** is **n**)
0xFF	RESET	None	Yes	19	Performs the reset sequence.

In the above notation, **mem(address)** means memory location specified by given address. **EXT(arg, oper)** means the result of ***EXT***ended operation when argument is **arg** and operation selector is **oper**. **IO_data** is the short form for **mem(0x8000 + s)**. All ***MUL*** and ***DIV*** operations are unsigned.

4. Instruction Operands

n	unsigned byte (uint8_t)
nn	unsigned int, little-endian (uint16_t)
d	signed byte (int8_t)
dd	destination address, same as nn
ss	source address, same as nn
oper	operation select, same as n
s	I/O device selector[3]

Note that operands in "Operands" column are listed in order in which they should be placed in memory, after an opcode which is always first.

5. Effect on Flags

No	Flags are not affected.
Yes	All flags are updated.

There are 8 flags:
C (carry, result is greater than 255), LSB in **F** register
OV (overflow, result is outside range [-128, 127]), valid only for signed
Z (result is zero)
NEG (result is negative), valid only for signed
POS (result is positive), valid only for signed
EQ (**arg1** equals **arg2**)
LT (**arg1** is less than **arg2**), valid only for unsigned
GT (**arg1** is greater than **arg2**), valid only for unsigned, MSB in **F** register

6. Reset Sequence

Reset is executed:

- After power-on.
- When reset button is pushed.
- After issuing *RESET* instruction.

Reset instruction loads registers **A**, **B**, **C**, **D** with 0, **F** with 0x24 (**Z** and **EQ** set), **SP** with 0xFFFF, **DP** (**D**ata **P**ointer) with 0x8080 and **PC** with 0x0000. Program execution starts from address 0x0000. Interrupts are disabled.

7. Interrupt Response

INT input is sampled at the beginning of every instruction execution. If sampled value is "1", the following sequence is executed:

1. First INTA pulse is issued, interrupt controller must NOT drive the data bus yet.
2. Current **PC** (**P**rogram **C**ounter) value is pushed onto the stack.
3. Second INTA pulse is issued, when INTA line is low, interrupt controller is expected to put the interrupt vector pointer onto data bus.
4. Captured pointer is multiplied by 2, data from memory location pointed to is loaded into **PC**.
5. Interrupts are disabled, interrupt flag is cleared and the program execution starts from new location loaded into **PC**.

Example: interrupt vector pointer is 0x08, memory location 0x10 (0x08 * 2) contains 0x02, memory location 0x11 (0x08 * 2 + 1) contains 0x40. After interrupt response sequence, program execution will resume from location 0x4002. The whole sequence takes 14 clock cycles.

Note: interrupt pointer (one byte) must be in range [0 – 127].

8. *RETI* Instruction

Pops **PC** from the stack to resume program execution from location where most recent interrupt was recognized. Enables the interrupts and starts program execution without sampling the INT input, so that at least one main program instruction can be executed before next interrupt is recognized. Same applies to returning to interrupt handler after a nested interrupt. It is worth noting that the only nesting limit is the stack size.

[3] I/O device selector **s** is used as a low byte of selected address location, while high byte is fixed at 0x80. There is no separate I/O address space, the point of I/O instructions is to provide "I/O friendly" timing. Standard memory access instructions are optimized for RAM & ROM operations, using them to access I/O is not recommended.

9. Relative Address Register Changes

There are various instructions that do not load absolute value into 16-bit registers (**PC**, **SP** and **DP**). For example ***JR*** (relative jump), ***ADD SP, d*** (add d to **S**tack **P**ointer), etc. For such instructions, upper 2 bits of modified register are fixed, so after executing ***JR -10*** from address 0x4000, **PC** will not be 0x3FF6, but 0x7FF6. For ***ADD SP, 20*** when **SP** is 0x7FF0, it will not become 0x8004, but 0x4004. Be careful!

10. Indexed Addressing

Operand ***(address_register + data_register)***, for example ***(SP + D)***, means "data at memory location pointed to by ***address_register*** after adding ***data_register*** to it". Value from ***data_register*** is always treated as signed and has range from -128 to 127. Subtracting is achieved by adding a negative value, there is no ***(SP - D)*** operation. For operand ***(SP + D + d)*** the range is [-256 – 254], as both D and d have [-128 – 127] range. Two upper bits of ***address_register*** are fixed, so for example 0x3FFA + 0x10 will roll back to 0x000A. 16-kilobyte page boundaries can be crossed only by loading a new value into ***address_register***.

11. Relative Jump and Call

d operand of ***JR*** (**J**ump **R**elative) and ***CALLR*** (**Call** **R**elative) is referenced to **PC** after fetching the instruction opcode and **d** value. So to jump back to ***JR*** opcode, **d** has to be -2 (which is 0xFE). To jump one byte before the opcode, **d** has to be -3. To jump forward 2 bytes ahead of **d** operand, **d** has to be 1. To jump forward 1 byte ahead of **d** (continue normal program execution) **d** has to be 0.

12. *EXT* Instruction

EXTended opcode is used to access additional ALU operations. **A** register is the operand, **B** selects the operation. For ***EXTDIR*** and ***EXTDIRL*** instructions, **B** value is provided as an instruction operand, so there is no need to load **B** before instruction execution. **B** is restored after ***EXTDIRL***. From now on, operation selector will be called **OP**, and single bits or bit slices from **OP** will be denoted as **OP**[x] and **OP**[x:y], respectively. For example, **OP**[4] is bit 4 of **OP**, **OP**[7:5] is a slice constructed from bits 7 – 5, where bit 7 is MSB. Bit 7 of **OP** is the MSB of **OP**, bit 0 is the LSB. Operation argument (**A** register) will be called **arg**, low byte of the 16-bit result will be called **res_L**, high byte will be called **res_H**. Whole 16-bit result is **res**. Available operations are described below.

12.1 Bitstream Formatting

When **OP**[7] is set to 0 (**OP** is 0xxxxxxx), bit insertion mode is selected. In this mode **arg** specifies inserted data, **res_L** is the bit mask that has to be applied to the current byte to which data is inserted, **res_H** is the bit mask for the next byte, which allows to cross byte boundaries with less additional operations. **OP**[6:0] selects the mask type, length and location of the insertion:

OP[6] – bit mask type. 0 selects reset mask, 1 selects set mask.

OP[5:3] – insertion length. Specifies how many bits from **arg** affect created masks.

OP[2:0] – insertion location. Specifies bit position from which the insertion will start. **Very important** detail is that MSB of byte into which **arg** is inserted is **bit 0**, not bit 7. This is because subsequent slices are usually inserted from left to right when creating a bitstream, so starting from MSB of the destination byte. To sum it up: MSB of the destination byte is the leftmost bit, but it has index 0, not 7.

Examples:

1. **arg** is 00000101, **OP** is 01011001 (**OP**[6] = 1, **OP**[5:3] = 011, **OP**[2:0] = 001). It means that selected mask type is set mask, length is 3 and location is 1. 3 lowest bits of **arg** will be inserted starting from bit 1 in the destination byte (2nd leftmost bit). Created mask will be 0xxx0000, where x denotes inserted value, which is 101 in this case. So the final mask is 01010000. By ORing this mask with the

destination byte, correct value will be inserted if the byte was initialized with a zero. If it was not, reset mask also has to be applied, and it will be 1xxx1111, so 11011111 in this case. By ANDing it with destination byte, bits will be set to 0s where 0s are inserted. **OP**[6] has to be set to 0 to get the reset mask. Whole inserted value fit into first destination byte, so masks for the next byte will be 00000000 for set mask and 11111111 for reset mask.

2. **arg** is 00110001, **OP** is 01110100 (**OP**[6] = 1, **OP**[5:3] = 110, **OP**[2:0] = 100). 110001 has to be inserted starting from bit 4, selected mask type is set mask. Created mask will be 0000xxxx xx000000, note that inserted value no longer fits into one byte and overflows into 2nd mask. Final mask will be 00001100 01000000. After applying the 1st one, current destination byte can be released (for example written into memory), and then 2nd mask has to be applied to the new destination byte. Reset mask would be 1111xxxx xx111111, so 11111100 01111111. When destination bytes are not initialized with 0, both masks (set and reset) have to be applied to ensure correct value insertion.

12.2 Bitstream Slicing

When **OP**[7:6] is 10 (**OP** is 10xxxxxx), bit slicing mode is selected. In this mode **arg** is the byte from which bits are sliced, **res_L** is the resulting slice and **res_H** specifies how many bits are left and have to be sliced from the next byte of the bitstream. **OP**[5:0] selects the length and starting position of the slice:

OP[5:3] – length of the slice.

OP[2:0] – starting position of the slice. Again **very important** detail: MSB of the input byte (leftmost bit of **arg**) has index 0, not 7.

Examples:

1. **arg** is 10100111, **OP** is 10101001 (**OP**[5:3] = 101, **OP**[2:0] = 001). 5 bits have to be sliced starting from bit 1 (2nd leftmost bit of **arg**). “x” will be used to mark sliced bits in **arg**: 1xxxxx11. Marked slice is 01001, and this value will be present when reading **res_L**. **res_H** will be 0 since all sliced bits were present in the input byte.
2. **arg** is 11010011, **OP** is 10111101 (**OP**[5:3] = 111, **OP**[2:0] = 101). 7 bits have to be sliced starting from bit 5. “x” marks the slice: 11010xxx. Slice is 011, so only 3 bits long, 4 bits are remaining. **res_L** will be 0110000, 3-bit slice is adjusted to allow for easy insertion of the remaining bits. **res_H** will be 00000100 (4 in decimal). It indicates that 4 bits have to be sliced from the next byte of the bitstream, and that byte should be now fetched. Assume that it is 10010111, and **OP** is 10100000 to slice 4 bits starting from bit 0. **res_L** will be 1001, **res_H** will be 0, since no bits are remaining. By ORing the previous slice (0110000) with **res_L**, correct cross-byte 7-bit slice is obtained, which is 0111001.

12.3 Bit Shifting

When **OP**[7:5] is 110 (**OP** is 110xxxxx), bit shifting mode is selected. In this mode **arg** is the input value, **res_L** and **res_H** is the shifting result. **OP**[4:0] selects the type of shift and by how many bits input will be shifted:

OP[4:3] – shift type.

OP[2:0] – specifies by how many bits **arg** will be shifted.

Shift types are described below.

OP[4:3] = 00 (**OP** = 11000xxx): shift left, **res_L** is the result, **res_H** stores the bits that overflowed from **res_L**. **res_H** and **res_L** together hold a 16-bit result. **res_H** can be ORed with byte preceding the currently shifted byte for easier multi-byte shifting.

OP[4:3] = 01 (**OP** = 11001xxx): shift right, **res_L** is the result, **res_H** stores the bits that overflowed from **res_L**. Its value can be ORed with byte following the currently shifted byte for easier multi-byte shifting.

OP[4:3] = 10 (**OP** = 11010xxx): rotate left. Bits that overflowed from the left side are put back on the right. **res_L** is the result, **res_H** is 0.

OP[4:3] = 11 (**OP** = 11011xxx): rotate right. Bits that overflowed from the right side are put back on the left. **res_L** is the result, **res_H** is 0.

12.4 Math and Auxiliary Bitwise Operations

When **OP**[7:5] is 111 (**OP** is 111xxxxx), math mode is selected with some additional bitwise operations. **arg** is the input value, **OP**[4:0] selects the operation (whole **OP** value is given for clarity):

OP	Operation
0xE0	Bitwise negation: **res_L** = ~**arg**, **res_H** = 0.
0xE1	Nibble swap: bits[7:4] of **arg** are exchanged with bits [3:0], result is present in **res_L**.
0xE2	Mirror bits: bit 7 of **arg** becomes bit 0, bit 1 becomes bit 6, etc. Result in **res_L**.
0xE3	Count set bits: **res_L** is equal to the number of 1s in **arg**.
0xE4	Arithmetic negation: **res_L** = -**arg**, **arg** is treated as signed (2's complement).
0xE5	Arithmetic shift right: **res_L** = **arg** >> 1, MSB is preserved, for example 0xF0 >> 1 = 0xF8. MSB of **res_H** contains the bit that overflowed from the right side.
0xE6	Absolute value: **res_L** = abs(**arg**), **arg** is treated as signed.
0xE7	**arg** conversion from 2's complement to sign-magnitude and vice-versa, result in **res_L**.
0xE8	Square: **res** = pow(**arg**, 2).
0xE9	Cube: **res** = pow(**arg**, 3).
0xEA	Square root: **res** = round(sqrt(**arg**) * 256), **res_H** is the integer part, **res_L** is fractional part.
0xEB	Cube root: **res** = round(cbrt(**arg**) * 256), **res_H** is the integer part, **res_L** is fractional part.
0xEC	Exponent: **res** = round(exp(**arg**)), where exp(x) is e^x.
0xED	Natural logarithm: **res** = round(log(**arg**) * 256), where log(x) is $\ln(x)$. **res** is signed. **res_H** is the integer part, **res_L** is the fractional part.
0xEE	Base 2 logarithm: **res** = round(log2(**arg**) * 256), where log2(x) is $\log_2(x)$. **res** is signed. **res_H** is the integer part, **res_L** is the fractional part.
0xEF	Base 10 logarithm: **res** = round(log10(**arg**) * 256), where log10(x) is $\log_{10}(x)$. **res** is signed. **res_H** is the integer part, **res_L** is the fractional part.
0xF0	Sine: **res** = round(sin(**arg_pi**) * 32768). **arg_pi** is **arg** mapped to range $[-\pi, \pi)$. -128 is -π, 127 is π (almost). **res** is signed and represents the fraction in range $[-1, 1)$.
0xF1	Inverse sine: **res** = round(asin(**arg_norm_sign**) * $\frac{32768}{\pi}$). **arg_norm_sign** is **arg** mapped to range $[-1, 1)$. -128 is -1, 127 is 1 (almost). **res** is signed and represents the range $[-\pi, \pi)$. This covers the whole set of asin value which is $\left[-\frac{\pi}{2}, \frac{\pi}{2}\right]$.
0xF2	Cosine: **res** = round(cos(**arg_pi**) * 32767). **res** is signed and represents the fraction in range $[-1, 1]$.
0xF3	Inverse cosine: **res** = round(acos(**arg_norm_sign1**) * $\frac{32768}{\pi}$). **arg_norm_sign1** is **arg** mapped to range $[-1, 1]$. -127 is -1, 127 is 1. **res** is signed and represents the range $[-\pi, \pi)$. This covers almost the whole set of acos values which is $[0, \pi]$.
0xF4	Tangent: **res** = round(tan(**arg_pi2**) * 256). **arg_pi2** is **arg** mapped to range $\left[-\frac{\pi}{2}, \frac{\pi}{2}\right)$. -128 is $-\frac{\pi}{2}$, 127 is $\frac{\pi}{2}$ (almost). **res** is signed and represents the range $[-128, 128)$.
0xF5	Inverse tangent: **res** = round(acos(**arg_norm_sign8**) * $\frac{65536}{\pi}$). **arg_norm_sign8** is **arg** mapped to range $[-8, 8)$. -128 is -8, 127 is 8 (almost). **res** is signed and represents the range $[-\frac{\pi}{2}, \frac{\pi}{2})$. This covers almost the whole set of atan values which is $[-\frac{\pi}{2}, \frac{\pi}{2}]$.
0xF6	Square signed: **res** = pow(**arg**, 2), **arg** is treated as signed.
0xF7	Square signed fractional: **res** = round(pow(**arg_norm_sign**, 2) * 65536). **res** represents a value in range [0, 1).

0xF8	Square fractional: **res** = round(pow(**arg_norm**, 2) * 65536). **arg_norm** is **arg** mapped to range $[0, 1)$. 0 is 0, 255 is 1 (almost). **res** represents a value in range [0, 1).
0xF9	Cube fractional: **res** = round(pow(**arg_norm**, 3) * 65536). **res** represents a value in range [0, 1).
0xFA	Square root fractional: **res** = round(sqrt(**arg_norm**) * 65536). **res** represents a value in range [0, 1).
0xFB	Cube root fractional: **res** = round(cbrt(**arg_norm**) * 65536). **res** represents a value in range [0, 1).
0xFC	Exponent fractional: **res** = round(exp(**arg_norm**) * 256), where exp(x) is e^x. **res** represents a value in range [0, 256).
0xFD	Natural logarithm fractional: **res** = round(log(**arg_norm**) * 256), where log(x) is $\ln(x)$. **res** represents a value in range [-128, 128).
0xFE	Base 2 logarithm fractional: **res** = round(log2(**arg_norm**) * 256), where log2(x) is $\log_2(x)$. **res** represents a value in range [-128, 128).
0xFF	Base 10 logarithm fractional: **res** = round(log10(**arg_norm**) * 256), where log10(x) is $\log_{10}(x)$. **res** represents a value in range [-128, 128).

In the above notation, **arg** is treated as unsigned unless otherwise specified. Same applies for **res**. For **OP** in range 0xE8 – 0xFF the result saturates when available range (0 – 65535 for unsigned, -32768 – 32767 for signed) is exceeded. So if the true result is 65536, actual result will be 65535, instead of wrapping back to 0. With true result equal to -40000, actual result will be -32768, etc.

12.5 Value Mapping Summary

Function	**OP**	**Input mapping**	**Output mapping**
Square	0xE8	1:1	1:1
Square fractional	0xF8	[0, 255] → [0, 1)	[0, 65535] → [0, 1)
Square signed	0xF6	1:1 (input signed)	1:1
Square signed fractional	0xF7	[-128, 127] → [-1, 1)	[0, 65535] → [0, 1)
Cube	0xE9	1:1	1:1
Cube fractional	0xF9	[0, 255] → [0, 1)	[0, 65535] → [0, 1)
Square root	0xEA	1:1	[0, 65535] → [0, 256)
Square root fractional	0xFA	[0, 255] → [0, 1)	[0, 65535] → [0, 1)
Cube root	0xEB	1:1	[0, 65535] → [0, 256)
Cube root fractional	0xFB	[0, 255] → [0, 1)	[0, 65535] → [0, 1)
Exponent	0xEC	1:1	1:1
Exponent fractional	0xFC	[0, 255] → [0, 1)	[0, 65535] → [0, 256)
Natural log	0xED	1:1	[-32768, 32767] → [-128, 128)
Natural log fractional	0xFD	[0, 255] → [0, 1)	[-32768, 32767] → [-128, 128)
Base 2 log	0xEE	1:1	[-32768, 32767] → [-128, 128)
Base 2 log fractional	0xFE	[0, 255] → [0, 1)	[-32768, 32767] → [-128, 128)
Base 10 log	0xEF	1:1	[-32768, 32767] → [-128, 128)
Base 10 log fractional	0xFF	[0, 255] → [0, 1)	[-32768, 32767] → [-128, 128)
Sine	0xF0	[-128, 127] → [-π, π)	[-32768, 32767] → [-1, 1)
Inverse sine	0xF1	[-128, 127] → [-1, 1)	[-32768, 32767] → [-π, π)
Cosine	0xF2	[-128, 127] → [-π, π)	[-32767, 32767] → [-1, 1]
Inverse cosine	0xF3	[-127, 127] → [-1, 1]	[-32768, 32767] → [-π, π)
Tangent	0xF4	[-128, 127] → $\left[-\frac{\pi}{2}, \frac{\pi}{2}\right)$	[-32768, 32767] → [-128, 128)
Inverse tangent	0xF5	[-128, 127] → [-8, 8)	[-32768, 32767] → $[-\frac{\pi}{2}, \frac{\pi}{2})$

All above mappings are linear. **1:1** means unsigned value with no mapping in-between, unless otherwise specified.