

CMOS 4-BIT SINGLE CHIP MICROCOMPUTER

# ***E0C63000 CORE CPU MANUAL***



## ***NOTICE***

---

*No part of this material may be reproduced or duplicated in any form or by any means without the written permission of Seiko Epson. Seiko Epson reserves the right to make changes to this material without notice. Seiko Epson does not assume any liability of any kind arising out of any inaccuracies contained in this material or due to its application or use in any product or circuit and, further, there is no representation that this material is applicable to products requiring high level reliability, such as medical products. Moreover, no license to any intellectual property rights is granted by implication or otherwise, and there is no representation or warranty that anything made in accordance with this material will be free from any patent or copyright infringement of a third party. This material or portions thereof may contain technology or the subject relating to strategic products under the control of the Foreign Exchange and Foreign Trade Law of Japan and may require an export license from the Ministry of International Trade and Industry or other approval from another government agency. Please note that "EOC" is the new name for the old product "SMC". If "SMC" appears in other manuals understand that it now reads "EOC".*

# E0C63000 CORE CPU MANUAL

## PREFACE

This manual explains the architecture, operation and instruction of the core CPU E0C63 of the CMOS 4-bit single chip microcomputer E0C63 Family.

Also, since the memory configuration and the peripheral circuit configuration is different for each device of the E0C63 Family, you should refer to the respective manuals for specific details other than the basic functions.

## CONTENTS

<b>CHAPTER 1</b>	<b>OUTLINE</b>	<b>1</b>
1.1	Features	1
1.2	Instruction Set Features	1
1.3	Block Diagram	2
1.4	Input-Output Signals	2
<b>CHAPTER 2</b>	<b>ARCHITECTURE</b>	<b>4</b>
2.1	ALU and Registers	4
2.1.1	ALU	4
2.1.2	Register configuration	4
2.1.3	Flags	5
2.1.4	Arithmetic operations with numbering system	7
2.1.5	EXT register and data extension	8
2.2	Program Memory	11
2.2.1	Configuration of program memory	11
2.2.2	PC (program counter)	11
2.2.3	Branch instructions	12
2.2.4	Table look-up instruction	16
2.3	Data Memory	17
2.3.1	Configuration of data memory	17
2.3.2	Addressing for data memory	18
2.3.3	Stack and stack pointer	19
2.3.4	Memory mapped I/O	21
<b>CHAPTER 3</b>	<b>CPU OPERATION</b>	<b>22</b>
3.1	Timing Generator and Bus Cycle	22
3.2	Instruction Fetch and Execution	22
3.3	Data Bus (Data Memory) Control	23
3.3.1	Data bus status	23
3.3.2	High-impedance control	23
3.3.3	Interrupt vector read	24
3.3.4	Memory write	24
3.3.5	Memory read	25
3.4	Initial Reset	25
3.4.1	Initial reset sequence	25
3.4.2	Initial setting of internal registers	26

**CONTENTS**

- 3.5 *Interrupts* ..... 26
  - 3.5.1 *Interrupt vectors* ..... 26
  - 3.5.2 *Interrupt sequence* ..... 27
  - 3.5.3 *Notes for interrupt processing* ..... 30
- 3.6 *Standby Status* ..... 31
  - 3.6.1 *HALT status* ..... 31
  - 3.6.2 *SLEEP status* ..... 31
- CHAPTER 4 *INSTRUCTION SET* ..... 33**
  - 4.1 *Addressing Mode* ..... 33
    - 4.1.1 *Basic addressing modes* ..... 33
    - 4.1.2 *Extended addressing mode* ..... 35
  - 4.2 *Instruction List* ..... 37
    - 4.2.1 *Function classification* ..... 37
    - 4.2.2 *Symbol meanings* ..... 38
    - 4.2.3 *Instruction list by function* ..... 40
    - 4.2.4 *List in alphabetical order* ..... 48
    - 4.2.5 *List of extended addressing instructions* ..... 55
  - 4.3 *Instruction Formats* ..... 59
  - 4.4 *Detailed Explanation of Instructions* ..... 60

# CHAPTER 1 OUTLINE

The E0C63000 is the core CPU of the 4-bit single chip microcomputer E0C63 Family that utilizes original EPSON architecture. It has a large and linear addressable space, maximum 64K words (13 bits/word) program memory (code ROM area) and maximum 64K words (4 bits/word) data memory (RAM, data ROM and I/O area), and high speed, abundant instruction sets. It operates in a wide range of supply voltage and features low power consumption. Furthermore, modularization of programs can be done easily because the program memory does not need bank and page management and relocatable programming is possible.

In addition, it has adopted a unified architecture and a peripheral circuit interface in memory mapped I/O method to flexibly meet future expansion of the E0C63 Family.

## 1.1 Features

---

The E0C63000 boasts the below features.

<i>Program memory</i>	Maximum 64K × 13 bits (linear address, non-page method)	
<i>Data memory</i>	Maximum 64K × 4 bits	
<i>Basic instruction set</i>	47 types with 5 types of basic addressing modes and 3 types of extended addressing modes	
<i>Instruction cycle</i>	1 cycle (2 clocks), 2 cycles (4 clocks) and 3 cycles (6 clocks)	
<i>Register configuration</i>	Data register	2 × 4 bits
	Index register	2 × 16 bits
	Address extension register	8 bits
	Program counter	16 bits
	Stack pointer	2 × 8 bits
	Condition flag	4 bits
	Queue register	16 bits
<i>Interrupt function</i>	NMI (Non Maskable Interrupt) vector	1
	Hardware interrupt vector	Maximum 15 vectors
	Software interrupt vector	Maximum 63 vectors
<i>Standby function</i>	HALT/SLEEP	
<i>Peripheral circuit interface</i>	Memory mapped I/O method	
<i>Pipeline processing</i>	2 stages (fetch and execution) pipeline processing	

## 1.2 Instruction Set Features

---

- (1) It adopts high efficiency machine cycles, high speed and abundant instruction set. Almost all standard instructions operate in 1 cycle (2 clock).
- (2) Both the program space and the data space are designed as a 64K-word linear space without page concept and can be addressed with 1 instruction.
- (3) The instruction system includes relocatable jump instructions and allows a relocatable programming. Thus modular programming and software library development can be realized easily, and it increases an efficiency for developing applications.
- (4) Memory management can be done easily by 5 types of basic addressing modes, 3 types of extended addressing modes with the address extension register and 16-bit operation function that is useful in address calculations.
- (5) 8-bit data processing is possible using the table look-up instruction and other instructions.
- (6) Some instructions support a numbering system, thus binary to hexadecimal software counters can be made easily.

### 1.3 Block Diagram

Figure 1.3.1 shows the E0C63000 block diagram.

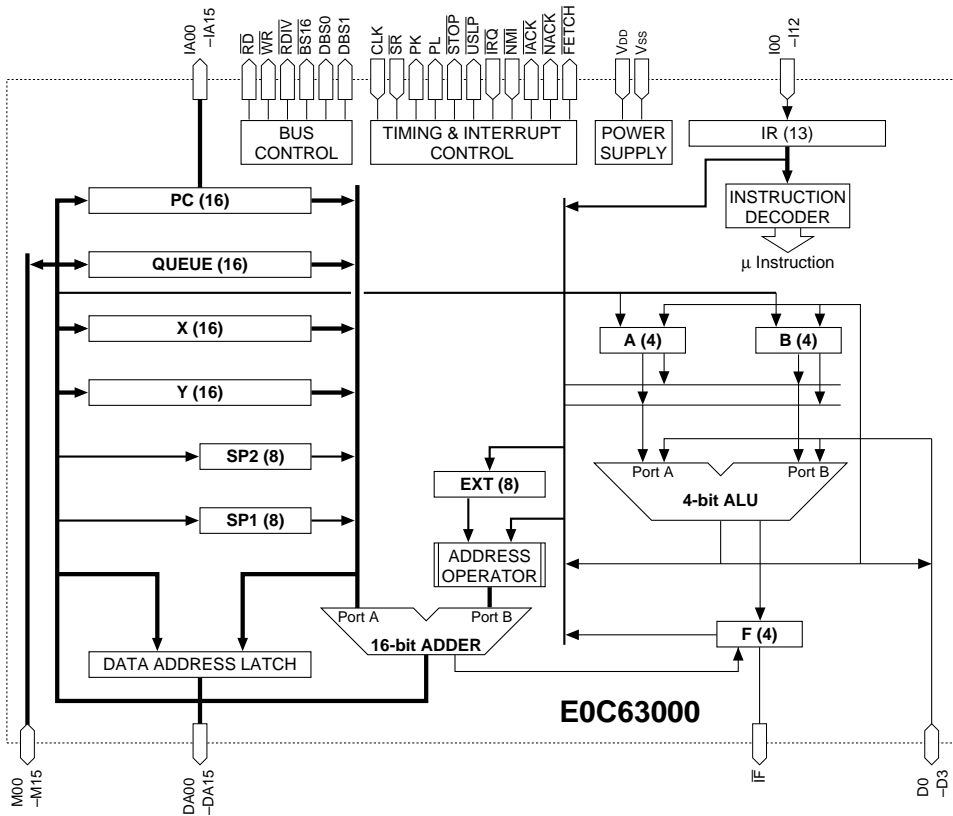


Fig. 1.3.1 E0C63000 block diagram

### 1.4 Input-Output Signals

Tables 1.4.1 (a) and 1.4.1 (b) show the input/output signals between the E0C63000 and peripheral circuits.

Table 1.4.1(a) Input/output signal list (1)

Type	Terminal name	I/O	Function
Power supply	VDD (VD1)	I	Power supply (+) Inputs a plus supply voltage.
	VSS (VS1)	I	Power supply (-) Inputs a minus supply voltage.
Clock	CLK	I	Clock input Inputs the system clock from the peripheral circuit.
	PK PL	O	2-phase divided clock output Outputs the 2-phase divided signals to be generated from the system clock input to the CLK terminal as following phase.  <div style="text-align: center;"> </div>
Address bus	IA00-IA15	O	Instruction address output Outputs an instruction (code ROM) address.
	DA00-DA15	O	Data address output Outputs a data (RAM, I/O) address.

Table 1.4.1(b) Input/output signal list (2)

Type	Terminal name	I/O	Function														
Data bus	I00–I12	I	Instruction bus Inputs an instruction code.														
	M00–M15	I/O	16-bit data bus A bidirectional data bus to connect to the RAM (stack RAM) for 16-bit accessing.														
	D0–D3	I/O	4-bit data bus A bidirectional data bus to connect to the RAM and I/O.														
Bus control signal	RD	O	Data read Goes to a low level when the CPU reads data (from RAM, I/O).														
	WR	O	Data write Goes to a low level when the CPU writes data (to RAM, I/O).														
	RDIV	O	Read interrupt vector Goes to a low level when the CPU reads an interrupt vector.														
System control signal	SR	I	Reset input A low level input resets the CPU.														
	USLP	O	Micro sleep Goes to a low level when the CPU executes the SLP instruction. The peripheral circuit stops oscillation on the basis of this signal.														
Interrupt signal	NMI	I	Non-maskable interrupt request An interrupt request terminal for an interrupt that cannot be masked by software. It is accepted at the falling edge of an input signal to this terminal.														
	IRQ	I	Interrupt request An interrupt request terminal for interrupts that can be masked by software. It is accepted by a low level signal input to this terminal.														
	IACK	O	Interrupt acknowledge Goes to a low level while executing an NMI or IRQ interrupt response cycle.														
	NACK	O	Non-maskable interrupt acknowledge Goes to a low level while executing a non-maskable interrupt response cycle.														
Status signal	FETCH	O	Fetch cycle Goes to a low level when the CPU fetches an instruction.														
	STOP	O	Stop signal Goes to a low level when the CPU is in stop status after executing the HALT or SLP instruction, or in reset status (SR is low).														
	IF	O	Interrupt flag Outputs a status (inverted value) of the interrupt flag in the flag (F) register.														
	BS16	O	16-bit access Goes to a low level when the CPU accesses to a 16-bit RAM.														
	DBS0 DBS1	O	Data bus status Outputs data bus status (for both the 4-bit and 16-bit data bus). <table border="1" style="margin-left: auto; margin-right: auto;"> <thead> <tr> <th>DBS1</th> <th>DBS0</th> <th>State</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>0</td> <td>High impedance</td> </tr> <tr> <td>0</td> <td>1</td> <td>Interrupt vector read</td> </tr> <tr> <td>1</td> <td>0</td> <td>Memory write</td> </tr> <tr> <td>1</td> <td>1</td> <td>Memory read</td> </tr> </tbody> </table>	DBS1	DBS0	State	0	0	High impedance	0	1	Interrupt vector read	1	0	Memory write	1	1
DBS1	DBS0	State															
0	0	High impedance															
0	1	Interrupt vector read															
1	0	Memory write															
1	1	Memory read															

See Chapter 3, "CPU OPERATION", for the timing of the signals.

# CHAPTER 2 ARCHITECTURE

This chapter explains the E0C63000 ALU, registers, configuration of the program memory area and data memory area, and addressing.

## 2.1 ALU and Registers

### 2.1.1 ALU

The ALU (Arithmetic and Logic Unit) loads 4-bit data from a memory or a register and operates the data according to the instruction. Table 2.1.1.1 shows the ALU operation functions.

Table 2.1.1.1 ALU operation functions

Function classification	Mnemonic	Operation
Arithmetic	ADD	Addition
	ADC	Addition with carry
	SUB	Subtraction
	SBC	Subtraction with carry
	CMP	Comparison
	INC	Increment (adds 1)
	DEC	Decrement (subtracts 1)
Logic	AND	Logical product
	OR	Logical sum
	XOR	Exclusive OR
	BIT	Bit test
	CLR	Bit clear
	SET	Bit set
	TST	Bit test
Rotate / shift	RL	Rotate to left with carry
	RR	Rotate to right with carry
	SLL	Logical shift to left
	SRL	Logical shift to right

The operation result is stored to a register or memory according to the instruction. In addition, the Z (zero) flag and C (carry) flag are set/reset according to the operation result.

### 2.1.2 Register configuration

Figure 2.1.2.1 shows the register configuration of the E0C63000.

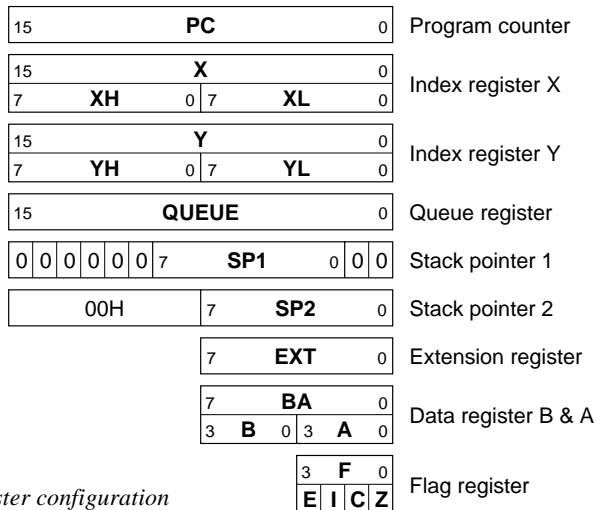


Fig. 2.1.2.1 Register configuration



- **A and B registers**

The A and B registers are respective 4-bit data registers that are used for data transfer and operation with other registers, data memories or immediate data. They are used independently for 4-bit transfer/operations and used in a BA pair that makes the B register the high-order 4 bits for 8-bit transfer/operations.

- **X and Y registers**

The X and Y registers are respective 16-bit index registers that are used for indirect addressing of the data memory. These registers are configured as an 8-bit register pair (high-order 8 bits: XH/YH, low-order 8 bits: XL/YL) and data transfer/operations can be done in an 8-bit unit or a 16-bit unit.

- **PC (program counter)**

The PC is a 16-bit counter to address a program memory and indicates the following address to be executed.

- **SP1 and SP2 (stack pointers)**

The SP1 and SP2 are respective 8-bit registers that indicate a stack address in the data memory. 8 bits of the SP1 correspond to the DA02 to DA09 bits of the address bus for 16-bit data accessing (address stacking) and it is used to operate the stack in a 4-word (16-bit) unit. 8 bits of the SP2 correspond to the low-order 8 bits (DA01 to DA07) of the address bus for 4-bit data accessing and it is used to operate stack in 1-word (4-bit) unit.

See Section 2.3.3, "Stack and stack pointer" for details of the stack operation.

- **EXT register**

The EXT register is an 8-bit data register that is used when an address or data is extended into 16 bits. See Section 2.1.5, "EXT register and data extension", for details.

- **F register**

The F register includes 4 bits of flags; Z and C flags that are changed by operation results, I flag that is used to enable/disable interrupts, and E flag that indicates extended addressing mode.

- **Queue register**

The queue register is used as a queue buffer for data when the SP1 processes 16-bit stack operations. This register is provided in order to process 16-bit data pop operations from the SP1 stack at high-speed. The queue register is accessed by the hardware, so it is not necessary to be aware of the register operation when programming.

### 2.1.3 Flags

The E0C63000 contains a 4-bit flag register (F register) that indicates such things as the operation result status within the CPU.

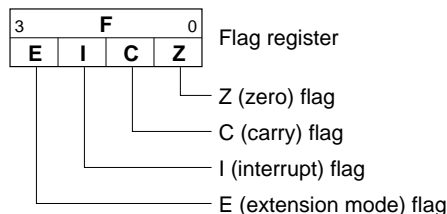


Fig. 2.1.3.1 F (flag) register

- **Z (zero) flag**

The Z flag is set to "1" when the execution result of an arithmetic instruction or a shift/rotate instruction has become "0" and is reset to "0" when the result is other than "0".

**Arithmetic instructions that change the Z flag:**

ADD, ADC, SUB, SBC, CMP, INC, DEC, AND, OR, XOR, BIT, CLR, SET, TST

*Shift/Rotate instructions that change the Z flag:*

SLL, SRL, RL, RR

The Z flag is used for condition judgments when executing the conditional jump ("JRZ sign8" and "JRNZ sign8") instructions, thus it is possible to branch processing to a routine according to the operation result.

• **C (carry) flag**

The C flag is set to "1" when a carry (carry from the most significant bit) or a borrow (the most significant bit borrows) has been generated by the execution of an arithmetic instruction and a shift/rotate instruction, otherwise the flag is set to "0".

*Arithmetic instructions that change the C flag:*

ADD, ADC, SUB, SBC, CMP, INC, DEC

(It is different from the Z flag, the logic operation instructions except for the instruction that operates the F register does not change the C flag. In addition, the ADD instructions for the X and Y register operations and the INC and DEC instructions for the stack pointer operation does not change the C flag.)

*Shift/Rotate instructions that change the C flag:*

SLL, SRL, RL, RR

The C flag is used for condition judgments when executing the conditional jump ("JRC sign8" and "JRNC sign8") instructions, thus it is possible to branch processing to a routine according to the operation result.

• **I flag**

The I flag permits and forbids the hardware interrupts except for the NMI. By setting the I flag to "1", the CPU enters in the EI (enable interrupts) status and the hardware interrupts are enabled. When the I flag is set to "0", the CPU is in the DI (disable interrupts) and the interrupts except for NMI are disabled. Furthermore, when a hardware interrupt (including the NMI) is generated, the I flag is reset to "0" and interrupts after that point are disabled. The multiple interrupts can be accepted by setting the I flag to "1" in the interrupt processing routine.

The NMI (non-maskable interrupt) is accepted regardless of the I flag setting.

The software interrupts are accepted regardless of the I flag and do not reset the I flag.

The I flag is set to "0" (DI status) at an initial reset, therefore it is necessary to set "1" before using interrupts by software.

See Section 3.5, "Interrupts" for details.

• **E (extension mode) flag**

The E flag indicates whether an extended addressing that uses the EXT (extension) register is valid or invalid. When data is loaded into the EXT register, this flag is set to "1" and the data of the instruction immediately after that (extended addressable instructions only) is extended with the EXT register.

Then the instruction is executed and the E flag is reset to "0".

See Section 2.1.5, "EXT register and data extension" for details.

• **Flag operations**

As described above, the flags are automatically set/reset by the hardware. However, it is necessary to set by software, especially the I flag. The following instructions are provided in order to operate the F flag.

LD	%A, %F	Reads all the flag data	XOR	%F, imm4	Inverts flag(s)
LD	%F, %A	Writes all the flag data	PUSH	%F	Evacuates the F register
LD	%F, imm4	Writes all the flag data	POP	%F	Returns the F register
AND	%F, imm4	Resets flag(s)	RETI		Returns the F register*
OR	%F, imm4	Sets flag(s)			

\* The RETI instruction is used to return from interrupt processing routines (including software interrupts), and returns the F register data that was evacuated when the interrupt was generated.

### 2.1.4 Arithmetic operations with numbering system

In the E0C63000, some instructions support a numbering system. These instructions are indicated with the following notations in the instruction list.

```
ADC  operand,n4
SBC  operand,n4
INC  operand,n4
DEC  operand,n4
```

(See "Instruction List" or "Detailed Explanation of Instructions" for the contents of the operand.)

"n4" is a radix, and can be specified from 1 to 16. The additions/subtractions are done in the numbering system with n4 as the radix. Various counters (such as binary, octal, decimal and hexadecimal) can be realized easily by software.

The Z flag indicates that an operation result is "0" or not in arithmetics with any numbering system. The C flag indicates a carry/borrow according to the radix.

The following shows examples of these operation.

Example 1) Octal addition `ADC %B,%A,8` (C flag is "0" before operation)

Setting value		Result	F register			
B register	A register	B register	E	I	C	Z
0010B(2)	0111B(7)	0001B(1)	0	-	1	0
0101B(5)	0011B(3)	0000B(0)	0	-	1	1

Example 2) Decimal subtractio `SBC %B,%A,10` (C flag is "0" before operation)

Setting value		Result	F register			
B register	A register	B register	E	I	C	Z
1001B(9)	0111B(7)	0010B(2)	0	-	0	0
0001B(1)	0010B(2)	1001B(9)	0	-	1	0

Example 3) 3-digit BCD down counter

```
LDB  %EXT,0           ; Counter base address [0010H]
LD   %XL,0x10
LDB  [%X]+,0         ; Initial value setting [100]
LDB  [%X]+,0
LDB  [%X]+,1
:
:
CTDOWN:               ; Count down subroutine-----
LDB  %EXT,0           ; Counter base address [0010H]
LD   %XL,0x10

DEC  [%X]+,10        ; Decrements digit 1
SBC  [%X]+,0,10     ; Decrements carry from digit 2
SBC  [%X],0,10      ; Decrements carry from digit 3
CALR CTDISP         ; Count number display routine
LD   %A,0           ; Zero check
ADD  %A,[%X]
ADD  %X,-1
ADD  %A,[%X]
ADD  %X,-1
JRNZ CTEXTIT       ; Return if counter is not zero
CALR CTOVER        ; Count over processing routine
CTEXIT:
RET
```

This routine constructs a 3-digit BCD counter using the decimal operation instructions underlined. Calling the CTDOWN subroutine decrements the counter, and then returns to the main routine. If the counter has to be zero, the CTOVER subroutine is called before returning to the main routine to process the end of counting.

• **Notes in numbering operations**

When performing a numbering operation, set operands in correct notation according to the radix before operation.

For example, if a decimal operation is done for hexadecimal values (AH to FH), the correct operation result is not obtained as shown in the following example.

Example: ADC %B, %A, 10

	Setting value		Result	F register				
	B register	A register		B register	E	I	C	
1	1001B(9)	1001B(9)	1000B(8)	0	-	1	0	○
2	0101B(AH)	1001B(9)	1001B(9)	0	-	1	0	△
3	1010B(AH)	1010B(AH)	1010B(AH)	0	-	1	0	×
4	1010B(AH)	1111B(FH)	1111B(FH)	0	-	1	0	×

Example 1 operates correctly because a decimal value is loaded in the B and A registers.

Examples 3 and 4 do not operate correctly.

Example 2 operates correctly even though it is a wrong setting.

**2.1.5 EXT register and data extension**

The E0C63000 has a linear 64K-word addressable space, therefore it is required to handle 16-bit address data. The EXT register and the F flag that extend 8-bit data into 16-bit data permit 16-bit data processing. The EXT register is an 8-bit register for storing extension data. The E flag indicates that the EXT register data is valid (extended addressing mode), and is set to "1" by writing data to the EXT register. The E flag is reset at 1 cycle after setting (during executing the next instruction), therefore an EXT register data is valid only for the executable instruction immediately after writing. However, that executable instruction must be a specific instruction which permits the extended addressing to extend the data using the EXT register. These instructions are specified in "Instruction List" and "Detailed Explanation of Instructions". Make sure of the instructions when programming.

*Note: Do not use instructions (see Instruction List) which are invalid for the extended addressing when the E flag is set to "1". (Do not use them following instructions that write data to the EXT register or that set the E flag.) Normal operations cannot be guaranteed if such instructions are used.*

**(1) Operation for EXT register and E flag (flag register)**

The following explains the operation for the EXT register and the E flag (flag register).

• **Data setting to the EXT register**

The following two instructions are provided to set data in the EXT register.

- LDB %EXT, imm8 Loads an 8-bit immediate data to the EXT register
- LDB %EXT, %BA Loads the content of the BA register to the EXT register

By executing the instruction, the EXT flag is set to "1" and it indicates that the content of the EXT register is valid (the content of the EXT register will be used for data extension in the following instructions).

Furthermore, the content of the EXT register can be read using the instruction below.

- LDB %BA, %EXT Loads the content of the EXT register to the BA register

• **Setting/resetting the E flag**

As mentioned above, the E flag is set to "1" by data setting to the EXT register and reset to "0" while executing the next instruction.

In addition, the E flag can be set/reset using the following instructions that operate the flags.

- LD %F, %A Writes all the flag data
- LD %F, imm4 Writes all the flag data
- AND %F, imm4 Resets flag(s)
- OR %F, imm4 Sets flag(s)
- XOR %F, imm4 Inverts flag(s)

The EXT register maintains the data set previously until new data is written or an initial reset. In other words, the content of the EXT register becomes valid by only setting the E flag using an above instruction without the register writing and is used for an extended addressing. However, the EXT register is undefined at an initial reset, therefore, do not directly set the E flag except when the content of the EXT register has been set for certain.

The following shows the other instructions related to flag data transfer.

LD	%A, %F	Reads all the flag data
PUSH	%F	Evacuates the F register
POP	%F	Returns the F register
RETI		Returns the F register *

- \* The RETI instruction is used to return from interrupt processing routines (including software interrupts), and returns the F register data that was evacuated when the interrupt was generated. If an interrupt (including NMI) is generated while fetching an instruction, such as a "LDB %EXT, ●●" instruction or an instruction which writes data to the flag register (the E flag may be set), the interrupt is accepted after fetching (and executing) the next instruction. In normal processing, data extension processing is not performed after returning from the interrupt service routine because the interrupt processing including the F register evacuation is performed after the data extension has finished (E flag is reset). However, if the stack data in the memory is directly changed in the interrupt service routine, the F register in which the E flag is set may return. In this case, the instruction immediately after returning by the RETI instruction is executed in the extended addressing mode by the E flag set to "1". Pay attention to the F register setting except when consciously describing such a processing. It is necessary to pay the same attention when returning the F register using the "POP %F" instruction.

## (2) Extension with E flag

The following explains the instructions that can be executed when the E flag is set to "1" and its operation.

- **Modifying the indirect addressing with the X and Y registers (for 4-bit data access)**

The indirect addressing instructions, which contain [%X] or [%Y] as an operand and accesses 4-bit data using the X or Y register, functions as an absolute addressing that uses the EXT register data together with the E flag (= "1").

When an 8-bit immediate data (imm8) is written to the EXT register and the E flag is set immediately before these instructions, the instruction is modified executing as [%X] = [0000H + imm8] or [%Y] = [FF00H + imm8]. Therefore, the addressable space with this function is data memory address from 0000H to 00FFH when [%X] is used, and from FF00H to FFFFH when [%Y] is used. Generally, data that are often used are allocated to the data memory from 0000H to 00FFH and the area from FF00H to FFFFH is assigned to the I/O memory area (for peripheral circuit control), so these areas are frequently accessed. To access these areas by a normal indirect addressing (if the E flag has not been set) using the X or Y register, two or three steps of instructions are necessary for setting an address data. In other words, using this function promotes efficiency of the entire program. See Section 2.3, "Data Memory" for details of the data memory.

Examples:

```
LDB  %EXT, 0x37
LD   %A, [%X]      ...Works as "LD %A, [0x0037]"

LDB  %EXT, 0x9C
ADD  [%Y], 5       ...Works as "ADD [0xFF9C], 5"
```

*Note: This function can be used by only the specific instructions which permits the extended addressing (see "Instruction List"). Be aware that the operation cannot be guaranteed if the instructions indicated below are used.*

1. Instructions which have a source and /or a destination operand with the post-increment function, [%X]+ and [%Y]+.
2. Instructions which have [%X] and/or [%Y] in both the source and destination operands.
3. The RETD instruction and the LDB instructions which transfers 8-bit data.

- **16-bit data transfer/arithmetic for the index registers X and Y**

The following six instructions, which handle the X or Y register and have an 8-bit immediate data as the operand, permit the extended addressing.

```
LDB  %XL,imm8      LDB  %YL,imm8
ADD  %X,sign8      ADD  %Y,sign8
CMP  %X,imm8       CMP  %Y,imm8
```

When data is written to the EXT register and the E flag is set immediately before these instructions, the data is processed after extending into 16-bit; imm8 (sign8) is used as the low-order 8 bits and the content of the EXT register is used as the high-order 8 bits.

**Examples:**

```
LDB  %EXT,0x15
LDB  %XL,0x7D      ...Works as "LD %X,0x157D"

LDB  %EXT,0xB8
ADD  %X,0x4F      ...Works as "ADD %X,0xB84F"

LDB  %EXT,0xE6
CMP  %X,0xA2      ...Works as "CMP %X,0x19A2"
                        * 19H = FFH - [EXT] (E6H)
```

Above examples use the X register, but work the same even when the Y register is used.

*Note: The CMP instruction performs a subtraction with a complement, therefore it is necessary to set the complement (1's complement) of the high-order 8-bit data in the EXT register.  
EXT register ← [FFH - High-order 8-bit data]*

- **Extending branch addresses**

The following PC relative branch instructions, which have a signed 8-bit relative address as the operand, permit extended addressing.

```
JR   sign8      JRC  sign8      JRNC sign8      JRZ   sign8      JRNZ  sign8
CALR sign8
```

When data is written to the EXT register and the E flag is set immediately before these instructions, the relative address is processed after extending into signed 16-bit; sign8 is used as the low-order 8 bits and the content of the EXT register is as the high-order 8 bits.

**Examples:**

```
LDB  %EXT,0x64
JR   0x29      ...Works as "JR 0x6429"

LDB  %EXT,0x00
JR   127      ...Works as "JR 127"

LDB  %EXT,0xFF
JR   -128     ...Works as "JR -128"

LDB  %EXT,0x3A
JR*  0x88     ...Works as "JR* 0x3A88" (* = C, NC, Z, or NZ)

LDB  %EXT,0xF8
CALR 0x62     ...Works as "CALR 0xF862"
```

See Section 2.2.3, "Branch instructions" for the branch instructions.

## 2.2 Program Memory

### 2.2.1 Configuration of program memory

The E0C63000 can access a maximum 64K-word ( $\times 13$  bits) program memory space. In the individual model of the E0C63 Family, the ROM of which size is decided depending on the model is connected to this space to write a program and static data.

Figure 2.2.1.1 shows the program memory map of the E0C63000.

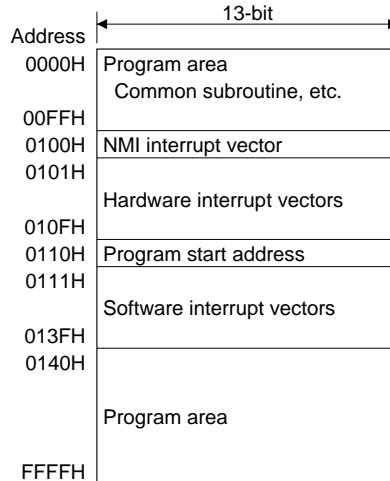


Fig. 2.2.1.1 E0C63000 program memory map

The E0C63000 can access 64K-word space linearly without any page management used in current 4-bit microcomputers.

As shown in Figure 2.2.1.1, the program start address after an initial reset is fixed at 0110H independent of the E0C63 Family models. Programming should be done so that the execution program starts from that address.

The address 0100H to 010FH is the hardware interrupt vector's area in which up to 16 interrupt vectors can be assigned. Address 0100H is for the exclusive use of NMI (non-maskable interrupt). The number of interrupt vectors is dependent on the interrupt function of the E0C63 Family models. Branch instructions to the interrupt service routines should be written in this area. See Section 3.5, "Interrupts" for details of the interrupts.

The address 0111H to 013FH is the software interrupt vector's area. Up to 63 software interrupts can be set up together with the hardware interrupt vector area. Set branch instructions to the interrupt service routines in this area similarly to the hardware interrupts.

Addresses from 0000H to 00FFH and from 0140H to FFFFH are program area. A call instruction (CALZ) that is for the exclusive use of the area from 0000H to 00FFH is provided so that the area is useful to store common subroutines that are called from relocatable modules.

### 2.2.2 PC (program counter)

The PC (program counter) is a 16-bit counter that keeps the program address to be executed next. The PC is incremented by executing every instruction step to execute a program sequentially. When a branch instruction is executed or an interrupt is generated, the content of the PC is modified to branch the process flow.

The PC covers the entire program memory space alone, therefore processing such as page management are unnecessary.

At initial reset, the PC is initialized to 0110H and the program starts executing from that address.

### 2.2.3 Branch instructions

Various branch instructions are provided for program repeat and subroutine calls that change a sequential program flow controlled with the PC. The branch instruction modifies the PC to branch the program to an optional address. The types of the branch instructions are classified as follows, according to their operation differences.

Table 2.2.3.1 Types of branch instructions

Type	Condition	Instruction
PC relative jump	Unconditional	JR
PC relative jump	Conditional	JRC, JRNC, JRZ, JRNZ
Indirect jump	Unconditional	JP
Absolute call	Unconditional	CALZ
PC relative call	Unconditional	CALR
Return	Unconditional	RET, RETS, RETD, RETI
Software interrupt	Unconditional	INT

• **PC relative jump instructions (JR)**

The PC relative jump instruction adds the relative address specified in the operand to the PC that has indicated the next address, and branches to that address. It permits relocatable programming. The relative address to be specified in the operand is a displacement from the PC value (address of the next instruction) when the branch instruction is executed to the branch destination address. When programming using the E0C63 Family assembler, it is not necessary to calculate displacements because a branch destination address can be defined as a label and it can be used as an operand. However, the range of branch destination addresses is different depending on the number of data bits that are handled as relative addresses.

The following explains the PC relative jump instructions and the relative addresses.

(1) *Instructions with a signed 8-bit immediate data sign8 that specifies a relative address*

```

Unconditional jump JR sign8
Conditional jump JRC sign8 JRNC sign8 JRZ sign8 JRNZ sign8
    
```

These instructions branch the program sequence with the sign8 specified in the operand as a signed 8-bit relative address. The range that can be branched is from the next instruction address -128 to +127. A value within the range from -128 to +127 should be used if specifying a value for jumping in the assembler. Generally branch destination labels such as "JR LABEL" are used, and they are expanded into the actual address by the assembler.

These instructions permit the extended addressing with the E flag, and the 8-bit relative address can be extended into 16 bits (the contents of the EXT register become the high-order 8 bits). In this case, the range that can be branched is from the next instruction address -32768 to +32767. Consequently, in the extended addressing mode these instructions can branch the entire 64K program memory.

Examples:

```

JR    -100    ...Jumps to the instruction 99 steps before
LDB  %EXT,100 ...((100 × 256) = 25600
JR    100    ...Jumps to the instruction 25701 steps after
    
```

The unconditional jump instruction "JR sign8" jumps to the branch destination unconditionally when it is executed.

The conditional jump instructions jump according to the status of C flag or the Z flag.

```

JRC  sign8    ...Jumps if the C flag is "1", or executes the next instruction if the C flag is "0"
JRNC sign8    ...Jumps if the C flag is "0", or executes the next instruction if the C flag is "1"
JRZ  sign8    ...Jumps if the Z flag is "1", or executes the next instruction if the Z flag is "0"
JRNZ sign8    ...Jumps if the Z flag is "0", or executes the next instruction if the Z flag is "1"
    
```



**(2) Instruction with a 4-bit A register data that specifies a relative address**

```
JR    %A
```

This instruction branches the program sequence with the content of the A register as an unsigned 4-bit relative address. The range that can be branched is from the next instruction address +0 to +15 (absolute value in the A register). This instruction is useful when operation results are used as the 4-bit relative addresses.

Example:

```
LD    %A, 4
JR    %A          ...Jumps to the instruction 5 steps after
```

**(3) Instruction with an 8-bit BA register data that specifies a relative address**

```
JR    %BA
```

This instruction branches the program sequence with the content of the BA register as an unsigned 8-bit relative address ( the B register data becomes the high-order 4 bits). The range that can be branched is from the next instruction address +0 to +255 (absolute value in the BA register). This instruction is useful when operation results are used as the 8-bit relative addresses.

Example:

```
LDB  %BA, 29
JR    %BA          ...Jumps to the instruction 30 steps after
```

**(4) Instruction with a data memory address within 0000H to 003FH in which the content specifies a 4-bit relative address**

```
JR    [addr6]
```

This instruction branches the program sequence with the content of the data memory specified by the [addr6] as an unsigned 4-bit relative address. The operand [addr6] can specify a data memory address within 0000H to 003FH. The range that can be branched is from the next instruction address +0 to +15 (absolute value in the specified data memory). For the data memory area that is specified with [addr6], bit operation instructions (CLR, SET, TST) are provided so that various flags can be set simply. This jump instruction can be used as a conditional jump according to these flags.

Example: When the content of the address 0010H is 4 (0100B).

```
SET  [0x0010], 0    ...Sets the bit 0 in the address 0010H to "1" ([0010H] = 5)
JR   [0x0010]      ...Jumps to the instruction 6 steps after
```

- **Indirect jump instruction (JP)**

The indirect jump instruction "JP %Y" loads the content of the Y register into the PC to branch to that address unconditionally. This instruction can branch entire 64K program memory because the 16-bit data in the Y register becomes a branch destination address as it is.

Example:

```
LDB  %EXT, 0x24
LDB  %YL, 0x00      ...Y = 2400H
JP   %Y            ...Jumps to the address 2400H
```

Figure 2.2.3.1 shows the operation of the jump instructions and the branch range.

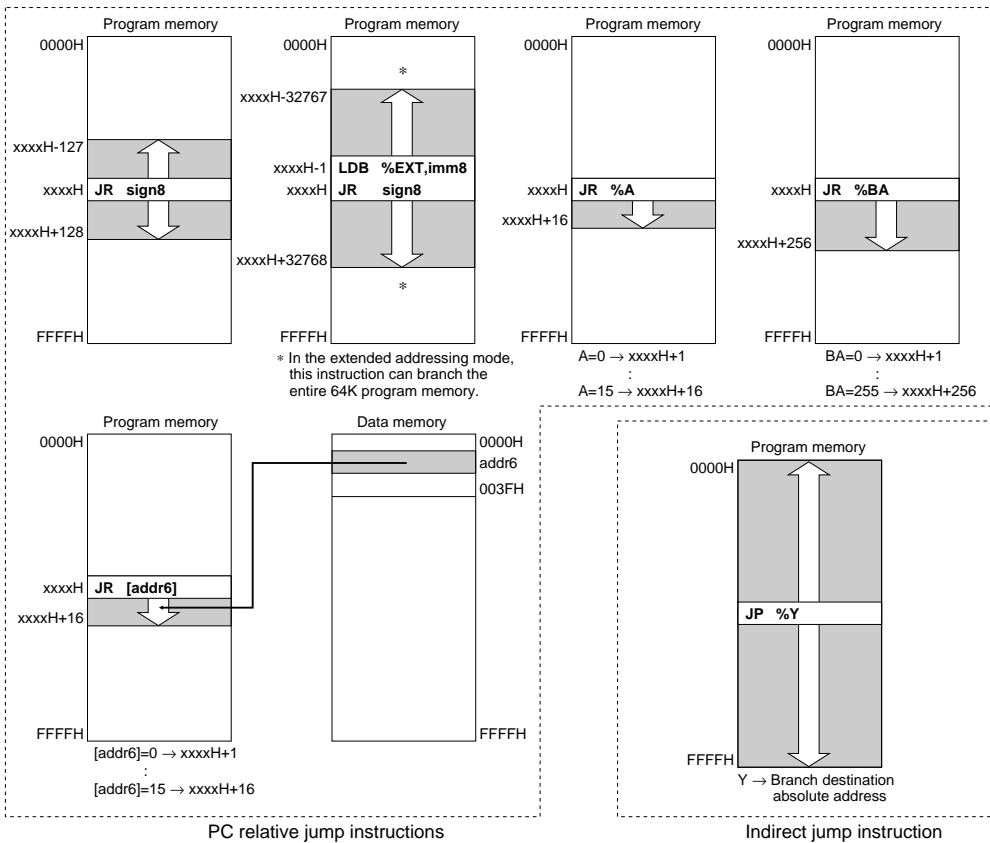


Fig. 2.2.3.1 Operation of jump instructions

• **Absolute call instruction (CALZ)**

The absolute call instruction "CALZ imm8" calls a subroutine within addresses 0000H to 00FFH. A subroutine start address (absolute address) should be specified to imm8. When the call instruction is executed, the PC value (address of the next instruction) is saved into the stack for return, then it branches to the specified address.

Generally common subroutines that are called from two or more modules are placed in this area when the program is developed as multiple modules.

Example:

```
CALZ 0x50                      ...Calls the subroutine located at the address 0050H
```

See Section 2.3.3, "Stack and stack pointer" for stack.

• **PC relative call instructions (CALR)**

The PC relative call instruction adds the relative address specified in the operand to the PC that has indicated the next address, and calls a subroutine started from that address. It permits relocatable programming.

The relative address to be specified in the operand is same as the PC related jump instruction.

The PC value (address of the next instruction) is saved into the stack before branching.

(1) *Instructions with a signed 8-bit immediate data sign8 that specifies a relative address*

```
CALR sign8
```

This instruction branches the program sequence with the sign8 specified in the operand as a signed 8-bit relative address. The range that can be branched is from the next instruction address - 128 to +127. A value within the range from -128 to +127 should be used if specifying a value for calling in the assembler. Generally branch destination labels such as "CALR LABEL" are used, and they are expanded into the actual address by the assembler.

This instruction permits the extended addressing with the E flag, and the 8-bit relative address can be extended into 16 bits (the contents of the EXT register becomes the high-order 8 bits). In this case, the range that can be branched is from the next instruction address -32768 to +32767. Consequently, in the extended addressing mode this instruction can call subroutines over a 64K program memory.

Examples:

```

CALR -50          ...Calls the subroutine 49 steps before
LDB  %EXT, 50    ...(50 × 256) = 17800
CALR  50         ...Calls the subroutine 17851 steps after
    
```

(2) *Instruction with a data memory address within 0000H to 003FH in which the content specifies a 4-bit relative address*

CALR [addr6]

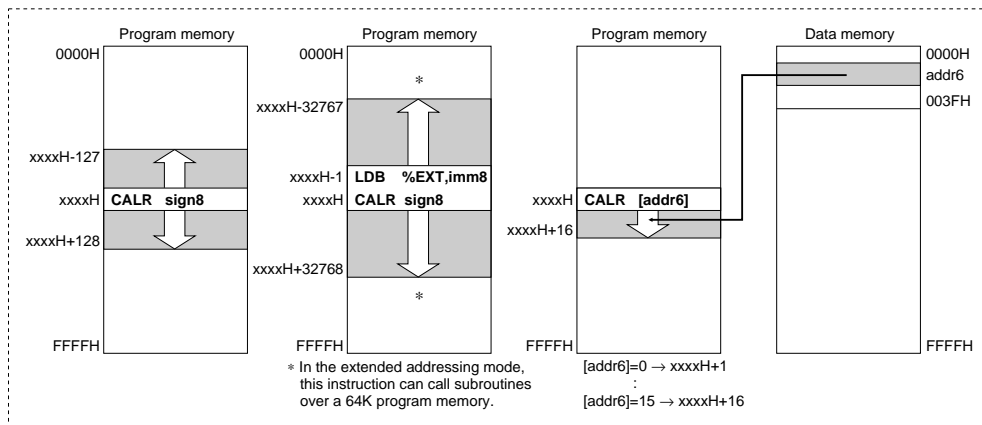
This instruction branches the program sequence with the content of the data memory specified by the [addr6] as an unsigned 4-bit relative address. The operand [addr6] can specify a data memory address within 0000H to 003FH. The range that can be branched is from the next instruction address +0 to +15. Same with the "JR [addr6]", this call instruction can be used as a conditional call according to the flags that are set in the memory specified with [addr6].

Example: When the content of the address 0010H is 4 (0100B).

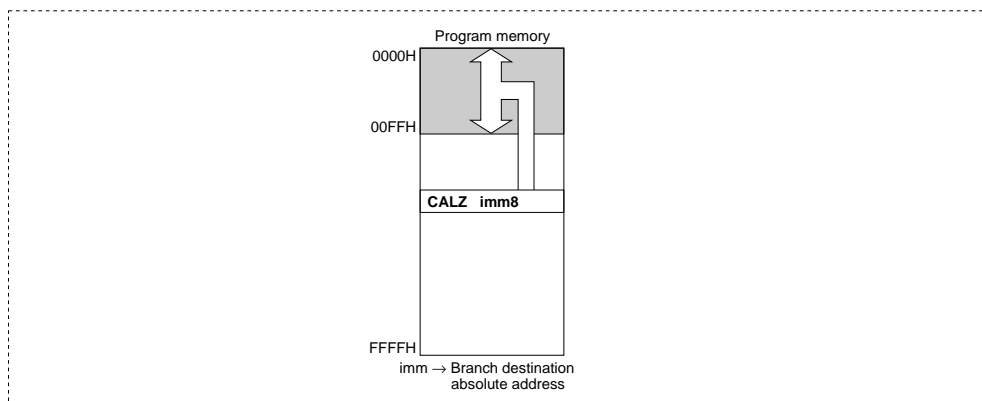
```

SET  [0x0010], 0 ...Sets the bit 0 in the address 0010H to "1" ([0010H] = 5)
CALR [0x0010]    ...Calls the subroutine 6 steps after
    
```

Figure 2.2.3.2 shows the operation of the call instructions and the branch range.



PC relative call instructions



Absolute call instruction

Fig. 2.2.3.2 Operation of call instructions

• **Return instructions (RET, RETS, RETD, RETI)**

A return instruction is used to return from a subroutine called by the call instruction to the routine that called the subroutine. Return operation is done by loading the PC value (address next to the call instruction) that was stored in the stack when the subroutine was called into the PC.

The RET instruction operates only to return the PC value in the stack, and the processing is continued from the address next to the call instruction.

The RETS instruction returns the PC value then adds "1" to the PC. It skips executing an instruction next to the call instruction.

Figure 2.2.3.3 shows return operations from a subroutine.

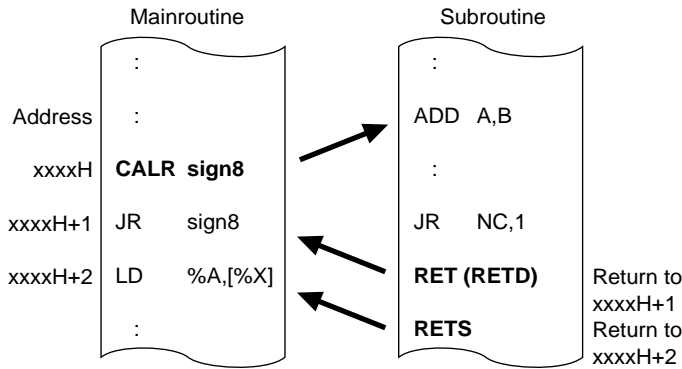


Fig. 2.2.3.3 Return from subroutine

The RETD instruction performs the same operation as the RET instruction, then stores the 8-bit data specified in the operand into the memory specified with the X register. This function is useful to create data tables that will be explained in the next section.

The RETI instruction is for the exclusive use of hardware and software interrupt service routines. When an interrupt is generated, the content of the F register is saved into the stack with the current PC value. The RETI instruction returns them.

• **Software interrupt instruction (INT)**

The software interrupt instruction "INT imm6" specifies a vector address within the addresses from 0111H to 013FH to execute its interrupt service routine. It can also call a hardware interrupt service routine because it can specify an address from 0100H. It performs the same operation with the call instruction, but the F register is also saved into the stack before branching. Consequently, the RETI instruction must be used for returning from interrupt service routines. See Section 3.5, "Interrupts" for details of the interrupt.

**2.2.4 Table look-up instruction**

The RETD instruction, one of the return instructions, has an 8-bit data in the operand, and stores the data in the memory specified with the X register (the low-order 8 bits are stored in [X] and the high-order 8 bits are stored in [X+1]) immediately after returning.

By using the RETD instruction combined with the "JR %BA" or "JR %A" instructions, an 8-bit data table for an LCD segment data conversion or similar can simply be constructed in the code ROM.

**Example:** The following is an example of a table for converting a BCD data (0 to 9) in the A register into an ASCII code (30H to 39H). The conversion result is stored in the addresses 0040H (low-order 4 bits) and 0041H (high-order 4 bits).

```
LD    %A,3      ;Sets data to be converted
CALR  TOASCII   ;Calls converting routine
LDB   %BA,[%X]+ ;Loads result from memory to BA register
:
:
```

```

TOASCII:                ;BCD to ASCII conversion
  LDB  %EXT,0x00        ;Sets address 0040H
  LDB  %XL,0x40
  JR   %A
  RETD 0x30            ;"0"
  RETD 0x31            ;"1"
  RETD 0x32            ;"2"
  RETD 0x33            ;"3"
  RETD 0x34            ;"4"
  RETD 0x35            ;"5"
  RETD 0x36            ;"6"
  RETD 0x37            ;"7"
  RETD 0x38            ;"8"
  RETD 0x39            ;"9"

```

As shown in the example, operation results in the A or BA register can simply be converted into other formats.

## 2.3 Data Memory

### 2.3.1 Configuration of data memory

In addition to the program memory space, the E0C63000 can also access 64K-word ( $\times 4$  bits) data memory. In the individual model of the E0C63 Family, RAM of which size is decided depending on the model and I/O memory are connected to this space.

Figure 2.3.1.1 shows the data memory map of the E0C63000.

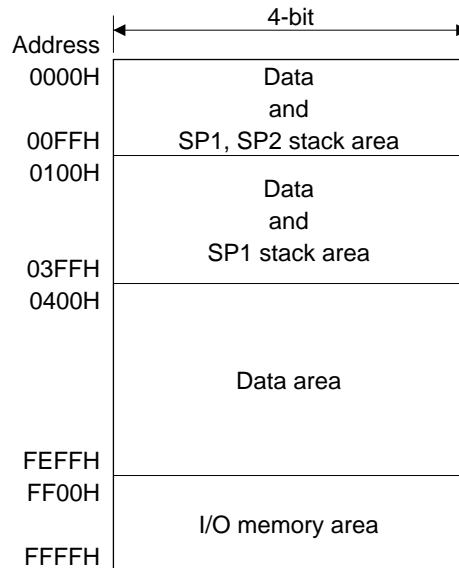


Fig. 2.3.1.1 E0C63000 data memory map

The E0C63000 can access 64K-word space linearly without any of the page management commonly used in current 4-bit microcomputers.

The E0C63000 has a built-in 16-bit data bus for the address stack (SP1), and a RAM that permits 16-bit data accessing can be connected to the addresses 0000H to 03FFH. The 16-bit accessible area is different depending on the individual models. That area permits normal 4-bit accessing. Switching between 4-bit accessing and 16-bit accessing is done according to the instruction by the hardware. A normal 4-bit data stack (SP2) is assigned within the addresses 0000H to 00FFH.

The addresses FF00H to FFFFH are used for an I/O memory area to control the peripheral circuits.

### 2.3.2 Addressing for data memory

For addressing to access the data memory, the index registers X and Y, and stack pointers SP1 and SP2 are used. (The next section will explain the stack pointers.)

Index registers X and Y are both 16-bit registers and cover the entire 64K data memory space. The data memory is accessed by setting an address in the register.

Example:

```
LDB  %EXT, 0x00
LDB  %XL, 0x10      ...Sets 0010H in the X register
LD   A, [%X]       ...Loads the content of the memory address 0010H into the A register
```

The indirect addressing with the X or Y register permits use of the post-increment function and processing for continuous addresses can be done efficiently. This function can be used in the instruction with [%X]+ or [%Y]+ as an operand. [%X]+ indicates that the content of the X register is incremented after end of transfer or operation, therefore the next address can be accessed without the X register re-setting. It is the same in case of the Y register.

Example: To copy the 3-word data from the address specified with the X register to the area specified with the Y register

```
LD [%Y]+, [%X]+
LD [%Y]+, [%X]+
LD [%Y], [%X]
```

In addition, the E0C63000 has also provided instructions in order to efficiently access only the area which is accessed frequently such as the I/O memory and lower addresses.

One of that is the addressing using the EXT register explained in Section 2.1.5.

- **Accessing for addresses 0000H to 00FFH**

For absolute addressing in this area, the EXT register and an indirect instruction with the X register ([%X]) are used. To access this area, first write an 8-bit low-order address (00H to FFH) in the EXT register, then execute an indirect addressing instruction with an operand [%X] (only the instruction that permits the extended addressing). In this case, the content of the X register does not affect the address to be accessed. Also the content of the X register is not changed.

Example:

```
LDB  %EXT, 0x37
LD   %A, [%X]      ...Works as "LD %A, [0x0037]"
```

- **Accessing for addresses FF00H to FFFFH (I/O memory area)**

For absolute addressing in this area, the EXT register and an indirect instruction with the Y register ([%Y]) are used. To access this area, first write an 8-bit low-order address (00H to FFH) in the EXT register, then execute an indirect addressing instruction with an operand [%Y] (only the instruction that permits the extended addressing). In this case, the content of the Y register does not affect the address to be accessed. Also the content of the Y register is not changed.

Example:

```
LDB  %EXT, 0x9C
ADD  [%Y], 5       ...Works as "ADD [0xFF9C], 5"
```

*Note: The extended addressing function using the EXT register is effective only for the instruction following immediately after writing data to the EXT register or setting the E flag to "1". For that instruction, do not use instructions other than the instructions that permit the extended addressing. Operation cannot be guaranteed if used.*

In addition to the above functions, some 6-bit addressing instructions are provided to directly access that area. These instructions have a [addr6] as the operand and can alone directly access the area 0000H to 003FH or FFC0H to FFFFH.

### • Accessing for addresses 0000H to 003FH

Data in this area is used for a relative address by the "JR [addr6]" and "CALR [addr6]" explained in Section 2.2.3. This area is suitable for setting up various flags and counters since the bit operation instructions (CLR, SET, TST) and increment/decrement instructions (INC, DEC) are provided for accessing this area.

### • Accessing for addresses FFC0H to FFFFH (I/O memory area)

The bit operation instructions (CLR, SET, TST) are provided for accessing this area. Therefore, control bits in the I/O memory can be operated simply.

Examples:

```
CLR  [ 0xFFC0 ], 0 ...Clears the D0 bit in the I/O memory address FFC0H to "0"
SET  [ 0xFFD2 ], 3 ...Sets the D3 bit in the I/O memory address FFD2H to "1"
```

### 2.3.3 Stack and stack pointer

The stack is a memory that is accessed in the LIFO (Last In, First Out) format and is allocated to the RAM area of the address 0000H to 03FFH. The stack area can be set from an optional address (toward the lower address) using the stack pointer.

The E0C63000 contains two stack pointers SP1 and SP2.

#### (1) Stack pointer SP1

The SP1 is used for the address data stack, and permits 16-bit data accessing.

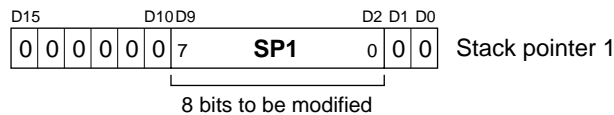


Fig. 2.3.3.1 SP1 configuration

As shown in the figure, the D0, D1 and D10–D15 within the 16 bits are fixed at "0". 8 bits of the D2–D9 can be set by software. Furthermore, the hardware also operates for this 8-bit field. Therefore, addressing by the SP1 is done in 4-word units, and a 16-bit address data can be transferred in one accessing. Since the SP1 performs 16-bit data accessing, this stack area is limited to the 16-bit accessible RAM area even though it is within the addresses 0000H to 03FFH.

This stack is used to evacuate return addresses when the call instructions are executed or the interrupts are generated. It is also used when the 16-bit data in the X or Y register is evacuated using the PUSH instruction. The return address data is written into the stack as shown in Figure 2.3.3.2.

The SP1 is decremented after the data is evacuated and is incremented when a return instruction is executed or after returning data by executing the POP instruction.

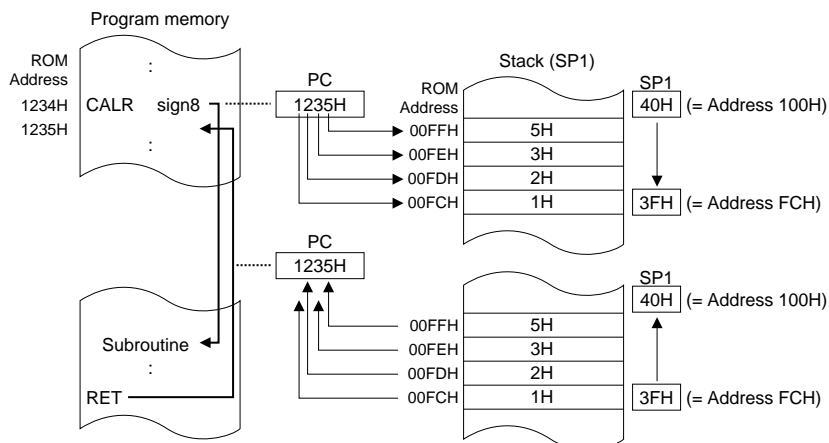


Fig. 2.3.3.2 Address stack operation

The SP1 increment/decrement affects only the 8-bit field shown in Figure 2.3.3.1, and its operation is performed cyclically. In other words, if the SP1 is decremented by the PUSH instruction or other conditions when the SP1 is 00H (indicating the memory address 0000H), the SP1 becomes FFH (indicating the memory address 03FCH). Similarly, if the SP1 is incremented by the POP instruction or other conditions when the SP1 is FFH (indicating the memory address 03FCH), the SP1 becomes 00H (indicating the memory address 0000H).

- **Queue register**

The queue register is provided in order to reduce the process time of the 16-bit data transfer by the SP1. The queue register retains 16-bit data in the RAM indicated with the SP1. It is accessed when the following instructions are executed, not by programs directly.

1. When the call instruction or the PUSH instruction is executed, and when an interrupt is generated  
When the CALR or CALZ instruction is executed, a software interrupt by the INT instruction is generated, and a hardware interrupt is generated, the PC value for returning is written in the memory [SP1-1]. When the "PUSH %X" or "PUSH %Y" instruction is executed, the content of the X register or Y register is written in the memory [SP1-1]. At this time, the same data which is written in the memory [SP1-1] is also written to the queue register.
2. When the return instruction or the POP instruction is executed  
When the RET, RETS, RETD, RETI, "POP %X" or "POP %Y" instructions are executed, the data retained in the queue register is returned to the PC, X register or Y register. Since the SP1 is incremented, the content of the queue register is renewed (it generates a bus cycle to load the content of the memory [SP1+1] to the queue register).
3. When the "LDB %SP1, %BA", "INC SP1" or "DEC SP1" instructions are executed  
When these instructions are executed, the content of the queue register is also renewed (it generates a bus cycle to load the content of the memory [SP1] to the queue register).

*Note: As shown above, the memory content that is indicated by the SP1 is written to the queue register according to the SP1 changes. Therefore, the queue register is not renewed even if the memory [SP1] is directly modified when the SP1 is not changed. Be aware that intended return and POP operations cannot be performed if such an operation is done.*

**(2) Stack pointer SP2**

The SP2 is used for the normal 4-bit data stack.

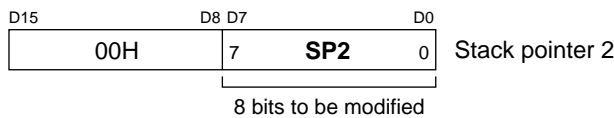


Fig. 2.3.3.3 SP2 configuration

In the case of the SP1, the D8–D15 within the 16 bits are fixed at "0". 8 bits of the D0–D7 can be set by software. Furthermore, the hardware also operates for this 8-bit field. The address range that can be used for the data stack is limited to within 0000H to 00FFH. Data evacuation/return is done in 1-word units.

This stack is used to evacuate the F register data when an interrupt is generated. It is also used when the 4-bit register data (A, B, F) is evacuated using the PUSH instruction. The register data is written into the stack as shown in Figure 2.3.3.4.

The SP2 is decremented after the data is evacuated and is incremented when the data is returned.



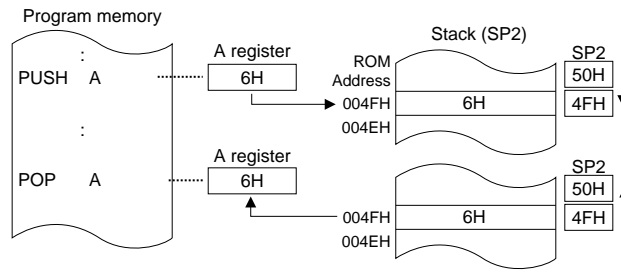


Fig. 2.3.3.4 4-bit stack operation

The SP2 increment/decrement affects only the 8-bit field shown in Figure 2.3.3.3, and its operation is performed cyclically. In other words, if the SP2 is decremented by the PUSH instruction or other conditions when the SP2 is 00H (indicating the memory address 0000H), the SP2 becomes FFH (indicating the memory address 00FFH). Similarly, if the SP2 is incremented by the POP instruction or other conditions when the SP2 is FFH (indicating the memory address 00FFH), the SP2 becomes 00H (indicating the memory address 0000H).

### (3) Notes for using the stack pointer

- The SP1 and SP2 are undefined at an initial reset. Therefore, both the stack pointers must be initialized by software.  
For safety, all the interrupts including NMI are masked until both the SP1 and SP2 are set by software. Furthermore, if either the SP1 or SP2 is re-set, all the interrupts are masked again until the other is re-set. Therefore be sure to set the SP1 and SP2 as a pair.
- The increment/decrement for the SP1 and SP2 is operated cyclically from 0000H to 03FFH (SP1) and from 0000H to 00FFH (SP2) regardless of the memory capacity/allocation set up in each model. Control with the program so that the stacks do not cross over the upper/lower limits of the mounted memory.
- The SP1 must be set in the RAM area that permits 16-bit accessing depending on the model. The SP1 address stack cannot be allocated to other than the 16-bit accessible area even if the address is less than 03FFH.
- The area management for the SP1 stack, SP2 stack and data RAM should be done by the user. Pay attention to these areas so that they do not overlap in the same addresses.

### 2.3.4 Memory mapped I/O

The E0C63 Family contains the E0C63000 as the core CPU and various types of peripheral circuits, such as input/output ports. The E0C63000 has adopted a memory mapped I/O system for controlling the peripheral circuits, and the control bits and the registers for exchanging data are arranged in the data memory area.

The I/O memory for controlling the peripheral circuits is assigned to the area from FF00H to FFFFH, and is distinguished from RAM and others. However, the accessing method is the same as RAM, so indirect addressing can be done using the X or Y register. In addition, since the I/O memory is accessed frequently, the exclusive instructions for this area are also provided. (See Section 2.3.2.)

Refer to the manual for the individual model of the E0C63 Family for the I/O memory and the peripheral circuits.

# CHAPTER 3 CPU OPERATION

This section explains the CPU operations and the operation timings.

## 3.1 Timing Generator and Bus Cycle

The E0C63000 has a built-in timing generator. The timing generator of the E0C63000 generates the two-phase divided signals PK and PL based on the clock (CLK) input externally (\*) to make states. One state is a 1/2 cycle of the CLK and the one bus cycle that becomes the instruction execution unit is composed of four states.

\* The clock that is input to the E0C63000 is generated by an oscillation circuit provided outside of the CPU. The E0C63 Family models have a built-in oscillation circuit.

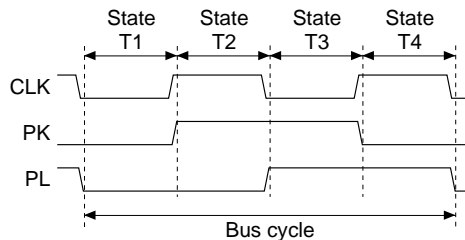


Fig. 3.1.1 State and bus cycle

The number of cycles which is stated in the instruction list indicates the number of bus cycles.

## 3.2 Instruction Fetch and Execution

The E0C63000 executes the instructions indicated with the PC (program counter) one by one. That operation for an instruction is divided into two stages; one is a fetch cycle to read an instruction, and another is an execution cycle to execute the instruction that has been read.

All the E0C63000 instructions are composed of one step (word), and are fetched in one bus cycle. An instruction code that is written in the ROM is read out during the fetch cycle and is analyzed by the instruction decoder. The  $\overline{\text{FETCH}}$  signal goes to a low level during that time. In addition, the PC is incremented at the end of each fetch.

The analyzed instruction is executed from the next bus cycle. The number of execution cycles is shown in the instruction list and it is one, two or three bus cycles depending on the instruction.

The E0C63000 contains two different buses for the program memory and the data memory. Consequently, a fetch cycle for the next instruction can be executed to overlap with the last execution cycle, and it increases the processing speed. In the one-cycle instructions, the next instruction is fetched at the same time an instruction is executed.

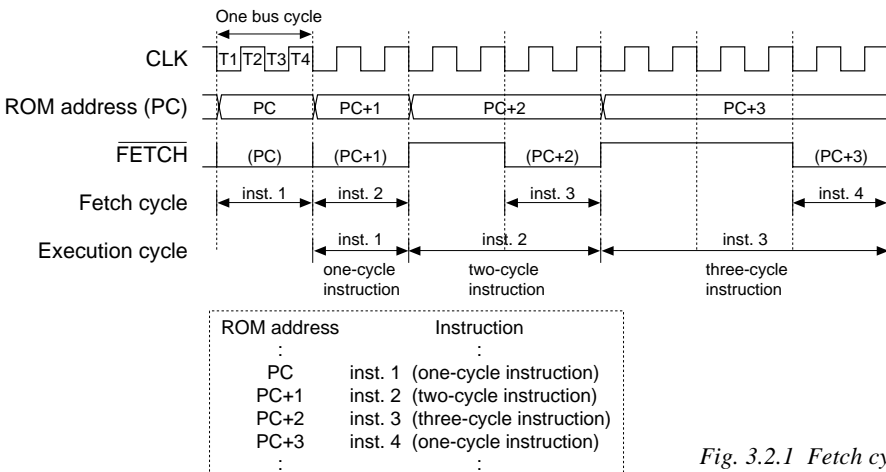


Fig. 3.2.1 Fetch cycle and execution cycle

### 3.3 Data Bus (Data Memory) Control

#### 3.3.1 Data bus status

The E0C63000 outputs the data bus status in each bus cycle externally on the DBS0 and DBS1 signals as a 2-bit status. The peripheral circuits perform the direction control of the bus driver and other controls with these signals. The data bus statuses indicated by the DBS0 and DBS1 are as shown in Table 3.3.1.1.

Table 3.3.1.1 Data bus status

DBS1	DBS0	State
0	0	High impedance
0	1	Interrupt vector read
1	0	Memory write
1	1	Memory read

#### 3.3.2 High-impedance control

The data bus goes to a high-impedance during an execution cycle (\*) that accesses only the internal registers in the CPU. During the bus cycle period, both the read signal  $\overline{RD}$  and write signal  $\overline{WR}$  are fixed at a high level and a dummy address is output on the address bus.

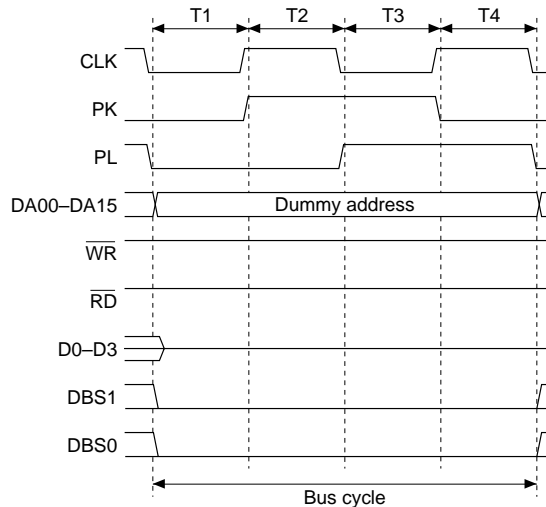


Fig. 3.3.2.1 Bus cycle during accessing internal register

- \* Data is output on the data bus only when the stack pointer SP1 is accessed because a data transfer is performed between the queue register and the data memory. In this case, the data bus status becomes a memory write or a memory read depending on the instruction that accesses the SP1.

### 3.3.3 Interrupt vector read

When an interrupt is generated, the CPU reads the interrupt vector output to the data bus by the peripheral circuit that has generated the interrupt. The interrupt vector read status indicates this bus cycle. The peripheral circuit outputs the interrupt vector to the data bus during this status, and the CPU reads the data between the T2 and T3 states. At this time, the CPU outputs the  $\overline{RDIV}$  signal (for exclusive use of the interrupt vector read) as a read signal, not the  $\overline{RD}$  signal that is used for normal data memory read. The address bus outputs a dummy address during this bus cycle. See Section 3.5 for the operation when an interrupt is generated.

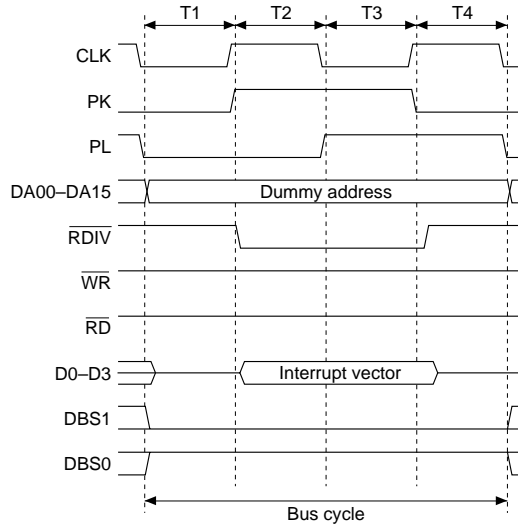


Fig. 3.3.3.1 Bus cycle during reading interrupt vector

### 3.3.4 Memory write

In an execution cycle that writes data to the data memory, the writing data is output to the data bus between the T2 and T4 states and the write signal  $\overline{WR}$  is output in the T3 state. The address bus outputs the target address during this bus cycle.

The E0C63000 contains a 4-bit data bus (D0-D3) and a 16-bit data bus (M00-M15) for an address stacking. The CPU switches the data bus according to the instruction. The  $\overline{BS16}$  signal is provided for this switching.

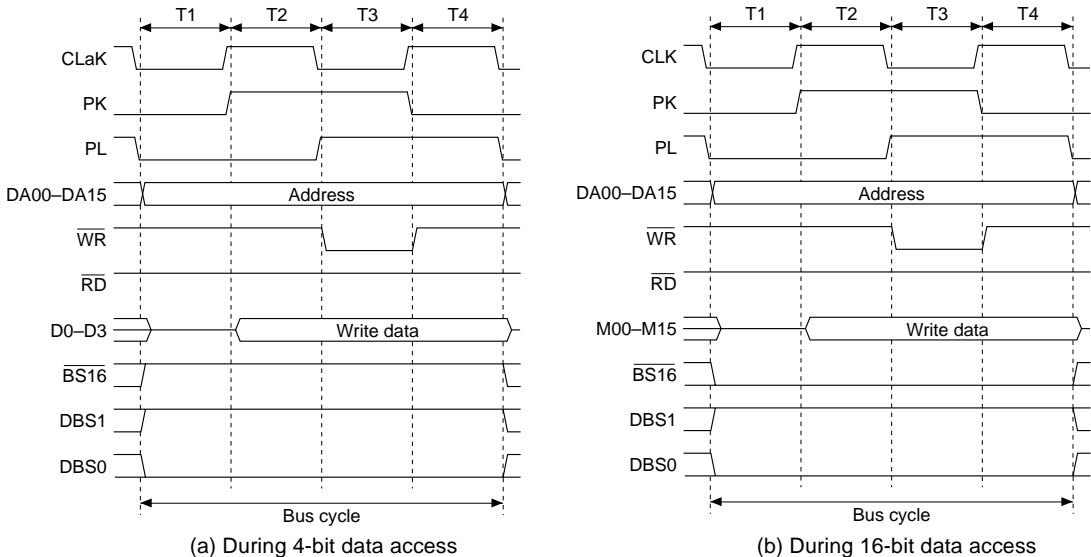


Fig. 3.3.4.1 Bus cycle during memory write

### 3.3.5 Memory read

In an execution cycle that reads data from the data memory, the read signal  $\overline{RD}$  is output between the T2 and T3 states and data is read from the data bus. The address bus outputs the target address during this bus cycle.

The 4-bit/16-bit access is the same as the memory write.

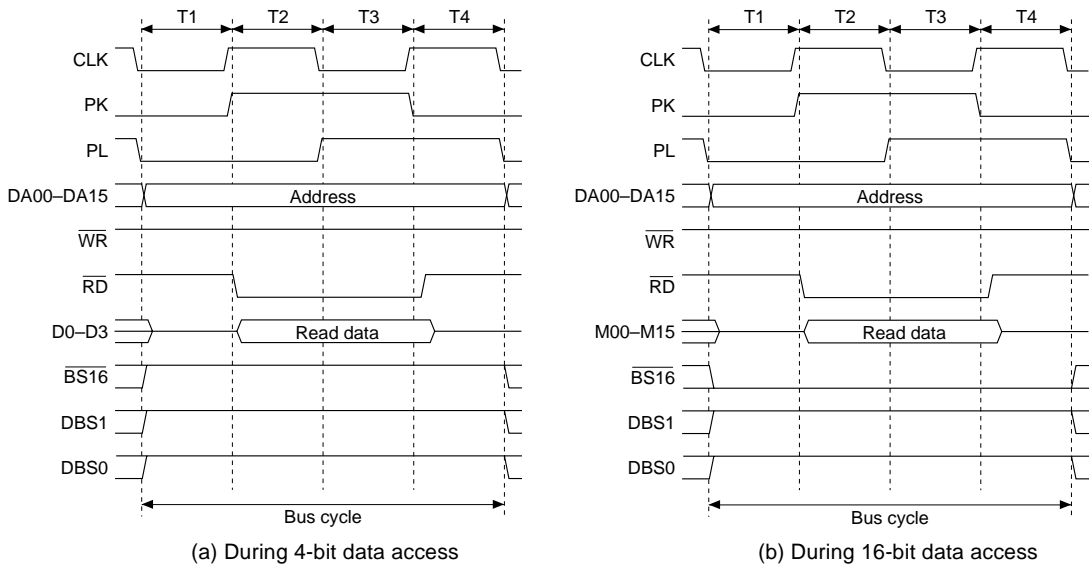


Fig. 3.3.5.1 Bus cycle during memory read

## 3.4 Initial Reset

The E0C63000 has a reset ( $\overline{SR}$ ) terminal in order to start the program after initializing the circuit when the power is turned on or other situations. The following explains the operation at an initial reset and the initial setting of the internal registers.

### 3.4.1 Initial reset sequence

The E0C63000 enters into an initial reset status immediately after setting the  $\overline{SR}$  terminal to a low level, and the internal circuits are initialized. During an initial reset, the data bus goes to a high-impedance and the  $\overline{RD}$  and  $\overline{WR}$  signals go to a high level.

When the  $\overline{SR}$  terminal goes to a high level, the initial reset is released and the program starts executing from address 0110H. The release of an initial reset (the  $\overline{SR}$  terminal goes a high level) is accepted at the rising edge of the CPU operation clock (CLK), and the first bus cycle ( fetching the instruction of the address 0110H) starts from 1 clock after.

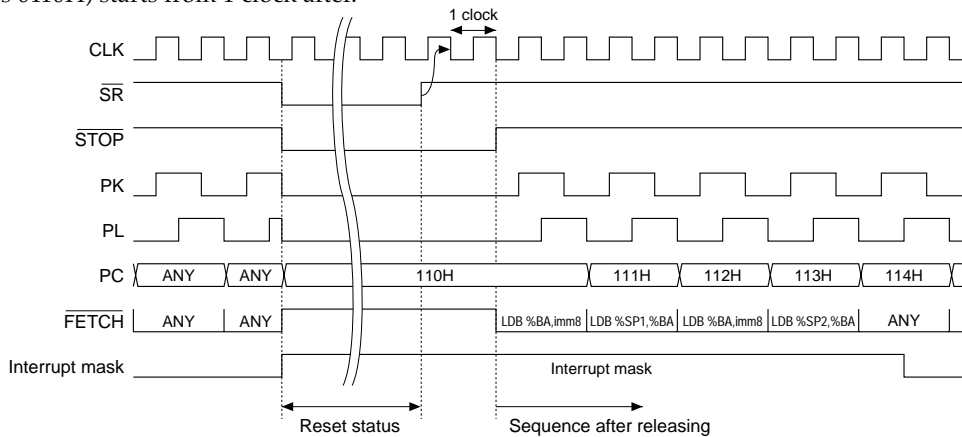


Fig. 3.4.1.1 Initial reset status and sequence after releasing

After an initial reset, all the interrupts including NMI are masked until both the stack pointers SP1 and SP2 are set by software.

**3.4.2 Initial setting of internal registers**

An initial reset initializes the internal registers in the CPU as shown in Table 3.4.2.1.

*Table 3.4.2.1 Initial setting of internal registers*

Name	Symbol	Number of bits	Setting value
Data register A	A	4	Undefined
Data register B	B	4	Undefined
Extension register EXT	EXT	8	Undefined
Index register X	X	16	Undefined
Index register Y	Y	16	Undefined
Program counter	PC	16	0110H
Stack pointer SP1	SP1	8	Undefined
Stack pointer SP2	SP2	8	Undefined
Zero flag	Z	1	Undefined
Carry flag	C	1	Undefined
Interrupt flag	I	1	0
Extension flag	E	1	0
Queue register	Q	16	Undefined

The registers and flags which are not initialized at an initial reset should be initialized in the program if necessary.

Be sure to set both the stack pointers SP1 and SP2. All the interrupts cannot be accepted if they are not set as a pair.

**3.5 Interrupts**

Interrupt is a function to process factors, that generate asynchronously with program execution, such as a key entry and an end of a peripheral circuit operation. When the CPU accepts an interrupt request that is sent by the hardware, the CPU stops executing the current sequence of the program and shifts into the interrupt processing. When all the interrupt processing has finished, the interrupted program is resumed.

The E0C63000 has the hardware interrupt function for the peripheral circuits including an NMI (non-maskable interrupt) and the hardware interrupt function. The hardware interrupts excluding the NMI can be set to the DI (disable interrupts) status by setting the I (interrupt) flag.

I flag = "1": EI (enable interrupts) status ...The CPU accepts interrupt requests from the peripheral circuits.

I flag = "0": DI (disable interrupts) status ...The CPU does not accept interrupt requests from the peripheral circuits. (excluding NMI and software interrupts)

The I flag is set to "0" at an initial reset. Furthermore, all the interrupts including NMI are masked and cannot be accepted regardless of the I flag setting until both the stack pointers SP1 and SP2 are set in the program after an initial reset.

**3.5.1 Interrupt vectors**

Interrupt vectors are provided to execute a interrupt service routine corresponding to the interrupt generated.

The interrupt vectors are assigned to the following addresses in the ROM.

- NMI interrupt vector: 0100H
- Hardware interrupt vectors: 0101H to 010FH
- Software interrupt vectors: 0111H to 013FH

Each of the addresses listed above corresponds to an interrupt factor individually. A branch (jump) instruction to the interrupt service routine should be written to these addresses.

Up to 15 hardware interrupt vectors are available, however, the number of vectors is different depending on the E0C63 Family models. The addresses, that are not assigned to the hardware interrupt vector within the addresses 0101H to 010FH, can be used as software interrupt vectors. In addition, since the hardware interrupt service routines can be executed using the software interrupt, up to 63 software interrupts can be used (excluding the address 0110H because it is the program start address).

### 3.5.2 Interrupt sequence

#### • Hardware interrupts

Hardware interrupts including NMI are generated by the peripheral circuits. The peripheral circuit that contains the interrupt function outputs an interrupt request to the CPU when the interrupt factor is generated. The  $\overline{\text{NMI}}$  terminal for NMI or  $\overline{\text{IRQ}}$  terminal for other interrupts goes low. Sampling the  $\overline{\text{NMI}}$  signal is done at the falling edge by the CPU. Sampling the  $\overline{\text{IRQ}}$  signal is done at the rising edge of the T3 state in the bus cycle. The CPU executes the following process after accepting an interrupt request.

- Bus cycle 0 Sampling the interrupt request.
- Bus cycle 1 The last execution cycle of the instruction under execution becomes a dummy fetch cycle. This cycle turns the interrupt acknowledge signal low (both  $\overline{\text{NACK}}$  and  $\overline{\text{IACK}}$  for NMI,  $\overline{\text{IACK}}$  only for a normal interrupt), which indicates that the interrupt has been accepted.
- Bus cycle 2 Saves the F register into the stack indicated by the SP2, then resets the I flag to "0" to prohibit following interrupts (excluding NMI).
- Bus cycle 3 Sets the data bus status DBS1/DBS0 to "01B". Then, turns the vector read signal  $\overline{\text{RDIV}}$  low and reads the interrupt vector (4 bits) output from the peripheral circuit to the data bus.  
When NMI is generated, this cycle becomes a dummy cycle because the interrupt vector is fixed at 0100H.  
The  $\overline{\text{NACK}}$  and/or  $\overline{\text{IACK}}$  are returned to high at the end of this cycle.
- Bus cycle 4 Fetches the instruction in the interrupt vector (data that is read in Bus cycle 3 becomes the low-order 4 bits of the vector) and saves the content of the PC (address immediately after the instruction that is executed in Bus cycle 0 or branch destination address when it is a branch instruction) to the stack indicated by the SP1.
- Bus cycle 5 Executes the instruction fetched in Bus cycle 4. (If it is 1-cycle instruction, the next instruction is fetched at the same time.)

#### • Exceptional acceptance of interrupt

For all the interrupts including NMI that are generated during fetching the following instructions are accepted after the next instruction is fetched (it is executed) even in the EI (enable interrupts) status.

##### 1. Instructions that set the E flag

```
LDB %EXT,imm8      LDB %EXT,%BA
```

##### 2. Instructions that write data in the F (flag) register

```
LD  %F,%A      LD  %F,imm4      AND %F,imm4      OR  %F,imm4
XOR %F,imm4    POP %F          RETI
```

These instructions set the E flag or may set it. Therefore, if an extended addressing instruction follows them, it is executed previous to the interrupt processing.

Further, these instructions may modify the content of the I flag. If these instructions set the I flag (EI status), the interrupt processing is done after executing the next instruction. If these instructions reset the I flag (DI status), interrupts generated after the instruction fetch cycle are masked.

3. Instructions that set the stack pointer

```
LDB %SP1, %BA      LDB %SP2, %BA
```

These two instructions are also accepted after fetching the next instruction. However, these instructions must be executed as a pair. When one of them is fetched at first, all the interrupts including NMI are masked (interrupts cannot be accepted). Then, when the other instruction is fetched, that mask is released and interrupts can be accepted after the next instruction is fetched.

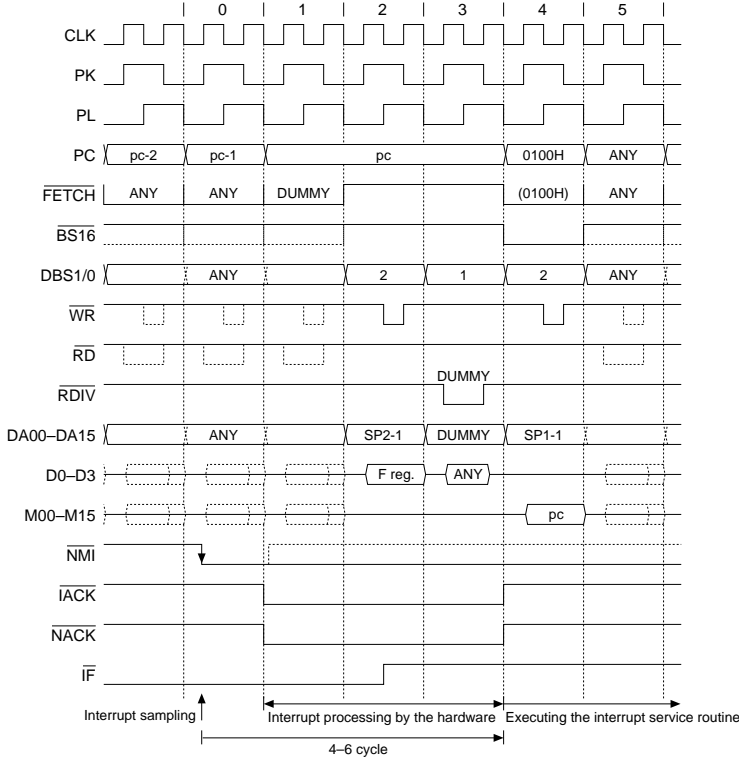
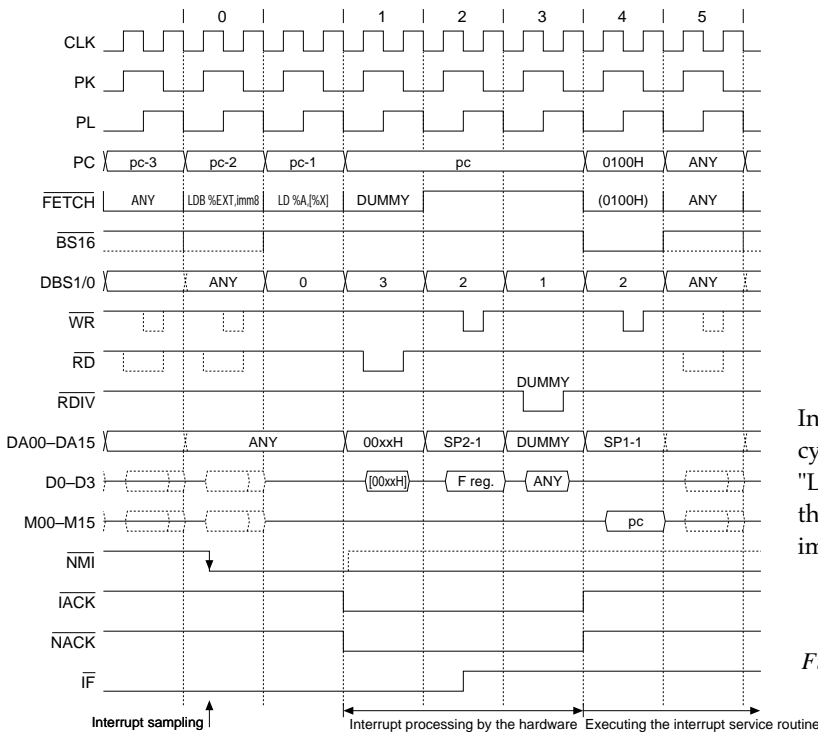


Fig. 3.5.2.1 NMI sequence (normal acceptance)



In this chart, the dummy fetch cycle starts after fetching the "LD %A, [%X]" instruction that follows the "LDB %EXT, imm8" instruction.

Fig. 3.5.2.2 NMI sequence (interrupt acceptance after 1 instruction)



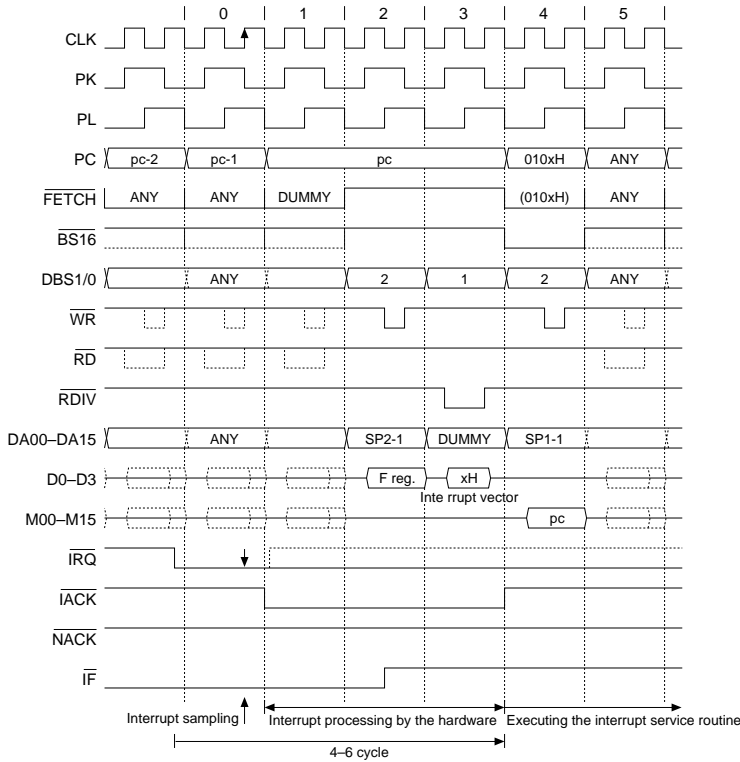
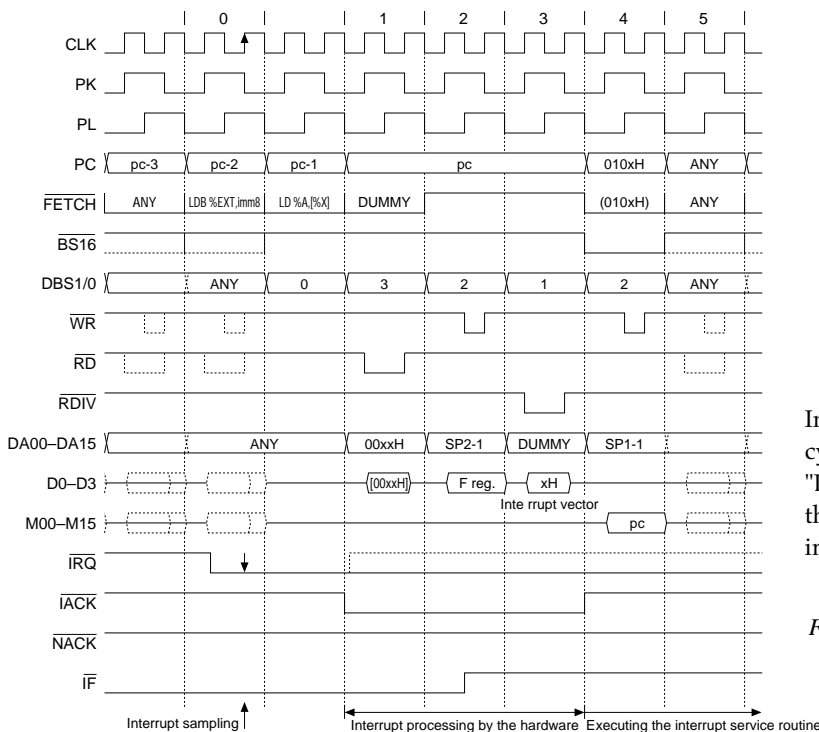


Fig. 3.5.2.3 Hardware interrupt (IRQ) sequence (normal acceptance)



In this chart, the dummy fetch cycle starts after fetching the "LD %A, [%X]" instruction that follows the "LDB %EXT, imm8" instruction.

Fig. 3.5.2.4 Hardware interrupt (IRQ) sequence (interrupt acceptance after 1 instruction)

• **Software interrupts**

The software interrupts are generated by the INT instruction. Time of the interrupt generation is determined by the software, so the I flag setting does not affect the interrupt. That processing is the same as the subroutine that evacuates the F register into the stack.

This interrupt does not change the interrupt control signals between the CPU and the peripheral circuits, or the I flag either. An address that is specified with the operand of the INT instruction is used as it is as the interrupt vector.

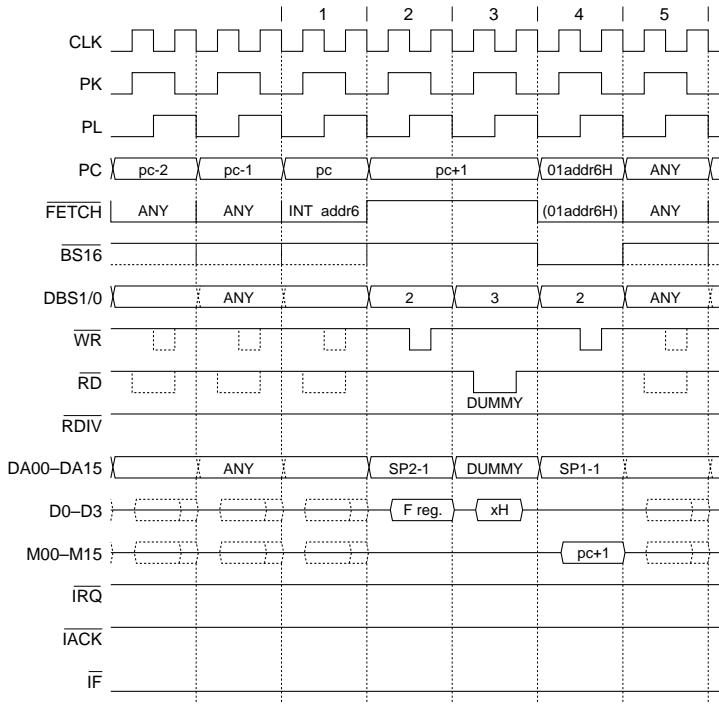


Fig. 3.5.2.5 Software interrupt sequence

**3.5.3 Notes for interrupt processing**

- (1) After an initial reset, all the interrupts including NMI are masked and cannot be accepted regardless of the I flag setting until both the stack pointers SP1 and SP2 are set in the program. Be sure to set the SP1 and SP2 in the initialize routine.  
Further, when re-setting the stack pointer, the SP1 and SP2 must be set as a pair. When one of them is set, all the interrupts including NMI are masked and interrupts cannot be accepted until the other one is set.
- (2) The interrupt processing is the same as a subroutine call that branches to the interrupt vector address. At that time, the F register is evacuated into the stack. Therefore, the interrupt service routine should be made as a subroutine and the RETI instruction that returns the F register must be used for return.
- (3) If an interrupt (including NMI) is generated while fetching an instruction, that sets the E flag or writes data to the F (flag) register, the interrupt is accepted after fetching (and executing) the next instruction. Therefore, the extended addressing with the EXT register is processed before executing the interrupt processing. However, if the stack data in the memory is directly changed in the interrupt service routine, the F register in which the E flag is set may return. In this case, the instruction immediately after returning by the RETI instruction is executed in the extended addressing mode by the E flag that is set to "1". Pay attention to the F register setting except when describing such a processing consciously.

## 3.6 Standby Status

The E0C63000 has a function that stops the CPU operation and it can greatly reduce power consumption. This function should be used to stop the CPU when there is no processing to be executed in the CPU, example while the application program waits an interrupt. This is a standby status where the CPU has been stopped to shift it to low power consumption.

This status is available in two types, a HALT status and a SLEEP status.

### 3.6.1 HALT status

The HALT status is the status in which only the CPU stops and shifting to it can be done using the HALT instruction. The HALT status is released by a hardware interrupt including NMI, and the program sequence returns to the step immediately after the HALT instruction by the RETI instruction in the interrupt service routine. The peripheral circuits including the oscillation circuit and timer operate all through the HALT status. Moreover during HALT status, the contents of the registers in the CPU that have been set before shifting are maintained.

Figure 3.6.1.1 shows the sequence of shifting to the HALT status and restarting.

In the HALT status the Th1 and Th2 states are continuously inserted. During this period, interrupt sampling is done at the falling edge of the Th2 state and the generation of an interrupt factor causes it to shift to the interrupt processing.

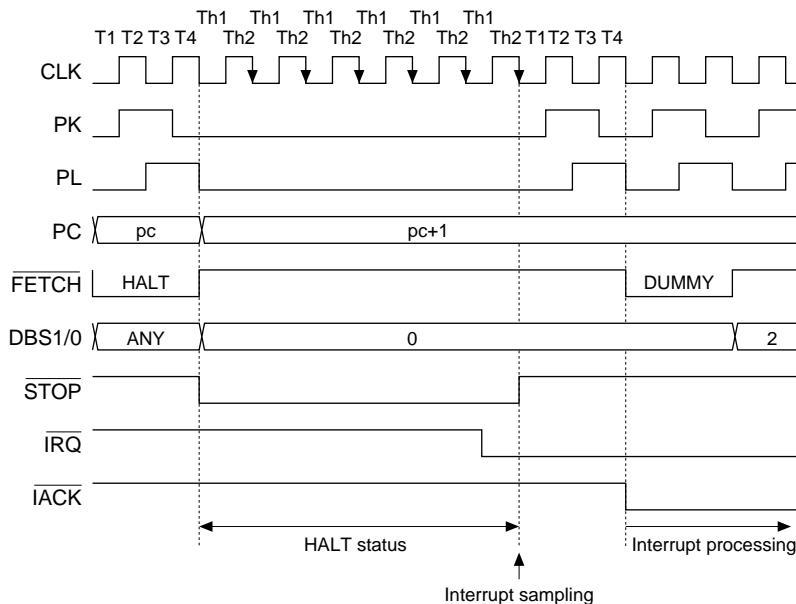


Fig. 3.6.1.1 Sequence of shifting to HALT status and restarting

### 3.6.2 SLEEP status

The SLEEP status is the status in which the CPU and the peripheral circuits within the MCU stop operating and shifting it can be done using the SLP instruction.

The SLEEP status is released by a reset or a specific interrupt (it differs depending on the model). When the SLEEP status is released by a reset, the program restarts from the program start address (0110H).

When it is released by an interrupt, the program sequence returns to the step immediately after the SLP instruction by the RETI instruction in the interrupt service routine.

Power consumption in the SLEEP status can be greatly reduced in comparison with the the HALT status, because such peripheral circuits as the oscillation circuit are also stopped. However, since stabilization time is needed for the oscillation circuit when restarting, it is effective when used for extended standby where instantaneous restarting is not necessary.

During SLEEP status, as in the HALT status, the contents of the registers in the CPU that have been set before shifting are maintained if rated voltage is supplied.

Figure 3.6.2.1 shows the sequence of shifting to the SLEEP status and restarting.

When an interrupt that releases the SLEEP status is generated, the oscillation circuit begins to oscillate. When the oscillation starts, the CLK input to the CPU is masked by the peripheral circuit and the input to the CPU begins after stabilization waiting time (several 10 msec–several msec) has elapsed. The CPU samples the interrupt at the falling edge of the initially input CLK and starts the interrupt processing.

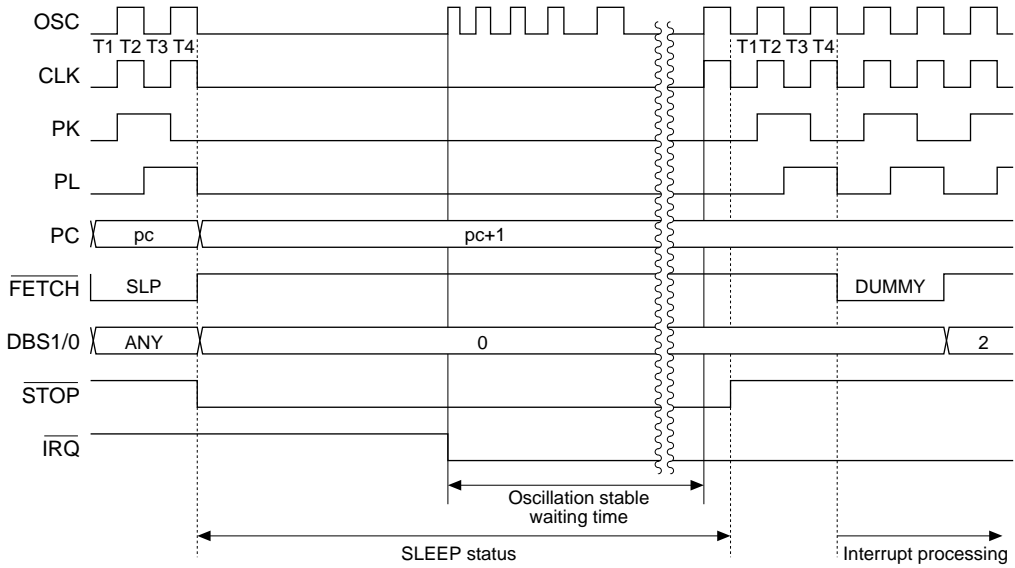


Fig. 3.6.2.1 Sequence of the shift to SLEEP status and restarting

# CHAPTER 4 INSTRUCTION SET

The E0C63000 offers high machine cycle efficiency and a high speed instruction set. It has 47 basic instructions (412 instructions in all) that are designed as an instruction system permitting relocatable programming.

This chapter explains about the addressing modes for memory management and about the details of each instruction.

## 4.1 Addressing Mode

The E0C63000 has the following 8 types of addressing modes and the address specifications corresponding to the various statuses are done concisely and accurately.

### • Types of addressing modes

Basic addressing modes (5 types)

- 1) Immediate data addressing
- 2) Register direct addressing
- 3) Register indirect addressing
- 4) 6-bit absolute addressing
- 5) Signed 8-bit PC relative addressing

Extended addressing modes (3 types)

- 1) 16-bit immediate data addressing
- 2) 8-bit absolute addressing
- 3) Signed 16-bit PC relative addressing

### 4.1.1 Basic addressing modes

The basic addressing mode is an addressing function independent of the instruction.

#### • Immediate data addressing

The immediate data addressing is the addressing mode in which the immediate data is used for operations and is used as transfer data. Values that are specified in the operand are directly used as data or addresses. In the instruction list, the following symbols are used to write immediate data.

Table 4.1.1.1 Symbol and size of immediate data

Symbol	Use	Size	Specifiable range
imm2	Specifying a bit No. in 4-bit data	2 bits	0–3
imm4	4-bit general-purpose data	4 bits	0–15
imm6	Specifying a software interrupt vector	6 bits	0–63
imm8	8-bit general-purpose data	8 bits	0–255
sign8	Signed 8-bit general-purpose data	8 bits	-128–127
n4	Specifying a radix	4 bits	1–16

Examples:

```
CLR [addr6], imm2 ...Clears a bit specified with imm2 within a 4-bit data in an address [addr6]
LD %A, imm4 ...Loads a 4-bit data imm4 into the A register
INT imm6 ...A software interrupt of which the vector address is specified with imm6
LDB %BA, imm8 ...Loads an 8-bit data imm8 into the BA register
CALZ imm8 ...Calls a subroutine that starts from an address imm8
        (Address specifiable range is 0000H to 00FFH.)
ADD %X, sign8 ...Adds a signed 8-bit data sign8 to the X register
ADC %B, %A, n4 ...Adds data in the A register to the B register with a radix n4 specification
```

• **Register direct addressing**

The register direct addressing is the addressing mode when specifying a register for the source and/ or destination. Register names should be written with % in front. Instructions in which the operand has the following register name operate in this addressing mode.

- 4-bit registers:    %A, %B, %F
- 8-bit registers:    %BA, %XH, %XL, %EXT, %SP1, %SP2
- 16-bit registers:  %X, %Y

Examples:

- ADD  %A, %B            ...Adds the data in the B register to the A register
- LDB  %BA, %XL        ...Loads the data in the XL register into the BA register
- DEC  %SP1            ...Decrements the stack pointer SP1
- JR   %A                ...Jumps using the content of the A register as a relative address
- JP   %Y                ...Jumps to the address indicated with the Y register

• **Register indirect addressing**

The register indirect addressing is the addressing mode for accessing the data memory and it indirectly specifies the data memory address with the index register X or Y. To write the instructions, place % in front of the index register name and enclose them with [ ].

- Indirect addressing with the X register:    Instructions which have [%X] or [%X]+ as the operand
- Indirect addressing with the Y register:    Instructions which have [%Y] or [%Y]+ as the operand

The content of the X register or Y register regarded as an address, and operations and transfers are performed for the data stored in the address or the address.

"+" in the [%X]+ and [%Y]+ indicates a post-increment function. Instructions that have these operands increment the content of the X register or Y register after executing the transfer or operation. This function is useful to access a continuous addresses in the data memory.

Examples:

- SUB  %A, [%X]         ...Subtracts the content of a memory specified with the X register from the A register
- LD   [%X]+, [%Y]+    ...Transfers the content of a memory specified with the Y register to a memory specified with the X register. Then increments the contents of the X register and Y register

• **6-bit absolute addressing**

The 6-bit absolute addressing is the addressing mode for accessing within the 6-bit address range from 0000H or FFC0H. Instructions that have [addr6] as the operand operate in this addressing mode. The address range that can be specified with the addr6 is 0000H to 003FH or FFC0H to FFFFH.

*(1) Instructions that access from 0000H to 003FH*

For this area, the following instructions, which are used in this area as counters and flags, are provided. An address within 0000H to 003FH is specified with the addr6.

- INC  [addr6]            ...Increments the content of a memory specified with the addr6
- DEC  [addr6]            ...Decrements the content of a memory specified with the addr6
- CLR  [addr6], imm2    ...Clears a bit specified with the imm2 in a memory specified with the addr6
- SET  [addr6], imm2    ...Sets a bit specified with the imm2 in a memory specified with the addr6
- TST  [addr6], imm2    ...Tests a bit specified with the imm2 in a memory specified with the addr6

In addition, the following branch instructions, which permit a conditional branch according to the contents of this area, are provided.

- JR   [addr6]            ...PC relative jump instruction that uses the content of a memory specified with addr6 as a relative address
- CALR [addr6]            ...PC relative call instruction that uses the content of a memory specified with addr6 as a relative address

These instructions perform a PC relative branch using the content (4 bits) of a memory specified with the [addr6] as a relative address. The branch destination address is [the address next to the branch instruction] + [the contents (0 to 15) of the memory specified with the addr6].

**(2) Instructions that access from FFC0H to FFFFH**

This area is reserved for the I/O memory in the E0C63 Family and the following instructions are provided to operate the control bits of the peripheral circuits.

An address within FFC0H to FFFFH is specified with the addr6. However the addr6 is handled as 0 to 3FH in the machine codes.

```
CLR  [addr6], imm2 ...Clears a bit specified with the imm2 in a memory specified with the addr6
SET  [addr6], imm2 ...Sets a bit specified with the imm2 in a memory specified with the addr6
TST  [addr6], imm2 ...Tests a bit specified with the imm2 in a memory specified with the addr6
```

Write only or read only control bits may have been assigned depending on the peripheral circuit. Pay attention when using the above-mentioned instructions for such bits or addresses containing such bits.

• **Signed 8-bit PC relative addressing**

The signed 8-bit PC relative addressing is the addressing mode used for the branch instructions. The signed 8-bit relative address (-128 to 127) that is specified in the operand is added to the address next to the branch instruction to branch to that address.

The following instructions operate in this addressing mode.

```
Jump instructions:  JR    sign8
                   JRC   sign8
                   JRNC  sign8
                   JRZ   sign8
                   JRNZ  sign8
Call instruction:  CALR  sign8
```

### 4.1.2 Extended addressing mode

In the E0C63000, when data is written to the EXT register (the E flag is set) and a specific instruction follows, the data specified by that instruction is extended with the EXT register data (see Section 2.1.5). When the E flag is set, instructions are extended in an addressing mode different from the mode that is specified in each instruction. This is the extended addressing mode that will be explained below.

However, instructions that can operate in the extended addressing mode are limited to those indicated in the instruction list, so check it when programming.

Further the extended addressing mode is effective only for the instruction following immediately after writing data to the EXT register and setting the E flag to "1" (the E flag is reset to "0" by executing that instruction). When using an instruction in the extended addressing mode, write data to be extended to the EXT register or set the E flag (when the E register has already been set).

• **16-bit immediate data addressing**

The addressing mode of the following instructions, which have an 8-bit immediate data as the operand, change to the 16-bit immediate data addressing when the E flag is set to "1". Consequently, it is possible to transfer and operate a 16-bit immediate data to the X or Y register.

**Instructions that operate in the 16-bit immediate data addressing mode with the E flag**

```
LDB  %XL, imm8      LDB  %Y, imm8
ADD  %X, sign8      ADD  %Y, sign8
CMP  %X, imm8      CMP  %X, imm8
```

The data is extended into 16 bits in which the E register data is the high-order 8 bits and the immediate data specified with the above instruction is the low-order 8 bit.

Examples:

```
LDB  %EXT, 0x15
LDB  %XL, 0x7D      ...Works as "LD %X, 0157D"
LDB  %EXT, 0xB8
ADD  %X, 0x4F      ...Works as "ADD %X, 0xB84F"

LDB  %EXT, 0xE6
CMP  %X, 0xA2      ...Works as "CMP %X, 0x19A2"
                        * 19H = FFH - [EXT] (E6H)
```

Above examples use the X register, but they work the same even when the Y register is used.

*Note: The CMP instruction performs a subtraction with a complement, therefore it is necessary to set the complement (1's complement) of the high-order 8-bit data in the EXT register.  
EXT register ← [FFH - High-order 8-bit data]*

• **8-bit absolute addressing**

The 8-bit absolute addressing is the addressing mode for accessing within the 8-bit address range from 0000H or FF00H. To enter this mode, write the low-order 8 bits (00H to FFH) of the address to the EXT register, then execute an indirect addressing instruction which has [%X] or [%Y] as the source operand or the destination operand. When [%X] is used, the memory from 0000H to 00FFH can be accessed, and when [%Y] is used, FF00H to FFFFH can be accessed.

*Instructions that operate in the 8-bit absolute addressing mode with the E flag*

Instruction	Operand
LD	%r, [%X] %r, [%Y] [%X], %r [%Y], %r [%X], imm4 [%Y], imm4
EX	%r, [%X] %r, [%Y]
ADD	%r, [%X] %r, [%Y] [%X], %r [%Y], %r [%X], imm4 [%Y], imm4
ADC	%r, [%X] %r, [%Y] [%X], %r [%Y], %r [%X], imm4 [%Y], imm4 %B, [%X], n4 %B, [%Y], n4 [%X], %B, n4 [%Y], %B, n4 [%X], 0, n4 [%Y], 0, n4
SUB	%r, [%X] %r, [%Y] [%X], %r [%Y], %r [%X], imm4 [%Y], imm4
SBC	%r, [%X] %r, [%Y] [%X], %r [%Y], %r [%X], imm4 [%Y], imm4 %B, [%X], n4 %B, [%Y], n4 [%X], %B, n4 [%Y], %B, n4 [%X], 0, n4 [%Y], 0, n4
INC	[%X], n4 [%Y], n4
DEC	[%X], n4 [%Y], n4
CMP	%r, [%X] %r, [%Y] [%X], %r [%Y], %r [%X], imm4 [%Y], imm4
AND	%r, [%X] %r, [%Y] [%X], %r [%Y], %r [%X], imm4 [%Y], imm4
OR	%r, [%X] %r, [%Y] [%X], %r [%Y], %r [%X], imm4 [%Y], imm4
XOR	%r, [%X] %r, [%Y] [%X], %r [%Y], %r [%X], imm4 [%Y], imm4
BIT	%r, [%X] %r, [%Y] [%X], %r [%Y], %r [%X], imm4 [%Y], imm4
SLL	[%X] [%Y]
SRL	[%X] [%Y]
RL	[%X] [%Y]
RR	[%X] [%Y]

\* "r" indicates the A or B register. Instructions with an operand other than above or the post-increment function do not have the extended addressing function.

Examples:

```
LDB  %EXT, 0x37
LD   %A, [%X]      ...Works as "LD %A, [0x0037]"

LDB  %EXT, 0x9C
ADD  [%Y], 5       ...Works as "ADD [0xFF9C]"
```



### • Signed 16-bit PC relative addressing

The addressing mode of the following branch instructions, which have an 8-bit relative address as the operand, change to the signed 16-bit PC relative addressing with the E flag set to "1". Consequently, it is possible to extend the branch range to the next address -32768 to +32767. (In this mode these instructions can branch the entire 64K program memory.)

*Instructions that operate in the signed 16-bit PC relative addressing mode with the E flag*

JR sign8 JRC sign8 JRNC sign8 JRZ sign8 JRNZ sign8  
CALR sign8

Examples:

LDB %EXT, 0x64  
JR 0x29 ...Works as "JR 0x6429"  
  
LDB %EXT, 0x3A  
JR\* 0x88 ...Works as "JR\* 0x3A88" (\* = C, NC, Z, or NZ)  
  
LDB %EXT, 0xF8  
CALR 0x62 ...Works as "CALR 0xF862"

## 4.2 Instruction List

### 4.2.1 Function classification

Table 4.2.1.1 lists the function classifications of the instructions.

Table 4.2.1.1 Instruction function classifications

Function classification	Mnemonic	Operation	Function classification	Mnemonic	Operation
Arithmetic	ADD	Addition	Rotate / shift	RL	Rotate to left with carry
	ADC	Addition with carry		RR	Rotate to right with carry
	SUB	Subtraction		SLL	Logical shift to left
	SBC	Subtraction with carry		SRL	Logical shift to right
	CMP	Comparison	Stack control	PUSH	Push
	INC	Increment (adds 1)		POP	Pop
	DEC	Decrement (subtracts 1)		Branch	JR
Logic	AND	Logical product	JP		Indirect jump
	OR	Logical sum	CALZ		Absolute call
	XOR	Exclusive OR	CALR		Rrelative call
	BIT	Bit test	RET		Return
	CLR	Bit clear	RETS		Return and skip
	SET	Bit set	RETD		Return and data set
	TST	Bit test	RETI	Interrupt return	
Transfer	LD	Load (4-bit data)	INT	Software interrupt	
	LDB	Load (8-bit data)	System control	NOP	No operation
	EX	Exchange (4-bit data)		HALT	Shift to HALT status
				SLP	Shift to SLEEP status

### 4.2.2 Symbol meanings

The following indicates the meanings of the symbols used in the instruction list.

#### Register names

- A ..... Data register A (4 bits)
- B ..... Data register B (4 bits)
- BA ..... BA register pair (8 bits, the B register is the high-order 4 bits)
- X ..... Index register X (16 bits)
- XH ..... XH register (high-order 8 bits of the X register)
- XL ..... XL register (low-order 8 bits of the X register)
- Y ..... Index register Y (16 bits)
- YH ..... YH register (high-order 8 bits of the Y register)
- YL ..... YL register (low-order 8 bits of the Y register)
- F ..... Flag register F (4 bits)
- EXT ..... Extension register EXT (8 bits)
- SP1 ..... Stack pointer SP1 (16 bits, however the setting data is 8 bits of D2 to D9)
- SP2 ..... Stack pointer SP2 (16 bits, however the setting data is 8 bits of D0 to D7)
- PC ..... Program counter PC (16 bits)

In the notation with mnemonics, the register names should be written with a % placed in front of them, according to the E0C63 Family assembler source format.

- %A ..... A register
- %B ..... B register
- %BA ..... BA register
- %X ..... X register
- %XH ..... XH register
- %XL ..... XL register
- %Y ..... Y register
- %YH ..... YH register
- %YL ..... YL register
- %F ..... F register
- %EXT ..... EXT register
- %SP1 ..... Stack pointer SP1
- %SP2 ..... Stack pointer SP2

#### Immediate data

- imm2 ..... 2-bit immediate data (0 to 3)
- imm4 ..... 4-bit immediate data (0 to 15)
- imm6 ..... Software interrupt vector (0100H to 013FH)
- imm8 ..... 8-bit immediate data (0 to 255)
- i7-i0 ..... Each bit in immX
- n4 ..... 4-bit radix specification data (1 to 16)
- n3-n0 ..... Each bit in n4
- sign8 ..... Signed 8-bit immediate data (-128 to 127)
- s7-s0 ..... Each bit in sign8
- addr6 ..... 6-bit address (00H to 3FH)
- a5-a0 ..... Each bit in addr6
- 00addr6 ..... addr6 which specifies an address within 0000H to 003FH
- FFaddr6 ..... addr6 which specifies an address within FFC0H to FFFFH

**Memory**

[%X], [X] .....	Memory where the X register specifies
[%Y], [Y] .....	Memory where the Y register specifies
[00addr6] .....	Memory within 0000H to 003FH where the addr6 specifies
[FFaddr6] .....	Memory within FFC0H to FFFFH where the addr6 specifies
[%SP1], [SP1] .....	16-bit address stack where the SP1 specifies
[%SP2], [SP2] .....	4-bit data stack where the SP2 specifies

**Flags**

Z .....	Zero flag
C .....	Carry flag
I .....	Interrupt flag
E .....	Extension flag
↑ .....	Flag is set
↓ .....	Flag is reset
↕ .....	Flag is set or reset
- .....	Flag is not changed

**Operations and others**

+	Addition
-	Subtraction
^	Logical product
∨	Logical sum
∇	Exclusive OR
←	Data load
↔	Data exchange

**Extended addressing mode (EXT.mode)**

○ .....	Can be used
× .....	Cannot be used (prohibit use)

### 4.2.3 Instruction list by function

4-bit data transfer

Mnemonic	Machine code										Operation	Cycle	Flag			EXT. mode	Page						
	12	11	10	9	8	7	6	5	4	3			2	1	0			E	I	C	Z		
LD	%A,%A	1	1	1	1	0	1	1	1	1	0	0	0	0	0	A ← A	1	↓	-	-	-	×	99
	%A,%B	1	1	1	1	0	1	1	1	1	0	0	0	1	0	A ← B	1	↓	-	-	-	×	99
	%A,%F	1	1	1	1	1	1	1	1	1	0	1	1	1	0	A ← F	1	↓	-	-	-	×	99
	%A,imm4	1	1	1	1	0	1	1	0	0	i3	i2	i1	i0	A ← imm4	1	↓	-	-	-	×	100	
	%A,[%X]	1	1	1	1	0	1	1	1	0	0	0	0	0	0	A ← [X]	1	↓	-	-	-	○	100
	%A,[%X]+	1	1	1	1	0	1	1	1	0	0	0	0	1	0	A ← [X], X ← X+1	1	↓	-	-	-	×	101
	%A,[%Y]	1	1	1	1	0	1	1	1	0	0	0	1	0	0	A ← [Y]	1	↓	-	-	-	○	100
%A,[%Y]+	1	1	1	1	0	1	1	1	0	0	0	1	1	0	A ← [Y], Y ← Y+1	1	↓	-	-	-	×	101	
LD	%B,%A	1	1	1	1	0	1	1	1	1	0	1	0	0	0	B ← A	1	↓	-	-	-	×	99
	%B,%B	1	1	1	1	0	1	1	1	1	0	1	1	0	0	B ← B	1	↓	-	-	-	×	99
	%B,imm4	1	1	1	1	0	1	1	0	1	i3	i2	i1	i0	B ← imm4	1	↓	-	-	-	×	100	
	%B,[%X]	1	1	1	1	0	1	1	1	0	0	1	0	0	0	B ← [X]	1	↓	-	-	-	○	100
	%B,[%X]+	1	1	1	1	0	1	1	1	0	0	1	0	1	0	B ← [X], X ← X+1	1	↓	-	-	-	×	101
	%B,[%Y]	1	1	1	1	0	1	1	1	0	0	1	1	0	0	B ← [Y]	1	↓	-	-	-	○	100
	%B,[%Y]+	1	1	1	1	0	1	1	1	0	0	1	1	1	0	B ← [Y], Y ← Y+1	1	↓	-	-	-	×	101
LD	%F,%A	1	1	1	1	1	1	1	1	1	0	1	0	1	0	F ← A	1	↑	↑	↑	↑	×	99
	%F,imm4	1	0	0	0	0	1	0	1	1	i3	i2	i1	i0	F ← imm4	1	↑	↑	↑	↑	×	100	
LD	[%X],%A	1	1	1	1	0	1	1	1	0	1	0	0	0	0	[X] ← A	1	↓	-	-	-	○	101
	[%X],%B	1	1	1	1	0	1	1	1	0	1	1	0	0	0	[X] ← B	1	↓	-	-	-	○	101
	[%X],imm4	1	1	1	1	0	1	0	0	0	i3	i2	i1	i0	[X] ← imm4	1	↓	-	-	-	○	102	
	[%X],[%Y]	1	1	1	1	0	1	1	1	1	1	0	1	0	0	[X] ← [Y]	2	↓	-	-	-	×	103
	[%X],[%Y]+	1	1	1	1	0	1	1	1	1	1	0	1	1	0	[X] ← [Y], Y ← Y+1	2	↓	-	-	-	×	104
	[%X]+,%A	1	1	1	1	0	1	1	1	0	1	0	0	1	0	[X] ← A, X ← X+1	1	↓	-	-	-	×	102
	[%X]+,%B	1	1	1	1	0	1	1	1	0	1	1	0	1	0	[X] ← B, X ← X+1	1	↓	-	-	-	×	102
	[%X]+,imm4	1	1	1	1	0	1	0	0	1	i3	i2	i1	i0	[X] ← imm4, X ← X+1	1	↓	-	-	-	×	103	
	[%X]+,[%Y]	1	1	1	1	0	1	1	1	1	1	1	1	0	0	[X] ← [Y], X ← X+1	2	↓	-	-	-	×	104
[%X]+,[%Y]+	1	1	1	1	0	1	1	1	1	1	1	1	1	0	[X] ← [Y], X ← X+1, Y ← Y+1	2	↓	-	-	-	×	105	
LD	[%Y],%A	1	1	1	1	0	1	1	1	0	1	0	1	0	0	[Y] ← A	1	↓	-	-	-	○	101
	[%Y],%B	1	1	1	1	0	1	1	1	0	1	1	1	0	0	[Y] ← B	1	↓	-	-	-	○	101
	[%Y],imm4	1	1	1	1	0	1	0	1	0	i3	i2	i1	i0	[Y] ← imm4	1	↓	-	-	-	○	102	
	[%Y],[%X]	1	1	1	1	0	1	1	1	1	1	0	0	0	0	[Y] ← [X]	2	↓	-	-	-	×	103
	[%Y],[%X]+	1	1	1	1	0	1	1	1	1	1	0	0	1	0	[Y] ← [X], X ← X+1	2	↓	-	-	-	×	104
	[%Y]+,%A	1	1	1	1	0	1	1	1	0	1	0	1	1	0	[Y] ← A, Y ← Y+1	1	↓	-	-	-	×	102
	[%Y]+,%B	1	1	1	1	0	1	1	1	0	1	1	1	1	0	[Y] ← B, Y ← Y+1	1	↓	-	-	-	×	102
	[%Y]+,imm4	1	1	1	1	0	1	0	1	1	i3	i2	i1	i0	[Y] ← imm4, Y ← Y+1	1	↓	-	-	-	×	103	
	[%Y]+,[%X]	1	1	1	1	0	1	1	1	1	1	1	0	0	0	[Y] ← [X], Y ← Y+1	2	↓	-	-	-	×	104
[%Y]+,[%X]+	1	1	1	1	0	1	1	1	1	1	1	0	1	0	[Y] ← [X], Y ← Y+1, X ← X+1	2	↓	-	-	-	×	105	
EX	%A,%B	1	1	1	1	1	1	1	1	1	0	1	1	1	1	A ↔ B	1	↓	-	-	-	×	90
EX	%A,[%X]	1	0	0	0	0	1	1	1	1	1	0	0	0	0	A ↔ [X]	2	↓	-	-	-	○	91
	%A,[%X]+	1	0	0	0	0	1	1	1	1	1	0	0	1	0	A ↔ [X], X ← X+1	2	↓	-	-	-	×	91
	%A,[%Y]	1	0	0	0	0	1	1	1	1	1	0	1	0	0	A ↔ [Y]	2	↓	-	-	-	○	91
	%A,[%Y]+	1	0	0	0	0	1	1	1	1	1	0	1	1	0	A ↔ [Y], Y ← Y+1	2	↓	-	-	-	×	91
EX	%B,[%X]	1	0	0	0	0	1	1	1	1	1	1	0	0	0	B ↔ [X]	2	↓	-	-	-	○	91
	%B,[%X]+	1	0	0	0	0	1	1	1	1	1	1	0	1	0	B ↔ [X], X ← X+1	2	↓	-	-	-	×	91
	%B,[%Y]	1	0	0	0	0	1	1	1	1	1	1	1	0	0	B ↔ [Y]	2	↓	-	-	-	○	91
	%B,[%Y]+	1	0	0	0	0	1	1	1	1	1	1	1	1	0	B ↔ [Y], Y ← Y+1	2	↓	-	-	-	×	91

## ALU arithmetic operation (1/3)

Mnemonic		Machine code										Operation	Cycle	Flag				EXT. mode	Page				
		12	11	10	9	8	7	6	5	4	3			2	1	0	E			I	C	Z	
ADD	%A,%A	1	1	0	0	1	0	1	1	1	0	0	0	X	A ← A+A	1	↓	-	↕	↕	×	68	
	%A,%B	1	1	0	0	1	0	1	1	1	0	0	1	X	A ← A+B	1	↓	-	↕	↕	×	68	
	%A,imm4	1	1	0	0	1	0	1	0	0	i3	i2	i1	i0	A ← A+imm4	1	↓	-	↕	↕	×	69	
	%A,[%X]	1	1	0	0	1	0	1	1	0	0	0	0	0	A ← A+[X]	1	↓	-	↕	↕	○	69	
	%A,[%X]+	1	1	0	0	1	0	1	1	0	0	0	0	1	A ← A+[X], X ← X+1	1	↓	-	↕	↕	×	70	
	%A,[%Y]	1	1	0	0	1	0	1	1	0	0	0	1	0	A ← A+[Y]	1	↓	-	↕	↕	○	69	
	%A,[%Y]+	1	1	0	0	1	0	1	1	0	0	0	1	1	A ← A+[Y], Y ← Y+1	1	↓	-	↕	↕	×	70	
ADD	%B,%A	1	1	0	0	1	0	1	1	1	0	1	0	X	B ← B+A	1	↓	-	↕	↕	×	68	
	%B,%B	1	1	0	0	1	0	1	1	1	0	1	1	X	B ← B+B	1	↓	-	↕	↕	×	68	
	%B,imm4	1	1	0	0	1	0	1	0	1	i3	i2	i1	i0	B ← B+imm4	1	↓	-	↕	↕	×	69	
	%B,[%X]	1	1	0	0	1	0	1	1	0	0	1	0	0	B ← B+[X]	1	↓	-	↕	↕	○	69	
	%B,[%X]+	1	1	0	0	1	0	1	1	0	0	1	0	1	B ← B+[X], X ← X+1	1	↓	-	↕	↕	×	70	
	%B,[%Y]	1	1	0	0	1	0	1	1	0	0	1	1	0	B ← B+[Y]	1	↓	-	↕	↕	○	69	
	%B,[%Y]+	1	1	0	0	1	0	1	1	0	0	1	1	1	B ← B+[Y], Y ← Y+1	1	↓	-	↕	↕	×	70	
ADD	[%X],%A	1	1	0	0	1	0	1	1	0	1	0	0	0	[X] ← [X]+A	2	↓	-	↕	↕	○	70	
	[%X],%B	1	1	0	0	1	0	1	1	0	1	1	0	0	[X] ← [X]+B	2	↓	-	↕	↕	○	70	
	[%X],imm4	1	1	0	0	1	0	0	0	1	i3	i2	i1	i0	[X] ← [X]+imm4	2	↓	-	↕	↕	○	71	
	[%X]+,%A	1	1	0	0	1	0	1	1	0	1	0	0	1	[X] ← [X]+A, X ← X+1	2	↓	-	↕	↕	×	71	
	[%X]+,%B	1	1	0	0	1	0	1	1	0	1	1	0	1	[X] ← [X]+B, X ← X+1	2	↓	-	↕	↕	×	71	
	[%X]+,imm4	1	1	0	0	1	0	0	0	1	i3	i2	i1	i0	[X] ← [X]+imm4, X ← X+1	2	↓	-	↕	↕	×	72	
	[%Y],%A	1	1	0	0	1	0	1	1	0	1	0	1	0	[Y] ← [Y]+A	2	↓	-	↕	↕	○	70	
ADD	[%Y],%B	1	1	0	0	1	0	1	1	0	1	1	1	0	[Y] ← [Y]+B	2	↓	-	↕	↕	○	70	
	[%Y],imm4	1	1	0	0	1	0	0	1	1	i3	i2	i1	i0	[Y] ← [Y]+imm4	2	↓	-	↕	↕	○	71	
	[%Y]+,%A	1	1	0	0	1	0	1	1	0	1	0	1	1	[Y] ← [Y]+A, Y ← Y+1	2	↓	-	↕	↕	×	71	
	[%Y]+,%B	1	1	0	0	1	0	1	1	0	1	1	1	1	[Y] ← [Y]+B, Y ← Y+1	2	↓	-	↕	↕	×	71	
	[%Y]+,imm4	1	1	0	0	1	0	0	1	1	i3	i2	i1	i0	[Y] ← [Y]+imm4, Y ← Y+1	2	↓	-	↕	↕	×	72	
	ADC	%A,%A	1	1	0	0	1	1	1	1	1	0	0	0	X	A ← A+A+C	1	↓	-	↕	↕	×	61
		%A,%B	1	1	0	0	1	1	1	1	1	0	0	1	X	A ← A+B+C	1	↓	-	↕	↕	×	61
%A,imm4		1	1	0	0	1	1	1	0	0	i3	i2	i1	i0	A ← A+imm4+C	1	↓	-	↕	↕	×	61	
%A,[%X]		1	1	0	0	1	1	1	1	0	0	0	0	0	A ← A+[X]+C	1	↓	-	↕	↕	○	62	
%A,[%X]+		1	1	0	0	1	1	1	1	0	0	0	0	1	A ← A+[X]+C, X ← X+1	1	↓	-	↕	↕	×	62	
%A,[%Y]		1	1	0	0	1	1	1	1	0	0	0	1	0	A ← A+[Y]+C	1	↓	-	↕	↕	○	62	
%A,[%Y]+		1	1	0	0	1	1	1	1	0	0	0	1	1	A ← A+[Y]+C, Y ← Y+1	1	↓	-	↕	↕	×	62	
ADC	%B,%A	1	1	0	0	1	1	1	1	1	0	1	0	X	B ← B+A+C	1	↓	-	↕	↕	×	61	
	%B,%B	1	1	0	0	1	1	1	1	1	0	1	1	X	B ← B+B+C	1	↓	-	↕	↕	×	61	
	%B,imm4	1	1	0	0	1	1	1	0	1	i3	i2	i1	i0	B ← B+imm4+C	1	↓	-	↕	↕	×	61	
	%B,[%X]	1	1	0	0	1	1	1	1	0	0	1	0	0	B ← B+[X]+C	1	↓	-	↕	↕	○	62	
	%B,[%X]+	1	1	0	0	1	1	1	1	0	0	1	0	1	B ← B+[X]+C, X ← X+1	1	↓	-	↕	↕	×	62	
	%B,[%Y]	1	1	0	0	1	1	1	1	0	0	1	1	0	B ← B+[Y]+C	1	↓	-	↕	↕	○	62	
	%B,[%Y]+	1	1	0	0	1	1	1	1	0	0	1	1	1	B ← B+[Y]+C, Y ← Y+1	1	↓	-	↕	↕	×	62	
ADC	[%X],%A	1	1	0	0	1	1	1	1	0	1	0	0	0	[X] ← [X]+A+C	2	↓	-	↕	↕	○	63	
	[%X],%B	1	1	0	0	1	1	1	1	0	1	1	0	0	[X] ← [X]+B+C	2	↓	-	↕	↕	○	63	
	[%X],imm4	1	1	0	0	1	1	0	0	0	i3	i2	i1	i0	[X] ← [X]+imm4+C	2	↓	-	↕	↕	○	64	
	[%X]+,%A	1	1	0	0	1	1	1	1	0	1	0	0	1	[X] ← [X]+A+C, X ← X+1	2	↓	-	↕	↕	×	63	
	[%X]+,%B	1	1	0	0	1	1	1	1	0	1	1	0	1	[X] ← [X]+B+C, X ← X+1	2	↓	-	↕	↕	×	63	
	[%X]+,imm4	1	1	0	0	1	1	0	0	1	i3	i2	i1	i0	[X] ← [X]+imm4+C, X ← X+1	2	↓	-	↕	↕	×	64	
	[%Y],%A	1	1	0	0	1	1	1	1	0	1	0	1	0	[Y] ← [Y]+A+C	2	↓	-	↕	↕	○	63	
ADC	[%Y],%B	1	1	0	0	1	1	1	1	0	1	1	1	0	[Y] ← [Y]+B+C	2	↓	-	↕	↕	○	63	
	[%Y],imm4	1	1	0	0	1	1	0	1	0	i3	i2	i1	i0	[Y] ← [Y]+imm4+C	2	↓	-	↕	↕	○	64	
	[%Y]+,%A	1	1	0	0	1	1	1	1	0	1	0	1	1	[Y] ← [Y]+A+C, Y ← Y+1	2	↓	-	↕	↕	×	63	
	[%Y]+,%B	1	1	0	0	1	1	1	1	0	1	1	1	1	[Y] ← [Y]+B+C, Y ← Y+1	2	↓	-	↕	↕	×	63	
	[%Y]+,imm4	1	1	0	0	1	1	0	1	1	i3	i2	i1	i0	[Y] ← [Y]+imm4+C, Y ← Y+1	2	↓	-	↕	↕	×	64	
	SUB	%A,%A	1	1	0	0	0	0	1	1	1	0	0	0	X	A ← A-A	1	↓	-	↕	↑	×	135
		%A,%B	1	1	0	0	0	0	1	1	1	0	0	1	X	A ← A-B	1	↓	-	↕	↑	×	135
%A,imm4		1	1	0	0	0	0	1	0	0	i3	i2	i1	i0	A ← A-imm4	1	↓	-	↕	↑	×	135	
%A,[%X]		1	1	0	0	0	0	1	1	0	0	0	0	0	A ← A-[X]	1	↓	-	↕	↑	○	136	
%A,[%X]+		1	1	0	0	0	0	1	1	0	0	0	0	1	A ← A-[X], X ← X+1	1	↓	-	↕	↑	×	136	
%A,[%Y]		1	1	0	0	0	0	1	1	0	0	0	1	0	A ← A-[Y]	1	↓	-	↕	↑	○	136	
%A,[%Y]+		1	1	0	0	0	0	1	1	0	0	0	1	1	A ← A-[Y], Y ← Y+1	1	↓	-	↕	↑	×	136	

**CHAPTER 4: INSTRUCTION SET**

**ALU arithmetic operation (2/3)**

Mnemonic		Machine code										Operation	Cycle	Flag			EXT. mode	Page							
		12	11	10	9	8	7	6	5	4	3			2	1	0			E	I	C	Z			
SUB	%B,%A	1	1	0	0	0	0	1	1	1	0	0	1	0	1	0	X	$B \leftarrow B-A$	1	↓	-	↓	↓	×	135
	%B,%B	1	1	0	0	0	0	0	1	1	1	0	0	1	1	1	X	$B \leftarrow B-B$	1	↓	-	↓	↑	×	135
	%B,imm4	1	1	0	0	0	0	1	0	1	0	1	i3	i2	i1	i0	$B \leftarrow B-imm4$	1	↓	-	↓	↓	×	135	
	%B,[%X]	1	1	0	0	0	0	1	1	1	0	0	0	1	0	0	0	$B \leftarrow B-[X]$	1	↓	-	↓	↓	○	136
	%B,[%X]+	1	1	0	0	0	0	1	1	1	0	0	0	1	0	1	1	$B \leftarrow B-[X], X \leftarrow X+1$	1	↓	-	↓	↓	×	136
	%B,[%Y]	1	1	0	0	0	0	1	1	1	0	0	0	1	1	1	0	$B \leftarrow B-[Y]$	1	↓	-	↓	↓	○	136
%B,[%Y]+	1	1	0	0	0	0	1	1	1	0	0	0	1	1	1	1	$B \leftarrow B-[Y], Y \leftarrow Y+1$	1	↓	-	↓	↓	×	136	
SUB	[%X],%A	1	1	0	0	0	0	1	1	0	0	1	0	0	0	0	0	$[X] \leftarrow [X]-A$	2	↓	-	↓	↓	○	137
	[%X],%B	1	1	0	0	0	0	1	1	1	0	1	1	0	0	0	0	$[X] \leftarrow [X]-B$	2	↓	-	↓	↓	○	137
	[%X],imm4	1	1	0	0	0	0	0	0	0	0	i3	i2	i1	i0	$[X] \leftarrow [X]-imm4$	2	↓	-	↓	↓	○	138		
	[%X]+,%A	1	1	0	0	0	0	1	1	1	0	1	0	0	1	1	0	$[X] \leftarrow [X]-A, X \leftarrow X+1$	2	↓	-	↓	↓	×	137
	[%X]+,%B	1	1	0	0	0	0	1	1	1	0	1	1	0	1	1	0	$[X] \leftarrow [X]-B, X \leftarrow X+1$	2	↓	-	↓	↓	×	137
	[%X]+,imm4	1	1	0	0	0	0	0	0	1	1	i3	i2	i1	i0	$[X] \leftarrow [X]-imm4, X \leftarrow X+1$	2	↓	-	↓	↓	×	138		
SUB	[%Y],%A	1	1	0	0	0	0	1	1	0	0	1	0	1	0	0	0	$[Y] \leftarrow [Y]-A$	2	↓	-	↓	↓	○	137
	[%Y],%B	1	1	0	0	0	0	1	1	1	0	1	1	0	0	0	0	$[Y] \leftarrow [Y]-B$	2	↓	-	↓	↓	○	137
	[%Y],imm4	1	1	0	0	0	0	0	1	0	1	i3	i2	i1	i0	$[Y] \leftarrow [Y]-imm4$	2	↓	-	↓	↓	○	138		
	[%Y]+,%A	1	1	0	0	0	0	1	1	1	0	1	0	1	1	1	0	$[Y] \leftarrow [Y]-A, Y \leftarrow Y+1$	2	↓	-	↓	↓	×	137
	[%Y]+,%B	1	1	0	0	0	0	1	1	1	0	1	1	1	1	1	1	$[Y] \leftarrow [Y]-B, Y \leftarrow Y+1$	2	↓	-	↓	↓	×	137
	[%Y]+,imm4	1	1	0	0	0	0	0	1	1	i3	i2	i1	i0	$[Y] \leftarrow [Y]-imm4, Y \leftarrow Y+1$	2	↓	-	↓	↓	×	138			
SBC	%A,%A	1	1	0	0	0	1	1	1	1	0	0	0	0	0	0	X	$A \leftarrow A-A-C$	1	↓	-	↓	↓	×	123
	%A,%B	1	1	0	0	0	1	1	1	1	0	0	0	1	1	1	X	$A \leftarrow A-B-C$	1	↓	-	↓	↓	×	123
	%A,imm4	1	1	0	0	0	1	1	0	0	i3	i2	i1	i0	$A \leftarrow A-imm4-C$	1	↓	-	↓	↓	×	124			
	%A,[%X]	1	1	0	0	0	1	1	1	0	0	0	0	0	0	0	0	$A \leftarrow A-[X]-C$	1	↓	-	↓	↓	○	124
	%A,[%X]+	1	1	0	0	0	1	1	1	0	0	0	0	1	1	0	1	$A \leftarrow A-[X]-C, X \leftarrow X+1$	1	↓	-	↓	↓	×	125
	%A,[%Y]	1	1	0	0	0	1	1	1	0	0	0	1	0	0	1	0	$A \leftarrow A-[Y]-C$	1	↓	-	↓	↓	○	124
%A,[%Y]+	1	1	0	0	0	1	1	1	0	0	0	1	1	1	0	1	$A \leftarrow A-[Y]-C, Y \leftarrow Y+1$	1	↓	-	↓	↓	×	125	
SBC	%B,%A	1	1	0	0	0	1	1	1	1	0	0	1	0	1	0	X	$B \leftarrow B-A-C$	1	↓	-	↓	↓	×	123
	%B,%B	1	1	0	0	0	1	1	1	1	0	0	1	1	1	1	X	$B \leftarrow B-B-C$	1	↓	-	↓	↓	×	123
	%B,imm4	1	1	0	0	0	1	1	0	1	i3	i2	i1	i0	$B \leftarrow B-imm4-C$	1	↓	-	↓	↓	×	124			
	%B,[%X]	1	1	0	0	0	1	1	1	0	0	1	0	0	0	0	0	$B \leftarrow B-[X]-C$	1	↓	-	↓	↓	○	124
	%B,[%X]+	1	1	0	0	0	1	1	1	0	0	1	0	1	0	1	1	$B \leftarrow B-[X]-C, X \leftarrow X+1$	1	↓	-	↓	↓	×	125
	%B,[%Y]	1	1	0	0	0	1	1	1	0	0	1	1	1	0	0	0	$B \leftarrow B-[Y]-C$	1	↓	-	↓	↓	○	124
%B,[%Y]+	1	1	0	0	0	1	1	1	0	0	1	1	1	1	1	1	$B \leftarrow B-[Y]-C, Y \leftarrow Y+1$	1	↓	-	↓	↓	×	125	
SBC	[%X],%A	1	1	0	0	0	1	1	1	0	1	0	0	0	0	0	0	$[X] \leftarrow [X]-A-C$	2	↓	-	↓	↓	○	125
	[%X],%B	1	1	0	0	0	1	1	1	0	1	1	0	0	0	0	0	$[X] \leftarrow [X]-B-C$	2	↓	-	↓	↓	○	125
	[%X],imm4	1	1	0	0	0	1	0	0	0	i3	i2	i1	i0	$[X] \leftarrow [X]-imm4-C$	2	↓	-	↓	↓	○	126			
	[%X]+,%A	1	1	0	0	0	1	1	1	0	1	0	0	1	0	1	1	$[X] \leftarrow [X]-A-C, X \leftarrow X+1$	2	↓	-	↓	↓	×	126
	[%X]+,%B	1	1	0	0	0	1	1	1	0	1	1	0	1	0	1	1	$[X] \leftarrow [X]-B-C, X \leftarrow X+1$	2	↓	-	↓	↓	×	126
	[%X]+,imm4	1	1	0	0	0	1	0	0	1	i3	i2	i1	i0	$[X] \leftarrow [X]-imm4-C, X \leftarrow X+1$	2	↓	-	↓	↓	×	127			
SBC	[%Y],%A	1	1	0	0	0	1	1	1	0	1	0	1	0	0	0	0	$[Y] \leftarrow [Y]-A-C$	2	↓	-	↓	↓	○	125
	[%Y],%B	1	1	0	0	0	1	1	1	0	1	1	1	0	0	0	0	$[Y] \leftarrow [Y]-B-C$	2	↓	-	↓	↓	○	125
	[%Y],imm4	1	1	0	0	0	1	0	1	0	i3	i2	i1	i0	$[Y] \leftarrow [Y]-imm4-C$	2	↓	-	↓	↓	○	126			
	[%Y]+,%A	1	1	0	0	0	1	1	1	0	1	0	1	1	1	0	1	$[Y] \leftarrow [Y]-A-C, Y \leftarrow Y+1$	2	↓	-	↓	↓	×	126
	[%Y]+,%B	1	1	0	0	0	1	1	1	0	1	1	1	1	1	1	1	$[Y] \leftarrow [Y]-B-C, Y \leftarrow Y+1$	2	↓	-	↓	↓	×	126
	[%Y]+,imm4	1	1	0	0	0	1	0	1	1	i3	i2	i1	i0	$[Y] \leftarrow [Y]-imm4-C, Y \leftarrow Y+1$	2	↓	-	↓	↓	×	127			
CMP	%A,%A	1	1	1	1	0	0	1	1	1	X	0	0	0	0	0	0	A-A	1	↓	-	↓	↑	×	84
	%A,%B	1	1	1	1	0	0	1	1	1	X	0	1	0	0	0	0	A-B	1	↓	-	↓	↓	×	84
	%A,imm4	1	1	1	1	0	0	1	0	1	i3	i2	i1	i0	A-imm4	1	↓	-	↓	↓	×	84			
	%A,[%X]	1	1	1	1	0	0	1	1	0	0	0	0	0	0	0	0	A-[X]	1	↓	-	↓	↓	○	85
	%A,[%X]+	1	1	1	1	0	0	1	1	0	0	0	0	1	1	0	0	A-[X], $X \leftarrow X+1$	1	↓	-	↓	↓	×	85
	%A,[%Y]	1	1	1	1	0	0	1	1	0	0	0	1	0	1	0	0	A-[Y]	1	↓	-	↓	↓	○	85
%A,[%Y]+	1	1	1	1	0	0	1	1	0	0	0	1	1	1	0	0	A-[Y], $Y \leftarrow Y+1$	1	↓	-	↓	↓	×	85	
CMP	%B,%A	1	1	1	1	0	0	1	1	1	X	1	0	0	0	0	0	B-A	1	↓	-	↓	↓	×	84
	%B,%B	1	1	1	1	0	0	1	1	1	X	1	1	0	0	0	0	B-B	1	↓	-	↓	↑	×	84
	%B,imm4	1	1	1	1	0	0	1	0	1	i3	i2	i1	i0	B-imm4	1	↓	-	↓	↓	×	84			
	%B,[%X]	1	1	1	1	0	0	1	1	0	0	1	0	0	0	0	0	B-[X]	1	↓	-	↓	↓	○	85
	%B,[%X]+	1	1	1	1	0	0	1	1	0	0	1	0	1	0	1	1	B-[X], $X \leftarrow X+1$	1	↓	-	↓	↓	×	85
	%B,[%Y]	1	1	1	1	0	0	1	1	0	0	1	1	1	0	0	0	B-[Y]	1	↓	-	↓	↓	○	85
%B,[%Y]+	1	1	1	1	0	0	1	1	0	0	1	1	1	1	1	1	B-[Y], $Y \leftarrow Y+1$	1	↓	-	↓	↓	×	85	

## ALU arithmetic operation (3/3)

Mnemonic		Machine code										Operation	Cycle	Flag			EXT. mode	Page				
		12	11	10	9	8	7	6	5	4	3			2	1	0			E	I	C	Z
CMP	[%X],%A	1	1	1	1	0	0	1	1	0	1	0	0	0	[X]-A	1	↓	-	↕	↕	○	86
	[%X],%B	1	1	1	1	0	0	1	1	0	1	1	0	0	[X]-B	1	↓	-	↕	↕	○	86
	[%X],imm4	1	1	1	1	0	0	0	0	0	i3	i2	i1	i0	[X]-imm4	1	↓	-	↕	↕	○	87
	[%X]+,%A	1	1	1	1	0	0	1	1	0	1	0	0	1	[X]-A, X ← X+1	1	↓	-	↕	↕	×	86
	[%X]+,%B	1	1	1	1	0	0	1	1	0	1	1	0	1	[X]-B, X ← X+1	1	↓	-	↕	↕	×	86
[%X]+,imm4	1	1	1	1	0	0	0	0	1	i3	i2	i1	i0	[X]-imm4, X ← X+1	1	↓	-	↕	↕	×	87	
CMP	[%Y],%A	1	1	1	1	0	0	1	1	0	1	0	1	0	[Y]-A	1	↓	-	↕	↕	○	86
	[%Y],%B	1	1	1	1	0	0	1	1	0	1	1	1	0	[Y]-B	1	↓	-	↕	↕	○	86
	[%Y],imm4	1	1	1	1	0	0	0	1	0	i3	i2	i1	i0	[Y]-imm4	1	↓	-	↕	↕	○	87
	[%Y]+,%A	1	1	1	1	0	0	1	1	0	1	0	1	1	[Y]-A, Y ← Y+1	1	↓	-	↕	↕	×	86
	[%Y]+,%B	1	1	1	1	0	0	1	1	0	1	1	1	1	[Y]-B, Y ← Y+1	1	↓	-	↕	↕	×	86
[%Y]+,imm4	1	1	1	1	0	0	0	1	1	i3	i2	i1	i0	[Y]-imm4, Y ← Y+1	1	↓	-	↕	↕	×	87	
INC	[00addr6]	1	0	0	0	0	0	1	a5a4	a3a2a1	a0			[00addr6] ← [00addr6]+1	2	↓	-	↕	↕	×	92	
DEC	[00addr6]	1	0	0	0	0	0	0	a5a4	a3a2a1	a0			[00addr6] ← [00addr6]-1	2	↓	-	↕	↕	×	88	
ADC *1	%B,%A,n4	1	0	0	0	0	1	0	0	0	1	[10H-n4]		B ← N's adjust (B+A+C)	2	↓	-	↕	↕	×	65	
	%B,[%X],n4	1	1	1	0	1	1	1	0	0	[10H-n4]		B ← N's adjust (B+[X]+C)	2	↓	-	↕	↕	○	65		
	%B,[%X]+,n4	1	1	1	0	1	1	1	0	1	[10H-n4]		B ← N's adjust (B+[X]+C), X ← X+1	2	↓	-	↕	↕	×	66		
	%B,[%Y],n4	1	1	1	0	1	1	1	1	0	[10H-n4]		B ← N's adjust (B+[Y]+C)	2	↓	-	↕	↕	○	65		
	%B,[%Y]+,n4	1	1	1	0	1	1	1	1	1	[10H-n4]		B ← N's adjust (B+[Y]+C), Y ← Y+1	2	↓	-	↕	↕	×	66		
ADC *1	[%X],%B,n4	1	1	1	0	1	0	1	0	0	[10H-n4]		[X] ← N's adjust ([X]+B+C)	2	↓	-	↕	↕	○	66		
	[%X],0,n4	1	1	1	0	1	0	0	0	0	[10H-n4]		[X] ← N's adjust ([X]+0+C)	2	↓	-	↕	↕	○	67		
	[%X]+,%B,n4	1	1	1	0	1	0	1	0	1	[10H-n4]		[X] ← N's adjust ([X]+B+C), X ← X+1	2	↓	-	↕	↕	×	67		
	[%X]+,0,n4	1	1	1	0	1	0	0	0	1	[10H-n4]		[X] ← N's adjust ([X]+0+C), X ← X+1	2	↓	-	↕	↕	×	68		
ADC *1	[%Y],%B,n4	1	1	1	0	1	0	1	1	0	[10H-n4]		[Y] ← N's adjust ([Y]+B+C)	2	↓	-	↕	↕	○	66		
	[%Y],0,n4	1	1	1	0	1	0	0	1	0	[10H-n4]		[Y] ← N's adjust ([Y]+0+C)	2	↓	-	↕	↕	○	67		
	[%Y]+,%B,n4	1	1	1	0	1	0	1	1	1	[10H-n4]		[Y] ← N's adjust ([Y]+B+C), Y ← Y+1	2	↓	-	↕	↕	×	67		
	[%Y]+,0,n4	1	1	1	0	1	0	0	1	1	[10H-n4]		[Y] ← N's adjust ([Y]+0+C), Y ← Y+1	2	↓	-	↕	↕	×	68		
SBC *1	%B,%A,n4	1	0	0	0	0	1	1	0	0	n3n2n1	n0		B ← N's adjust (B-A-C)	2	↓	-	↕	↕	×	127	
	%B,[%X],n4	1	1	1	0	0	1	1	0	0	n3n2n1	n0		B ← N's adjust (B-[X]-C)	2	↓	-	↕	↕	○	128	
	%B,[%X]+,n4	1	1	1	0	0	1	1	0	1	n3n2n1	n0		B ← N's adjust (B-[X]-C), X ← X+1	2	↓	-	↕	↕	×	128	
	%B,[%Y],n4	1	1	1	0	0	1	1	1	0	n3n2n1	n0		B ← N's adjust (B-[Y]-C)	2	↓	-	↕	↕	○	128	
	%B,[%Y]+,n4	1	1	1	0	0	1	1	1	1	n3n2n1	n0		B ← N's adjust (B-[Y]-C), Y ← Y+1	2	↓	-	↕	↕	×	128	
SBC *1	[%X],%B,n4	1	1	1	0	0	1	1	0	0	n3n2n1	n0		[X] ← N's adjust ([X]-B-C)	2	↓	-	↕	↕	○	129	
	[%X],0,n4	1	1	1	0	0	0	0	0	0	n3n2n1	n0		[X] ← N's adjust ([X]-0-C)	2	↓	-	↕	↕	○	130	
	[%X]+,%B,n4	1	1	1	0	0	1	1	0	1	n3n2n1	n0		[X] ← N's adjust ([X]-B-C), X ← X+1	2	↓	-	↕	↕	×	129	
	[%X]+,0,n4	1	1	1	0	0	0	0	1	0	n3n2n1	n0		[X] ← N's adjust ([X]-0-C), X ← X+1	2	↓	-	↕	↕	×	130	
SBC *1	[%Y],%B,n4	1	1	1	0	0	1	1	1	0	n3n2n1	n0		[Y] ← N's adjust ([Y]-B-C)	2	↓	-	↕	↕	○	129	
	[%Y],0,n4	1	1	1	0	0	0	1	0	n3n2n1	n0		[Y] ← N's adjust ([Y]-0-C)	2	↓	-	↕	↕	○	130		
	[%Y]+,%B,n4	1	1	1	0	0	1	1	1	n3n2n1	n0		[Y] ← N's adjust ([Y]-B-C), Y ← Y+1	2	↓	-	↕	↕	×	129		
	[%Y]+,0,n4	1	1	1	0	0	0	1	1	n3n2n1	n0		[Y] ← N's adjust ([Y]-0-C), Y ← Y+1	2	↓	-	↕	↕	×	130		
INC *1	[%X],n4	1	1	1	0	1	1	0	0	0	[10H-n4]		[X] ← N's adjust ([X]+1)	2	↓	-	↕	↕	○	93		
	[%X]+,n4	1	1	1	0	1	1	0	1	[10H-n4]			[X] ← N's adjust ([X]+1), X ← X+1	2	↓	-	↕	↕	×	93		
INC *1	[%Y],n4	1	1	1	0	1	1	0	1	0	[10H-n4]		[Y] ← N's adjust ([Y]+1)	2	↓	-	↕	↕	○	93		
	[%Y]+,n4	1	1	1	0	1	1	0	1	1	[10H-n4]		[Y] ← N's adjust ([Y]+1), Y ← Y+1	2	↓	-	↕	↕	×	93		
DEC *1	[%X],n4	1	1	1	0	0	1	0	0	0	n3n2n1	n0		[X] ← N's adjust ([X]-1)	2	↓	-	↕	↕	○	89	
	[%X]+,n4	1	1	1	0	0	1	0	0	1	n3n2n1	n0		[X] ← N's adjust ([X]-1), X ← X+1	2	↓	-	↕	↕	×	89	
DEC *1	[%Y],n4	1	1	1	0	0	1	0	1	0	n3n2n1	n0		[Y] ← N's adjust ([Y]-1)	2	↓	-	↕	↕	○	89	
	[%Y]+,n4	1	1	1	0	0	1	0	1	1	n3n2n1	n0		[Y] ← N's adjust ([Y]-1), Y ← Y+1	2	↓	-	↕	↕	×	89	

\*1 "n4" should be specified with a value between 1 and 16 that indicates a radix.

In the ADC and INC instructions, the assembler converts the "n4" into a complement, and places it at the low-order 4 bits in the machine code.

In the SBC and DEC instructions, the "n4" is placed as it is at the low-order 4 bits in the machine code.

(However, when 16 is specified to n4, the machine code is generated with 0000H as the low-order 4 bits.)

CHAPTER 4: INSTRUCTION SET

ALU logic operation (1/2)

Mnemonic		Machine code										Operation	Cycle	Flag			EXT. mode	Page					
		12	11	10	9	8	7	6	5	4	3			2	1	0			E	I	C	Z	
AND	%A,%A	1	1	0	1	0	0	1	1	1	1	0	0	0	0	X	$A \leftarrow A \wedge A$	1	↓	--	↓	×	73
	%A,%B	1	1	0	1	0	0	1	1	1	0	0	0	1	X	$A \leftarrow A \wedge B$	1	↓	--	↓	×	73	
	%A,imm4	1	1	0	1	0	0	1	0	0	1	i3	i2	i1	i0	$A \leftarrow A \wedge \text{imm4}$	1	↓	--	↓	×	74	
	%A,[%X]	1	1	0	1	0	0	1	1	0	0	0	0	0	0	$A \leftarrow A \wedge [X]$	1	↓	--	↓	○	75	
	%A,[%X]+	1	1	0	1	0	0	1	1	0	0	0	0	1	$A \leftarrow A \wedge [X], X \leftarrow X+1$	1	↓	--	↓	×	75		
	%A,[%Y]	1	1	0	1	0	0	1	1	0	0	0	1	0	$A \leftarrow A \wedge [Y]$	1	↓	--	↓	○	75		
%A,[%Y]+	1	1	0	1	0	0	1	1	0	0	0	1	1	$A \leftarrow A \wedge [Y], Y \leftarrow Y+1$	1	↓	--	↓	×	75			
AND	%B,%A	1	1	0	1	0	0	1	1	1	0	1	0	1	X	$B \leftarrow B \wedge A$	1	↓	--	↓	×	73	
	%B,%B	1	1	0	1	0	0	1	1	1	0	1	1	X	$B \leftarrow B \wedge B$	1	↓	--	↓	×	73		
	%B,imm4	1	1	0	1	0	0	1	0	1	i3	i2	i1	i0	$B \leftarrow B \wedge \text{imm4}$	1	↓	--	↓	×	74		
	%B,[%X]	1	1	0	1	0	0	1	1	0	0	1	0	0	$B \leftarrow B \wedge [X]$	1	↓	--	↓	○	75		
	%B,[%X]+	1	1	0	1	0	0	1	1	0	0	1	0	1	$B \leftarrow B \wedge [X], X \leftarrow X+1$	1	↓	--	↓	×	75		
	%B,[%Y]	1	1	0	1	0	0	1	1	0	0	1	1	0	$B \leftarrow B \wedge [Y]$	1	↓	--	↓	○	75		
%B,[%Y]+	1	1	0	1	0	0	1	1	0	0	1	1	1	$B \leftarrow B \wedge [Y], Y \leftarrow Y+1$	1	↓	--	↓	×	75			
AND	%F,imm4	1	0	0	0	0	1	0	0	0	i3	i2	i1	i0	$F \leftarrow F \wedge \text{imm4}$	1	↓	↓	↓	↓	×	74	
AND	[%X],%A	1	1	0	1	0	0	1	1	0	1	0	0	0	$[X] \leftarrow [X] \wedge A$	2	↓	--	↓	○	76		
	[%X],%B	1	1	0	1	0	0	1	1	0	1	1	0	0	$[X] \leftarrow [X] \wedge B$	2	↓	--	↓	○	76		
	[%X],imm4	1	1	0	1	0	0	0	0	0	i3	i2	i1	i0	$[X] \leftarrow [X] \wedge \text{imm4}$	2	↓	--	↓	○	77		
	[%X]+,%A	1	1	0	1	0	0	1	1	0	1	0	0	1	$[X] \leftarrow [X] \wedge A, X \leftarrow X+1$	2	↓	--	↓	×	76		
	[%X]+,%B	1	1	0	1	0	0	1	1	0	1	1	0	1	$[X] \leftarrow [X] \wedge B, X \leftarrow X+1$	2	↓	--	↓	×	76		
	[%X]+,imm4	1	1	0	1	0	0	0	0	1	i3	i2	i1	i0	$[X] \leftarrow [X] \wedge \text{imm4}, X \leftarrow X+1$	2	↓	--	↓	×	77		
AND	[%Y],%A	1	1	0	1	0	0	1	1	0	1	0	1	0	$[Y] \leftarrow [Y] \wedge A$	2	↓	--	↓	○	76		
	[%Y],%B	1	1	0	1	0	0	1	1	0	1	1	1	0	$[Y] \leftarrow [Y] \wedge B$	2	↓	--	↓	○	76		
	[%Y],imm4	1	1	0	1	0	0	0	1	0	i3	i2	i1	i0	$[Y] \leftarrow [Y] \wedge \text{imm4}$	2	↓	--	↓	○	77		
	[%Y]+,%A	1	1	0	1	0	0	1	1	0	1	0	1	1	$[Y] \leftarrow [Y] \wedge A, Y \leftarrow Y+1$	2	↓	--	↓	×	76		
	[%Y]+,%B	1	1	0	1	0	0	1	1	0	1	1	1	1	$[Y] \leftarrow [Y] \wedge B, Y \leftarrow Y+1$	2	↓	--	↓	×	76		
	[%Y]+,imm4	1	1	0	1	0	0	0	1	1	i3	i2	i1	i0	$[Y] \leftarrow [Y] \wedge \text{imm4}, Y \leftarrow Y+1$	2	↓	--	↓	×	77		
OR	%A,%A	1	1	0	1	1	0	1	1	1	0	0	0	X	$A \leftarrow A \vee A$	1	↓	--	↓	×	112		
	%A,%B	1	1	0	1	1	0	1	1	1	0	0	1	X	$A \leftarrow A \vee B$	1	↓	--	↓	×	112		
	%A,imm4	1	1	0	1	1	0	1	0	1	i3	i2	i1	i0	$A \leftarrow A \vee \text{imm4}$	1	↓	--	↓	×	112		
	%A,[%X]	1	1	0	1	1	0	1	1	0	0	0	0	0	$A \leftarrow A \vee [X]$	1	↓	--	↓	○	113		
	%A,[%X]+	1	1	0	1	1	0	1	1	0	0	0	0	1	$A \leftarrow A \vee [X], X \leftarrow X+1$	1	↓	--	↓	×	114		
	%A,[%Y]	1	1	0	1	1	0	1	1	0	0	0	1	0	$A \leftarrow A \vee [Y]$	1	↓	--	↓	○	113		
	%A,[%Y]+	1	1	0	1	1	0	1	1	0	0	0	1	1	$A \leftarrow A \vee [Y], Y \leftarrow Y+1$	1	↓	--	↓	×	114		
	%B,%A	1	1	0	1	1	0	1	1	1	0	1	0	X	$B \leftarrow B \vee A$	1	↓	--	↓	×	112		
%B,%B	1	1	0	1	1	0	1	1	1	0	1	1	X	$B \leftarrow B \vee B$	1	↓	--	↓	×	112			
%B,imm4	1	1	0	1	1	0	1	0	1	i3	i2	i1	i0	$B \leftarrow B \vee \text{imm4}$	1	↓	--	↓	×	112			
%B,[%X]	1	1	0	1	1	0	1	1	0	0	1	0	0	$B \leftarrow B \vee [X]$	1	↓	--	↓	○	113			
%B,[%X]+	1	1	0	1	1	0	1	1	0	0	1	0	1	$B \leftarrow B \vee [X], X \leftarrow X+1$	1	↓	--	↓	×	114			
%B,[%Y]	1	1	0	1	1	0	1	1	0	0	1	1	0	$B \leftarrow B \vee [Y]$	1	↓	--	↓	○	113			
%B,[%Y]+	1	1	0	1	1	0	1	1	0	0	1	1	1	$B \leftarrow B \vee [Y], Y \leftarrow Y+1$	1	↓	--	↓	×	114			
OR	%F,imm4	1	0	0	0	0	1	0	0	1	i3	i2	i1	i0	$F \leftarrow F \vee \text{imm4}$	1	↑	↑	↑	↑	×	113	
OR	[%X],%A	1	1	0	1	1	0	1	1	0	1	0	0	0	$[X] \leftarrow [X] \vee A$	2	↓	--	↓	○	114		
	[%X],%B	1	1	0	1	1	0	1	1	0	1	1	0	0	$[X] \leftarrow [X] \vee B$	2	↓	--	↓	○	114		
	[%X],imm4	1	1	0	1	1	0	0	0	0	i3	i2	i1	i0	$[X] \leftarrow [X] \vee \text{imm4}$	2	↓	--	↓	○	115		
	[%X]+,%A	1	1	0	1	1	0	1	1	0	1	0	0	1	$[X] \leftarrow [X] \vee A, X \leftarrow X+1$	2	↓	--	↓	×	115		
	[%X]+,%B	1	1	0	1	1	0	1	1	0	1	1	0	1	$[X] \leftarrow [X] \vee B, X \leftarrow X+1$	2	↓	--	↓	×	115		
	[%X]+,imm4	1	1	0	1	1	0	0	0	1	i3	i2	i1	i0	$[X] \leftarrow [X] \vee \text{imm4}, X \leftarrow X+1$	2	↓	--	↓	×	116		
OR	[%Y],%A	1	1	0	1	1	0	1	1	0	1	0	1	0	$[Y] \leftarrow [Y] \vee A$	2	↓	--	↓	○	114		
	[%Y],%B	1	1	0	1	1	0	1	1	0	1	1	1	0	$[Y] \leftarrow [Y] \vee B$	2	↓	--	↓	○	114		
	[%Y],imm4	1	1	0	1	1	0	0	1	0	i3	i2	i1	i0	$[Y] \leftarrow [Y] \vee \text{imm4}$	2	↓	--	↓	○	115		
	[%Y]+,%A	1	1	0	1	1	0	1	1	0	1	0	1	1	$[Y] \leftarrow [Y] \vee A, Y \leftarrow Y+1$	2	↓	--	↓	×	115		
	[%Y]+,%B	1	1	0	1	1	0	1	1	0	1	1	1	1	$[Y] \leftarrow [Y] \vee B, Y \leftarrow Y+1$	2	↓	--	↓	×	115		
	[%Y]+,imm4	1	1	0	1	1	0	0	1	1	i3	i2	i1	i0	$[Y] \leftarrow [Y] \vee \text{imm4}, Y \leftarrow Y+1$	2	↓	--	↓	×	116		



ALU logic operation (2/2)

Mnemonic		Machine code										Operation	Cycle	Flag			EXT. mode	Page				
		12	11	10	9	8	7	6	5	4	3			2	1	0			E	I	C	Z
XOR	%A,%A	1	1	0	1	1	1	1	1	1	0	0	0	X	$A \leftarrow A \nabla A$	1	↓	--	↑	×	139	
	%A,%B	1	1	0	1	1	1	1	1	1	0	0	1	X	$A \leftarrow A \nabla B$	1	↓	--	↑	×	139	
	%A,imm4	1	1	0	1	1	1	0	0	i3	i2	i1	i0	$A \leftarrow A \nabla \text{imm4}$	1	↓	--	↑	×	140		
	%A,[%X]	1	1	0	1	1	1	1	0	0	0	0	0	0	$A \leftarrow A \nabla [X]$	1	↓	--	↑	○	141	
	%A,[%X]+	1	1	0	1	1	1	1	0	0	0	0	0	1	$A \leftarrow A \nabla [X], X \leftarrow X+1$	1	↓	--	↑	×	141	
	%A,[%Y]	1	1	0	1	1	1	1	0	0	0	1	0	0	$A \leftarrow A \nabla [Y]$	1	↓	--	↑	○	141	
XOR	%A,[%Y]+	1	1	0	1	1	1	1	0	0	0	1	1	$A \leftarrow A \nabla [Y], Y \leftarrow Y+1$	1	↓	--	↑	×	141		
	%B,%A	1	1	0	1	1	1	1	1	0	1	0	X	$B \leftarrow B \nabla A$	1	↓	--	↑	×	139		
	%B,%B	1	1	0	1	1	1	1	1	0	1	0	1	X	$B \leftarrow B \nabla B$	1	↓	--	↑	×	139	
	%B,imm4	1	1	0	1	1	1	0	1	i3	i2	i1	i0	$B \leftarrow B \nabla \text{imm4}$	1	↓	--	↑	×	140		
	%B,[%X]	1	1	0	1	1	1	1	0	0	1	0	0	0	$B \leftarrow B \nabla [X]$	1	↓	--	↑	○	141	
	%B,[%X]+	1	1	0	1	1	1	1	0	0	1	0	1	0	$B \leftarrow B \nabla [X], X \leftarrow X+1$	1	↓	--	↑	×	141	
XOR	%B,[%Y]	1	1	0	1	1	1	1	0	0	1	1	0	$B \leftarrow B \nabla [Y]$	1	↓	--	↑	○	141		
	%B,[%Y]+	1	1	0	1	1	1	1	0	0	1	1	1	$B \leftarrow B \nabla [Y], Y \leftarrow Y+1$	1	↓	--	↑	×	141		
	%F,imm4	1	0	0	0	0	1	0	1	0	i3	i2	i1	i0	$F \leftarrow F \nabla \text{imm4}$	1	↑	↓	↓	↓	×	140
	XOR	[%X],%A	1	1	0	1	1	1	1	0	1	0	0	0	$[X] \leftarrow [X] \nabla A$	2	↓	--	↑	○	142	
		[%X],%B	1	1	0	1	1	1	1	0	1	1	0	0	$[X] \leftarrow [X] \nabla B$	2	↓	--	↑	○	142	
		[%X],imm4	1	1	0	1	1	0	0	0	i3	i2	i1	i0	$[X] \leftarrow [X] \nabla \text{imm4}$	2	↓	--	↑	○	143	
[%X]+,%A		1	1	0	1	1	1	1	0	1	0	0	1	$[X] \leftarrow [X] \nabla A, X \leftarrow X+1$	2	↓	--	↑	×	142		
[%X]+,%B		1	1	0	1	1	1	1	0	1	1	0	1	$[X] \leftarrow [X] \nabla B, X \leftarrow X+1$	2	↓	--	↑	×	142		
[%X]+,imm4		1	1	0	1	1	0	0	1	i3	i2	i1	i0	$[X] \leftarrow [X] \nabla \text{imm4}, X \leftarrow X+1$	2	↓	--	↑	×	143		
XOR	[%Y],%A	1	1	0	1	1	1	1	0	1	0	1	0	$[Y] \leftarrow [Y] \nabla A$	2	↓	--	↑	○	142		
	[%Y],%B	1	1	0	1	1	1	1	0	1	1	1	0	$[Y] \leftarrow [Y] \nabla B$	2	↓	--	↑	○	142		
	[%Y],imm4	1	1	0	1	1	0	1	0	i3	i2	i1	i0	$[Y] \leftarrow [Y] \nabla \text{imm4}$	2	↓	--	↑	○	143		
	[%Y]+,%A	1	1	0	1	1	1	1	0	1	0	1	1	$[Y] \leftarrow [Y] \nabla A, Y \leftarrow Y+1$	2	↓	--	↑	×	142		
	[%Y]+,%B	1	1	0	1	1	1	1	0	1	1	1	1	$[Y] \leftarrow [Y] \nabla B, Y \leftarrow Y+1$	2	↓	--	↑	×	142		
	[%Y]+,imm4	1	1	0	1	1	0	1	1	i3	i2	i1	i0	$[Y] \leftarrow [Y] \nabla \text{imm4}, Y \leftarrow Y+1$	2	↓	--	↑	×	143		
BIT	%A,%A	1	1	0	1	0	1	1	1	1	0	0	0	X	$A \wedge A$	1	↓	--	↑	×	78	
	%A,%B	1	1	0	1	0	1	1	1	1	0	0	1	X	$A \wedge B$	1	↓	--	↑	×	78	
	%A,imm4	1	1	0	1	0	1	1	0	0	i3	i2	i1	i0	$A \wedge \text{imm4}$	1	↓	--	↑	×	78	
	%A,[%X]	1	1	0	1	0	1	1	1	0	0	0	0	0	$A \wedge [X]$	1	↓	--	↑	○	79	
	%A,[%X]+	1	1	0	1	0	1	1	1	0	0	0	0	1	$A \wedge [X], X \leftarrow X+1$	1	↓	--	↑	×	79	
	%A,[%Y]	1	1	0	1	0	1	1	1	0	0	0	1	0	$A \wedge [Y]$	1	↓	--	↑	○	79	
	%A,[%Y]+	1	1	0	1	0	1	1	1	0	0	0	1	1	$A \wedge [Y], Y \leftarrow Y+1$	1	↓	--	↑	×	79	
	BIT	%B,%A	1	1	0	1	0	1	1	1	1	0	1	0	X	$B \wedge A$	1	↓	--	↑	×	78
%B,%B		1	1	0	1	0	1	1	1	1	0	1	1	X	$B \wedge B$	1	↓	--	↑	×	78	
%B,imm4		1	1	0	1	0	1	1	0	1	i3	i2	i1	i0	$B \wedge \text{imm4}$	1	↓	--	↑	×	78	
%B,[%X]		1	1	0	1	0	1	1	1	0	0	1	0	0	$B \wedge [X]$	1	↓	--	↑	○	79	
%B,[%X]+		1	1	0	1	0	1	1	1	0	0	1	0	1	$B \wedge [X], X \leftarrow X+1$	1	↓	--	↑	×	79	
%B,[%Y]		1	1	0	1	0	1	1	1	0	0	1	1	0	$B \wedge [Y]$	1	↓	--	↑	○	79	
%B,[%Y]+		1	1	0	1	0	1	1	1	0	0	1	1	1	$B \wedge [Y], Y \leftarrow Y+1$	1	↓	--	↑	×	79	
BIT		[%X],%A	1	1	0	1	0	1	1	1	0	1	0	0	0	$[X] \wedge A$	1	↓	--	↑	○	80
	[%X],%B	1	1	0	1	0	1	1	1	0	1	1	0	0	$[X] \wedge B$	1	↓	--	↑	○	80	
	[%X],imm4	1	1	0	1	0	1	0	0	0	i3	i2	i1	i0	$[X] \wedge \text{imm4}$	1	↓	--	↑	○	81	
	[%X]+,%A	1	1	0	1	0	1	1	1	0	1	0	0	1	$[X] \wedge A, X \leftarrow X+1$	1	↓	--	↑	×	80	
	[%X]+,%B	1	1	0	1	0	1	1	1	0	1	1	0	1	$[X] \wedge B, X \leftarrow X+1$	1	↓	--	↑	×	80	
	[%X]+,imm4	1	1	0	1	0	1	0	1	1	i3	i2	i1	i0	$[X] \wedge \text{imm4}, X \leftarrow X+1$	1	↓	--	↑	×	81	
	BIT	[%Y],%A	1	1	0	1	0	1	1	1	0	1	0	1	0	$[Y] \wedge A$	1	↓	--	↑	○	80
[%Y],%B		1	1	0	1	0	1	1	1	0	1	1	1	0	$[Y] \wedge B$	1	↓	--	↑	○	80	
[%Y],imm4		1	1	0	1	0	1	0	1	0	i3	i2	i1	i0	$[Y] \wedge \text{imm4}$	1	↓	--	↑	○	81	
[%Y]+,%A		1	1	0	1	0	1	1	1	0	1	0	1	1	$[Y] \wedge A, Y \leftarrow Y+1$	1	↓	--	↑	×	80	
[%Y]+,%B		1	1	0	1	0	1	1	1	0	1	1	1	1	$[Y] \wedge B, Y \leftarrow Y+1$	1	↓	--	↑	×	80	
[%Y]+,imm4		1	1	0	1	0	1	0	1	1	i3	i2	i1	i0	$[Y] \wedge \text{imm4}, Y \leftarrow Y+1$	1	↓	--	↑	×	81	
CLR	[00addr6],imm2	1	0	1	0	0	i1	i0	a5	a4	a3	a2	a1	a0	$[00addr6] \leftarrow [00addr6] \wedge \text{not } (2^{\text{imm2}})$	2	↓	--	↑	×	83	
	[FFaddr6],imm2	1	0	1	0	1	i1	i0	a5	a4	a3	a2	a1	a0	$[FFaddr6] \leftarrow [FFaddr6] \wedge \text{not } (2^{\text{imm2}})$	2	↓	--	↑	×	83	
SET	[00addr6],imm2	1	0	1	1	0	i1	i0	a5	a4	a3	a2	a1	a0	$[00addr6] \leftarrow [00addr6] \wedge (2^{\text{imm2}})$	2	↓	--	↑	×	131	
	[FFaddr6],imm2	1	0	1	1	1	i1	i0	a5	a4	a3	a2	a1	a0	$[FFaddr6] \leftarrow [FFaddr6] \wedge (2^{\text{imm2}})$	2	↓	--	↑	×	131	
TST	[00addr6],imm2	1	0	0	1	0	i1	i0	a5	a4	a3	a2	a1	a0	$[00addr6] \wedge (2^{\text{imm2}})$	1	↓	--	↑	×	139	
	[FFaddr6],imm2	1	0	0	1	1	i1	i0	a5	a4	a3	a2	a1	a0	$[FFaddr6] \wedge (2^{\text{imm2}})$	1	↓	--	↑	×	139	

**CHAPTER 4: INSTRUCTION SET**

**ALU shift and rotate operation**

Mnemonic		Machine code										Operation	Cycle	Flag			EXT. mode	Page				
		12	11	10	9	8	7	6	5	4	3			2	1	0			E	I	C	Z
SLL	%A	1	0	0	0	0	1	1	1	1	0	0	0	0	A (C←D3←D2←D1←D0←0)	1	↓	-	↓	↓	×	131
	%B	1	0	0	0	0	1	1	1	1	0	1	0	0	B (C←D3←D2←D1←D0←0)	1	↓	-	↓	↓	×	131
	[%X]	1	0	0	0	0	1	1	1	0	0	0	0	0	[X] (C←D3←D2←D1←D0←0)	2	↓	-	↓	↓	○	132
	[%X]+	1	0	0	0	0	1	1	1	0	0	0	0	1	[X] (C←D3←D2←D1←D0←0), X ← X+1	2	↓	-	↓	↓	×	132
	[%Y]	1	0	0	0	0	1	1	1	0	0	0	1	0	[Y] (C←D3←D2←D1←D0←0)	2	↓	-	↓	↓	○	132
[%Y]+	1	0	0	0	0	1	1	1	0	0	0	1	1	[Y] (C←D3←D2←D1←D0←0), Y ← Y+1	2	↓	-	↓	↓	×	132	
SRL	%A	1	0	0	0	0	1	1	1	1	0	0	0	1	A (0→D3→D2→D1→D0→C)	1	↓	-	↓	↓	×	133
	%B	1	0	0	0	0	1	1	1	1	0	1	0	1	B (0→D3→D2→D1→D0→C)	1	↓	-	↓	↓	×	133
	[%X]	1	0	0	0	0	1	1	1	0	0	1	0	0	[X] (0→D3→D2→D1→D0→C)	2	↓	-	↓	↓	○	134
	[%X]+	1	0	0	0	0	1	1	1	0	0	1	0	1	[X] (0→D3→D2→D1→D0→C), X ← X+1	2	↓	-	↓	↓	×	134
	[%Y]	1	0	0	0	0	1	1	1	0	0	1	0	0	[Y] (0→D3→D2→D1→D0→C)	2	↓	-	↓	↓	○	134
[%Y]+	1	0	0	0	0	1	1	1	0	0	1	1	1	[Y] (0→D3→D2→D1→D0→C), Y ← Y+1	2	↓	-	↓	↓	×	134	
RL	%A	1	0	0	0	0	1	1	1	1	0	0	1	0	A (C←D3←D2←D1←D0←C)	1	↓	-	↓	↓	×	120
	%B	1	0	0	0	0	1	1	1	1	0	1	0	1	B (C←D3←D2←D1←D0←C)	1	↓	-	↓	↓	×	120
	[%X]	1	0	0	0	0	1	1	1	0	1	0	0	0	[X] (C←D3←D2←D1←D0←C)	2	↓	-	↓	↓	○	121
	[%X]+	1	0	0	0	0	1	1	1	0	1	0	0	1	[X] (C←D3←D2←D1←D0←C), X ← X+1	2	↓	-	↓	↓	×	121
	[%Y]	1	0	0	0	0	1	1	1	0	1	0	1	0	[Y] (C←D3←D2←D1←D0←C)	2	↓	-	↓	↓	○	121
[%Y]+	1	0	0	0	0	1	1	1	0	1	0	1	1	[Y] (C←D3←D2←D1←D0←C), Y ← Y+1	2	↓	-	↓	↓	×	121	
RR	%A	1	0	0	0	0	1	1	1	1	0	0	1	0	A (C→D3→D2→D1→D0→C)	1	↓	-	↓	↓	×	122
	%B	1	0	0	0	0	1	1	1	1	0	1	1	1	B (C→D3→D2→D1→D0→C)	1	↓	-	↓	↓	×	122
	[%X]	1	0	0	0	0	1	1	1	0	1	1	0	0	[X] (C→D3→D2→D1→D0→C)	2	↓	-	↓	↓	○	122
	[%X]+	1	0	0	0	0	1	1	1	0	1	1	0	1	[X] (C→D3→D2→D1→D0→C), X ← X+1	2	↓	-	↓	↓	×	123
	[%Y]	1	0	0	0	0	1	1	1	0	1	1	1	0	[Y] (C→D3→D2→D1→D0→C)	2	↓	-	↓	↓	○	122
[%Y]+	1	0	0	0	0	1	1	1	0	1	1	1	1	[Y] (C→D3→D2→D1→D0→C), Y ← Y+1	2	↓	-	↓	↓	×	123	

**8/16-bit operation**

Mnemonic		Machine code										Operation	Cycle	Flag			EXT. mode	Page				
		12	11	10	9	8	7	6	5	4	3			2	1	0			E	I	C	Z
LDB	%BA,%XL	1	1	1	1	1	1	1	0	0	1	0	0	0	BA ← XL	1	↓	-	-	-	×	107
	%BA,%XH	1	1	1	1	1	1	1	0	0	1	0	0	1	BA ← XH	1	↓	-	-	-	×	107
	%BA,%YL	1	1	1	1	1	1	1	0	0	1	0	1	0	BA ← YL	1	↓	-	-	-	×	107
	%BA,%YH	1	1	1	1	1	1	1	0	0	1	0	1	1	BA ← YH	1	↓	-	-	-	×	107
	%BA,%EXT	1	1	1	1	1	1	1	0	1	0	1	1	X	BA ← EXT	1	↓	-	-	-	×	106
	%BA,%SP1	1	1	1	1	1	1	1	0	0	1	1	0	X	BA ← SP1	1	↓	-	-	-	×	107
	%BA,%SP2	1	1	1	1	1	1	1	0	0	1	1	1	X	BA ← SP2	1	↓	-	-	-	×	107
	%BA,imm8	0	1	0	0	1	i7	i6	i5	i4	i3	i2	i1	i0	BA ← imm8	1	↓	-	-	-	×	105
	%BA,[%X]+	1	1	1	1	1	1	0	1	1	0	0	0	0	A ← [X], B ← [X+1], X ← X+2	2	↓	-	-	-	×	106
%BA,[%Y]+	1	1	1	1	1	1	0	1	0	1	0	0	0	A ← [Y], B ← [Y+1], Y ← Y+2	2	↓	-	-	-	×	106	
LDB	%XL,%BA	1	1	1	1	1	1	0	0	0	0	0	0	XL ← BA	1	↓	-	-	-	×	110	
	%XL,imm8	0	1	0	1	0	i7	i6	i5	i4	i3	i2	i1	i0	XL ← imm8	1	↓	-	-	-	○	110
	%XH,%BA	1	1	1	1	1	1	1	0	0	0	0	1	1	XH ← BA	1	↓	-	-	-	×	110
LDB	%YL,%BA	1	1	1	1	1	1	0	0	0	1	0	1	0	YL ← BA	1	↓	-	-	-	×	110
	%YL,imm8	0	1	0	1	1	i7	i6	i5	i4	i3	i2	i1	i0	YL ← imm8	1	↓	-	-	-	○	110
	%YH,%BA	1	1	1	1	1	1	0	0	0	1	1	0	0	YH ← BA	1	↓	-	-	-	×	110
LDB	%EXT,%BA	1	1	1	1	1	1	0	1	0	1	0	X	EXT ← BA	1	↑	-	-	-	×	109	
	%EXT,imm8	0	1	0	0	0	i7	i6	i5	i4	i3	i2	i1	i0	EXT ← imm8	1	↑	-	-	-	×	109
LDB	%SP1,%BA	1	1	1	1	1	1	0	0	0	1	0	X	SP1 ← BA	1	↓	-	-	-	×	111	
	%SP2,%BA	1	1	1	1	1	1	0	0	1	1	1	X	SP2 ← BA	1	↓	-	-	-	×	111	
LDB	[%X]+,%BA	1	1	1	1	1	1	0	1	0	1	0	0	1	[X] ← A, [X+1] ← B, X ← X+2	2	↓	-	-	-	×	108
	[%X]+,imm8	0	0	0	0	1	i7	i6	i5	i4	i3	i2	i1	i0	[X] ← i3-0, [X+1] ← i7-4, X ← X+2	2	↓	-	-	-	×	108
LDB	[%Y]+,%BA	1	1	1	1	1	1	0	1	0	1	0	1	1	[Y] ← A, [Y+1] ← B, Y ← Y+2	2	↓	-	-	-	×	108
ADD	%X,%BA	1	1	1	1	1	1	0	1	0	0	0	X	X ← X+BA	1	↓	-	-	↓	×	72	
	%X,sign8	0	1	1	0	0	s7	s6	s5	s4	s3	s2	s1	s0	X ← X+sign8 (sign8=-128~127)	1	↓	-	-	↓	○	72
	%Y,%BA	1	1	1	1	1	1	0	1	0	0	1	X	Y ← Y+BA	1	↓	-	-	↓	×	72	
	%Y,sign8	0	1	1	0	0	s7	s6	s5	s4	s3	s2	s1	s0	Y ← Y+sign8 (sign8=-128~127)	1	↓	-	-	↓	○	72
CMP	%X,imm8	0	1	1	1	0	[ FFH - imm8 ]								X-imm8 (imm8=0~255)	1	↓	-	↓	↓	○	88
	%Y,imm8	0	1	1	1	1	[ FFH - imm8 ]								Y-imm8 (imm8=0~255)	1	↓	-	↓	↓	○	88
INC	%SP1	1	1	1	1	1	1	1	0	1	0	0	0	0	SP1 ← SP1+1	1	↓	-	-	↓	×	94
	%SP2	1	1	1	1	1	1	1	0	1	1	0	0	0	SP2 ← SP2+1	1	↓	-	-	↓	×	94
DEC	%SP1	1	1	1	1	1	1	1	0	0	0	0	0	0	SP1 ← SP1-1	1	↓	-	-	↓	×	90
	%SP2	1	1	1	1	1	1	1	0	0	1	0	0	0	SP2 ← SP2-1	1	↓	-	-	↓	×	90

## Stack operation

Mnemonic	Machine code	Operation	Cycle	Flag				EXT. mode	Page
				E	I	C	Z		
PUSH	%A	$[SP2-1] \leftarrow A, SP2 \leftarrow SP2-1$	1	↓	-	-	-	×	117
	%B	$[SP2-1] \leftarrow B, SP2 \leftarrow SP2-1$	1	↓	-	-	-	×	117
	%F	$[SP2-1] \leftarrow F, SP2 \leftarrow SP2-1$	1	↓	-	-	-	×	117
	%X	$(([SP1-1]*4+3)-[SP1-1]*4) \leftarrow X, SP1 \leftarrow SP1-1$	1	↓	-	-	-	×	118
POP	%A	$A \leftarrow [SP2], SP2 \leftarrow SP2+1$	1	↓	-	-	-	×	116
	%B	$B \leftarrow [SP2], SP2 \leftarrow SP2+1$	1	↓	-	-	-	×	116
	%F	$F \leftarrow [SP2], SP2 \leftarrow SP2+1$	1	↑	↑	↑	↑	×	116
	%X	$X \leftarrow ([SP1*4+3]-[SP1*4]), SP1 \leftarrow SP1+1$	1	↓	-	-	-	×	117
	%Y	$Y \leftarrow ([SP1*4+3]-[SP1*4]), SP1 \leftarrow SP1+1$	1	↓	-	-	-	×	117

## Branch control

Mnemonic	Machine code	Operation	Cycle	Flag				EXT. mode	Page
				E	I	C	Z		
JR	sign8	$PC \leftarrow PC+sign8+1$ (sign8=-128~127)	1	↓	-	-	-	○	97
JR	%A	$PC \leftarrow PC+A+1$	1	↓	-	-	-	×	95
	%BA	$PC \leftarrow PC+BA+1$	1	↓	-	-	-	×	96
JR	[00addr6]	$PC \leftarrow PC+[00addr6]+1$	2	↓	-	-	-	×	96
JRC	sign8	If C=1 then $PC \leftarrow PC+sign8+1$ (sign8=-128~127)	1	↓	-	-	-	○	97
JRNC	sign8	If C=0 then $PC \leftarrow PC+sign8+1$ (sign8=-128~127)	1	↓	-	-	-	○	98
JRZ	sign8	If Z=1 then $PC \leftarrow PC+sign8+1$ (sign8=-128~127)	1	↓	-	-	-	○	99
JRNZ	sign8	If Z=0 then $PC \leftarrow PC+sign8+1$ (sign8=-128~127)	1	↓	-	-	-	○	98
JP	%Y	$PC \leftarrow Y$	1	↓	-	-	-	×	95
CALZ	imm8	$(([SP1-1]*4+3)-[SP1-1]*4) \leftarrow PC+1, SP1 \leftarrow SP1-1, PC \leftarrow imm8$	1	↓	-	-	-	×	83
CALR	sign8	$(([SP1-1]*4+3)-[SP1-1]*4) \leftarrow PC+1, SP1 \leftarrow SP1-1, PC \leftarrow PC+sign8+1$ (sign8=-128~127)	1	↓	-	-	-	○	82
CALR	[00addr6]	$(([SP1-1]*4+3)-[SP1-1]*4) \leftarrow PC+1, SP1 \leftarrow SP1-1, PC \leftarrow PC+[00addr6]+1$	2	↓	-	-	-	×	82
INT	imm6	$[SP2-1] \leftarrow F, SP2 \leftarrow SP2-1$ $(([SP1-1]*4+3)-[SP1-1]*4) \leftarrow PC+1, SP1 \leftarrow SP1-1, PC \leftarrow imm6$ (imm6=0100H~013FH)	3	↓	-	-	-	×	94
RET		$PC \leftarrow ([SP1*4+3]-[SP1*4]), SP1 \leftarrow SP1+1$	1	↓	-	-	-	×	118
RETS		$PC \leftarrow ([SP1*4+3]-[SP1*4]), SP1 \leftarrow SP1+1$ $PC \leftarrow PC+1$	2	↓	-	-	-	×	120
RETD	imm8	$PC \leftarrow ([SP1*4+3]-[SP1*4]), SP1 \leftarrow SP1+1$ $[X] \leftarrow i3-0, [X+1] \leftarrow i7-4, X \leftarrow X+2$	3	↓	-	-	-	×	119
RETI		$PC \leftarrow ([SP1*4+3]-[SP1*4]), SP1 \leftarrow SP1+1$ $F \leftarrow [SP2], SP2 \leftarrow SP2+1$	2	↑	↑	↑	↑	×	119

## System control

Mnemonic	Machine code	Operation	Cycle	Flag				EXT. mode	Page
				E	I	C	Z		
HALT		Halt	2	↓	-	-	-	×	92
SLP		Sleep	2	↓	-	-	-	×	133
NOP		No operation ( $PC \leftarrow PC+1$ )	1	↓	-	-	-	×	111

Note:

- The extended addressing (combined with the E flag) is available only for the instructions indicated with ○ in the EXT. mode row. Operation of other instructions (indicated with ×) cannot be guaranteed, therefore do not write data to the EXT register or do not set the E flag immediately before those instructions.

- X in the machine code row indicates that the bit is valid even though it is "0" or "1", but the assembler generates it as "0". When entering the code directly, such as for debugging, "0" should be entered.

4.2.4 List in alphabetical order

Mnemonic	Machine code										Operation	Cycle	Flag			EXT. mode	Page					
	12	11	10	9	8	7	6	5	4	3			2	1	0			E	I	Z		
ADC	%A,%A	1	1	0	0	1	1	1	1	1	0	0	0	X	A ← A+A+C	1	↓	-	↓	↓	×	61
	%A,%B	1	1	0	0	1	1	1	1	1	0	0	1	X	A ← A+B+C	1	↓	-	↓	↓	×	61
	%A,imm4	1	1	0	0	1	1	1	0	0	i3	i2	i1	i0	A ← A+imm4+C	1	↓	-	↓	↓	×	61
	%A,[%X]	1	1	0	0	1	1	1	1	0	0	0	0	0	A ← A+[X]+C	1	↓	-	↓	↓	○	62
	%A,[%X]+	1	1	0	0	1	1	1	1	0	0	0	0	1	A ← A+[X]+C, X ← X+1	1	↓	-	↓	↓	×	62
	%A,[%Y]	1	1	0	0	1	1	1	1	0	0	0	1	0	A ← A+[Y]+C	1	↓	-	↓	↓	○	62
	%A,[%Y]+	1	1	0	0	1	1	1	1	0	0	0	1	1	A ← A+[Y]+C, Y ← Y+1	1	↓	-	↓	↓	×	62
	%B,%A	1	1	0	0	1	1	1	1	1	0	1	0	X	B ← B+A+C	1	↓	-	↓	↓	×	61
	%B,%A,n4	1	0	0	0	0	1	1	0	1	[	10H	-	n4]	B ← N's adjust (B+A+C)	2	↓	-	↓	↓	×	65
	%B,%B	1	1	0	0	1	1	1	1	1	0	1	1	X	B ← B+B+C	1	↓	-	↓	↓	×	61
	%B,imm4	1	1	0	0	1	1	1	0	1	i3	i2	i1	i0	B ← B+imm4+C	1	↓	-	↓	↓	×	61
	%B,[%X]	1	1	0	0	1	1	1	1	0	0	1	0	0	B ← B+[X]+C	1	↓	-	↓	↓	○	62
	%B,[%X],n4	1	1	1	0	1	1	1	0	0	[	10H	-	n4]	B ← N's adjust (B+[X]+C)	2	↓	-	↓	↓	○	65
	%B,[%X]+	1	1	0	0	1	1	1	1	0	0	1	0	1	B ← B+[X]+C, X ← X+1	1	↓	-	↓	↓	×	62
	%B,[%X]+,n4	1	1	1	0	1	1	1	0	1	[	10H	-	n4]	B ← N's adjust (B+[X]+C), X ← X+1	2	↓	-	↓	↓	×	66
	%B,[%Y]	1	1	0	0	1	1	1	1	0	0	1	1	0	B ← B+[Y]+C	1	↓	-	↓	↓	○	62
	%B,[%Y],n4	1	1	1	0	1	1	1	1	0	[	10H	-	n4]	B ← N's adjust (B+[Y]+C)	2	↓	-	↓	↓	○	65
	%B,[%Y]+	1	1	0	0	1	1	1	1	0	0	1	1	1	B ← B+[Y]+C, Y ← Y+1	1	↓	-	↓	↓	×	62
	%B,[%Y]+,n4	1	1	1	0	1	1	1	1	1	[	10H	-	n4]	B ← N's adjust (B+[Y]+C), Y ← Y+1	2	↓	-	↓	↓	×	66
	[%X],%A	1	1	0	0	1	1	1	1	0	1	0	0	0	[X] ← [X]+A+C	2	↓	-	↓	↓	○	63
	[%X],%B	1	1	0	0	1	1	1	1	0	1	1	0	0	[X] ← [X]+B+C	2	↓	-	↓	↓	○	63
	[%X],%B,n4	1	1	1	0	1	0	1	0	0	[	10H	-	n4]	[X] ← N's adjust ([X]+B+C)	2	↓	-	↓	↓	○	66
	[%X],imm4	1	1	0	0	1	1	0	0	0	i3	i2	i1	i0	[X] ← [X]+imm4+C	2	↓	-	↓	↓	○	64
	[%X],0,n4	1	1	1	0	1	0	0	0	0	[	10H	-	n4]	[X] ← N's adjust ([X]+0+C)	2	↓	-	↓	↓	○	67
	[%X]+,%A	1	1	0	0	1	1	1	1	0	1	0	0	1	[X] ← [X]+A+C, X ← X+1	2	↓	-	↓	↓	×	63
	[%X]+,%B	1	1	0	0	1	1	1	1	0	1	1	0	1	[X] ← [X]+B+C, X ← X+1	2	↓	-	↓	↓	×	63
	[%X]+,%B,n4	1	1	1	0	1	0	1	0	1	[	10H	-	n4]	[X] ← N's adjust ([X]+B+C), X ← X+1	2	↓	-	↓	↓	×	67
	[%X]+,imm4	1	1	0	0	1	1	0	0	1	i3	i2	i1	i0	[X] ← [X]+imm4+C, X ← X+1	2	↓	-	↓	↓	×	64
	[%X]+,0,n4	1	1	1	0	1	0	0	0	1	[	10H	-	n4]	[X] ← N's adjust ([X]+0+C), X ← X+1	2	↓	-	↓	↓	×	68
	[%Y],%A	1	1	0	0	1	1	1	1	0	1	0	1	0	[Y] ← [Y]+A+C	2	↓	-	↓	↓	○	63
	[%Y],%B	1	1	0	0	1	1	1	1	0	1	1	1	0	[Y] ← [Y]+B+C	2	↓	-	↓	↓	○	63
	[%Y],%B,n4	1	1	1	0	1	0	1	1	0	[	10H	-	n4]	[Y] ← N's adjust ([Y]+B+C)	2	↓	-	↓	↓	○	66
[%Y],imm4	1	1	0	0	1	1	0	1	0	i3	i2	i1	i0	[Y] ← [Y]+imm4+C	2	↓	-	↓	↓	○	64	
[%Y],0,n4	1	1	1	0	1	0	0	1	0	[	10H	-	n4]	[Y] ← N's adjust ([Y]+0+C)	2	↓	-	↓	↓	○	67	
[%Y]+,%A	1	1	0	0	1	1	1	1	0	1	0	1	1	[Y] ← [Y]+A+C, Y ← Y+1	2	↓	-	↓	↓	×	63	
[%Y]+,%B	1	1	0	0	1	1	1	1	0	1	1	1	1	[Y] ← [Y]+B+C, Y ← Y+1	2	↓	-	↓	↓	×	63	
[%Y]+,%B,n4	1	1	1	0	1	0	1	1	1	[	10H	-	n4]	[Y] ← N's adjust ([Y]+B+C), Y ← Y+1	2	↓	-	↓	↓	×	67	
[%Y]+,imm4	1	1	0	0	1	1	0	1	1	i3	i2	i1	i0	[Y] ← [Y]+imm4+C, Y ← Y+1	2	↓	-	↓	↓	×	64	
[%Y]+,0,n4	1	1	1	0	1	0	0	1	1	[	10H	-	n4]	[Y] ← N's adjust ([Y]+0+C), Y ← Y+1	2	↓	-	↓	↓	×	67	
ADD	%A,%A	1	1	0	0	1	0	1	1	1	0	0	0	X	A ← A+A	1	↓	-	↓	↓	×	68
	%A,%B	1	1	0	0	1	0	1	1	1	0	0	1	X	A ← A+B	1	↓	-	↓	↓	×	68
	%A,imm4	1	1	0	0	1	0	1	0	0	i3	i2	i1	i0	A ← A+imm4	1	↓	-	↓	↓	×	69
	%A,[%X]	1	1	0	0	1	0	1	1	0	0	0	0	0	A ← A+[X]	1	↓	-	↓	↓	○	69
	%A,[%X]+	1	1	0	0	1	0	1	1	0	0	0	0	1	A ← A+[X], X ← X+1	1	↓	-	↓	↓	×	70
	%A,[%Y]	1	1	0	0	1	0	1	1	0	0	0	1	0	A ← A+[Y]	1	↓	-	↓	↓	○	69
	%A,[%Y]+	1	1	0	0	1	0	1	1	0	0	0	1	1	A ← A+[Y], Y ← Y+1	1	↓	-	↓	↓	×	70
	%B,%A	1	1	0	0	1	0	1	1	1	0	1	0	X	B ← B+A	1	↓	-	↓	↓	×	68
	%B,%B	1	1	0	0	1	0	1	1	1	0	1	1	X	B ← B+B	1	↓	-	↓	↓	×	68
	%B,imm4	1	1	0	0	1	0	1	0	1	i3	i2	i1	i0	B ← B+imm4	1	↓	-	↓	↓	×	69
	%B,[%X]	1	1	0	0	1	0	1	1	0	0	1	0	0	B ← B+[X]	1	↓	-	↓	↓	○	69
	%B,[%X]+	1	1	0	0	1	0	1	1	0	0	1	0	1	B ← B+[X], X ← X+1	1	↓	-	↓	↓	×	70
	%B,[%Y]	1	1	0	0	1	0	1	1	0	0	1	1	0	B ← B+[Y]	1	↓	-	↓	↓	○	69
	%B,[%Y]+	1	1	0	0	1	0	1	1	0	0	1	1	1	B ← B+[Y], Y ← Y+1	1	↓	-	↓	↓	×	70
	%X,%BA	1	1	1	1	1	1	1	0	1	0	0	0	X	X ← X+BA	1	↓	-	-	↓	×	72
	%X,sign8	0	1	1	0	0	s7	s6	s5	s4	s3	s2	s1	s0	X ← X+sign8 (sign8=-128~127)	1	↓	-	-	↓	○	73
	%Y,%BA	1	1	1	1	1	1	1	0	1	0	0	1	X	Y ← Y+BA	1	↓	-	-	↓	×	72
	%Y,sign8	0	1	1	0	1	s7	s6	s5	s4	s3	s2	s1	s0	Y ← Y+sign8 (sign8=-128~127)	1	↓	-	-	↓	○	73
	[%X],%A	1	1	0	0	1	0	1	1	0	1	0	0	0	[X] ← [X]+A	2	↓	-	↓	↓	○	70
	[%X],%B	1	1	0	0	1	0	1	1	0	1	1	0	0	[X] ← [X]+B	2	↓	-	↓	↓	○	70

Mnemonic	Machine code										Operation	Cycle	Flag			EXT. mode	Page					
	12	11	10	9	8	7	6	5	4	3			2	1	0			E	I	C	Z	
ADD	[%X],imm4	1	1	0	0	1	0	0	0	0	i3	i2	i1	i0	$X \leftarrow [X] + \text{imm4}$	2	↓	-	↓	↓	○	71
	[%X]+,%A	1	1	0	0	1	0	1	1	0	0	0	0	1	$X \leftarrow [X] + A, X \leftarrow X + 1$	2	↓	-	↓	↓	×	71
	[%X]+,%B	1	1	0	0	1	0	1	1	0	1	1	0	1	$X \leftarrow [X] + B, X \leftarrow X + 1$	2	↓	-	↓	↓	×	71
	[%X]+,imm4	1	1	0	0	1	0	0	0	1	i3	i2	i1	i0	$X \leftarrow [X] + \text{imm4}, X \leftarrow X + 1$	2	↓	-	↓	↓	×	72
	[%Y],%A	1	1	0	0	1	0	1	1	0	1	0	1	0	$Y \leftarrow [Y] + A$	2	↓	-	↓	↓	○	70
	[%Y],%B	1	1	0	0	1	0	1	1	0	1	1	1	0	$Y \leftarrow [Y] + B$	2	↓	-	↓	↓	○	70
	[%Y],imm4	1	1	0	0	1	0	0	1	0	i3	i2	i1	i0	$Y \leftarrow [Y] + \text{imm4}$	2	↓	-	↓	↓	○	71
	[%Y]+,%A	1	1	0	0	1	0	1	1	0	1	0	1	1	$Y \leftarrow [Y] + A, Y \leftarrow Y + 1$	2	↓	-	↓	↓	×	71
	[%Y]+,%B	1	1	0	0	1	0	1	1	0	1	1	1	1	$Y \leftarrow [Y] + B, Y \leftarrow Y + 1$	2	↓	-	↓	↓	×	71
[%Y]+,imm4	1	1	0	0	1	0	0	1	1	i3	i2	i1	i0	$Y \leftarrow [Y] + \text{imm4}, Y \leftarrow Y + 1$	2	↓	-	↓	↓	×	72	
AND	%A,%A	1	1	0	1	0	0	1	1	1	0	0	0	X	$A \leftarrow A \wedge A$	1	↓	-	-	↓	×	73
	%A,%B	1	1	0	1	0	0	1	1	1	0	0	1	X	$A \leftarrow A \wedge B$	1	↓	-	-	↓	×	73
	%A,imm4	1	1	0	1	0	0	1	0	0	i3	i2	i1	i0	$A \leftarrow A \wedge \text{imm4}$	1	↓	-	-	↓	×	74
	%A,[%X]	1	1	0	1	0	0	1	1	0	0	0	0	0	$A \leftarrow A \wedge [X]$	1	↓	-	-	↓	○	75
	%A,[%X]+	1	1	0	1	0	0	1	1	0	0	0	0	1	$A \leftarrow A \wedge [X], X \leftarrow X + 1$	1	↓	-	-	↓	×	75
	%A,[%Y]	1	1	0	1	0	0	1	1	0	0	0	1	0	$A \leftarrow A \wedge [Y]$	1	↓	-	-	↓	○	75
	%A,[%Y]+	1	1	0	1	0	0	1	1	0	0	0	1	1	$A \leftarrow A \wedge [Y], Y \leftarrow Y + 1$	1	↓	-	-	↓	×	75
	%B,%A	1	1	0	1	0	0	1	1	1	0	1	0	X	$B \leftarrow B \wedge A$	1	↓	-	-	↓	×	73
	%B,%B	1	1	0	1	0	0	1	1	1	0	1	1	X	$B \leftarrow B \wedge B$	1	↓	-	-	↓	×	73
	%B,imm4	1	1	0	1	0	0	1	1	i3	i2	i1	i0	$B \leftarrow B \wedge \text{imm4}$	1	↓	-	-	↓	×	74	
	%B,[%X]	1	1	0	1	0	0	1	1	0	1	0	0	$B \leftarrow B \wedge [X]$	1	↓	-	-	↓	○	75	
	%B,[%X]+	1	1	0	1	0	0	1	1	0	1	0	1	$B \leftarrow B \wedge [X], X \leftarrow X + 1$	1	↓	-	-	↓	×	75	
	%B,[%Y]	1	1	0	1	0	0	1	1	0	1	1	0	$B \leftarrow B \wedge [Y]$	1	↓	-	-	↓	○	75	
	%B,[%Y]+	1	1	0	1	0	0	1	1	1	1	1	1	$B \leftarrow B \wedge [Y], Y \leftarrow Y + 1$	1	↓	-	-	↓	×	75	
	%F,imm4	1	0	0	0	0	1	0	0	i3	i2	i1	i0	$F \leftarrow F \wedge \text{imm4}$	1	↓	↓	↓	↓	×	74	
	[%X],%A	1	1	0	1	0	0	1	1	0	0	0	0	$X \leftarrow [X] \wedge A$	2	↓	-	-	↓	○	76	
	[%X],%B	1	1	0	1	0	0	1	1	0	1	0	0	$X \leftarrow [X] \wedge B$	2	↓	-	-	↓	○	76	
	[%X],imm4	1	1	0	1	0	0	0	0	i3	i2	i1	i0	$X \leftarrow [X] \wedge \text{imm4}$	2	↓	-	-	↓	○	77	
	[%X]+,%A	1	1	0	1	0	0	1	1	0	1	0	1	$X \leftarrow [X] \wedge A, X \leftarrow X + 1$	2	↓	-	-	↓	×	76	
	[%X]+,%B	1	1	0	1	0	0	1	1	0	1	0	1	$X \leftarrow [X] \wedge B, X \leftarrow X + 1$	2	↓	-	-	↓	×	76	
	[%X]+,imm4	1	1	0	1	0	0	0	1	i3	i2	i1	i0	$X \leftarrow [X] \wedge \text{imm4}, X \leftarrow X + 1$	2	↓	-	-	↓	×	77	
	[%Y],%A	1	1	0	1	0	0	1	1	0	1	0	1	$Y \leftarrow [Y] \wedge A$	2	↓	-	-	↓	○	76	
	[%Y],%B	1	1	0	1	0	0	1	1	0	1	1	0	$Y \leftarrow [Y] \wedge B$	2	↓	-	-	↓	○	76	
	[%Y],imm4	1	1	0	1	0	0	0	1	i3	i2	i1	i0	$Y \leftarrow [Y] \wedge \text{imm4}$	2	↓	-	-	↓	○	77	
	[%Y]+,%A	1	1	0	1	0	0	1	1	0	1	1	1	$Y \leftarrow [Y] \wedge A, Y \leftarrow Y + 1$	2	↓	-	-	↓	×	76	
	[%Y]+,%B	1	1	0	1	0	0	1	1	0	1	1	1	$Y \leftarrow [Y] \wedge B, Y \leftarrow Y + 1$	2	↓	-	-	↓	×	76	
	[%Y]+,imm4	1	1	0	1	0	0	1	1	i3	i2	i1	i0	$Y \leftarrow [Y] \wedge \text{imm4}, Y \leftarrow Y + 1$	2	↓	-	-	↓	×	77	
BIT	%A,%A	1	1	0	1	0	1	1	1	1	0	0	X	$A \wedge A$	1	↓	-	-	↓	×	78	
	%A,%B	1	1	0	1	0	1	1	1	1	0	1	X	$A \wedge B$	1	↓	-	-	↓	×	78	
	%A,imm4	1	1	0	1	0	1	1	0	i3	i2	i1	i0	$A \wedge \text{imm4}$	1	↓	-	-	↓	×	78	
	%A,[%X]	1	1	0	1	0	1	1	0	0	0	0	0	$A \wedge [X]$	1	↓	-	-	↓	○	79	
	%A,[%X]+	1	1	0	1	0	1	1	0	0	0	0	1	$A \wedge [X], X \leftarrow X + 1$	1	↓	-	-	↓	×	79	
	%A,[%Y]	1	1	0	1	0	1	1	0	0	0	1	0	$A \wedge [Y]$	1	↓	-	-	↓	○	79	
	%A,[%Y]+	1	1	0	1	0	1	1	0	0	0	1	1	$A \wedge [Y], Y \leftarrow Y + 1$	1	↓	-	-	↓	×	79	
	%B,%A	1	1	0	1	0	1	1	1	1	0	1	X	$B \wedge A$	1	↓	-	-	↓	×	78	
	%B,%B	1	1	0	1	0	1	1	1	1	0	1	X	$B \wedge B$	1	↓	-	-	↓	×	78	
	%B,imm4	1	1	0	1	0	1	1	0	i3	i2	i1	i0	$B \wedge \text{imm4}$	1	↓	-	-	↓	×	78	
	%B,[%X]	1	1	0	1	0	1	1	0	0	1	0	0	$B \wedge [X]$	1	↓	-	-	↓	○	79	
	%B,[%X]+	1	1	0	1	0	1	1	0	0	1	0	1	$B \wedge [X], X \leftarrow X + 1$	1	↓	-	-	↓	×	79	
	%B,[%Y]	1	1	0	1	0	1	1	0	0	1	1	0	$B \wedge [Y]$	1	↓	-	-	↓	○	79	
	%B,[%Y]+	1	1	0	1	0	1	1	0	0	1	1	1	$B \wedge [Y], Y \leftarrow Y + 1$	1	↓	-	-	↓	×	79	
	[%X],%A	1	1	0	1	0	1	1	1	0	0	0	0	$X \wedge A$	1	↓	-	-	↓	○	80	
	[%X],%B	1	1	0	1	0	1	1	1	0	1	0	0	$X \wedge B$	1	↓	-	-	↓	○	80	
	[%X],imm4	1	1	0	1	0	1	0	0	i3	i2	i1	i0	$X \wedge \text{imm4}$	1	↓	-	-	↓	○	81	
	[%X]+,%A	1	1	0	1	0	1	1	1	0	1	0	1	$X \wedge A, X \leftarrow X + 1$	1	↓	-	-	↓	×	80	
	[%X]+,%B	1	1	0	1	0	1	1	1	0	1	0	1	$X \wedge B, X \leftarrow X + 1$	1	↓	-	-	↓	×	80	
	[%X]+,imm4	1	1	0	1	0	1	0	0	i3	i2	i1	i0	$X \wedge \text{imm4}, X \leftarrow X + 1$	1	↓	-	-	↓	×	81	
	[%Y],%A	1	1	0	1	0	1	1	1	0	1	0	1	$Y \wedge A$	1	↓	-	-	↓	○	80	
	[%Y],%B	1	1	0	1	0	1	1	1	0	1	1	0	$Y \wedge B$	1	↓	-	-	↓	○	80	

CHAPTER 4: INSTRUCTION SET

Mnemonic	Machine code										Operation	Cycle	Flag			EXT. mode	Page					
	12	11	10	9	8	7	6	5	4	3			2	1	0			E	I	C	Z	
BIT	[%Y],imm4	1	1	0	1	0	1	0	1	0	i3	i2	i1	i0	[Y] $\wedge$ imm4	1	↓	-	-	↓	○	81
	[%Y]+,%A	1	1	0	1	0	1	1	1	0	1	0	1	1	[Y] $\wedge$ A, Y $\leftarrow$ Y+1	1	↓	-	-	↓	×	80
	[%Y]+,%B	1	1	0	1	0	1	1	1	0	1	1	1	1	[Y] $\wedge$ B, Y $\leftarrow$ Y+1	1	↓	-	-	↓	×	80
	[%Y]+,imm4	1	1	0	1	0	1	0	1	1	i3	i2	i1	i0	[Y] $\wedge$ imm4, Y $\leftarrow$ Y+1	1	↓	-	-	↓	×	81
CALR	[00addr6]	1	1	1	1	1	0	0	a5	a4	a3	a2	a1	a0	(((SP1-1)*4+3)-((SP1-1)*4)) $\leftarrow$ PC+1, SP1 $\leftarrow$ SP1-1, PC $\leftarrow$ PC+[00addr6]+1	2	↓	-	-	-	×	82
CALR	sign8	0	0	0	1	0	s7	s6	s5	s4	s3	s2	s1	s0	(((SP1-1)*4+3)-((SP1-1)*4)) $\leftarrow$ PC+1, SP1 $\leftarrow$ SP1-1, PC $\leftarrow$ PC+sign8+1 (sign8=128-127)	1	↓	-	-	-	○	82
CALZ	imm8	0	0	0	1	1	i7	i6	i5	i4	i3	i2	i1	i0	(((SP1-1)*4+3)-((SP1-1)*4)) $\leftarrow$ PC+1, SP1 $\leftarrow$ SP1-1, PC $\leftarrow$ imm8	1	↓	-	-	-	×	83
CLR	[00addr6],imm2	1	0	1	0	0	i1	i0	a5	a4	a3	a2	a1	a0	[00addr6] $\leftarrow$ [00addr6] $\wedge$ not (2 <sup>imm2</sup> )	2	↓	-	-	↓	×	83
	[FFaddr6],imm2	1	0	1	0	1	i1	i0	a5	a4	a3	a2	a1	a0	[FFaddr6] $\leftarrow$ [FFaddr6] $\wedge$ not (2 <sup>imm2</sup> )	2	↓	-	-	↓	×	83
CMP	%A,%A	1	1	1	1	0	0	1	1	1	X	0	0	0	A-A	1	↓	-	↓	↑	×	84
	%A,%B	1	1	1	1	0	0	1	1	1	X	0	1	1	A-B	1	↓	-	↓	↓	×	84
	%A,imm4	1	1	1	1	0	0	1	0	0	i3	i2	i1	i0	A-imm4	1	↓	-	↓	↓	×	84
	%A,[%X]	1	1	1	1	0	0	1	1	0	0	0	0	0	A-[X]	1	↓	-	↓	↓	○	85
	%A,[%X]+	1	1	1	1	0	0	1	1	0	0	0	0	1	A-[X], X $\leftarrow$ X+1	1	↓	-	↓	↓	×	85
	%A,[%Y]	1	1	1	1	0	0	1	1	0	0	0	1	0	A-[Y]	1	↓	-	↓	↓	○	85
	%A,[%Y]+	1	1	1	1	0	0	1	1	0	0	0	1	1	A-[Y], Y $\leftarrow$ Y+1	1	↓	-	↓	↓	×	85
	%B,%A	1	1	1	1	0	0	1	1	1	X	1	0	0	B-A	1	↓	-	↓	↓	×	84
	%B,%B	1	1	1	1	0	0	1	1	1	X	1	1	0	B-B	1	↓	-	↓	↑	×	84
	%B,imm4	1	1	1	1	0	0	1	0	1	i3	i2	i1	i0	B-imm4	1	↓	-	↓	↓	×	84
	%B,[%X]	1	1	1	1	0	0	1	1	0	0	1	0	0	B-[X]	1	↓	-	↓	↓	○	85
	%B,[%X]+	1	1	1	1	0	0	1	1	0	0	1	0	1	B-[X], X $\leftarrow$ X+1	1	↓	-	↓	↓	×	85
	%B,[%Y]	1	1	1	1	0	0	1	1	0	0	1	1	0	B-[Y]	1	↓	-	↓	↓	○	85
	%B,[%Y]+	1	1	1	1	0	0	1	1	0	0	1	1	1	B-[Y], Y $\leftarrow$ Y+1	1	↓	-	↓	↓	×	85
	%X,imm8	0	1	1	1	0	[	FFH	-	imm8	]	X-imm8 (imm8=0~255)			1	↓	-	↓	↓	○	88	
	%Y,imm8	0	1	1	1	1	[	FFH	-	imm8	]	Y-imm8 (imm8=0~255)			1	↓	-	↓	↓	○	88	
	[%X],%A	1	1	1	1	0	0	1	1	0	1	0	0	0	[X]-A	1	↓	-	↓	↓	○	86
	[%X],%B	1	1	1	1	0	0	1	1	0	1	1	0	0	[X]-B	1	↓	-	↓	↓	○	86
	[%X],imm4	1	1	1	1	0	0	0	0	0	i3	i2	i1	i0	[X]-imm4	1	↓	-	↓	↓	○	87
	[%X]+,%A	1	1	1	1	0	0	1	0	0	1	0	0	1	[X]-A, X $\leftarrow$ X+1	1	↓	-	↓	↓	×	86
[%X]+,%B	1	1	1	1	0	0	1	1	0	1	1	0	1	[X]-B, X $\leftarrow$ X+1	1	↓	-	↓	↓	×	86	
[%X]+,imm4	1	1	1	1	0	0	0	0	1	i3	i2	i1	i0	[X]-imm4, X $\leftarrow$ X+1	1	↓	-	↓	↓	×	87	
[%Y],%A	1	1	1	1	0	0	1	1	0	1	0	1	0	[Y]-A	1	↓	-	↓	↓	○	86	
[%Y],%B	1	1	1	1	0	0	1	1	0	1	1	1	0	[Y]-B	1	↓	-	↓	↓	○	86	
[%Y],imm4	1	1	1	1	0	0	0	1	0	i3	i2	i1	i0	[Y]-imm4	1	↓	-	↓	↓	○	87	
[%Y]+,%A	1	1	1	1	0	0	1	1	0	1	0	1	1	[Y]-A, Y $\leftarrow$ Y+1	1	↓	-	↓	↓	×	86	
[%Y]+,%B	1	1	1	1	0	0	1	1	0	1	1	1	1	[Y]-B, Y $\leftarrow$ Y+1	1	↓	-	↓	↓	×	86	
[%Y]+,imm4	1	1	1	1	0	0	0	1	1	i3	i2	i1	i0	[Y]-imm4, Y $\leftarrow$ Y+1	1	↓	-	↓	↓	×	87	
DEC	%SP1	1	1	1	1	1	1	1	1	0	0	0	0	0	SP1 $\leftarrow$ SP1-1	1	↓	-	-	↓	×	90
	%SP2	1	1	1	1	1	1	1	1	0	0	1	0	0	SP2 $\leftarrow$ SP2-1	1	↓	-	-	↓	×	90
	[%X],n4	1	1	1	0	0	1	0	0	0	n3	n2	n1	n0	[X] $\leftarrow$ N's adjust ([X]-1)	2	↓	-	↓	↓	○	89
	[%X]+,n4	1	1	1	0	0	1	0	0	1	n3	n2	n1	n0	[X] $\leftarrow$ N's adjust ([X]-1), X $\leftarrow$ X+1	2	↓	-	↓	↓	×	89
	[%Y],n4	1	1	1	0	0	1	0	1	0	n3	n2	n1	n0	[Y] $\leftarrow$ N's adjust ([Y]-1)	2	↓	-	↓	↓	○	89
	[%Y]+,n4	1	1	1	0	0	1	0	1	1	n3	n2	n1	n0	[Y] $\leftarrow$ N's adjust ([Y]-1), Y $\leftarrow$ Y+1	2	↓	-	↓	↓	×	89
	[00addr6]	1	0	0	0	0	0	0	a5	a4	a3	a2	a1	a0	[00addr6] $\leftarrow$ [00addr6]-1	2	↓	-	↓	↓	×	88
EX	%A,%B	1	1	1	1	1	1	1	1	1	0	1	1	1	A $\leftrightarrow$ B	1	↓	-	-	-	×	90
	%A,[%X]	1	0	0	0	1	1	1	1	1	0	0	0	A $\leftrightarrow$ [X]	2	↓	-	-	-	○	91	
	%A,[%X]+	1	0	0	0	1	1	1	1	1	0	0	1	A $\leftrightarrow$ [X], X $\leftarrow$ X+1	2	↓	-	-	-	×	91	
	%A,[%Y]	1	0	0	0	1	1	1	1	1	0	1	0	A $\leftrightarrow$ [Y]	2	↓	-	-	-	○	91	
	%A,[%Y]+	1	0	0	0	1	1	1	1	1	0	1	1	A $\leftrightarrow$ [Y], Y $\leftarrow$ Y+1	2	↓	-	-	-	×	91	
	%B,[%X]	1	0	0	0	1	1	1	1	1	1	0	0	B $\leftrightarrow$ [X]	2	↓	-	-	-	○	91	
	%B,[%X]+	1	0	0	0	1	1	1	1	1	1	0	1	B $\leftrightarrow$ [X], X $\leftarrow$ X+1	2	↓	-	-	-	×	91	
	%B,[%Y]	1	0	0	0	1	1	1	1	1	1	1	0	B $\leftrightarrow$ [Y]	2	↓	-	-	-	○	91	
%B,[%Y]+	1	0	0	0	1	1	1	1	1	1	1	1	B $\leftrightarrow$ [Y], Y $\leftarrow$ Y+1	2	↓	-	-	-	×	91		
HALT		1	1	1	1	1	1	1	1	1	1	1	0	0	Halt	2	↓	-	-	-	×	92
INC	%SP1	1	1	1	1	1	1	1	1	0	1	0	0	0	SP1 $\leftarrow$ SP1+1	1	↓	-	-	↓	×	94
	%SP2	1	1	1	1	1	1	1	1	0	1	1	0	0	SP2 $\leftarrow$ SP2+1	1	↓	-	-	↓	×	94

Mnemonic		Machine code										Operation	Cycle	Flag			EXT. mode	Page					
		12	11	10	9	8	7	6	5	4	3			2	1	0			E	I	C	Z	
INC	[%X],n4	1	1	1	0	0	1	1	0	0	0	0	[10H-n4]	[X] ← N's adjust ([X]+1)	2	↓	-	↕	↕	○	93		
	[%X]+,n4	1	1	1	0	0	1	1	0	0	0	0	[10H-n4]	[X] ← N's adjust ([X]+1), X ← X+1	2	↓	-	↕	↕	×	93		
	[%Y],n4	1	1	1	0	0	1	1	0	0	0	0	[10H-n4]	[Y] ← N's adjust ([Y]+1)	2	↓	-	↕	↕	○	93		
	[%Y]+,n4	1	1	1	0	0	1	1	0	0	1	1	[10H-n4]	[Y] ← N's adjust ([Y]+1), Y ← Y+1	2	↓	-	↕	↕	×	93		
	[00addr6]	1	0	0	0	0	0	1	a5a4	a3a2a1a0			[00addr6] ← [00addr6]+1	2	↓	-	↕	↕	×	92			
INT	imm6	1	1	1	1	1	1	0	i5i4	i3i2i1i0			[SP2-1] ← F, SP2 ← SP2-1 (((SP1-1)*4+3)-[(SP1-1)*4]) ← PC+1, SP1 ← SP1-1, PC ← imm6 (imm6=0100H-013FH)	3	↓	-	-	-	×	94			
JP	%Y	1	1	1	1	1	1	1	1	1	1	0	0 0 1 X	PC ← Y	1	↓	-	-	-	×	95		
JR	%A	1	1	1	1	1	1	1	1	1	1	0	0 0 0 1	PC ← PC+A+1	1	↓	-	-	-	×	95		
	%BA	1	1	1	1	1	1	1	1	1	1	0	0 0 0 0	PC ← PC+BA+1	1	↓	-	-	-	×	96		
	sign8	0	0	0	0	0	s7	s6	s5	s4	s3	s2	s1	s0		PC ← PC+sign8+1 (sign8=-128~127)	1	↓	-	-	-	○	97
	[00addr6]	1	1	1	1	1	0	1	a5a4	a3a2a1a0			PC ← PC+[00addr6]+1	2	↓	-	-	-	×	96			
JRC	sign8	0	0	1	0	0	s7	s6	s5	s4	s3	s2	s1	s0		If C=1 then PC ← PC+sign8+1 (sign8=-128~127)	1	↓	-	-	-	○	97
JRNC	sign8	0	0	1	0	1	s7	s6	s5	s4	s3	s2	s1	s0		If C=0 then PC ← PC+sign8+1 (sign8=-128~127)	1	↓	-	-	-	○	98
JRNZ	sign8	0	0	1	1	1	s7	s6	s5	s4	s3	s2	s1	s0		If Z=0 then PC ← PC+sign8+1 (sign8=-128~127)	1	↓	-	-	-	○	98
JRZ	sign8	0	0	1	1	0	s7	s6	s5	s4	s3	s2	s1	s0		If Z=1 then PC ← PC+sign8+1 (sign8=-128~127)	1	↓	-	-	-	○	99
LD	%A,%A	1	1	1	1	0	1	1	1	1	1	0	0 0 0 0	A ← A	1	↓	-	-	-	×	99		
	%A,%B	1	1	1	1	0	1	1	1	1	1	0	0 0 1 0	A ← B	1	↓	-	-	-	×	99		
	%A,%F	1	1	1	1	1	1	1	1	1	1	0	0 1 1 0	A ← F	1	↓	-	-	-	×	99		
	%A,imm4	1	1	1	1	0	1	1	1	0	i3	i2 i1 i0		A ← imm4	1	↓	-	-	-	×	100		
	%A,[%X]	1	1	1	1	0	1	1	1	1	0	0	0 0 0 0	A ← [X]	1	↓	-	-	-	○	100		
	%A,[%X]+	1	1	1	1	0	1	1	1	1	0	0	0 0 0 1	A ← [X], X ← X+1	1	↓	-	-	-	×	101		
	%A,[%Y]	1	1	1	1	0	1	1	1	1	0	0	0 0 1 0	A ← [Y]	1	↓	-	-	-	○	100		
	%A,[%Y]+	1	1	1	1	0	1	1	1	1	0	0	0 1 1 0	A ← [Y], Y ← Y+1	1	↓	-	-	-	×	101		
	%B,%A	1	1	1	1	0	1	1	1	1	1	0	0 1 0 0	B ← A	1	↓	-	-	-	×	99		
	%B,%B	1	1	1	1	0	1	1	1	1	1	0	0 1 1 0	B ← B	1	↓	-	-	-	×	99		
	%B,imm4	1	1	1	1	0	1	1	1	0	i3	i2 i1 i0		B ← imm4	1	↓	-	-	-	×	100		
	%B,[%X]	1	1	1	1	0	1	1	1	1	0	0	1 0 0 0	B ← [X]	1	↓	-	-	-	○	100		
	%B,[%X]+	1	1	1	1	0	1	1	1	1	0	0	1 0 1 0	B ← [X], X ← X+1	1	↓	-	-	-	×	101		
	%B,[%Y]	1	1	1	1	0	1	1	1	1	0	0	1 1 0 0	B ← [Y]	1	↓	-	-	-	○	100		
	%B,[%Y]+	1	1	1	1	0	1	1	1	1	0	0	1 1 1 1	B ← [Y], Y ← Y+1	1	↓	-	-	-	×	101		
	%F,%A	1	1	1	1	1	1	1	1	1	1	0	1 0 1 0	F ← A	1	↕	↕	↕	↕	×	99		
	%F,imm4	1	0	0	0	0	1	0	1	1	i3	i2 i1 i0		F ← imm4	1	↕	↕	↕	↕	×	100		
	[%X],%A	1	1	1	1	0	1	1	1	1	0	0	1 0 0 0	[X] ← A	1	↓	-	-	-	○	101		
	[%X],%B	1	1	1	1	0	1	1	1	1	0	0	1 1 0 0	[X] ← B	1	↓	-	-	-	○	101		
	[%X],imm4	1	1	1	1	0	1	0	0	0	i3	i2 i1 i0		[X] ← imm4	1	↓	-	-	-	○	102		
	[%X],[%Y]	1	1	1	1	0	1	1	1	1	0	1	0 1 0 0	[X] ← [Y]	2	↓	-	-	-	×	103		
	[%X],[%Y]+	1	1	1	1	0	1	1	1	1	0	1	1 0 1 1	[X] ← [Y], Y ← Y+1	2	↓	-	-	-	×	104		
	[%X]+,%A	1	1	1	1	0	1	1	1	0	0	0	1 0 0 1	[X] ← A, X ← X+1	1	↓	-	-	-	×	102		
	[%X]+,%B	1	1	1	1	0	1	1	1	0	1	0	1 1 0 1	[X] ← B, X ← X+1	1	↓	-	-	-	×	102		
	[%X]+,imm4	1	1	1	1	0	1	0	0	1	i3	i2 i1 i0		[X] ← imm4, X ← X+1	1	↓	-	-	-	×	103		
	[%X]+,[%Y]	1	1	1	1	0	1	1	1	1	0	1	1 1 0 0	[X] ← [Y], X ← X+1	2	↓	-	-	-	×	104		
	[%X]+,[%Y]+	1	1	1	1	0	1	1	1	1	1	1	1 1 1 1	[X] ← [Y], X ← X+1, Y ← Y+1	2	↓	-	-	-	×	105		
[%Y],%A	1	1	1	1	0	1	1	1	1	0	0	1 0 1 0	[Y] ← A	1	↓	-	-	-	○	101			
[%Y],%B	1	1	1	1	0	1	1	1	1	0	0	1 1 1 0	[Y] ← B	1	↓	-	-	-	○	101			
[%Y],imm4	1	1	1	1	0	1	0	1	0	i3	i2 i1 i0		[Y] ← imm4	1	↓	-	-	-	○	102			
[%Y],[%X]	1	1	1	1	0	1	1	1	1	0	0	1 0 0 0	[Y] ← [X]	2	↓	-	-	-	×	103			
[%Y],[%X]+	1	1	1	1	0	1	1	1	1	0	0	1 0 0 1	[Y] ← [X], X ← X+1	2	↓	-	-	-	×	104			
[%Y]+,%A	1	1	1	1	0	1	1	1	0	0	1	1 0 1 1	[Y] ← A, Y ← Y+1	1	↓	-	-	-	×	102			
[%Y]+,%B	1	1	1	1	0	1	1	1	0	1	1	1 1 1 1	[Y] ← B, Y ← Y+1	1	↓	-	-	-	×	102			
[%Y]+,imm4	1	1	1	1	0	1	0	1	1	i3	i2 i1 i0		[Y] ← imm4, Y ← Y+1	1	↓	-	-	-	×	103			
[%Y]+,[%X]	1	1	1	1	0	1	1	1	1	0	0	1 1 0 0	[Y] ← [X], Y ← Y+1	2	↓	-	-	-	×	104			
[%Y]+,[%X]+	1	1	1	1	0	1	1	1	1	1	0	1 1 0 1	[Y] ← [X], Y ← Y+1, X ← X+1	2	↓	-	-	-	×	105			
LDB	%BA,%EXT	1	1	1	1	1	1	1	0	0	1	1	0 1 X	BA ← EXT	1	↓	-	-	-	×	106		
	%BA,%SP1	1	1	1	1	1	1	1	0	0	1	1	0 X	BA ← SP1	1	↓	-	-	-	×	107		
	%BA,%SP2	1	1	1	1	1	1	1	0	0	1	1	1 X	BA ← SP2	1	↓	-	-	-	×	107		
	%BA,%XH	1	1	1	1	1	1	1	0	0	1	0	0 1	BA ← XH	1	↓	-	-	-	×	107		
	%BA,%XL	1	1	1	1	1	1	1	0	0	1	0	0 0	BA ← XL	1	↓	-	-	-	×	107		

CHAPTER 4: INSTRUCTION SET

Mnemonic	Machine code										Operation	Cycle	Flag			EXT. mode	Page					
	12	11	10	9	8	7	6	5	4	3			2	1	0			E	I	C	Z	
LDB	%BA,%YH	1	1	1	1	1	1	1	0	1	0	1	1	BA ← YH	1	↓	-	-	-	x	107	
	%BA,%YL	1	1	1	1	1	1	1	0	0	1	0	1	BA ← YL	1	↓	-	-	-	x	107	
	%BA,imm8	0	1	0	0	1	i7	i6	i5	i4	i3	i2	i1	i0	BA ← imm8	1	↓	-	-	-	x	105
	%BA,[%X]+	1	1	1	1	1	1	1	0	1	1	0	0	0	A ← [X], B ← [X+1], X ← X+2	2	↓	-	-	-	x	106
	%BA,[%Y]+	1	1	1	1	1	1	1	0	1	1	0	1	0	A ← [Y], B ← [Y+1], Y ← Y+2	2	↓	-	-	-	x	106
	%EXT,%BA	1	1	1	1	1	1	1	0	1	0	1	0	X	EXT ← BA	1	↑	-	-	-	x	109
	%EXT,imm8	0	1	0	0	0	i7	i6	i5	i4	i3	i2	i1	i0	EXT ← imm8	1	↑	-	-	-	x	109
	%SP1,%BA	1	1	1	1	1	1	1	0	0	1	0	0	X	SP1 ← BA	1	↓	-	-	-	x	111
	%SP2,%BA	1	1	1	1	1	1	1	0	0	1	1	X	SP2 ← BA	1	↓	-	-	-	x	111	
	%XH,%BA	1	1	1	1	1	1	1	0	0	0	0	1	XH ← BA	1	↓	-	-	-	x	110	
	%XL,%BA	1	1	1	1	1	1	1	0	0	0	0	0	XL ← BA	1	↓	-	-	-	x	110	
	%XL,imm8	0	1	0	1	0	i7	i6	i5	i4	i3	i2	i1	i0	XL ← imm8	1	↓	-	-	-	○	110
	%YH,%BA	1	1	1	1	1	1	1	0	0	0	1	1	YH ← BA	1	↓	-	-	-	x	110	
	%YL,%BA	1	1	1	1	1	1	1	0	0	0	1	0	YL ← BA	1	↓	-	-	-	x	110	
%YL,imm8	0	1	0	1	0	i7	i6	i5	i4	i3	i2	i1	i0	YL ← imm8	1	↓	-	-	-	○	110	
[%X]+,%BA	1	1	1	1	1	1	1	0	1	0	0	1	[X] ← A, [X+1] ← B, X ← X+2	2	↓	-	-	-	x	108		
[%X]+,imm8	0	0	0	0	1	i7	i6	i5	i4	i3	i2	i1	i0	[X] ← i3-0, [X+1] ← i7-4, X ← X+2	2	↓	-	-	-	x	108	
[%Y]+,%BA	1	1	1	1	1	1	1	0	1	0	1	1	[Y] ← A, [Y+1] ← B, Y ← Y+2	2	↓	-	-	-	x	108		
NOP	1	1	1	1	1	1	1	1	1	1	1	1	X	No operation (PC ← PC+1)	1	↓	-	-	-	x	111	
OR	%A,%A	1	1	0	1	1	0	1	1	1	0	0	0	X	A ← A∨A	1	↓	-	-	↓	x	112
	%A,%B	1	1	0	1	1	0	1	1	1	0	0	1	X	A ← A∨B	1	↓	-	-	↓	x	112
	%A,imm4	1	1	0	1	1	0	1	0	0	i3	i2	i1	i0	A ← A∨imm4	1	↓	-	-	↓	x	112
	%A,[%X]	1	1	0	1	1	0	1	1	0	0	0	0	A ← A∨[X]	1	↓	-	-	↓	○	113	
	%A,[%X]+	1	1	0	1	1	0	1	1	0	0	0	1	A ← A∨[X], X ← X+1	1	↓	-	-	↓	x	114	
	%A,[%Y]	1	1	0	1	1	0	1	1	0	0	0	1	0	A ← A∨[Y]	1	↓	-	-	↓	○	113
	%A,[%Y]+	1	1	0	1	1	0	0	1	1	0	0	1	1	A ← A∨[Y], Y ← Y+1	1	↓	-	-	↓	x	114
	%B,%A	1	1	0	1	1	0	1	1	1	0	1	0	X	B ← B∨A	1	↓	-	-	↓	x	112
	%B,%B	1	1	0	1	1	0	1	1	1	0	1	1	X	B ← B∨B	1	↓	-	-	↓	x	112
	%B,imm4	1	1	0	1	1	0	1	0	1	i3	i2	i1	i0	B ← B∨imm4	1	↓	-	-	↓	x	112
	%B,[%X]	1	1	0	1	1	0	1	1	0	0	1	0	0	B ← B∨[X]	1	↓	-	-	↓	○	113
	%B,[%X]+	1	1	0	1	1	0	1	1	0	0	1	0	1	B ← B∨[X], X ← X+1	1	↓	-	-	↓	x	114
	%B,[%Y]	1	1	0	1	1	0	0	1	1	0	0	1	0	B ← B∨[Y]	1	↓	-	-	↓	○	113
	%B,[%Y]+	1	1	0	1	1	0	0	1	1	0	1	1	1	B ← B∨[Y], Y ← Y+1	1	↓	-	-	↓	x	114
	%F,imm4	1	0	0	0	0	1	0	0	1	i3	i2	i1	i0	F ← F∨imm4	1	↑	↑	↑	↑	x	113
	[%X],%A	1	1	0	1	1	0	1	1	0	1	0	0	0	[X] ← [X]∨A	2	↓	-	-	↓	○	114
	[%X],%B	1	1	0	1	1	0	1	1	0	1	1	0	0	[X] ← [X]∨B	2	↓	-	-	↓	○	114
	[%X],imm4	1	1	0	1	1	0	0	0	0	i3	i2	i1	i0	[X] ← [X]∨imm4	2	↓	-	-	↓	○	115
	[%X]+,%A	1	1	0	1	1	0	1	1	0	1	0	0	1	[X] ← [X]∨A, X ← X+1	2	↓	-	-	↓	x	115
	[%X]+,%B	1	1	0	1	1	0	1	1	0	1	1	0	1	[X] ← [X]∨B, X ← X+1	2	↓	-	-	↓	x	115
[%X]+,imm4	1	1	0	1	1	0	0	0	1	i3	i2	i1	i0	[X] ← [X]∨imm4, X ← X+1	2	↓	-	-	↓	x	116	
[%Y],%A	1	1	0	1	1	0	1	1	0	1	0	1	0	[Y] ← [Y]∨A	2	↓	-	-	↓	○	114	
[%Y],%B	1	1	0	1	1	0	1	1	0	1	1	0	0	[Y] ← [Y]∨B	2	↓	-	-	↓	○	114	
[%Y],imm4	1	1	0	1	1	0	0	1	0	i3	i2	i1	i0	[Y] ← [Y]∨imm4	2	↓	-	-	↓	○	115	
[%Y]+,%A	1	1	0	1	1	0	1	1	0	1	1	1	1	[Y] ← [Y]∨A, Y ← Y+1	2	↓	-	-	↓	x	115	
[%Y]+,%B	1	1	0	1	1	0	1	1	0	1	1	1	1	[Y] ← [Y]∨B, Y ← Y+1	2	↓	-	-	↓	x	115	
[%Y]+,imm4	1	1	0	1	1	0	0	1	1	i3	i2	i1	i0	[Y] ← [Y]∨imm4, Y ← Y+1	2	↓	-	-	↓	x	116	
POP	%A	1	1	1	1	1	1	1	0	1	1	1	1	A ← [SP2], SP2 ← SP2+1	1	↓	-	-	-	x	116	
	%B	1	1	1	1	1	1	1	0	1	1	1	0	B ← [SP2], SP2 ← SP2+1	1	↓	-	-	-	x	116	
	%F	1	1	1	1	1	1	1	0	0	1	1	0	1	F ← [SP2], SP2 ← SP2+1	1	↑	↑	↑	↑	x	116
	%X	1	1	1	1	1	1	1	0	0	1	0	0	1	X ← (([SP1*4+3]-[SP1*4]), SP1 ← SP1+1	1	↓	-	-	-	x	117
	%Y	1	1	1	1	1	1	1	0	0	1	0	1	X	Y ← (([SP1*4+3]-[SP1*4]), SP1 ← SP1+1	1	↓	-	-	-	x	117
PUSH	%A	1	1	1	1	1	1	1	0	0	1	1	1	[SP2-1] ← A, SP2 ← SP2-1	1	↓	-	-	-	x	117	
	%B	1	1	1	1	1	1	1	0	0	1	1	0	[SP2-1] ← B, SP2 ← SP2-1	1	↓	-	-	-	x	117	
	%F	1	1	1	1	1	1	1	0	0	1	0	1	0	[SP2-1] ← F, SP2 ← SP2-1	1	↓	-	-	-	x	117
	%X	1	1	1	1	1	1	1	0	0	0	0	1	(((SP1-1)*4+3)-((SP1-1)*4)) ← X, SP1 ← SP1-1	1	↓	-	-	-	x	118	
	%Y	1	1	1	1	1	1	1	0	0	0	1	X	(((SP1-1)*4+3)-((SP1-1)*4)) ← Y, SP1 ← SP1-1	1	↓	-	-	-	x	118	
RET	1	1	1	1	1	1	1	1	0	0	1	0	X	PC ← (([SP1*4+3]-[SP1*4]), SP1 ← SP1+1	1	↓	-	-	-	x	118	
RETD	imm8	1	0	0	0	1	i7	i6	i5	i4	i3	i2	i1	i0	PC ← (([SP1*4+3]-[SP1*4]), SP1 ← SP1+1 [X] ← i3-0, [X+1] ← i7-4, X ← X+2	3	↓	-	-	-	x	119



Mnemonic	Machine code										Operation	Cycle	Flag				EXT. mode	Page				
	12	11	10	9	8	7	6	5	4	3			2	1	0	E			I	C	Z	
RETI		1	1	1	1	1	1	1	1	1	0	0	0	1	PC ← ([SP1*4+3]-[SP1*4]), SP1 ← SP1+1 F ← [SP2], SP2 ← SP2+1	2	↑	↓	↑	↓	×	119
RETS		1	1	1	1	1	1	1	1	1	0	1	1	1	PC ← ([SP1*4+3]-[SP1*4]), SP1 ← SP1+1 PC ← PC+1	2	↓	-	-	-	×	120
RL	%A	1	0	0	0	0	1	1	1	1	0	0	1	0	A (C←D3←D2←D1←D0←C)	1	↓	-	↓	↓	×	120
	%B	1	0	0	0	0	1	1	1	1	0	1	1	0	B (C←D3←D2←D1←D0←C)	1	↓	-	↓	↓	×	120
	[%X]	1	0	0	0	0	1	1	1	0	1	0	0	0	[X] (C←D3←D2←D1←D0←C)	2	↓	-	↓	↓	○	121
	[%X]+	1	0	0	0	0	1	1	1	0	1	0	0	1	[X] (C←D3←D2←D1←D0←C), X ← X+1	2	↓	-	↓	↓	×	121
	[%Y]	1	0	0	0	0	1	1	1	0	1	0	1	0	[Y] (C←D3←D2←D1←D0←C)	2	↓	-	↓	↓	○	121
RR	%A	1	0	0	0	0	1	1	1	1	0	0	1	1	A (C→D3→D2→D1→D0→C)	1	↓	-	↓	↓	×	122
	%B	1	0	0	0	0	1	1	1	1	0	1	1	1	B (C→D3→D2→D1→D0→C)	1	↓	-	↓	↓	×	122
	[%X]	1	0	0	0	0	1	1	1	0	1	1	0	0	[X] (C→D3→D2→D1→D0→C)	2	↓	-	↓	↓	○	122
	[%X]+	1	0	0	0	0	1	1	1	0	1	1	0	1	[X] (C→D3→D2→D1→D0→C), X ← X+1	2	↓	-	↓	↓	×	123
	[%Y]	1	0	0	0	0	1	1	1	0	1	1	1	0	[Y] (C→D3→D2→D1→D0→C)	2	↓	-	↓	↓	○	122
SBC	%A,%A	1	1	0	0	0	1	1	1	1	0	0	0	X	A ← A-A-C	1	↓	-	↓	↓	×	123
	%A,%B	1	1	0	0	0	1	1	1	1	0	0	1	X	A ← A-B-C	1	↓	-	↓	↓	×	123
	%A,imm4	1	1	0	0	0	1	1	0	0	i3	i2	i1	i0	A ← A-imm4-C	1	↓	-	↓	↓	×	124
	%A,[%X]	1	1	0	0	0	1	1	1	0	0	0	0	0	A ← A-[X]-C	1	↓	-	↓	↓	○	124
	%A,[%X]+	1	1	0	0	0	1	1	1	0	0	0	0	1	A ← A-[X]-C, X ← X+1	1	↓	-	↓	↓	×	125
%B,%A	%A,[%Y]	1	1	0	0	0	1	1	1	0	0	0	1	0	A ← A-[Y]-C	1	↓	-	↓	↓	○	124
	%A,[%Y]+	1	1	0	0	0	1	1	1	0	0	0	1	1	A ← A-[Y]-C, Y ← Y+1	1	↓	-	↓	↓	×	125
	%B,%A	1	1	0	0	0	1	1	1	1	0	1	0	X	B ← B-A-C	1	↓	-	↓	↓	×	123
	%B,%A,n4	1	0	0	0	0	1	1	0	0	n3	n2	n1	n0	B ← N's adjust (B-A-C)	2	↓	-	↓	↓	×	127
	%B,%B	1	1	0	0	0	1	1	1	1	0	1	1	X	B ← B-B-C	1	↓	-	↓	↓	×	123
	%B,imm4	1	1	0	0	0	1	1	0	1	i3	i2	i1	i0	B ← B-imm4-C	1	↓	-	↓	↓	×	124
	%B,[%X]	1	1	0	0	0	1	1	1	0	0	1	0	0	B ← B-[X]-C	1	↓	-	↓	↓	○	124
	%B,[%X],n4	1	1	1	0	0	1	1	0	0	n3	n2	n1	n0	B ← N's adjust (B-[X]-C)	2	↓	-	↓	↓	○	128
	%B,[%X]+	1	1	0	0	0	1	1	1	0	0	1	0	1	B ← B-[X]-C, X ← X+1	1	↓	-	↓	↓	×	125
	%B,[%X]+,n4	1	1	1	0	0	1	1	0	1	n3	n2	n1	n0	B ← N's adjust (B-[X]-C), X ← X+1	2	↓	-	↓	↓	×	128
	%B,[%Y]	1	1	0	0	0	1	1	1	0	0	1	1	0	B ← B-[Y]-C	1	↓	-	↓	↓	○	124
	%B,[%Y],n4	1	1	1	0	0	1	1	1	0	n3	n2	n1	n0	B ← N's adjust (B-[Y]-C)	2	↓	-	↓	↓	○	128
	%B,[%Y]+	1	1	0	0	0	1	1	1	0	0	1	1	1	B ← B-[Y]-C, Y ← Y+1	1	↓	-	↓	↓	×	125
	%B,[%Y]+,n4	1	1	1	0	0	1	1	1	1	n3	n2	n1	n0	B ← N's adjust (B-[Y]-C), Y ← Y+1	2	↓	-	↓	↓	×	128
	[%X],%A	1	1	0	0	0	1	1	1	0	1	0	0	0	[X] ← [X]-A-C	2	↓	-	↓	↓	○	125
	[%X],%B	1	1	0	0	0	1	1	1	0	1	1	0	0	[X] ← [X]-B-C	2	↓	-	↓	↓	○	125
	[%X],%B,n4	1	1	1	0	0	1	1	0	0	n3	n2	n1	n0	[X] ← N's adjust ([X]-B-C)	2	↓	-	↓	↓	○	129
	[%X],imm4	1	1	0	0	0	1	0	0	0	i3	i2	i1	i0	[X] ← [X]-imm4-C	2	↓	-	↓	↓	○	126
	[%X],0,n4	1	1	1	0	0	0	0	0	0	n3	n2	n1	n0	[X] ← N's adjust ([X]-0-C)	2	↓	-	↓	↓	○	130
	[%X]+,%A	1	1	0	0	0	1	1	1	0	1	0	0	1	[X] ← [X]-A-C, X ← X+1	2	↓	-	↓	↓	×	126
[%X]+,%B	1	1	0	0	0	1	1	1	0	1	1	0	1	[X] ← [X]-B-C, X ← X+1	2	↓	-	↓	↓	×	126	
[%X]+,%B,n4	1	1	1	0	0	1	1	0	1	n3	n2	n1	n0	[X] ← N's adjust ([X]-B-C), X ← X+1	2	↓	-	↓	↓	×	129	
[%X]+,imm4	1	1	0	0	0	1	0	0	1	i3	i2	i1	i0	[X] ← [X]-imm4-C, X ← X+1	2	↓	-	↓	↓	×	127	
[%X]+,0,n4	1	1	1	0	0	0	0	0	1	n3	n2	n1	n0	[X] ← N's adjust ([X]-0-C), X ← X+1	2	↓	-	↓	↓	×	130	
[%Y],%A	1	1	0	0	0	1	1	1	0	1	0	1	0	[Y] ← [Y]-A-C	2	↓	-	↓	↓	○	125	
[%Y],%B	1	1	0	0	0	1	1	1	0	1	1	1	0	[Y] ← [Y]-B-C	2	↓	-	↓	↓	○	125	
[%Y],%B,n4	1	1	1	0	0	0	1	1	0	n3	n2	n1	n0	[Y] ← N's adjust ([Y]-B-C)	2	↓	-	↓	↓	○	129	
[%Y],imm4	1	1	0	0	0	1	0	1	0	i3	i2	i1	i0	[Y] ← [Y]-imm4-C	2	↓	-	↓	↓	○	126	
[%Y],0,n4	1	1	1	0	0	0	0	1	0	n3	n2	n1	n0	[Y] ← N's adjust ([Y]-0-C)	2	↓	-	↓	↓	○	130	
[%Y]+,%A	1	1	0	0	0	1	1	1	0	1	0	1	1	[Y] ← [Y]-A-C, Y ← Y+1	2	↓	-	↓	↓	×	126	
[%Y]+,%B	1	1	0	0	0	1	1	1	0	1	1	1	1	[Y] ← [Y]-B-C, Y ← Y+1	2	↓	-	↓	↓	×	126	
[%Y]+,%B,n4	1	1	1	0	0	0	1	1	1	n3	n2	n1	n0	[Y] ← N's adjust ([Y]-B-C), Y ← Y+1	2	↓	-	↓	↓	×	130	
[%Y]+,imm4	1	1	0	0	0	1	0	1	1	i3	i2	i1	i0	[Y] ← [Y]-imm4-C, Y ← Y+1	2	↓	-	↓	↓	×	127	
[%Y]+,0,n4	1	1	1	0	0	0	0	1	1	n3	n2	n1	n0	[Y] ← N's adjust ([Y]-0-C), Y ← Y+1	2	↓	-	↓	↓	×	130	
SET	[00addr6],imm2	1	0	1	1	0	i1	i0	a5	a4	a3	a2	a1	a0	[00addr6] ← [00addr6]v(2imm2)	2	↓	-	-	↓	×	131
	[FFaddr6],imm2	1	0	1	1	1	i1	i0	a5	a4	a3	a2	a1	a0	[FFaddr6] ← [FFaddr6]v(2imm2)	2	↓	-	-	↓	×	131
SLL	%A	1	0	0	0	0	1	1	1	1	0	0	0	0	A (C←D3←D2←D1←D0←0)	1	↓	-	↓	↓	×	131
	%B	1	0	0	0	0	1	1	1	1	0	1	0	0	B (C←D3←D2←D1←D0←0)	1	↓	-	↓	↓	×	131

CHAPTER 4: INSTRUCTION SET

Mnemonic	Machine code										Operation	Cycle	Flag			EXT. mode	Page								
	12	11	10	9	8	7	6	5	4	3			2	1	0			E	I	C	Z				
SLL	[%X]	1	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0	[X] (C←D3←D2←D1←D0←0)	2	↓	-	↑	↓	○	132
	[%X]+	1	0	0	0	0	1	1	1	0	0	0	0	0	1	0	[X] (C←D3←D2←D1←D0←0), X ← X+1	2	↓	-	↓	↓	×	132	
	[%Y]	1	0	0	0	0	1	1	1	0	0	0	1	0	0	0	[Y] (C←D3←D2←D1←D0←0)	2	↓	-	↑	↓	○	132	
	[%Y]+	1	0	0	0	0	1	1	1	0	0	0	1	1	0	[Y] (C←D3←D2←D1←D0←0), Y ← Y+1	2	↓	-	↓	↓	×	132		
SLP		1	1	1	1	1	1	1	1	1	0	1	1	0	1	0	Sleep	2	↓	-	-	↓	×	133	
SRL	%A	1	0	0	0	0	1	1	1	1	0	0	0	0	1	0	A (0→D3→D2→D1→D0→C)	1	↓	-	↓	↓	×	133	
	%B	1	0	0	0	0	1	1	1	1	0	1	0	1	0	B (0→D3→D2→D1→D0→C)	1	↓	-	↓	↓	×	133		
	[%X]	1	0	0	0	0	1	1	1	0	0	1	0	0	[X] (0→D3→D2→D1→D0→C)	2	↓	-	↓	↓	○	134			
	[%X]+	1	0	0	0	0	1	1	1	0	0	1	0	1	[X] (0→D3→D2→D1→D0→C), X ← X+1	2	↓	-	↓	↓	×	134			
	[%Y]	1	0	0	0	0	1	1	1	0	0	1	1	0	[Y] (0→D3→D2→D1→D0→C)	2	↓	-	↓	↓	○	134			
	[%Y]+	1	0	0	0	0	1	1	1	0	0	1	1	1	[Y] (0→D3→D2→D1→D0→C), Y ← Y+1	2	↓	-	↓	↓	×	134			
SUB	%A,%A	1	1	0	0	0	0	1	1	1	0	0	0	X	A ← A-A	1	↓	-	↑	↑	×	135			
	%A,%B	1	1	0	0	0	0	1	1	1	0	0	0	X	A ← A-B	1	↓	-	↓	↓	×	135			
	%A,imm4	1	1	0	0	0	0	1	0	1	0	i3	i2	i1	i0	A ← A-imm4	1	↓	-	↓	↓	×	135		
	%A,[%X]	1	1	0	0	0	0	1	1	0	0	0	0	0	A ← A-[X]	1	↓	-	↓	↓	○	136			
	%A,[%X]+	1	1	0	0	0	0	1	1	0	0	0	0	1	A ← A-[X], X ← X+1	1	↓	-	↓	↓	×	136			
	%A,[%Y]	1	1	0	0	0	0	1	1	0	0	0	1	0	A ← A-[Y]	1	↓	-	↓	↓	○	136			
	%A,[%Y]+	1	1	0	0	0	0	1	1	0	0	0	1	1	A ← A-[Y], Y ← Y+1	1	↓	-	↓	↓	×	136			
	%B,%A	1	1	0	0	0	0	1	1	1	0	0	1	X	B ← B-A	1	↓	-	↓	↓	×	135			
	%B,%B	1	1	0	0	0	0	1	1	1	0	1	1	X	B ← B-B	1	↓	-	↓	↑	×	135			
	%B,imm4	1	1	0	0	0	0	1	0	1	0	i3	i2	i1	i0	B ← B-imm4	1	↓	-	↓	↓	×	135		
	%B,[%X]	1	1	0	0	0	0	1	1	0	0	0	1	0	B ← B-[X]	1	↓	-	↓	↓	○	136			
	%B,[%X]+	1	1	0	0	0	0	1	1	0	0	0	1	0	B ← B-[X], X ← X+1	1	↓	-	↓	↓	×	136			
	%B,[%Y]	1	1	0	0	0	0	1	1	0	0	0	1	1	B ← B-[Y]	1	↓	-	↓	↓	○	136			
	%B,[%Y]+	1	1	0	0	0	0	1	1	0	0	1	1	1	B ← B-[Y], Y ← Y+1	1	↓	-	↓	↓	×	136			
	[%X],%A	1	1	0	0	0	0	1	1	0	1	0	0	0	[X] ← [X]-A	2	↓	-	↓	↓	○	137			
	[%X],%B	1	1	0	0	0	0	1	1	0	1	1	0	0	[X] ← [X]-B	2	↓	-	↓	↓	○	137			
	[%X],imm4	1	1	0	0	0	0	0	0	0	0	i3	i2	i1	i0	[X] ← [X]-imm4	2	↓	-	↓	↓	○	138		
	[%X]+,%A	1	1	0	0	0	0	1	1	0	1	0	0	1	[X] ← [X]-A, X ← X+1	2	↓	-	↓	↓	×	137			
	[%X]+,%B	1	1	0	0	0	0	1	1	0	1	1	0	1	[X] ← [X]-B, X ← X+1	2	↓	-	↓	↓	×	137			
	[%X]+,imm4	1	1	0	0	0	0	0	0	1	i3	i2	i1	i0	[X] ← [X]-imm4, X ← X+1	2	↓	-	↓	↓	×	138			
	[%Y],%A	1	1	0	0	0	0	1	1	0	1	0	1	0	[Y] ← [Y]-A	2	↓	-	↓	↓	○	137			
	[%Y],%B	1	1	0	0	0	0	1	1	0	1	1	1	0	[Y] ← [Y]-B	2	↓	-	↓	↓	○	137			
	[%Y],imm4	1	1	0	0	0	0	0	1	0	i3	i2	i1	i0	[Y] ← [Y]-imm4	2	↓	-	↓	↓	○	138			
	[%Y]+,%A	1	1	0	0	0	0	1	1	0	1	0	1	1	[Y] ← [Y]-A, Y ← Y+1	2	↓	-	↓	↓	×	137			
[%Y]+,%B	1	1	0	0	0	0	1	1	0	1	1	1	1	[Y] ← [Y]-B, Y ← Y+1	2	↓	-	↓	↓	×	137				
[%Y]+,imm4	1	1	0	0	0	0	0	1	1	i3	i2	i1	i0	[Y] ← [Y]-imm4, Y ← Y+1	2	↓	-	↓	↓	×	138				
TST	[00addr6],imm2	1	0	0	1	0	i1	i0	a5	a4	a3	a2	a1	a0	[00addr6]^(2imm2)	1	↓	-	-	↓	×	139			
	[FFaddr6],imm2	1	0	0	1	1	i1	i0	a5	a4	a3	a2	a1	a0	[FFaddr6]^(2imm2)	1	↓	-	-	↓	×	139			
XOR	%A,%A	1	1	0	1	1	1	1	1	1	0	0	0	X	A ← A∨A	1	↓	-	-	↑	×	139			
	%A,%B	1	1	0	1	1	1	1	1	1	0	0	1	X	A ← A∨B	1	↓	-	-	↓	×	139			
	%A,imm4	1	1	0	1	1	1	1	0	0	i3	i2	i1	i0	A ← A∨imm4	1	↓	-	-	↓	×	140			
	%A,[%X]	1	1	0	1	1	1	1	0	0	0	0	0	0	A ← A∨[X]	1	↓	-	-	↓	○	141			
	%A,[%X]+	1	1	0	1	1	1	1	0	0	0	0	0	1	A ← A∨[X], X ← X+1	1	↓	-	-	↓	×	141			
	%A,[%Y]	1	1	0	1	1	1	1	0	0	0	0	1	0	A ← A∨[Y]	1	↓	-	-	↓	○	141			
	%A,[%Y]+	1	1	0	1	1	1	1	0	0	0	1	1	0	A ← A∨[Y], Y ← Y+1	1	↓	-	-	↓	×	141			
	%B,%A	1	1	0	1	1	1	1	1	1	0	0	1	X	B ← B∨A	1	↓	-	-	↓	×	139			
	%B,%B	1	1	0	1	1	1	1	1	1	0	1	1	X	B ← B∨B	1	↓	-	-	↑	×	139			
	%B,imm4	1	1	0	1	1	1	0	1	i3	i2	i1	i0	B ← B∨imm4	1	↓	-	-	↓	×	140				
	%B,[%X]	1	1	0	1	1	1	1	0	0	0	1	0	0	B ← B∨[X]	1	↓	-	-	↓	○	141			
	%B,[%X]+	1	1	0	1	1	1	1	0	0	0	1	0	1	B ← B∨[X], X ← X+1	1	↓	-	-	↓	×	141			
	%B,[%Y]	1	1	0	1	1	1	1	0	0	0	1	1	0	B ← B∨[Y]	1	↓	-	-	↓	○	141			
	%B,[%Y]+	1	1	0	1	1	1	1	0	0	1	1	1	1	B ← B∨[Y], Y ← Y+1	1	↓	-	-	↓	×	141			
	%F,imm4	1	0	0	0	0	1	0	1	0	i3	i2	i1	i0	F ← F∨imm4	1	↓	↓	↓	↓	×	140			
	[%X],%A	1	1	0	1	1	1	1	0	1	0	0	0	0	[X] ← [X]∨A	2	↓	-	-	↓	○	142			
	[%X],%B	1	1	0	1	1	1	1	0	1	0	0	0	0	[X] ← [X]∨B	2	↓	-	-	↓	○	142			
	[%X],imm4	1	1	0	1	1	1	0	0	0	i3	i2	i1	i0	[X] ← [X]∨imm4	2	↓	-	-	↓	○	143			
	[%X]+,%A	1	1	0	1	1	1	1	0	1	0	0	1	0	[X] ← [X]∨A, X ← X+1	2	↓	-	-	↓	×	142			
	[%X]+,%B	1	1	0	1	1	1	1	0	1	0	1	0	1	[X] ← [X]∨B, X ← X+1	2	↓	-	-	↓	×	142			
	[%X]+,imm4	1	1	0	1	1	0	0	1	i3	i2	i1	i0	[X] ← [X]∨imm4, X ← X+1	2	↓	-	-	↓	×	143				
	[%Y],%A	1	1	0	1	1	1	1	0	1	0	0	1	0	[Y] ← [Y]∨A	2	↓	-	-	↓	○	142			
	[%Y],%B	1	1	0	1	1	1	1	0	1	1	0	0	0	[Y] ← [Y]∨B	2	↓	-	-	↓	○	142			
	[%Y],imm4	1	1	0	1	1	0	1	0	0	i3	i2	i1	i0	[Y] ← [Y]∨imm4	2	↓	-	-	↓	○	143			
[%Y]+,%A	1	1	0	1	1	1	1	0	1	0	1	1	0	[Y] ← [Y]∨A, Y ← Y+1	2	↓	-	-	↓	×	142				
[%Y]+,%B	1	1	0	1	1	1	1	0	1	1	1	1	1	[Y] ← [Y]∨B, Y ← Y+1	2	↓	-	-	↓	×	142				
[%Y]+,imm4	1	1	0	1	1	0	1	1	i3	i2	i1	i0	[Y] ← [Y]∨imm4, Y ← Y+1	2	↓	-	-	↓	×	143					

### 4.2.5 List of extended addressing instructions

8-bit absolute addressing (1/4)

	Mnemonic	Operation	Flag			
			E	I	C	Z
LDB	%EXT,imm8					
LD	%A,[%X]	$A \leftarrow [00imm8]$ (00imm8 = 0000H ~ 00FFH)	↓	-	-	-
LDB	%EXT,imm8					
LD	%A,[%Y]	$A \leftarrow [FFimm8]$ (FFimm8 = FF00H + 00H ~ FFH)	↓	-	-	-
LDB	%EXT,imm8					
LD	%B,[%X]	$B \leftarrow [00imm8]$	↓	-	-	-
LDB	%EXT,imm8					
LD	%B,[%Y]	$B \leftarrow [FFimm8]$	↓	-	-	-
LDB	%EXT,imm8					
LD	[%X],%A	$[00imm8] \leftarrow A$	↓	-	-	-
LDB	%EXT,imm8					
LD	[%X],%B	$[00imm8] \leftarrow B$	↓	-	-	-
LDB	%EXT,imm8					
LD	[%X],imm4	$[00imm8] \leftarrow imm4$	↓	-	-	-
LDB	%EXT,imm8					
LD	[%Y],%A	$[FFimm8] \leftarrow A$	↓	-	-	-
LDB	%EXT,imm8					
LD	[%Y],%B	$[FFimm8] \leftarrow B$	↓	-	-	-
LDB	%EXT,imm8					
LD	[%Y],imm4	$[FFimm8] \leftarrow imm4$	↓	-	-	-
LDB	%EXT,imm8					
EX	%A,[%X]	$A \leftrightarrow [00imm8]$	↓	-	-	-
LDB	%EXT,imm8					
EX	%A,[%Y]	$A \leftrightarrow [FFimm8]$	↓	-	-	-
LDB	%EXT,imm8					
EX	%B,[%X]	$B \leftrightarrow [00imm8]$	↓	-	-	-
LDB	%EXT,imm8					
EX	%B,[%Y]	$B \leftrightarrow [FFimm8]$	↓	-	-	-
LDB	%EXT,imm8					
ADD	%A,[%X]	$A \leftarrow A + [00imm8]$	↓	-	↑	↓
LDB	%EXT,imm8					
ADD	%A,[%Y]	$A \leftarrow A + [FFimm8]$	↓	-	↑	↓
LDB	%EXT,imm8					
ADD	%B,[%X]	$B \leftarrow B + [00imm8]$	↓	-	↑	↓
LDB	%EXT,imm8					
ADD	%B,[%Y]	$B \leftarrow B + [FFimm8]$	↓	-	↑	↓
LDB	%EXT,imm8					
ADD	[%X],%A	$[00imm8] \leftarrow [00imm8] + A$	↓	-	↑	↓
LDB	%EXT,imm8					
ADD	[%X],%B	$[00imm8] \leftarrow [00imm8] + B$	↓	-	↑	↓
LDB	%EXT,imm8					
ADD	[%X],imm4	$[00imm8] \leftarrow [00imm8] + imm4$	↓	-	↑	↓
LDB	%EXT,imm8					
ADD	[%Y],%A	$[FFimm8] \leftarrow [FFimm8] + A$	↓	-	↑	↓
LDB	%EXT,imm8					
ADD	[%Y],%B	$[FFimm8] \leftarrow [FFimm8] + B$	↓	-	↑	↓
LDB	%EXT,imm8					
ADD	[%Y],imm4	$[FFimm8] \leftarrow [FFimm8] + imm4$	↓	-	↑	↓
LDB	%EXT,imm8					
ADC	%A,[%X]	$A \leftarrow A + [00imm8] + C$	↓	-	↑	↓
LDB	%EXT,imm8					
ADC	%A,[%Y]	$A \leftarrow A + [FFimm8] + C$	↓	-	↑	↓
LDB	%EXT,imm8					
ADC	%B,[%X]	$B \leftarrow B + [00imm8] + C$	↓	-	↑	↓
LDB	%EXT,imm8					
ADC	%B,[%Y]	$B \leftarrow B + [FFimm8] + C$	↓	-	↑	↓
LDB	%EXT,imm8					
ADC	[%X],%A	$[00imm8] \leftarrow [00imm8] + A + C$	↓	-	↑	↓
LDB	%EXT,imm8					
ADC	[%X],%B	$[00imm8] \leftarrow [00imm8] + B + C$	↓	-	↑	↓
LDB	%EXT,imm8					
ADC	[%X],imm4	$[00imm8] \leftarrow [00imm8] + imm4 + C$	↓	-	↑	↓

## CHAPTER 4: INSTRUCTION SET

### 8-bit absolute addressing (2/4)

Mnemonic		Operation	Flag E I C Z
LDB	%EXT,imm8		
ADC	[%Y],%A	[FFimm8] ← [FFimm8] + A + C	↓ - ↓ ↓
LDB	%EXT,imm8		
ADC	[%Y],%B	[FFimm8] ← [FFimm8] + B + C	↓ - ↓ ↓
LDB	%EXT,imm8		
ADC	[%Y],imm4	[FFimm8] ← [FFimm8] + imm4 + C	↓ - ↓ ↓
LDB	%EXT,imm8		
SUB	%A,[%X]	A ← A - [00imm8] (00imm8 = 0000H ~ 00FFH)	↓ - ↓ ↓
LDB	%EXT,imm8		
SUB	%A,[%Y]	A ← A - [FFimm8] (FFimm8 = FF00H + 00H ~ FFH)	↓ - ↓ ↓
LDB	%EXT,imm8		
SUB	%B,[%X]	B ← B - [00imm8]	↓ - ↓ ↓
LDB	%EXT,imm8		
SUB	%B,[%Y]	B ← B - [FFimm8]	↓ - ↓ ↓
LDB	%EXT,imm8		
SUB	[%X],%A	[00imm8] ← [00imm8] - A	↓ - ↓ ↓
LDB	%EXT,imm8		
SUB	[%X],%B	[00imm8] ← [00imm8] - B	↓ - ↓ ↓
LDB	%EXT,imm8		
SUB	[%X],imm4	[00imm8] ← [00imm8] - imm4	↓ - ↓ ↓
LDB	%EXT,imm8		
SUB	[%Y],%A	[FFimm8] ← [FFimm8] - A	↓ - ↓ ↓
LDB	%EXT,imm8		
SUB	[%Y],%B	[FFimm8] ← [FFimm8] - B	↓ - ↓ ↓
LDB	%EXT,imm8		
SUB	[%Y],imm4	[FFimm8] ← [FFimm8] - imm4	↓ - ↓ ↓
LDB	%EXT,imm8		
SBC	%A,[%X]	A ← A - [00imm8] - C	↓ - ↓ ↓
LDB	%EXT,imm8		
SBC	%A,[%Y]	A ← A - [FFimm8] - C	↓ - ↓ ↓
LDB	%EXT,imm8		
SBC	%B,[%X]	B ← B - [00imm8] - C	↓ - ↓ ↓
LDB	%EXT,imm8		
SBC	%B,[%Y]	B ← B - [FFimm8] - C	↓ - ↓ ↓
LDB	%EXT,imm8		
SBC	[%X],%A	[00imm8] ← [00imm8] - A - C	↓ - ↓ ↓
LDB	%EXT,imm8		
SBC	[%X],%B	[00imm8] ← [00imm8] - B - C	↓ - ↓ ↓
LDB	%EXT,imm8		
SBC	[%X],imm4	[00imm8] ← [00imm8] - imm4 - C	↓ - ↓ ↓
LDB	%EXT,imm8		
SBC	[%Y],%A	[FFimm8] ← [FFimm8] - A - C	↓ - ↓ ↓
LDB	%EXT,imm8		
SBC	[%Y],%B	[FFimm8] ← [FFimm8] - B - C	↓ - ↓ ↓
LDB	%EXT,imm8		
SBC	[%Y],imm4	[FFimm8] ← [FFimm8] - imm4 - C	↓ - ↓ ↓
LDB	%EXT,imm8		
CMP	%A,[%X]	A - [00imm8]	↓ - ↓ ↓
LDB	%EXT,imm8		
CMP	%A,[%Y]	A - [FFimm8]	↓ - ↓ ↓
LDB	%EXT,imm8		
CMP	%B,[%X]	B - [00imm8]	↓ - ↓ ↓
LDB	%EXT,imm8		
CMP	%B,[%Y]	B - [FFimm8]	↓ - ↓ ↓
LDB	%EXT,imm8		
CMP	[%X],%A	[00imm8] - A	↓ - ↓ ↓
LDB	%EXT,imm8		
CMP	[%X],%B	[00imm8] - B	↓ - ↓ ↓
LDB	%EXT,imm8		
CMP	[%X],imm4	[00imm8] - imm4	↓ - ↓ ↓
LDB	%EXT,imm8		
CMP	[%Y],%A	[FFimm8] - A	↓ - ↓ ↓
LDB	%EXT,imm8		
CMP	[%Y],%B	[FFimm8] - B	↓ - ↓ ↓
LDB	%EXT,imm8		
CMP	[%Y],imm4	[FFimm8] - imm4	↓ - ↓ ↓

## 8-bit absolute addressing (3/4)

	Mnemonic	Operation	Flag			
			E	I	C	Z
LDB	%EXT,imm8					
ADC	%B,[%X],n4	$B \leftarrow N\text{'s adjust } (B + [00imm8] + C)$ (00imm8 = 0000H ~ 00FFH)	↓	-	↑	↓
LDB	%EXT,imm8					
ADC	%B,[%Y],n4	$B \leftarrow N\text{'s adjust } (B + [FFimm8] + C)$ (FFimm8 = FF00H + 00H ~ FFH)	↓	-	↑	↓
LDB	%EXT,imm8					
ADC	[%X],%B,n4	$[00imm8] \leftarrow N\text{'s adjust } ([00imm8] + B + C)$	↓	-	↑	↓
LDB	%EXT,imm8					
ADC	[%X],0,n4	$[00imm8] \leftarrow N\text{'s adjust } ([00imm8] + 0 + C)$	↓	-	↑	↓
LDB	%EXT,imm8					
ADC	[%Y],%B,n4	$[FFimm8] \leftarrow N\text{'s adjust } ([FFimm8] + B + C)$	↓	-	↑	↓
LDB	%EXT,imm8					
ADC	[%Y],0,n4	$[FFimm8] \leftarrow N\text{'s adjust } ([FFimm8] + 0 + C)$	↓	-	↑	↓
LDB	%EXT,imm8					
SBC	%B,[%X],n4	$B \leftarrow N\text{'s adjust } (B - [00imm8] - C)$	↓	-	↑	↓
LDB	%EXT,imm8					
SBC	%B,[%Y],n4	$B \leftarrow N\text{'s adjust } (B - [FFimm8] - C)$	↓	-	↑	↓
LDB	%EXT,imm8					
SBC	[%X],%B,n4	$[00imm8] \leftarrow N\text{'s adjust } ([00imm8] - B - C)$	↓	-	↑	↓
LDB	%EXT,imm8					
SBC	[%X],0,n4	$[00imm8] \leftarrow N\text{'s adjust } ([00imm8] - 0 - C)$	↓	-	↑	↓
LDB	%EXT,imm8					
SBC	[%Y],%B,n4	$[FFimm8] \leftarrow N\text{'s adjust } ([FFimm8] - B - C)$	↓	-	↑	↓
LDB	%EXT,imm8					
SBC	[%Y],0,n4	$[FFimm8] \leftarrow N\text{'s adjust } ([FFimm8] - 0 - C)$	↓	-	↑	↓
LDB	%EXT,imm8					
INC	[%X],n4	$[00imm8] \leftarrow N\text{'s adjust } ([00imm8] + 1)$	↓	-	↑	↓
LDB	%EXT,imm8					
INC	[%Y],n4	$[FFimm8] \leftarrow N\text{'s adjust } ([FFimm8] + 1)$	↓	-	↑	↓
LDB	%EXT,imm8					
DEC	[%X],n4	$[00imm8] \leftarrow N\text{'s adjust } ([00imm8] - 1)$	↓	-	↑	↓
LDB	%EXT,imm8					
DEC	[%Y],n4	$[FFimm8] \leftarrow N\text{'s adjust } ([FFimm8] - 1)$	↓	-	↑	↓
LDB	%EXT,imm8					
AND	%A,[%X]	$A \leftarrow A \wedge [00imm8]$	↓	-	-	↓
LDB	%EXT,imm8					
AND	%A,[%Y]	$A \leftarrow A \wedge [FFimm8]$	↓	-	-	↓
LDB	%EXT,imm8					
AND	%B,[%X]	$B \leftarrow B \wedge [00imm8]$	↓	-	-	↓
LDB	%EXT,imm8					
AND	%B,[%Y]	$B \leftarrow B \wedge [FFimm8]$	↓	-	-	↓
LDB	%EXT,imm8					
AND	[%X],%A	$[00imm8] \leftarrow [00imm8] \wedge A$	↓	-	-	↓
LDB	%EXT,imm8					
AND	[%X],%B	$[00imm8] \leftarrow [00imm8] \wedge B$	↓	-	-	↓
LDB	%EXT,imm8					
AND	[%X],imm4	$[00imm8] \leftarrow [00imm8] \wedge imm4$	↓	-	-	↓
LDB	%EXT,imm8					
AND	[%Y],%A	$[FFimm8] \leftarrow [FFimm8] \wedge A$	↓	-	-	↓
LDB	%EXT,imm8					
AND	[%Y],%B	$[FFimm8] \leftarrow [FFimm8] \wedge B$	↓	-	-	↓
LDB	%EXT,imm8					
AND	[%Y],imm4	$[FFimm8] \leftarrow [FFimm8] \wedge imm4$	↓	-	-	↓
LDB	%EXT,imm8					
OR	%A,[%X]	$A \leftarrow A \vee [00imm8]$	↓	-	-	↓
LDB	%EXT,imm8					
OR	%A,[%Y]	$A \leftarrow A \vee [FFimm8]$	↓	-	-	↓
LDB	%EXT,imm8					
OR	%B,[%X]	$B \leftarrow B \vee [00imm8]$	↓	-	-	↓
LDB	%EXT,imm8					
OR	%B,[%Y]	$B \leftarrow B \vee [FFimm8]$	↓	-	-	↓
LDB	%EXT,imm8					
OR	[%X],%A	$[00imm8] \leftarrow [00imm8] \vee A$	↓	-	-	↓
LDB	%EXT,imm8					
OR	[%X],%B	$[00imm8] \leftarrow [00imm8] \vee B$	↓	-	-	↓
LDB	%EXT,imm8					
OR	[%X],imm4	$[00imm8] \leftarrow [00imm8] \vee imm4$	↓	-	-	↓

8-bit absolute addressing (4/4)

Mnemonic		Operation	Flag			
			E	I	C	Z
LDB	%EXT,imm8					
OR	[%Y],%A	$[FFimm8] \leftarrow [FFimm8] \vee A$ (FFimm8 = FF00H + 00H ~ FFH)	↓	–	–	↑
LDB	%EXT,imm8					
OR	[%Y],%B	$[FFimm8] \leftarrow [FFimm8] \vee B$	↓	–	–	↑
LDB	%EXT,imm8					
OR	[%Y],imm4	$[FFimm8] \leftarrow [FFimm8] \vee imm4$	↓	–	–	↑
LDB	%EXT,imm8					
XOR	%A,[%X]	$A \leftarrow A \vee [00imm8]$ (00imm8 = 0000H ~ 00FFH)	↓	–	–	↑
LDB	%EXT,imm8					
XOR	%A,[%Y]	$A \leftarrow A \vee [FFimm8]$	↓	–	–	↑
LDB	%EXT,imm8					
XOR	%B,[%X]	$B \leftarrow B \vee [00imm8]$	↓	–	–	↑
LDB	%EXT,imm8					
XOR	%B,[%Y]	$B \leftarrow B \vee [FFimm8]$	↓	–	–	↑
LDB	%EXT,imm8					
XOR	[%X],%A	$[00imm8] \leftarrow [00imm8] \vee A$	↓	–	–	↑
LDB	%EXT,imm8					
XOR	[%X],%B	$[00imm8] \leftarrow [00imm8] \vee B$	↓	–	–	↑
LDB	%EXT,imm8					
XOR	[%X],imm4	$[00imm8] \leftarrow [00imm8] \vee imm4$	↓	–	–	↑
LDB	%EXT,imm8					
XOR	[%Y],%A	$[FFimm8] \leftarrow [FFimm8] \vee A$	↓	–	–	↑
LDB	%EXT,imm8					
XOR	[%Y],%B	$[FFimm8] \leftarrow [FFimm8] \vee B$	↓	–	–	↑
LDB	%EXT,imm8					
XOR	[%Y],imm4	$[FFimm8] \leftarrow [FFimm8] \vee imm4$	↓	–	–	↑
LDB	%EXT,imm8					
BIT	%A,[%X]	$A \wedge [00imm8]$	↓	–	–	↑
LDB	%EXT,imm8					
BIT	%A,[%Y]	$A \wedge [FFimm8]$	↓	–	–	↑
LDB	%EXT,imm8					
BIT	%B,[%X]	$B \wedge [00imm8]$	↓	–	–	↑
LDB	%EXT,imm8					
BIT	%B,[%Y]	$B \wedge [FFimm8]$	↓	–	–	↑
LDB	%EXT,imm8					
BIT	[%X],%A	$[00imm8] \wedge A$	↓	–	–	↑
LDB	%EXT,imm8					
BIT	[%X],%B	$[00imm8] \wedge B$	↓	–	–	↑
LDB	%EXT,imm8					
BIT	[%X],imm4	$[00imm8] \wedge imm4$	↓	–	–	↑
LDB	%EXT,imm8					
BIT	[%Y],%A	$[FFimm8] \wedge A$	↓	–	–	↑
LDB	%EXT,imm8					
BIT	[%Y],%B	$[FFimm8] \wedge B$	↓	–	–	↑
LDB	%EXT,imm8					
BIT	[%Y],imm4	$[FFimm8] \wedge imm4$	↓	–	–	↑
LDB	%EXT,imm8					
SLL	[%X]	$[00imm8]$ (C ← D3 ← D2 ← D1 ← D0 ← 0)	↓	–	↑	↑
LDB	%EXT,imm8					
SLL	[%Y]	$[FFimm8]$ (C ← D3 ← D2 ← D1 ← D0 ← 0)	↓	–	↑	↑
LDB	%EXT,imm8					
SRL	[%X]	$[00imm8]$ (0 → D3 → D2 → D1 → D0 → C)	↓	–	↑	↑
LDB	%EXT,imm8					
SRL	[%Y]	$[FFimm8]$ (0 → D3 → D2 → D1 → D0 → C)	↓	–	↑	↑
LDB	%EXT,imm8					
RL	[%X]	$[00imm8]$ (C ← D3 ← D2 ← D1 ← D0 ← C)	↓	–	↑	↑
LDB	%EXT,imm8					
RL	[%Y]	$[FFimm8]$ (C ← D3 ← D2 ← D1 ← D0 ← C)	↓	–	↑	↑
LDB	%EXT,imm8					
RR	[%X]	$[00imm8]$ (C → D3 → D2 → D1 → D0 → C)	↓	–	↑	↑
LDB	%EXT,imm8					
RR	[%Y]	$[FFimm8]$ (C → D3 → D2 → D1 → D0 → C)	↓	–	↑	↑

## 16-bit immediate data addressing

Mnemonic	Operation	Flag			
		E	I	C	Z
LDB %EXT,imm8 *1					
LDB %XL,imm8 *2	$X \leftarrow \text{imm16}$ (*1 is upper 8-bit, *2 is lower 8-bit)	↓	-	-	-
LDB %EXT,imm8 *1					
LDB %YL,imm8 *2	$Y \leftarrow \text{imm16}$ (*1 is upper 8-bit, *2 is lower 8-bit)	↓	-	-	-
LDB %EXT,imm8 *1					
ADD %X,sign8 *2	$X \leftarrow X + \text{imm16}$ (*1 is upper 8-bit, *2 is lower 8-bit)	↓	-	-	↑
LDB %EXT,imm8 *1					
ADD %Y,sign8 *2	$Y \leftarrow Y + \text{imm16}$ (*1 is upper 8-bit, *2 is lower 8-bit)	↓	-	-	↑
LDB %EXT,imm8 *1					
CMP %X,imm8 *2	$X - \text{imm16}$ (FFH - *1 is upper 8-bit, *2 is lower 8-bit)	↓	-	↑	↓
LDB %EXT,imm8 *1					
CMP %Y,imm8 *2	$Y - \text{imm16}$ (FFH - *1 is upper 8-bit, *2 is lower 8-bit)	↓	-	↑	↓

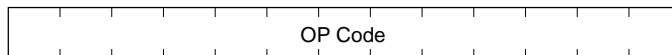
## signed 16-bit PC relative addressing

Mnemonic	Operation	Flag			
		E	I	C	Z
LDB %EXT,imm8	(sign16 : imm8 is upper 8-bit, sign8 is lower 8-bit)				
JR sign8	$PC \leftarrow PC + \text{sign16} + 1$ (sign16 = 32767~-32768)	↓	-	-	-
LDB %EXT,imm8					
JRC sign8	If C = 1 then $PC \leftarrow PC + \text{sign16} + 1$ (sign16 = 32767 ~ -32768)	↓	-	-	-
LDB %EXT,imm8					
JRNC sign8	If C = 0 then $PC \leftarrow PC + \text{sign16} + 1$ (sign16 = 32767 ~ -32768)	↓	-	-	-
LDB %EXT,imm8					
JRZ sign8	If Z = 1 then $PC \leftarrow PC + \text{sign16} + 1$ (sign16 = 32767 ~ -32768)	↓	-	-	-
LDB %EXT,imm8					
JRNZ sign8	If Z = 0 then $PC \leftarrow PC + \text{sign16} + 1$ (sign16 = 32767 ~ -32768)	↓	-	-	-
LDB %EXT,imm8					
CALR sign8	( [SP1 - 1 *4 + 3] ~ [ (SP1 - 1) *4] ) $\leftarrow PC + 1$ , SP1 $\leftarrow$ SP1 - 1 $PC \leftarrow PC + \text{sign16} + 1$ (sign16 = 32767 ~ -32768)	↓	-	-	-

### 4.3 Instruction Formats

All the instructions of the E0C63000 are configured with 1 word (13 bits) as follows:

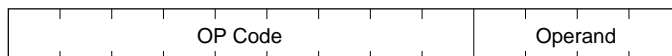
## I 13-bit operation code



Examples:

LD %A,%B  
ADD %A,[%X]  
PUSH %F

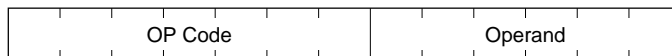
## II 9-bit operation code + 4-bit immediate data



Examples:

LD %A,imm4  
ADC [%Y],%B,n4  
BIT %B,imm4

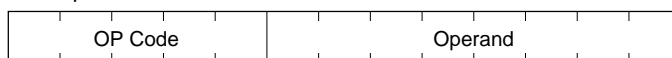
## III 7-bit operation code + 6-bit immediate data



Examples:

INC [addr6]  
CALR [addr6]  
INT imm6

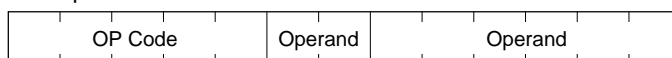
## IV 5-bit operation code + 8-bit immediate data



Examples:

LDB %BA,imm8  
CALZ imm8  
JR sign8

## V 5-bit operation code + 2-bit immediate data + 6-bit immediate data



Examples:

CLR [addr6],imm2  
SET [addr6],imm2  
TST [addr6],imm2

### 4.4 Detailed Explanation of Instructions

This section explains the individual instructions in alphabetic order according to the following format.

View of the explanation

	Mnemonic	Mnemonic meaning	Number of bus cycles																																																																											
	<b>ADC %r,%r'</b>	<i>Add with carry r' reg. to r reg.</i>	<i>1 cycle</i>																																																																											
<b>Function explanation</b>	<p><i>Function:</i> <math>r \leftarrow r + r' + C</math>                  Adds the content of the r' register (A or B) and carry (C) to the r register (A or B).</p>																																																																													
<b>Mnemonic and object codes</b>	<p><i>Code:</i></p> <table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th style="width: 10%;">Mnemonic</th> <th colspan="10" style="text-align: center;">MSB</th> <th colspan="4" style="text-align: center;">LSB</th> </tr> </thead> <tbody> <tr> <td>ADC %A,%A</td> <td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>X</td> <td>19F0H, (19F1H)</td> </tr> <tr> <td>ADC %A,%B</td> <td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>X</td> <td>19F2H, (19F3H)</td> </tr> <tr> <td>ADC %B,%A</td> <td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>X</td> <td>19F4H, (19F5H)</td> </tr> <tr> <td>ADC %B,%B</td> <td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td><td>X</td> <td>19F6H, (19F7H)</td> </tr> </tbody> </table>			Mnemonic	MSB										LSB				ADC %A,%A	1	1	0	0	1	1	1	1	1	0	0	0	X	19F0H, (19F1H)	ADC %A,%B	1	1	0	0	1	1	1	1	1	0	0	1	X	19F2H, (19F3H)	ADC %B,%A	1	1	0	0	1	1	1	1	1	0	1	0	X	19F4H, (19F5H)	ADC %B,%B	1	1	0	0	1	1	1	1	1	0	1	1	X	19F6H, (19F7H)
Mnemonic	MSB										LSB																																																																			
ADC %A,%A	1	1	0	0	1	1	1	1	1	0	0	0	X	19F0H, (19F1H)																																																																
ADC %A,%B	1	1	0	0	1	1	1	1	1	0	0	1	X	19F2H, (19F3H)																																																																
ADC %B,%A	1	1	0	0	1	1	1	1	1	0	1	0	X	19F4H, (19F5H)																																																																
ADC %B,%B	1	1	0	0	1	1	1	1	1	0	1	1	X	19F6H, (19F7H)																																																																
<b>Addressing mode</b>	<p><i>Mode:</i> Src: Register direct                  Dst: Register direct                  Extended addressing: Invalid</p>																																																																													
<b>Src indicates the source and Dst indicates the destination</b>	<p><i>Flags:</i></p> <table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th style="width: 10%;">E</th> <th style="width: 10%;">I</th> <th style="width: 10%;">C</th> <th style="width: 10%;">Z</th> </tr> </thead> <tbody> <tr> <td style="text-align: center;">↓</td> <td style="text-align: center;">-</td> <td style="text-align: center;">↑</td> <td style="text-align: center;">↑</td> </tr> </tbody> </table> <p style="margin-left: 20px;"><b>Status of the flag</b>                  - Does not change                  ↓ Reset                  ↑ Set                  ↑↓ Set/reset</p>			E	I	C	Z	↓	-	↑	↑																																																																			
E	I	C	Z																																																																											
↓	-	↑	↑																																																																											

The meaning of the symbols are the same as for the instruction list.  
 The following symbols are used to explain two or more registers as aggregations.

- r* ..... Data registers A, B, or flag register F
- ir* ..... Index registers X or Y
- rr* ..... Index registers XL, XH, YL or YH
- sp* ..... Stack pointers SP1 or SP2



**ADC %r,%r'***Add with carry r' reg. to r reg.**1 cycle***Function:**  $r \leftarrow r + r' + C$ 

Adds the content of the r' register (A or B) and carry (C) to the r register (A or B).

**Code:**

Mnemonic	MSB										LSB			
ADC %A,%A	1	1	0	0	1	1	1	1	1	0	0	0	X	19F0H, (19F1H)
ADC %A,%B	1	1	0	0	1	1	1	1	1	0	0	1	X	19F2H, (19F3H)
ADC %B,%A	1	1	0	0	1	1	1	1	1	0	1	0	X	19F4H, (19F5H)
ADC %B,%B	1	1	0	0	1	1	1	1	1	0	1	1	X	19F6H, (19F7H)

**Flags:**

E	I	C	Z
↓	–	↑	↑

**Mode:**

Src: Register direct

Dst: Register direct

Extended addressing: Invalid

**ADC %r,imm4***Add with carry immediate data imm4 to r reg.**1 cycle***Function:**  $r \leftarrow r + \text{imm4} + C$ 

Adds the 4-bit immediate data imm4 and carry (C) to the r register (A or B).

**Code:**

Mnemonic	MSB										LSB			
ADC %A,imm4	1	1	0	0	1	1	1	0	0	i3	i2	i1	i0	19C0H–19CFH
ADC %B,imm4	1	1	0	0	1	1	1	0	1	i3	i2	i1	i0	19D0H–19DFH

**Flags:**

E	I	C	Z
↓	–	↑	↑

**Mode:**

Src: Immediate data

Dst: Register direct

Extended addressing: Invalid

**ADC %r,[%ir]**

*Add with carry location [ir reg.] to r reg.*

*1 cycle*

**Function:**  $r \leftarrow r + [ir] + C$

Adds the content of the data memory addressed by the ir register (X or Y) and carry (C) to the r register (A or B).

**Code:**

Mnemonic	MSB												LSB			
ADC %A,[%X]	1	1	0	0	1	1	1	1	0	0	0	0	0	19E0H		
ADC %A,[%Y]	1	1	0	0	1	1	1	1	0	0	0	1	0	19E2H		
ADC %B,[%X]	1	1	0	0	1	1	1	1	0	0	1	0	0	19E4H		
ADC %B,[%Y]	1	1	0	0	1	1	1	1	0	0	1	1	0	19E6H		

**Flags:**

E	I	C	Z
↓	–	↑	↑

**Mode:** Src: Register indirect  
 Dst: Register direct  
 Extended addressing: Valid

**Extended LDB** %EXT,imm8

**operation:** ADC %r,[%X]  $r \leftarrow r + [00imm8] + C$  (00imm8 = 0000H + 00H to FFH)

LDB %EXT,imm8

ADC %r,[%Y]  $r \leftarrow r + [FFimm8] + C$  (FFimm8 = FF00H + 00H to FFH)

**ADC %r,[%ir]+**

*Add with carry location [ir reg.] to r reg. and increment ir reg.*

*1 cycle*

**Function:**  $r \leftarrow r + [ir] + C, ir \leftarrow ir + 1$

Adds the content of the data memory addressed by the ir register (X or Y) and carry (C) to the r register (A or B). Then increments the ir register (X or Y). The flags change due to the operation result of the r register and the increment result of the ir register does not affect the flags.

**Code:**

Mnemonic	MSB												LSB			
ADC %A,[%X]+	1	1	0	0	1	1	1	1	0	0	0	0	1	19E1H		
ADC %A,[%Y]+	1	1	0	0	1	1	1	1	0	0	0	1	1	19E3H		
ADC %B,[%X]+	1	1	0	0	1	1	1	1	0	0	1	0	1	19E5H		
ADC %B,[%Y]+	1	1	0	0	1	1	1	1	0	0	1	1	1	19E7H		

**Flags:**

E	I	C	Z
↓	–	↑	↑

**Mode:** Src: Register indirect  
 Dst: Register direct  
 Extended addressing: Invalid

**ADC [%ir],%r***Add with carry r reg. to location [ir reg.]**2 cycles***Function:**  $[ir] \leftarrow [ir] + r + C$ 

Adds the content of the r register (A or B) and carry (C) to the data memory addressed by the ir register (X or Y).

Code:	Mnemonic	MSB												LSB			
	ADC [%X],%A	1	1	0	0	1	1	1	1	0	1	0	0	0	19E8H		
	ADC [%X],%B	1	1	0	0	1	1	1	1	0	1	1	0	0	19ECH		
	ADC [%Y],%A	1	1	0	0	1	1	1	1	0	1	0	1	0	19EAH		
	ADC [%Y],%B	1	1	0	0	1	1	1	1	0	1	1	1	0	19EEH		

Flags:	E	I	C	Z
	↓	-	↑	↑

**Mode:** Src: Register direct  
 Dst: Register indirect  
 Extended addressing: Valid

**Extended LDB** %EXT,imm8**operation:** ADC [%X],%r       $[00imm8] \leftarrow [00imm8] + r + C$  (00imm8 = 0000H + 00H to FFH)

LDB %EXT,imm8

ADC [%Y],%r       $[FFimm8] \leftarrow [FFimm8] + r + C$  (FFimm8 = FF00H + 00H to FFH)**ADC [%ir]+,%r***Add with carry r reg. to location [ir reg.] and increment ir reg.**2 cycles***Function:**  $[ir] \leftarrow [ir] + r + C, ir \leftarrow ir + 1$ 

Adds the content of the r register (A or B) and carry (C) to the data memory addressed by the ir register (X or Y). Then increments the ir register (X or Y). The flags change due to the operation result of the data memory and the increment result of the ir register does not affect the flags.

Code:	Mnemonic	MSB												LSB			
	ADC [%X]+,%A	1	1	0	0	1	1	1	1	0	1	0	0	1	19E9H		
	ADC [%X]+,%B	1	1	0	0	1	1	1	1	0	1	1	0	1	19EDH		
	ADC [%Y]+,%A	1	1	0	0	1	1	1	1	0	1	0	1	1	19EBH		
	ADC [%Y]+,%B	1	1	0	0	1	1	1	1	0	1	1	1	1	19EFH		

Flags:	E	I	C	Z
	↓	-	↑	↑

**Mode:** Src: Register direct  
 Dst: Register indirect  
 Extended addressing: Invalid



---

**ADC %B,%A,n4**     *Add with carry A reg. to B reg. in specified radix*     *2 cycles*

**Function:**  $B \leftarrow N\text{'s adjust } (B + A + C)$

Adds the content of the A register and carry (C) to the B register. The operation result is adjusted with n4 as the radix. The C flag is set by a carry according to the radix.

**Code:**

Mnemonic	MSB										LSB	
ADC %B,%A,n4	1	0	0	0	0	1	1	0	1	[10H-n4]	10D0H–10DFH	

**Flags:**

E	I	C	Z
↓	–	↑	↑

**Mode:** Src: Register direct  
 Dst: Register direct  
 Extended addressing: Invalid

**Note:** n4 should be specified with a value from 1 to 16.

---

**ADC %B,[%ir],n4**     *Add with carry location [ir reg.] to B reg. in specified radix*     *2 cycles*

**Function:**  $B \leftarrow N\text{'s adjust } (B + [ir] + C)$

Adds the content of the data memory addressed by the ir register (X or Y) and carry (C) to the B register. The operation result is adjusted with n4 as the radix. The C flag is set by a carry according to the radix.

**Code:**

Mnemonic	MSB										LSB	
ADC %B,[%X],n4	1	1	1	0	1	1	1	0	0	[10H-n4]	1DC0H–1DCFH	
ADC %B,[%Y],n4	1	1	1	0	1	1	1	1	0	[10H-n4]	1DE0H–1DEFH	

**Flags:**

E	I	C	Z
↓	–	↑	↑

**Mode:** Src: Register indirect  
 Dst: Register direct  
 Extended addressing: Valid

**Extended operation:** LDB %EXT,imm8

ADC %B,[%X],n4      $B \leftarrow N\text{'s adjust } (B + [00imm8] + C)$  (00imm8 = 0000H + 00H to FFH)

LDB %EXT,imm8

ADC %B,[%Y],n4      $B \leftarrow N\text{'s adjust } (B + [FFimm8] + C)$  (FFimm8 = FF00H + 00H to FFH)

**Note:** n4 should be specified with a value from 1 to 16.

**ADC %B,[%ir]+,n4** Add with carry location [ir reg.] to B reg. in specified radix and increment ir reg. 2 cycles

**Function:**  $B \leftarrow N's\ adjust\ (B + [ir] + C)$ ,  $ir \leftarrow ir + 1$

Adds the content of the data memory addressed by the ir register (X or Y) and carry (C) to the B register. The operation result is adjusted with n4 as the radix. Then increments the ir register (X or Y). The flags change due to the operation result of the B register and the increment result of the ir register does not affect the flags. The C flag is set by a carry according to the radix.

**Code:**

Mnemonic	MSB										LSB	
ADC %B,[%X]+,n4	1	1	1	1	0	1	1	1	0	1	[10H-n4]	1DD0H-1DDFH
ADC %B,[%Y]+,n4	1	1	1	1	0	1	1	1	1	1	[10H-n4]	1DF0H-1DFFH

**Flags:**

E	I	C	Z
↓	-	↑	↑

**Mode:** Src: Register indirect  
 Dst: Register direct  
 Extended addressing: Invalid

**Note:** n4 should be specified with a value from 1 to 16.

**ADC [%ir],%B,n4** Add with carry B reg. to location [ir reg.] in specified radix 2 cycles

**Function:**  $[ir] \leftarrow N's\ adjust\ ([ir] + B + C)$

Adds the content of the B register and carry (C) to the data memory addressed by the ir register (X or Y). The operation result is adjusted with n4 as the radix. The C flag is set by a carry according to the radix.

**Code:**

Mnemonic	MSB										LSB	
ADC [%X],%B,n4	1	1	1	0	1	0	1	0	0	[10H-n4]	1D40H-1D4FH	
ADC [%Y],%B,n4	1	1	1	0	1	0	1	1	0	[10H-n4]	1D60H-1D6FH	

**Flags:**

E	I	C	Z
↓	-	↑	↑

**Mode:** Src: Register direct  
 Dst: Register indirect  
 Extended addressing: Valid

**Extended LDB** %EXT,imm8

**operation:** ADC [%X],%B,n4  $[00imm8] \leftarrow N's\ adjust\ ([00imm8] + B + C)$   
 (00imm8 = 0000H + 00H to FFH)

LDB %EXT,imm8

ADC [%Y],%B,n4  $[FFimm8] \leftarrow N's\ adjust\ ([FFimm8] + B + C)$   
 (FFimm8 = FF00H + 00H to FFH)

**Note:** n4 should be specified with a value from 1 to 16.

**ADC [%ir]+,%B,n4** Add with carry B reg. to location [ir reg.] in specified radix and increment ir reg. 2 cycles

**Function:** [ir] ← N's adjust ([ir] + B + C), ir ← ir + 1

Adds the content of the B register and carry (C) to the data memory addressed by the ir register (X or Y). The operation result is adjusted with n4 as the radix. Then increments the ir register (X or Y). The flags change due to the operation result of the data memory and the increment result of the ir register does not affect the flags. The C flag is set by a carry according to the radix.

Code:	Mnemonic	MSB										LSB	
	ADC [%X]+,%B,n4	1	1	1	0	1	0	1	0	1	[10H-n4]	1D50H–1D5FH	
	ADC [%Y]+,%B,n4	1	1	1	0	1	0	1	1	1	[10H-n4]	1D70H–1D7FH	

Flags:	E	I	C	Z
	↓	–	↑	↑

**Mode:** Src: Register direct  
 Dst: Register indirect  
 Extended addressing: Invalid

**Note:** n4 should be specified with a value from 1 to 16.

**ADC [%ir],0,n4** Add carry to location [ir reg.] in specified radix 2 cycles

**Function:** [ir] ← N's adjust ([ir] + 0 + C)

Adds the carry (C) to the data memory addressed by the ir register (X or Y). The operation result is adjusted with n4 as the radix. The C flag is set by a carry according to the radix. This instruction is useful for a carry processing to the highest digit of n based counters.

Code:	Mnemonic	MSB										LSB	
	ADC [%X],0,n4	1	1	1	0	1	0	0	0	0	[10H-n4]	1D00H–1D0FH	
	ADC [%Y],0,n4	1	1	1	0	1	0	0	1	0	[10H-n4]	1D20H–1D2FH	

Flags:	E	I	C	Z
	↓	–	↑	↑

**Mode:** Src: Register direct  
 Dst: Register indirect  
 Extended addressing: Valid

**Extended LDB** %EXT,imm8

**operation:** ADC [%X],0,n4 [00imm8] ← N's adjust ([00imm8] + 0 + C)  
 (00imm8 = 0000H + 00H to FFH)

LDB %EXT,imm8

ADC [%Y],0,n4 [FFimm8] ← N's adjust ([FFimm8] + 0 + C)  
 (FFimm8 = FF00H + 00H to FFH)

**Note:** n4 should be specified with a value from 1 to 16.

**ADC [%ir]+,0,n4**      *Add carry to location [ir reg.] in specified radix and increment ir reg. 2 cycles*

**Function:**  $[ir] \leftarrow N's \text{ adjust } ([ir] + 0 + C), ir \leftarrow ir + 1$

Adds the carry (C) to the data memory addressed by the ir register (X or Y). The operation result is adjusted with n4 as the radix. Then increments the ir register (X or Y). The flags change due to the operation result of the data memory and the increment result of the ir register does not affect the flags. The C flag is set by a carry according to the radix. This instruction is useful for a carry processing of n based counters.

**Code:**

Mnemonic	MSB										LSB	
ADC [%X]+,0,n4	1	1	1	1	0	1	0	0	0	1	[10H-n4]	1D10H-1D1FH
ADC [%Y]+,0,n4	1	1	1	1	0	1	0	0	1	1	[10H-n4]	1D30H-1D3FH

**Flags:**

E	I	C	Z
↓	-	↑	↑

**Mode:** Src: Register direct  
 Dst: Register indirect  
 Extended addressing: Invalid

**Note:** n4 should be specified with a value from 1 to 16.

**ADD %r,%r'**      *Add r' reg. to r reg. 1 cycle*

**Function:**  $r \leftarrow r + r'$

Adds the content of the r' register (A or B) to the r register (A or B).

**Code:**

Mnemonic	MSB										LSB			
ADD %A,%A	1	1	0	0	1	0	1	1	1	0	0	0	X	1970H, (1971H)
ADD %A,%B	1	1	0	0	1	0	1	1	1	0	0	1	X	1972H, (1973H)
ADD %B,%A	1	1	0	0	1	0	1	1	1	0	1	0	X	1974H, (1975H)
ADD %B,%B	1	1	0	0	1	0	1	1	1	0	1	1	X	1976H, (1977H)

**Flags:**

E	I	C	Z
↓	-	↑	↑

**Mode:** Src: Register direct  
 Dst: Register direct  
 Extended addressing: Invalid



**ADD %r,imm4***Add immediate data imm4 to r reg.**1 cycle***Function:**  $r \leftarrow r + \text{imm4}$ 

Adds the 4-bit immediate data imm4 to the r register (A or B).

**Code:**

Mnemonic	MSB										LSB			
ADD %A,imm4	1	1	0	0	1	0	1	0	0	i3	i2	i1	i0	1940H–194FH
ADD %B,imm4	1	1	0	0	1	0	1	0	1	i3	i2	i1	i0	1950H–195FH

**Flags:**

E	I	C	Z
↓	–	↑	↑

**Mode:**

Src: Immediate data

Dst: Register direct

Extended addressing: Invalid

**ADD %r,[%ir]***Add location [ir reg.] to r reg.**1 cycle***Function:**  $r \leftarrow r + [\text{ir}]$ 

Adds the content of the data memory addressed by the ir register (X or Y) to the r register (A or B).

**Code:**

Mnemonic	MSB										LSB			
ADD %A,[%X]	1	1	0	0	1	0	1	1	0	0	0	0	0	1960H
ADD %A,[%Y]	1	1	0	0	1	0	1	1	0	0	0	1	0	1962H
ADD %B,[%X]	1	1	0	0	1	0	1	1	0	0	1	0	0	1964H
ADD %B,[%Y]	1	1	0	0	1	0	1	1	0	0	1	1	0	1966H

**Flags:**

E	I	C	Z
↓	–	↑	↑

**Mode:**

Src: Register indirect

Dst: Register direct

Extended addressing: Valid

**Extended LDB %EXT,imm8****operation:** ADD %r,[%X]  $r \leftarrow r + [00\text{imm8}]$  (00imm8 = 0000H + 00H to FFH)

LDB %EXT,imm8

ADD %r,[%Y]  $r \leftarrow r + [\text{FFimm8}]$  (FFimm8 = FF00H + 00H to FFH)

**ADD %r,[%ir]+** *Add location [ir reg.] to r reg. and increment ir reg.* 1 cycle

**Function:**  $r \leftarrow r + [ir]$ ,  $ir \leftarrow ir + 1$

Adds the content of the data memory addressed by the ir register (X or Y) to the r register (A or B). Then increments the ir register (X or Y). The flags change due to the operation result of the r register and the increment result of the ir register does not affect the flags.

**Code:**

Mnemonic	MSB												LSB		
ADD %A,[%X]+	1	1	0	0	1	0	1	1	0	0	0	0	1	1961H	
ADD %A,[%Y]+	1	1	0	0	1	0	1	1	0	0	0	1	1	1963H	
ADD %B,[%X]+	1	1	0	0	1	0	1	1	0	0	1	0	1	1965H	
ADD %B,[%Y]+	1	1	0	0	1	0	1	1	0	0	1	1	1	1967H	

**Flags:**

E	I	C	Z
↓	–	↑	↑

**Mode:** Src: Register indirect  
 Dst: Register direct  
 Extended addressing: Invalid

**ADD [%ir],%r** *Add r reg. to location [ir reg.]* 2 cycles

**Function:**  $[ir] \leftarrow [ir] + r$

Adds the content of the r register (A or B) to the data memory addressed by the ir register (X or Y).

**Code:**

Mnemonic	MSB												LSB		
ADD [%X],%A	1	1	0	0	1	0	1	1	0	1	0	0	0	1968H	
ADD [%X],%B	1	1	0	0	1	0	1	1	0	1	1	0	0	196CH	
ADD [%Y],%A	1	1	0	0	1	0	1	1	0	1	0	1	0	196AH	
ADD [%Y],%B	1	1	0	0	1	0	1	1	0	1	1	1	0	196EH	

**Flags:**

E	I	C	Z
↓	–	↑	↑

**Mode:** Src: Register direct  
 Dst: Register indirect  
 Extended addressing: Valid

**Extended operation:** LDB %EXT,imm8  
 ADD [%X],%r  $[00imm8] \leftarrow [00imm8] + r$  (00imm8 = 0000H + 00H to FFH)  
 LDB %EXT,imm8  
 ADD [%Y],%r  $[FFimm8] \leftarrow [FFimm8] + r$  (FFimm8 = FF00H + 00H to FFH)

---

**ADD [%ir]+,%r**      *Add r reg. to location [ir reg.] and increment ir reg.*      2 cycles

**Function:**  $[ir] \leftarrow [ir] + r$ ,  $ir \leftarrow ir + 1$

Adds the content of the r register (A or B) to the data memory addressed by the ir register (X or Y). Then increments the ir register (X or Y). The flags change due to the operation result of the data memory and the increment result of the ir register does not affect the flags.

**Code:**

Mnemonic	MSB												LSB	
ADD [%X]+,%A	1	1	0	0	1	0	1	1	0	1	0	0	1	1969H
ADD [%X]+,%B	1	1	0	0	1	0	1	1	0	1	1	0	1	196DH
ADD [%Y]+,%A	1	1	0	0	1	0	1	1	0	1	0	1	1	196BH
ADD [%Y]+,%B	1	1	0	0	1	0	1	1	0	1	1	1	1	196FH

**Flags:**

E	I	C	Z
↓	–	↑	↑

**Mode:** Src: Register direct  
 Dst: Register indirect  
 Extended addressing: Invalid

---

**ADD [%ir],imm4**      *Add immediate data imm4 to location [ir reg.]*      2 cycles

**Function:**  $[ir] \leftarrow [ir] + imm4$

Adds the 4-bit immediate data imm4 to the data memory addressed by the ir register (X or Y).

**Code:**

Mnemonic	MSB												LSB			
ADD [%X],imm4	1	1	0	0	1	0	0	0	0	i3	i2	i1	i0	1900H–190FH		
ADD [%Y],imm4	1	1	0	0	1	0	0	1	0	i3	i2	i1	i0	1920H–192FH		

**Flags:**

E	I	C	Z
↓	–	↑	↑

**Mode:** Src: Immediate data  
 Dst: Register indirect  
 Extended addressing: Valid

**Extended** LDB %EXT,imm8

**operation:** ADD [%X],imm4       $[00imm8] \leftarrow [00imm8] + imm4$  (00imm8 = 0000H + 00H to FFH)

LDB %EXT,imm8

ADD [%Y],imm4       $[FFimm8] \leftarrow [FFimm8] + imm4$  (FFimm8 = FF00H + 00H to FFH)

**ADD [%ir]+,imm4** *Add immediate data imm4 to location [ir reg.] and increment ir reg. 2 cycles*

**Function:**  $[ir] \leftarrow [ir] + imm4, ir \leftarrow ir + 1$

Adds the 4-bit immediate data imm4 to the data memory addressed by the ir register (X or Y). Then increments the ir register (X or Y). The flags change due to the operation result of the data memory and the increment result of the ir register does not affect the flags.

**Code:**

Mnemonic	MSB												LSB			
ADD [%X]+,imm4	1	1	0	0	1	0	0	0	1	i3	i2	i1	i0	1910H–191FH		
ADD [%Y]+,imm4	1	1	0	0	1	0	0	1	1	i3	i2	i1	i0	1930H–193FH		

**Flags:**

E	I	C	Z
↓	–	↑	↑

**Mode:** Src: Immediate data  
 Dst: Register indirect  
 Extended addressing: Invalid

**ADD %ir,%BA** *Add BA reg. to ir reg. 1 cycle*

**Function:**  $ir \leftarrow ir + BA$

Adds the content of the BA register to the ir register (X or Y). This instruction does not affect the C flag regardless of the operation result.

**Code:**

Mnemonic	MSB												LSB	
ADD %X,%BA	1	1	1	1	1	1	1	0	1	0	0	0	X	1FD0H, (1FD1H)
ADD %Y,%BA	1	1	1	1	1	1	1	0	1	0	0	1	X	1FD2H, (1FD3H)

**Flags:**

E	I	C	Z
↓	–	–	↑

**Mode:** Src: Register direct  
 Dst: Register direct  
 Extended addressing: Invalid

**ADD %ir,sign8***Add immediate data sign8 to ir reg.**1 cycle***Function:**  $ir \leftarrow ir + \text{sign8}$ 

Adds the signed 8-bit immediate data sign8 (-128 to 127) to the ir register (X or Y). This instruction does not affect the C flag regardless of the operation result.

Code:	Mnemonic	MSB											LSB			
	ADD %X,sign8	0	1	1	0	0	s7	s6	s5	s4	s3	s2	s1	s0	0C00H–0CFFH	
	ADD %Y,sign8	0	1	1	0	1	s7	s6	s5	s4	s3	s2	s1	s0	0D00H–0DFFH	

Flags:	E	I	C	Z
	↓	–	–	↑

**Mode:** Src: Immediate data  
 Dst: Register direct  
 Extended addressing: Valid

**Extended LDB** %EXT,imm8**operation:** ADD %ir,sign8       $ir \leftarrow ir + \text{sign16}$  (upper 8-bit: imm8, lower 8-bit: sign8)**AND %r,%r'***Logical AND of r' reg. and r reg.**1 cycle***Function:**  $r \leftarrow r \wedge r'$ 

Performs a logical AND operation of the content of the r' register (A or B) and the content of the r register (A or B), and stores the result in the r register.

Code:	Mnemonic	MSB											LSB			
	AND %A,%A	1	1	0	1	0	0	1	1	1	0	0	0	X	1A70H, (1A71H)	
	AND %A,%B	1	1	0	1	0	0	1	1	1	0	0	1	X	1A72H, (1A73H)	
	AND %B,%A	1	1	0	1	0	0	1	1	1	0	1	0	X	1A74H, (1A75H)	
	AND %B,%B	1	1	0	1	0	0	1	1	1	0	1	1	X	1A76H, (1A77H)	

Flags:	E	I	C	Z
	↓	–	–	↑

**Mode:** Src: Register direct  
 Dst: Register direct  
 Extended addressing: Invalid

**AND %r,imm4**

*Logical AND of immediate data imm4 and r reg.*

*1 cycle*

**Function:**  $r \leftarrow r \wedge \text{imm4}$

Performs a logical AND operation of the 4-bit immediate data imm4 and the content of the r register (A or B), and stores the result in the r register.

**Code:**

Mnemonic	MSB										LSB			
AND %A,imm4	1	1	0	1	0	0	1	0	0	i3	i2	i1	i0	1A40H–1A4FH
AND %B,imm4	1	1	0	1	0	0	1	0	1	i3	i2	i1	i0	1A50H–1A5FH

**Flags:**

E	I	C	Z
↓	–	–	↑

**Mode:**

Src: Immediate data  
 Dst: Register direct  
 Extended addressing: Invalid

**AND %F,imm4**

*Logical AND of immediate data imm4 and F reg.*

*1 cycle*

**Function:**  $F \leftarrow F \wedge \text{imm4}$

Performs a logical AND operation of the 4-bit immediate data imm4 and the content of the F (flag) register, and stores the result in the r register. It is possible to reset any flag.

**Code:**

Mnemonic	MSB										LSB			
AND %F,imm4	1	0	0	0	0	1	0	0	0	i3	i2	i1	i0	1080H–108FH

**Flags:**

E	I	C	Z
↓	↓	↓	↓

**Mode:**

Src: Immediate data  
 Dst: Register direct  
 Extended addressing: Invalid

**AND %r,[%ir]***Logical AND of location [ir reg.] and r reg.**1 cycle***Function:**  $r \leftarrow r \wedge [ir]$ 

Performs a logical AND operation of the content of the data memory addressed by the ir register (X or Y) and the content of the r register (A or B), and stores the result in the r register.

Code:	Mnemonic	MSB												LSB			
	AND %A,[%X]	1	1	0	1	0	0	1	1	0	0	0	0	0	0	1A60H	
	AND %A,[%Y]	1	1	0	1	0	0	1	1	0	0	0	1	0	1A62H		
	AND %B,[%X]	1	1	0	1	0	0	1	1	0	0	1	0	0	1A64H		
	AND %B,[%Y]	1	1	0	1	0	0	1	1	0	0	1	1	0	1A66H		

Flags:	E	I	C	Z
	↓	-	-	↑

**Mode:** Src: Register indirect  
 Dst: Register direct  
 Extended addressing: Valid

**Extended LDB** %EXT,imm8**operation:** AND %r,[%X]  $r \leftarrow r \wedge [00imm8]$  (00imm8 = 0000H + 00H to FFH)

LDB %EXT,imm8

AND %r,[%Y]  $r \leftarrow r \wedge [FFimm8]$  (FFimm8 = FF00H + 00H to FFH)**AND %r,[%ir]+***Logical AND of location [ir reg.] and r reg. and increment ir reg.**1 cycle***Function:**  $r \leftarrow r \wedge [ir], ir \leftarrow ir + 1$ 

Performs a logical AND operation of the content of the data memory addressed by the ir register (X or Y) and the content of the r register (A or B), and stores the result in the r register. Then increments the ir register (X or Y). The flags change due to the operation result of the r register and the increment result of the ir register does not affect the flags.

Code:	Mnemonic	MSB												LSB			
	AND %A,[%X]+	1	1	0	1	0	0	1	1	0	0	0	0	1	1A61H		
	AND %A,[%Y]+	1	1	0	1	0	0	1	1	0	0	0	1	1	1A63H		
	AND %B,[%X]+	1	1	0	1	0	0	1	1	0	0	1	0	1	1A65H		
	AND %B,[%Y]+	1	1	0	1	0	0	1	1	0	0	1	1	1	1A67H		

Flags:	E	I	C	Z
	↓	-	-	↑

**Mode:** Src: Register indirect  
 Dst: Register direct  
 Extended addressing: Invalid

**AND [%ir],%r**

*Logical AND of r reg. and location [ir reg.]*

*2 cycles*

**Function:**  $[ir] \leftarrow [ir] \wedge r$

Performs a logical AND operation of the content of the r register (A or B) and the content of the data memory addressed by the ir register (X or Y), and stores the result in that address.

**Code:**

Mnemonic	MSB												LSB			
AND [%X],%A	1	1	0	1	0	0	1	1	0	1	0	0	0	1A68H		
AND [%X],%B	1	1	0	1	0	0	1	1	0	1	1	0	0	1A6CH		
AND [%Y],%A	1	1	0	1	0	0	1	1	0	1	0	1	0	1A6AH		
AND [%Y],%B	1	1	0	1	0	0	1	1	0	1	1	1	0	1A6EH		

**Flags:**

E	I	C	Z
↓	-	-	↑

**Mode:** Src: Register direct  
 Dst: Register indirect  
 Extended addressing: Valid

**Extended operation:** LDB %EXT,imm8  
 AND [%X],%r  $[00imm8] \leftarrow [00imm8] \wedge r$  (00imm8 = 0000H + 00H to FFH)  
 LDB %EXT,imm8  
 AND [%Y],%r  $[FFimm8] \leftarrow [FFimm8] \wedge r$  (FFimm8 = FF00H + 00H to FFH)

**AND [%ir]+,%r**

*Logical AND of r reg. and location [ir reg.] and increment ir reg.*

*2 cycles*

**Function:**  $[ir] \leftarrow [ir] \wedge r, ir \leftarrow ir + 1$

Performs a logical AND operation of the content of the r register (A or B) and the content of the data memory addressed by the ir register (X or Y), and stores the result in that address. Then increments the ir register (X or Y). The flags change due to the operation result of the data memory and the increment result of the ir register does not affect the flags.

**Code:**

Mnemonic	MSB												LSB			
AND [%X]+,%A	1	1	0	1	0	0	1	1	0	1	0	0	1	1A69H		
AND [%X]+,%B	1	1	0	1	0	0	1	1	0	1	1	0	1	1A6DH		
AND [%Y]+,%A	1	1	0	1	0	0	1	1	0	1	0	1	1	1A6BH		
AND [%Y]+,%B	1	1	0	1	0	0	1	1	0	1	1	1	1	1A6FH		

**Flags:**

E	I	C	Z
↓	-	-	↑

**Mode:** Src: Register direct  
 Dst: Register indirect  
 Extended addressing: Invalid



**AND [%ir],imm4**     *Logical AND of immediate data imm4 and location [ir reg.]*     2 cycles**Function:**  $[ir] \leftarrow [ir] \wedge imm4$ 

Performs a logical AND operation of the 4-bit immediate data imm4 and the content of the data memory addressed by the ir register (X or Y), and stores the result in that address.

Code:	Mnemonic	MSB								LSB					
	AND [%X],imm4	1	1	0	1	0	0	0	0	0	i3	i2	i1	i0	1A00H–1A0FH
	AND [%Y],imm4	1	1	0	1	0	0	0	1	0	i3	i2	i1	i0	1A20H–1A2FH

Flags:	E	I	C	Z
	↓	–	–	↑

**Mode:** Src: Immediate data  
 Dst: Register indirect  
 Extended addressing: Valid

**Extended LDB** %EXT,imm8**operation:** AND [%X],imm4      $[00imm8] \leftarrow [00imm8] \wedge imm4$  (00imm8 = 0000H + 00H to FFH)

LDB %EXT,imm8

AND [%Y],imm4      $[FFimm8] \leftarrow [FFimm8] \wedge imm4$  (FFimm8 = FF00H + 00H to FFH)**AND [%ir]+,imm4**     *Logical AND of immediate data imm4 and location [ir reg.] and increment ir reg. 2 cycles***Function:**  $[ir] \leftarrow [ir] \wedge imm4, ir \leftarrow ir + 1$ 

Performs a logical AND operation of the 4-bit immediate data imm4 and the content of the data memory addressed by the ir register (X or Y), and stores the result in that address. Then increments the ir register (X or Y). The flags change due to the operation result of the data memory and the increment result of the ir register does not affect the flags.

Code:	Mnemonic	MSB								LSB					
	AND [%X]+,imm4	1	1	0	1	0	0	0	1	1	i3	i2	i1	i0	1A10H–1A1FH
	AND [%Y]+,imm4	1	1	0	1	0	0	0	1	1	i3	i2	i1	i0	1A30H–1A3FH

Flags:	E	I	C	Z
	↓	–	–	↑

**Mode:** Src: Immediate data  
 Dst: Register indirect  
 Extended addressing: Invalid

**BIT %r,%r'**

*Test bit of r reg. with r' reg.*

*1 cycle*

**Function:**  $r \wedge r'$

Performs a logical AND of the content of the r' register (A or B) and the content of the r register (A or B) to check the bits of the r register. The Z flag is changed due to the operation result, but the content of the register is not changed.

**Code:**

Mnemonic	MSB												LSB		
BIT %A,%A	1	1	0	1	0	1	1	1	1	0	0	0	X	1AF0H, (1AF1H)	
BIT %A,%B	1	1	0	1	0	1	1	1	1	0	0	1	X	1AF2H, (1AF3H)	
BIT %B,%A	1	1	0	1	0	1	1	1	1	0	1	0	X	1AF4H, (1AF5H)	
BIT %B,%B	1	1	0	1	0	1	1	1	1	0	1	1	X	1AF6H, (1AF7H)	

**Flags:**

E	I	C	Z
↓	-	-	↑

**Mode:**

Src: Register direct  
 Dst: Register direct  
 Extended addressing: Invalid

**BIT %r,imm4**

*Test bit of r reg. with immediate data imm4*

*1 cycle*

**Function:**  $r \wedge imm4$

Performs a logical AND of the 4-bit immediate data imm4 and the content of the r register (A or B) to check the bits of the r register. The Z flag is changed due to the operation result, but the content of the register is not changed.

**Code:**

Mnemonic	MSB												LSB				
BIT %A,imm4	1	1	0	1	0	1	1	0	0	i3	i2	i1	i0	1AC0H-1ACFH			
BIT %B,imm4	1	1	0	1	0	1	1	0	1	i3	i2	i1	i0	1AD0H-1ADFH			

**Flags:**

E	I	C	Z
↓	-	-	↑

**Mode:**

Src: Immediate data  
 Dst: Register direct  
 Extended addressing: Invalid

**BIT %r,[%ir]***Test bit of r reg. with location [ir reg.]**1 cycle***Function:**  $r \wedge [ir]$ 

Performs a logical AND of the content of the data memory addressed by the ir register (X or Y) and the content of the r register (A or B) to check the bits of the r register. The Z flag is changed due to the operation result, but the content of the register is not changed.

Code:	Mnemonic	MSB												LSB			
	BIT %A,[%X]	1	1	0	1	0	1	1	1	1	0	0	0	0	0	1AE0H	
	BIT %A,[%Y]	1	1	0	1	0	1	1	1	0	0	0	1	0	1AE2H		
	BIT %B,[%X]	1	1	0	1	0	1	1	1	0	0	1	0	0	1AE4H		
	BIT %B,[%Y]	1	1	0	1	0	1	1	1	0	0	1	1	0	1AE6H		

Flags:	E	I	C	Z
	↓	-	-	↑

**Mode:** Src: Register indirect  
 Dst: Register direct  
 Extended addressing: Valid

**Extended LDB** %EXT,imm8**operation:** BIT %r,[%X]  $r \wedge [00imm8]$  (00imm8 = 0000H + 00H to FFH)

LDB %EXT,imm8

BIT %r,[%Y]  $r \wedge [FFimm8]$  (FFimm8 = FF00H + 00H to FFH)**BIT %r,[%ir]+***Test bit of r reg. with location [ir reg.] and increment ir reg.**1 cycle***Function:**  $r \wedge [ir], ir \leftarrow ir + 1$ 

Performs a logical AND of the content of the data memory addressed by the ir register (X or Y) and the content of the r register (A or B) to check the bits of the r register. The Z flag is changed due to the operation result, but the content of the register is not changed. Then increments the ir register (X or Y). The increment result of the ir register does not affect the flags.

Code:	Mnemonic	MSB												LSB			
	BIT %A,[%X]+	1	1	0	1	0	1	1	1	0	0	0	0	1	1AE1H		
	BIT %A,[%Y]+	1	1	0	1	0	1	1	1	0	0	0	1	1	1AE3H		
	BIT %B,[%X]+	1	1	0	1	0	1	1	1	0	0	1	0	1	1AE5H		
	BIT %B,[%Y]+	1	1	0	1	0	1	1	1	0	0	1	1	1	1AE7H		

Flags:	E	I	C	Z
	↓	-	-	↑

**Mode:** Src: Register indirect  
 Dst: Register direct  
 Extended addressing: Invalid

**BIT [%ir],%r**

*Test bit of location [ir reg.] with r reg.*

*1 cycle*

**Function:** [ir]  $\wedge$  r

Performs a logical AND of the content of the r register (A or B) and the content of the data memory addressed by the ir register (X or Y) to check the bits of the memory. The Z flag is changed due to the operation result, but the content of the memory is not changed.

**Code:**

Mnemonic	MSB												LSB	
BIT [%X],%A	1	1	0	1	0	1	1	1	0	1	0	0	0	1AE8H
BIT [%X],%B	1	1	0	1	0	1	1	1	0	1	1	0	0	1AECB
BIT [%Y],%A	1	1	0	1	0	1	1	1	0	1	0	1	0	1AEA
BIT [%Y],%B	1	1	0	1	0	1	1	1	0	1	1	1	0	1AEEH

**Flags:**

E	I	C	Z
↓	-	-	↑

**Mode:** Src: Register direct  
 Dst: Register indirect  
 Extended addressing: Valid

**Extended operation:** LDB %EXT,imm8  
 BIT [%X],%r [00imm8]  $\wedge$  r (00imm8 = 0000H + 00H to FFH)  
 LDB %EXT,imm8  
 BIT [%Y],%r [FFimm8]  $\wedge$  r (FFimm8 = FF00H + 00H to FFH)

**BIT [%ir]+,%r**

*Test bit of location [ir reg.] with r reg. and increment ir reg.*

*1 cycle*

**Function:** [ir]  $\wedge$  r, ir  $\leftarrow$  ir + 1

Performs a logical AND of the content of the r register (A or B) and the content of the data memory addressed by the ir register (X or Y) to check the bits of the memory. The Z flag is changed due to the operation result, but the content of the memory is not changed. Then increments the ir register (X or Y). The increment result of the ir register does not affect the flags.

**Code:**

Mnemonic	MSB												LSB	
BIT [%X]+,%A	1	1	0	1	0	1	1	1	0	1	0	0	1	1AE9H
BIT [%X]+,%B	1	1	0	1	0	1	1	1	0	1	1	0	1	1AEDH
BIT [%Y]+,%A	1	1	0	1	0	1	1	1	0	1	0	1	1	1AEBH
BIT [%Y]+,%B	1	1	0	1	0	1	1	1	0	1	1	1	1	1AEFH

**Flags:**

E	I	C	Z
↓	-	-	↑

**Mode:** Src: Register direct  
 Dst: Register indirect  
 Extended addressing: Invalid

---

**BIT [%ir],imm4**      *Test bit of location [ir reg.] with immediate data imm4*      *1 cycle*

**Function:**  $[ir] \wedge imm4$

Performs a logical AND of the 4-bit immediate data imm4 and the content of the data memory addressed by the ir register (X or Y) to check the bits of the memory. The Z flag is changed due to the operation result, but the content of the memory is not changed.

**Code:**

Mnemonic	MSB										LSB					
BIT [%X],imm4	1	1	0	1	0	1	0	0	0	0	i3	i2	i1	i0	1A80H–1A8FH	
BIT [%Y],imm4	1	1	0	1	0	1	0	1	0	1	0	i3	i2	i1	i0	1AA0H–1AAFH

**Flags:**

E	I	C	Z
↓	–	–	↑

**Mode:** Src: Immediate data  
 Dst: Register indirect  
 Extended addressing: Valid

**Extended LDB** %EXT,imm8

**operation:** BIT [%X],imm4  $[00imm8] \wedge imm4$  (00imm8 = 0000H + 00H to FFH)

LDB %EXT,imm8

BIT [%Y],imm4  $[FFimm8] \wedge imm4$  (FFimm8 = FF00H + 00H to FFH)

---

**BIT [%ir]+,imm4**      *Test bit of location [ir reg.] with immediate data imm4 and increment ir reg. 1 cycle*

**Function:**  $[ir] \wedge imm4, ir \leftarrow ir + 1$

Performs a logical AND of the 4-bit immediate data imm4 and the content of the data memory addressed by the ir register (X or Y) to check the bits of the memory. The Z flag is changed due to the operation result, but the content of the memory is not changed. Then increments the ir register (X or Y). The increment result of the ir register does not affect the flags.

**Code:**

Mnemonic	MSB										LSB				
BIT [%X]+,imm4	1	1	0	1	0	1	0	0	1	1	i3	i2	i1	i0	1A90H–1A9FH
BIT [%Y]+,imm4	1	1	0	1	0	1	0	1	1	1	i3	i2	i1	i0	1AB0H–1ABFH

**Flags:**

E	I	C	Z
↓	–	–	↑

**Mode:** Src: Immediate data  
 Dst: Register indirect  
 Extended addressing: Invalid

**CALR [addr6]**

*Call subroutine at relative location [addr6]*

*2 cycles*

**Function:**  $((SP1-1)*4+3) \sim ((SP1-1)*4) \leftarrow PC + 1, SP1 \leftarrow SP1 - 1, PC \leftarrow PC + [addr6] + 1$   
 (addr6 = 0000H–003FH)

Saves the address next to this instruction to the stack as a return address, then adds the content of the data memory (0000H–003FH) specified with the addr6 to that address to unconditionally call the subroutine started from the address. Branch destination range is the next address of this instruction +0 to 15.

**Code:**

Mnemonic	MSB								LSB							
CALR [addr6]	1	1	1	1	1	0	0	a5	a4	a3	a2	a1	a0	1F00H–1F3FH		

**Flags:**

E	I	C	Z
↓	–	–	–

**Mode:** 6-bit absolute  
 Extended addressing: Invalid

**CALR sign8**

*Call subroutine at relative location sign8*

*1 cycle*

**Function:**  $((SP1-1)*4+3) \sim ((SP1-1)*4) \leftarrow PC + 1, SP1 \leftarrow SP1 - 1, PC \leftarrow PC + sign8 + 1$  (sign8 = -128~127)

Saves the address next to this instruction to the stack as a return address, then adds the related address specified with the sign8 to that address to unconditionally call the subroutine started from the address. Branch destination range is the next address of this instruction -128 to +127.

**Code:**

Mnemonic	MSB								LSB							
CALR sign8	0	0	0	1	0	s7	s6	s5	s4	s3	s2	s1	s0	0200H–02FFH		

**Flags:**

E	I	C	Z
↓	–	–	–

**Mode:** Signed 8-bit PC relative  
 Extended addressing: Valid

**Extended LDB** %EXT,imm8

**operation:** CALR sign8  $((SP1-1)*4+3) \sim ((SP1-1)*4) \leftarrow PC + 1, SP1 \leftarrow SP1 - 1,$   
 $PC \leftarrow PC + sign16 + 1$   
 (sign16 = -32768 to 32767, upper 8-bit: imm8, lower 8-bit: sign8)

**CALZ imm8***Call subroutine at location imm8**1 cycle***Function:**  $[(SP1-1)*4+3] \sim [(SP1-1)*4] \leftarrow PC + 1, SP1 \leftarrow SP1 - 1, PC \leftarrow imm8$ 

Saves the address next to this instruction to the stack as a return address, then unconditionally calls the subroutine started from the address (0000H–00FFH) specified with the imm8.

Code:	Mnemonic	MSB										LSB			
	CALZ imm8	0	0	0	1	1	i7	i6	i5	i4	i3	i2	i1	i0	0300H–03FFH

Flags:	E	I	C	Z
	↓	–	–	–

**Mode:** Immediate data  
 Extended addressing: Invalid

**CLR [addr6],imm2** *Clear bit imm2 in location [addr6]**2 cycles***Function:**  $[addr6] \leftarrow [addr6] \wedge \text{not}(2^{imm2})$ 

(addr6 = 0000H–003FH or FFC0H–FFFFH)

Clears the bit specified with the imm2 in the data memory specified with the addr6 to "0".

Code:	Mnemonic	MSB										LSB			
	CLR [00addr6],imm2	1	0	1	0	0	i1	i0	a5	a4	a3	a2	a1	a0	1400H–14FFH
	CLR [FFaddr6],imm2	1	0	1	0	1	i1	i0	a5	a4	a3	a2	a1	a0	1500H–15FFH

Flags:	E	I	C	Z
	↓	–	–	↑

**Mode:** Src: Immediate data  
 Dst: 6-bit absolute  
 Extended addressing: Invalid

**CMP %r,%r'**

*Compare r reg. with r' reg.*

*1 cycle*

**Function:**  $r - r'$

Subtracts the content of the r' register (A or B) from the content of the r register (A or B). It changes the flags (Z and C), but does not change the content of the register.

**Code:**

Mnemonic	MSB											LSB		
CMP %A,%A	1	1	1	1	0	0	1	1	1	X	0	0	0	1E70H, (1E78H)
CMP %A,%B	1	1	1	1	0	0	1	1	1	X	0	1	0	1E72H, (1E7AH)
CMP %B,%A	1	1	1	1	0	0	1	1	1	X	1	0	0	1E74H, (1E7CH)
CMP %B,%B	1	1	1	1	0	0	1	1	1	X	1	1	0	1E76H, (1E7EH)

**Flags:**

E	I	C	Z
↓	–	↑	↑
↓	–	↓	↑

( $r \neq r'$ )  
( $r = r'$ )

**Mode:**

Src: Register direct  
 Dst: Register direct  
 Extended addressing: Invalid

**CMP %r,imm4**

*Compare r reg. with immediate data imm4*

*1 cycle*

**Function:**  $r - imm4$

Subtracts the 4-bit immediate data imm4 from the content of the r register (A or B). It changes the flags (Z and C), but does not change the content of the register.

**Code:**

Mnemonic	MSB											LSB			
CMP %A,imm4	1	1	1	1	0	0	1	0	0	i3	i2	i1	i0	1E40H–1E4FH	
CMP %B,imm4	1	1	1	1	0	0	1	0	1	i3	i2	i1	i0	1E50H–1E5FH	

**Flags:**

E	I	C	Z
↓	–	↑	↑

**Mode:**

Src: Immediate data  
 Dst: Register direct  
 Extended addressing: Invalid



**CMP %r,[%ir]***Compare r reg. with location [ir reg.]**1 cycle***Function:** r - [ir]

Subtracts the content of the data memory addressed by the ir register (X or Y) from the content of the r register (A or B). It changes the flags (Z and C), but does not change the content of the register.

Code:	Mnemonic	MSB												LSB	
	CMP %A,[%X]	1	1	1	1	0	0	1	1	0	0	0	0	0	1E60H
	CMP %A,[%Y]	1	1	1	1	0	0	1	1	0	0	0	1	0	1E62H
	CMP %B,[%X]	1	1	1	1	0	0	1	1	0	0	1	0	0	1E64H
	CMP %B,[%Y]	1	1	1	1	0	0	1	1	0	0	1	1	0	1E66H

Flags:	E	I	C	Z
	↓	-	↑	↑

**Mode:** Src: Register indirect  
 Dst: Register direct  
 Extended addressing: Valid

**Extended LDB %EXT,imm8****operation:** CMP %r,[%X] r - [00imm8] (00imm8 = 0000H + 00H to FFH)

LDB %EXT,imm8

CMP %r,[%Y] r - [FFimm8] (FFimm8 = FF00H + 00H to FFH)

**CMP %r,[%ir]+***Compare r reg. with location [ir reg.] and increment ir reg.**1 cycle***Function:** r - [ir], ir ← ir + 1

Subtracts the content of the data memory addressed by the ir register (X or Y) from the content of the r register (A or B). It changes the flags (Z and C), but does not change the content of the register. Then increments the ir register (X or Y). The increment result of the ir register does not affect the flags.

Code:	Mnemonic	MSB												LSB	
	CMP %A,[%X]+	1	1	1	1	0	0	1	1	0	0	0	0	1	1E61H
	CMP %A,[%Y]+	1	1	1	1	0	0	1	1	0	0	0	1	1	1E63H
	CMP %B,[%X]+	1	1	1	1	0	0	1	1	0	0	1	0	1	1E65H
	CMP %B,[%Y]+	1	1	1	1	0	0	1	1	0	0	1	1	1	1E67H

Flags:	E	I	C	Z
	↓	-	↑	↑

**Mode:** Src: Register indirect  
 Dst: Register direct  
 Extended addressing: Invalid

**CMP [%ir],%r** *Compare location [ir reg.] with r reg.* 1 cycle

**Function:** [ir] - r

Subtracts the content of the r register (A or B) from the content of the data memory addressed by the ir register (X or Y). It changes the flags (Z and C), but does not change the content of the memory.

**Code:**

Mnemonic	MSB												LSB			
CMP [%X],%A	1	1	1	1	1	0	0	1	1	0	1	0	0	0	1E68H	
CMP [%X],%B	1	1	1	1	0	0	1	1	0	1	1	0	0	0	1E6CH	
CMP [%Y],%A	1	1	1	1	0	0	1	1	0	1	0	1	0	0	1E6AH	
CMP [%Y],%B	1	1	1	1	0	0	1	1	0	1	1	1	1	0	1E6EH	

**Flags:**

E	I	C	Z
↓	—	↑	↑

**Mode:** Src: Register direct  
 Dst: Register indirect  
 Extended addressing: Valid

**Extended operation:** LDB %EXT,imm8  
 CMP [%X],%r [00imm8] - r (00imm8 = 0000H + 00H to FFH)  
 LDB %EXT,imm8  
 CMP [%Y],%r [FFimm8] - r (FFimm8 = FF00H + 00H to FFH)

**CMP [%ir]+,%r** *Compare location [ir reg.] with r reg. and increment ir reg.* 1 cycle

**Function:** [ir] - r, ir ← ir + 1

Subtracts the content of the r register (A or B) from the content of the data memory addressed by the ir register (X or Y). It changes the flags (Z and C), but does not change the content of the memory. Then increments the ir register (X or Y). The increment result of the ir register does not affect the flags.

**Code:**

Mnemonic	MSB												LSB			
CMP [%X]+,%A	1	1	1	1	0	0	1	1	0	1	0	0	1	1E69H		
CMP [%X]+,%B	1	1	1	1	0	0	1	1	0	1	1	0	1	1E6DH		
CMP [%Y]+,%A	1	1	1	1	0	0	1	1	0	1	0	1	1	1E6BH		
CMP [%Y]+,%B	1	1	1	1	0	0	1	1	0	1	1	1	1	1E6FH		

**Flags:**

E	I	C	Z
↓	—	↑	↑

**Mode:** Src: Register direct  
 Dst: Register indirect  
 Extended addressing: Invalid

---

**CMP [%ir],imm4**    *Compare location [ir reg.] with immediate data imm4*    *1 cycle*

**Function:** [ir] - imm4

Subtracts the 4-bit immediate data imm4 from the content of the data memory addressed by the ir register (X or Y). It changes the flags (Z and C), but does not change the content of the memory.

**Code:**

Mnemonic	MSB										LSB				
CMP [%X],imm4	1	1	1	1	0	0	0	0	0	0	i3	i2	i1	i0	1E00H–1E0FH
CMP [%Y],imm4	1	1	1	1	0	0	0	1	0	0	i3	i2	i1	i0	1E20H–1E2FH

**Flags:**

E	I	C	Z
↓	–	↑	↑

**Mode:** Src: Immediate data  
 Dst: Register indirect  
 Extended addressing: Valid

**Extended operation:** LDB %EXT,imm8  
 CMP [%X],imm4    [00imm8] - imm4 (00imm8 = 0000H + 00H to FFH)  
 LDB %EXT,imm8  
 CMP [%Y],imm4    [FFimm8] - imm4 (FFimm8 = FF00H + 00H to FFH)

---

**CMP [%ir]+,imm4**    *Compare location [ir reg.] with immediate data imm4 and increment ir reg. 1 cycle*

**Function:** [ir] - imm4, ir ← ir + 1

Subtracts the 4-bit immediate data imm4 from the content of the data memory addressed by the ir register (X or Y). It changes the flags (Z and C), but does not change the content of the memory. Then increments the ir register (X or Y). The increment result of the ir register does not affect the flags.

**Code:**

Mnemonic	MSB										LSB				
CMP [%X]+,imm4	1	1	1	1	0	0	0	0	1	0	i3	i2	i1	i0	1E10H–1E1FH
CMP [%Y]+,imm4	1	1	1	1	0	0	0	1	1	0	i3	i2	i1	i0	1E30H–1E3FH

**Flags:**

E	I	C	Z
↓	–	↑	↑

**Mode:** Src: Immediate data  
 Dst: Register indirect  
 Extended addressing: Invalid

**CMP %ir,imm8**

*Compare ir reg. with immediate data imm8*

*1 cycle*

**Function:** ir - imm8

Subtracts the 8-bit immediate data imm8 from the content of the ir register (X or Y). It changes the flags (Z and C), but does not change the register.

**Code:**

Mnemonic	MSB						LSB					
CMP %X,imm8	0	1	1	1	0	[	FFH-imm8	]	0E00H-0EFFH			
CMP %Y,imm8	0	1	1	1	1	[	FFH-imm8	]	0F00H-0FFFH			

**Flags:**

E	I	C	Z
↓	-	↑	↑

**Mode:**

Src: Immediate data  
 Dst: Register direct  
 Extended addressing: Valid

**Extended LDB** %EXT,imm8

**operation:** CMP %ir,imm8' ir - imm16 (upper 8-bit: FFH - imm8, lower 8-bit: imm8')

**DEC [addr6]**

*Decrement location [addr6]*

*2 cycles*

**Function:** [addr6] ← [addr6] - 1

(addr6 = 0000H-003FH)

Decrements (-1) the content of the data memory addressed by the addr6.

**Code:**

Mnemonic	MSB										LSB						
DEC [addr6]	1	0	0	0	0	0	0	0	a5	a4	a3	a2	a1	a0	1000H-103FH		

**Flags:**

E	I	C	Z
↓	-	↑	↑

**Mode:**

6-bit absolute addressing  
 Extended addressing: Invalid

**DEC [ir],n4***Decrement location [ir] in specified radix**2 cycles***Function:**  $[ir] \leftarrow N's \text{ adjust } ([ir] - 1)$ 

Decrements (-1) the content of the data memory addressed by the ir register (X or Y). The operation result is adjusted with n4 as the radix.

Code:	Mnemonic	MSB								LSB					
	DEC [%X],n4	1	1	1	0	0	1	0	0	0	n3	n2	n1	n0	1C80H–1C8FH
	DEC [%Y],n4	1	1	1	0	0	1	0	1	0	n3	n2	n1	n0	1CA0H–1CAFH

Flags:	E	I	C	Z
	↓	–	↑	↑

**Mode:** Src: Immediate data  
 Dst: Register indirect  
 Extended addressing: Valid

**Extended LDB** %EXT,imm8**operation:** DEC [%X],n4 [00imm8] ← N's adjust ([00imm8] - 1) (00imm8 = 0000H + 00H to FFH)

LDB %EXT,imm8

DEC [%Y],n4 [FFimm8] ← N's adjust ([FFimm8] - 1) (FFimm8 = FF00H + 00H to FFH)

**Note:** n4 should be specified with a value from 1 to 16. When 16 is specified for n4, the low-order 4 bits of the machine code (n3–n0) become 0000B.

**DEC [ir]+,n4***Decrement location [ir] in specified radix and increment ir reg.**2 cycles***Function:**  $[ir] \leftarrow N's \text{ adjust } ([ir] - 1), ir \leftarrow ir + 1$ 

Decrements (-1) the content of the data memory addressed by the ir register (X or Y). The operation result is adjusted with n4 as the radix. Then increments the ir register (X or Y). The increment result of the ir register does not affect the flags.

Code:	Mnemonic	MSB								LSB					
	DEC [%X]+,n4	1	1	1	0	0	1	0	0	1	n3	n2	n1	n0	1C90H–1C9FH
	DEC [%Y]+,n4	1	1	1	0	0	1	0	1	1	n3	n2	n1	n0	1CB0H–1CBFH

Flags:	E	I	C	Z
	↓	–	↑	↑

**Mode:** Src: Immediate data  
 Dst: Register indirect  
 Extended addressing: Invalid

**Note:** n4 should be specified with a value from 1 to 16. When 16 is specified for n4, the low-order 4 bits of the machine code (n3–n0) become 0000B.

**DEC %sp**

*Decrement stack pointer*

*1 cycle*

**Function:**  $sp \leftarrow sp - 1$

Decrements (-1) the content of the stack pointer sp (SP1 or SP2). This instruction does not change the C flag regardless of the operation result.

**Code:**

Mnemonic	MSB											LSB		
DEC %SP1	1	1	1	1	1	1	1	1	1	0	0	0	0	1FE0H
DEC %SP2	1	1	1	1	1	1	1	1	1	0	0	1	0	1FE4H

**Flags:**

E	I	C	Z
↓	-	-	↑

**Mode:** Register direct  
 Extended addressing: Invalid

**EX %A,%B**

*Exchange A reg. and B reg.*

*1 cycle*

**Function:**  $A \leftrightarrow B$

Exchanges the contents of the A register and B register.

**Code:**

Mnemonic	MSB											LSB			
EX %A,%B	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1FF7H

**Flags:**

E	I	C	Z
↓	-	-	-

**Mode:** Src: Register direct  
 Dst: Register direct  
 Extended addressing: Invalid

**EX %r,[%ir]***Exchange r reg. and location [ir reg.]**2 cycles***Function:**  $r \leftrightarrow [ir]$ 

Exchanges the contents of the r register (A or B) and data memory addressed by the ir register (X or Y).

**Code:**

Mnemonic	MSB												LSB		
EX %A,[%X]	1	0	0	0	0	1	1	1	1	1	1	0	0	0	10F8H
EX %A,[%Y]	1	0	0	0	0	1	1	1	1	1	1	0	1	0	10FAH
EX %B,[%X]	1	0	0	0	0	1	1	1	1	1	1	1	0	0	10FCH
EX %B,[%Y]	1	0	0	0	0	1	1	1	1	1	1	1	1	0	10FEH

**Flags:**

E	I	C	Z
↓	-	-	-

**Mode:**

Src: Register indirect

Dst: Register direct

Extended addressing: Valid

**Extended LDB** %EXT,imm8**operation:** EX %r,[%X]  $r \leftrightarrow [00imm8]$  (00imm8 = 0000H + 00H to FFH)

LDB %EXT,imm8

EX %r,[%Y]  $r \leftrightarrow [FFimm8]$  (FFimm8 = FF00H + 00H to FFH)**EX %r,[%ir]+***Exchange r reg. and location [ir reg.] and increment ir reg.**2 cycles***Function:**  $r \leftrightarrow [ir], ir \leftarrow ir + 1$ 

Exchanges the contents of the r register (A or B) and data memory addressed by the ir register (X or Y). Then increments the ir register (X or Y). The increment result of the ir register does not affect the flags.

**Code:**

Mnemonic	MSB												LSB	
EX %A,[%X]+	1	0	0	0	0	1	1	1	1	1	0	0	1	10F9H
EX %A,[%Y]+	1	0	0	0	0	1	1	1	1	1	0	1	1	10FBH
EX %B,[%X]+	1	0	0	0	0	1	1	1	1	1	1	0	1	10FDH
EX %B,[%Y]+	1	0	0	0	0	1	1	1	1	1	1	1	1	10FFH

**Flags:**

E	I	C	Z
↓	-	-	-

**Mode:**

Src: Register indirect

Dst: Register direct

Extended addressing: Invalid

# HALT

*Set CPU to HALT mode*

*2 cycles*

**Function:** Halt

Sets the CPU to HALT status.

The CPU stops operating, thus the power consumption is reduced. Peripheral circuits such as the oscillation circuit still operate.

An interrupt causes it to return from HALT status to the normal program execution status.

**Code:**

Mnemonic	MSB												LSB			
HALT	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	1FFCH

**Flags:**

E	I	C	Z
↓	-	-	-

# INC [addr6]

*Increment location [addr6]*

*2 cycles*

**Function:** [addr6] ← [addr6] + 1

(addr6 = 0000H–003FH)

Increments (+1) the content of the data memory addressed by the addr6.

**Code:**

Mnemonic	MSB												LSB			
INC [addr6]	1	0	0	0	0	0	0	1	a5	a4	a3	a2	a1	a0	1040H–107FH	

**Flags:**

E	I	C	Z
↓	-	↑	↑

**Mode:**

6-bit absolute

Extended addressing: Invalid



**INC [ir],n4***Increment location [ir] in specified radix**2 cycles***Function:**  $[ir] \leftarrow N's \text{ adjust } ([ir] + 1)$ 

Increments (+1) the content of the data memory addressed by the ir register (X or Y). The operation result is adjusted with n4 as the radix.

Code:	Mnemonic	MSB								LSB	
	INC [%X],n4	1	1	1	0	1	1	0	0	[10H-n4]	1D80H–1D8FH
	INC [%Y],n4	1	1	1	0	1	1	0	1	[10H-n4]	1DA0H–1DAFH

Flags:	E	I	C	Z
	↓	–	↑	↑

**Mode:** Src: Immediate data  
 Dst: Register indirect  
 Extended addressing: Valid

**Extended LDB** %EXT,imm8**operation:** INC [%X],n4 [00imm8] ← N's adjust ([00imm8] + 1) (00imm8 = 0000H + 00H to FFH)

LDB %EXT,imm8

INC [%Y],n4 [FFimm8] ← N's adjust ([FFimm8] + 1) (FFimm8 = FF00H + 00H to FFH)

**Note:** n4 should be specified with a value from 1 to 16.**INC [ir]+,n4***Increment location [ir] in specified radix and increment ir reg.**2 cycles***Function:**  $[ir] \leftarrow N's \text{ adjust } ([ir] + 1), ir \leftarrow ir + 1$ 

Increments (+1) the content of the data memory addressed by the ir register (X or Y). The operation result is adjusted with n4 as the radix. Then increments the ir register (X or Y). The increment result of the ir register does not affect the flags.

Code:	Mnemonic	MSB								LSB		
	INC [%X]+,n4	1	1	1	0	1	1	0	0	1	[10H-n4]	1D90H–1D9FH
	INC [%Y]+,n4	1	1	1	0	1	1	0	1	1	[10H-n4]	1DB0H–1DBFH

Flags:	E	I	C	Z
	↓	–	↑	↑

**Mode:** Src: Immediate data  
 Dst: Register indirect  
 Extended addressing: Invalid

**Note:** n4 should be specified with a value from 1 to 16.

**INC %sp**

*Increment stack pointer*

*1 cycle*

**Function:**  $sp \leftarrow sp + 1$

Increments (+1) the content of the stack pointer sp (SP1 or SP2). This instruction does not change the C flag regardless of the operation result.

**Code:**

Mnemonic	MSB											LSB			
INC %SP1	1	1	1	1	1	1	1	1	1	0	1	0	0	1FE8H	
INC %SP2	1	1	1	1	1	1	1	1	1	0	1	1	0	1FECH	

**Flags:**

E	I	C	Z
↓	-	-	↑

**Mode:** Register direct  
 Extended addressing: Invalid

**INT imm6**

*Software interrupt*

*3 cycles*

**Function:**  $[SP2-1] \leftarrow F, SP2 \leftarrow SP2 - 1, ([ (SP1-1)*4+3 ] \sim [ (SP1-1)*4 ]) \leftarrow PC + 1, SP1 \leftarrow SP1 - 1, PC \leftarrow imm6$   
 (imm6 = 0100H–013FH)

Saves the content of the F register and the return address (this instruction address + 1) to the stack, then executes the software interrupt routine that starts from the vector address (0100H–013FH) specified by the imm6.

**Code:**

Mnemonic	MSB											LSB			
INT imm6	1	1	1	1	1	1	0	i5	i4	i3	i2	i1	i0	1F80H–1FBFH	

**Flags:**

E	I	C	Z
↓	-	-	-

**Mode:** Immediate data  
 Extended addressing: Invalid

**Note:** The RETI instruction, which returns the content of the F register, should be used for returning from the interrupt routine that is executed by this instruction.

**JP %Y***Indirect jump using Y reg.**1 cycle***Function:**  $PC \leftarrow Y$ 

Loads the content of the Y register into the PC to branch unconditionally.

**Code:**

Mnemonic	MSB										LSB				
JP %Y	1	1	1	1	1	1	1	1	1	1	0	0	1	X	1FF2H, (1FF3H)

**Flags:**

E	I	C	Z
↓	-	-	-

**Mode:**

Register direct

Extended addressing: Invalid

**JR %A***Jump to relative location A reg.**1 cycle***Function:**  $PC \leftarrow PC + A + 1$ 

Adds the content of the A register to the address next to this instruction, to unconditionally branch to that address. Branch destination range is the next address of this instruction +0 to 15.

**Code:**

Mnemonic	MSB										LSB				
JR %A	1	1	1	1	1	1	1	1	1	1	0	0	0	1	1FF1H

**Flags:**

E	I	C	Z
↓	-	-	-

**Mode:**

Register direct

Extended addressing: Invalid

**JR %BA**

*Jump to relative location BA reg.*

*1 cycle*

**Function:**  $PC \leftarrow PC + BA + 1$

Adds the content of the BA register to the address next to this instruction, to unconditionally branch to that address. Branch destination range is the next address of this instruction +0 to 255.

**Code:**

Mnemonic	MSB											LSB				
JR %BA	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	1F40H

**Flags:**

E	I	C	Z
↓	-	-	-

**Mode:** Register direct  
 Extended addressing: Invalid

**JR [addr6]**

*Jump to relative location [addr6]*

*2 cycles*

**Function:**  $PC \leftarrow PC + [addr6] + 1$  ( $addr6 = 0000H-003FH$ )

Adds the content of the data memory (0000H–003FH) specified with the addr6 to the address next to this instruction, to unconditionally branch to that address. Branch destination range is the next address of this instruction +0 to 15.

**Code:**

Mnemonic	MSB											LSB				
JR [addr6]	1	1	1	1	1	0	1	a5	a4	a3	a2	a1	a0	1F40H–1F7FH		

**Flags:**

E	I	C	Z
↓	-	-	-

**Mode:** 6-bit absolute  
 Extended addressing: Invalid

**JR sign8***Jump to relative location sign8**1 cycle***Function:**  $PC \leftarrow PC + \text{sign8} + 1$  (sign8 = -128~127)

Adds the relative address specified with the sign8 to the address next to this instruction, to unconditionally branch to that address. Branch destination range is the next address of this instruction -128 to +127.

Code:	Mnemonic	MSB											LSB			
JR	sign8	0	0	0	0	0	s7	s6	s5	s4	s3	s2	s1	s0	0000H-00FFH	

Flags:	E	I	C	Z
	↓	-	-	-

**Mode:** Signed 8-bit PC relative  
 Extended addressing: Valid

**Extended LDB** %EXT,imm8

**operation:** JR sign8  $PC \leftarrow PC + \text{sign16} + 1$   
 (sign16 = -32768 to 32767, upper 8-bit: imm8, lower 8-bit: sign8)

**JRC sign8***Jump to relative location sign8 if C flag is set**1 cycle***Function:** If C = 1 then  $PC \leftarrow PC + \text{sign8} + 1$  (sign8 = -128~127)

Executes the "JR sign8" instruction if the C (carry) flag has been set to "1", otherwise executes the next instruction.

Code:	Mnemonic	MSB											LSB			
JRC	sign8	0	0	1	0	0	s7	s6	s5	s4	s3	s2	s1	s0	0400H-04FFH	

Flags:	E	I	C	Z
	↓	-	-	-

**Mode:** Signed 8-bit PC relative  
 Extended addressing: Valid

**Extended LDB** %EXT,imm8

**operation:** JRC sign8 If C = 1 then  $PC \leftarrow PC + \text{sign16} + 1$   
 (sign16 = -32768 to 32767, upper 8-bit: imm8, lower 8-bit: sign8)

## JRNC sign8

*Jump to relative location sign8 if C flag is reset*

*1 cycle*

**Function:** If C = 0 then PC ← PC + sign8 + 1 (sign8 = -128~127)

Executes the "JR sign8" instruction if the C (carry) flag has been reset to "0", otherwise executes the next instruction.

**Code:**

Mnemonic	MSB										LSB					
JRNC sign8	0	0	1	0	1	s7	s6	s5	s4	s3	s2	s1	s0	0500H–05FFH		

**Flags:**

E	I	C	Z
↓	–	–	–

**Mode:** Signed 8-bit PC relative  
 Extended addressing: Valid

**Extended** LDB %EXT,imm8

**operation:** JRNC sign8      If C = 0 then PC ← PC + sign16 + 1  
 (sign16 = -32768 to 32767, upper 8-bit: imm8, lower 8-bit: sign8)

## JRNZ sign8

*Jump to relative location sign8 if Z flag is reset*

*1 cycle*

**Function:** If Z = 0 then PC ← PC + sign8 + 1 (sign8 = -128~127)

Executes the "JR sign8" instruction if the Z (zero) flag has been set to "1", otherwise executes the next instruction.

**Code:**

Mnemonic	MSB										LSB					
JRNZ sign8	0	0	1	1	1	s7	s6	s5	s4	s3	s2	s1	s0	0700H–07FFH		

**Flags:**

E	I	C	Z
↓	–	–	–

**Mode:** Signed 8-bit PC relative  
 Extended addressing: Valid

**Extended** LDB %EXT,imm8

**operation:** JRNZ sign8      If Z = 0 then PC ← PC + sign16 + 1  
 (sign16 = -32768 to 32767, upper 8-bit: imm8, lower 8-bit: sign8)

**JRZ sign8***Jump to relative location sign8 if Z flag is set**1 cycle***Function:** If Z = 1 then PC ← PC + sign8 + 1 (sign8 = -128~127)

Executes the "JR sign8" instruction if the Z (zero) flag has been reset to "0", otherwise executes the next instruction.

Code:	Mnemonic	MSB											LSB			
	JRZ sign8	0	0	1	1	0	s7	s6	s5	s4	s3	s2	s1	s0	0600H–06FFH	

Flags:	E	I	C	Z
	↓	–	–	–

**Mode:** Signed 8-bit PC relative  
Extended addressing: Valid**Extended LDB** %EXT,imm8**operation:** JRZ sign8      If Z = 1 then PC ← PC + sign16 + 1  
(sign16 = -32768 to 32767, upper 8-bit: imm8, lower 8-bit: sign8)**LD %r,%r'***Load r' reg. into r reg.**1 cycle***Function:** r ← r'

Loads the content of the r' register (A, B or F) into the r register (A, B or F).

Code:	Mnemonic	MSB											LSB			
	LD %A,%A	1	1	1	1	0	1	1	1	1	0	0	0	0	1EF0H	
	LD %A,%B	1	1	1	1	0	1	1	1	1	0	0	1	0	1EF2H	
	LD %A,%F	1	1	1	1	1	1	1	1	1	0	1	1	0	1FF6H	
	LD %B,%A	1	1	1	1	0	1	1	1	1	0	1	0	0	1EF4H	
	LD %B,%B	1	1	1	1	0	1	1	1	1	0	1	1	0	1EF6H	
	LD %F,%A	1	1	1	1	1	1	1	1	1	0	1	0	1	1FF5H	

Flags:	E	I	C	Z
	↓	–	–	–
	↑	↑	↑	↑

(r = F)

**Mode:** Src: Register direct  
Dst: Register direct  
Extended addressing: Invalid

**LD %r,imm4**

*Load immediate data imm4 into r reg.*

*1 cycle*

**Function:**  $r \leftarrow \text{imm4}$

Loads the 4-bit immediate data imm4 into the r register (A, B or F).

**Code:**

Mnemonic	MSB												LSB			
LD %A,imm4	1	1	1	1	0	1	1	0	0	i3	i2	i1	i0	1EC0H–1ECFH		
LD %B,imm4	1	1	1	1	0	1	1	0	1	i3	i2	i1	i0	1ED0H–1EDFH		
LD %F,imm4	1	0	0	0	0	1	0	1	1	i3	i2	i1	i0	10B0H–10BFH		

**Flags:**

E	I	C	Z
↓	–	–	–
↑	↑	↑	↑

(r = F)

**Mode:** Src: Immediate data  
 Dst: Register direct  
 Extended addressing: Invalid

**LD %r,[%ir]**

*Load location [ir reg.] into r reg.*

*1 cycle*

**Function:**  $r \leftarrow [ir]$

Loads the content of the data memory addressed by the ir register (X or Y) into the r register (A or B).

**Code:**

Mnemonic	MSB												LSB			
LD %A,[%X]	1	1	1	1	0	1	1	1	0	0	0	0	0	1EE0H		
LD %A,[%Y]	1	1	1	1	0	1	1	1	0	0	0	1	0	1EE2H		
LD %B,[%X]	1	1	1	1	0	1	1	1	0	0	1	0	0	1EE4H		
LD %B,[%Y]	1	1	1	1	0	1	1	1	0	0	1	1	0	1EE6H		

**Flags:**

E	I	C	Z
↓	–	–	–

**Mode:** Src: Register indirect  
 Dst: Register direct  
 Extended addressing: Valid

**Extended operation:** LDB %EXT,imm8  
 LD %r,[%X]  $r \leftarrow [00imm8]$  (00imm8 = 0000H + 00H to FFH)  
 LDB %EXT,imm8  
 LD %r,[%Y]  $r \leftarrow [FFimm8]$  (FFimm8 = FF00H + 00H to FFH)



**LD %r,[%ir]+***Load location [ir reg.] into r reg. and increment ir reg.**1 cycle***Function:**  $r \leftarrow [ir], ir \leftarrow ir + 1$ 

Loads the content of the data memory addressed by the ir register (X or Y) into the r register (A or B). Then increments the ir register (X or Y).

Code:	Mnemonic	MSB										LSB			
	LD %A,[%X]+	1	1	1	1	0	1	1	1	0	0	0	0	1	1EE1H
	LD %A,[%Y]+	1	1	1	1	0	1	1	1	0	0	0	1	1	1EE3H
	LD %B,[%X]+	1	1	1	1	0	1	1	1	0	0	1	0	1	1EE5H
	LD %B,[%Y]+	1	1	1	1	0	1	1	1	0	0	1	1	1	1EE7H

Flags:	E	I	C	Z
	↓	-	-	-

**Mode:** Src: Register indirect  
 Dst: Register direct  
 Extended addressing: Invalid

**LD [%ir],%r***Load r reg. into location [ir reg.]**1 cycle***Function:**  $[ir] \leftarrow r$ 

Loads the content of the r register (A or B) into the data memory addressed by the ir register (X or Y).

Code:	Mnemonic	MSB										LSB			
	LD [%X],%A	1	1	1	1	0	1	1	1	0	1	0	0	0	1EE8H
	LD [%X],%B	1	1	1	1	0	1	1	1	0	1	1	0	0	1EECH
	LD [%Y],%A	1	1	1	1	0	1	1	1	0	1	0	1	0	1EEAH
	LD [%Y],%B	1	1	1	1	0	1	1	1	0	1	1	1	0	1EEEH

Flags:	E	I	C	Z
	↓	-	-	-

**Mode:** Src: Register direct  
 Dst: Register indirect  
 Extended addressing: Valid

**Extended** LDB %EXT,imm8**operation:** LD [%X],%r [00imm8] ← r (00imm8 = 0000H + 00H to FFH)

LDB %EXT,imm8

LD [%Y],%r [FFimm8] ← r (FFimm8 = FF00H + 00H to FFH)

**LD [%ir]+,%r**      *Load r reg. into location [ir reg.] and increment ir reg.*      *1 cycle*

**Function:** [ir] ← r, ir ← ir + 1  
 Loads the content of the r register (A or B) into the data memory addressed by the ir register (X or Y). Then increments the ir register (X or Y).

**Code:**

Mnemonic	MSB												LSB			
LD [%X]+,%A	1	1	1	1	0	1	1	1	0	1	0	0	1	1E	E9	H
LD [%X]+,%B	1	1	1	1	0	1	1	1	0	1	1	0	1	1E	ED	H
LD [%Y]+,%A	1	1	1	1	0	1	1	1	0	1	0	1	1	1E	EB	H
LD [%Y]+,%B	1	1	1	1	0	1	1	1	0	1	1	1	1	1E	EF	H

**Flags:**

E	I	C	Z
↓	-	-	-

**Mode:** Src: Register direct  
 Dst: Register indirect  
 Extended addressing: Invalid

**LD [%ir],imm4**      *Load immediate data imm4 into location [ir reg.]*      *1 cycle*

**Function:** [ir] ← imm4  
 Loads the 4-bit immediate data imm4 into the data memory addressed by the ir register (X or Y).

**Code:**

Mnemonic	MSB												LSB					
LD [%X],imm4	1	1	1	1	0	1	0	0	0	i3	i2	i1	i0	1E	80	H-1E	8F	H
LD [%Y],imm4	1	1	1	1	0	1	0	1	0	i3	i2	i1	i0	1E	A0	H-1E	AF	H

**Flags:**

E	I	C	Z
↓	-	-	-

**Mode:** Src: Immediate data  
 Dst: Register indirect  
 Extended addressing: Valid

**Extended operation:** LDB %EXT,imm8  
 LD [%X],imm4      [00imm8] ← imm4 (00imm8 = 0000H + 00H to FFH)  
 LDB %EXT,imm8  
 LD [%Y],imm4      [FFimm8] ← imm4 (FFimm8 = FF00H + 00H to FFH)

**LD [%ir]+,imm4**      *Load immediate data imm4 into location [ir reg.] and increment ir reg. 1 cycle***Function:** [ir] ← imm4, ir ← ir + 1

Loads the 4-bit immediate data imm4 into the data memory addressed by the ir register (X or Y). Then increments the ir register (X or Y).

Code:	Mnemonic	MSB										LSB			
	LD [%X]+,imm4	1	1	1	1	0	1	0	0	1	i3	i2	i1	i0	1E90H–1E9FH
	LD [%Y]+,imm4	1	1	1	1	0	1	0	1	1	i3	i2	i1	i0	1EB0H–1EBFH

Flags:	E	I	C	Z
	↓	–	–	–

**Mode:** Src: Immediate data  
 Dst: Register indirect  
 Extended addressing: Invalid

**LD [%ir],[%ir']**      *Load location [ir' reg.] into location [ir reg.] 2 cycles***Function:** [ir] ← [ir']

Loads the content of the data memory addressed by the ir' register (X or Y) into the data memory addressed by the ir register (Y or X).

Code:	Mnemonic	MSB										LSB			
	LD [%X],[%Y]	1	1	1	1	0	1	1	1	1	1	0	1	0	1EFAH
	LD [%Y],[%X]	1	1	1	1	0	1	1	1	1	1	0	0	0	1EF8H

Flags:	E	I	C	Z
	↓	–	–	–

**Mode:** Src: Register indirect  
 Dst: Register indirect  
 Extended addressing: Invalid



**LD [%ir]+,[%ir']+** *Load location [ir' reg.] into location [ir reg.] and increment ir and ir' reg. 2 cycles***Function:**  $[ir] \leftarrow [ir']$ ,  $ir \leftarrow ir + 1$ ,  $ir' \leftarrow ir' + 1$ 

Loads the content of the data memory addressed by the ir' register (X or Y) into the data memory addressed by the ir register (Y or X). Then increments both the ir and ir' registers.

Code:	Mnemonic	MSB												LSB				
	LD [%X]+,[%Y]+	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1EFFH
	LD [%Y]+,[%X]+	1	1	1	1	0	1	1	1	1	1	1	1	0	1	1	1	1EFDH

Flags:	E	I	C	Z
	↓	-	-	-

**Mode:** Src: Register indirect  
 Dst: Register indirect  
 Extended addressing: Invalid

**LDB %BA,imm8** *Load immediate data imm8 into BA reg. 1 cycle***Function:**  $BA \leftarrow imm8$ 

Loads the 8-bit immediate data imm8 into the BA register.

Code:	Mnemonic	MSB												LSB			
	LDB %BA,imm8	0	1	0	0	1	i7	i6	i5	i4	i3	i2	i1	i0	0900H-09FFH		

Flags:	E	I	C	Z
	↓	-	-	-

**Mode:** Src: Immediate data  
 Dst: Register direct  
 Extended addressing: Invalid

---

**LDB %BA,[%ir]+**    *Load location [ir reg.] into BA reg. and increment ir reg.*    *2 cycles*

**Function:**  $A \leftarrow [ir], B \leftarrow [ir + 1], ir \leftarrow ir + 2$

Loads the 2-word data in the data memory into the BA register. The content of the data memory addressed by the ir register (X or Y) is loaded into the A register as the low-order 4 bits, and the content of the next address is loaded into the B register as the high-order 4 bits. The ir register (X or Y) is incremented by 2 words.

**Code:**

Mnemonic	MSB												LSB		
LDB %BA,[%X]+	1	1	1	1	1	1	1	1	0	1	1	0	0	0	1FD8H
LDB %BA,[%Y]+	1	1	1	1	1	1	1	1	0	1	1	0	1	0	1FDAH

**Flags:**

E	I	C	Z
↓	-	-	-

**Mode:** Src: Register indirect  
 Dst: Register direct  
 Extended addressing: Invalid

---

**LDB %BA,%EXT**    *Load EXT reg. into BA reg.*    *1 cycle*

**Function:**  $BA \leftarrow EXT$

Loads the content of the EXT register into the BA register.

**Code:**

Mnemonic	MSB												LSB		
LDB %BA,%EXT	1	1	1	1	1	1	1	1	0	1	0	1	1	X	1FD6H, (1FD7H)

**Flags:**

E	I	C	Z
↓	-	-	-

**Mode:** Src: Register direct  
 Dst: Register direct  
 Extended addressing: Invalid

**LDB %BA,%rr***Load rr reg. into BA reg.**1 cycle***Function:** BA ← rr

Loads the content of the rr register (XL, XH, YL or YH) into the BA register.

**Code:**

Mnemonic	MSB											LSB						
LDB %BA,%XL	1	1	1	1	1	1	1	0	0	1	0	0	0	1	1	0	0	1FC8H
LDB %BA,%XH	1	1	1	1	1	1	1	0	0	1	0	0	1	1	1	0	0	1FC9H
LDB %BA,%YL	1	1	1	1	1	1	1	0	0	1	0	1	0	1	1	0	0	1FCAH
LDB %BA,%YH	1	1	1	1	1	1	1	0	0	1	0	1	1	1	1	0	0	1FCBH

**Flags:**

E	I	C	Z
↓	-	-	-

**Mode:**

Src: Register direct

Dst: Register direct

Extended addressing: Invalid

**LDB %BA,%sp***Load stack pointer into BA reg.**1 cycle***Function:** BA ← sp

Loads the content of the stack pointer sp (SP1 or SP2) into the BA register.

**Code:**

Mnemonic	MSB											LSB						
LDB %BA,%SP1	1	1	1	1	1	1	1	0	0	1	1	0	X	1	1	0	X	1FCCH, (1FCDH)
LDB %BA,%SP2	1	1	1	1	1	1	1	0	0	1	1	1	X	1	1	0	X	1FCEH, (1FCFH)

**Flags:**

E	I	C	Z
↓	-	-	-

**Mode:**

Src: Register direct

Dst: Register direct

Extended addressing: Invalid





---

**LDB %EXT,imm8**    *Load immediate data imm8 into EXT reg.*    *1 cycle*

**Function:** EXT ← imm8

Loads the 8-bit immediate data into the EXT register. The E flag is set to "1".

**Code:**

Mnemonic	MSB										LSB			
LDB %EXT,imm8	0	1	0	0	0	i7	i6	i5	i4	i3	i2	i1	i0	0800H–08FFH

**Flags:**

E	I	C	Z
↑	–	–	–

**Mode:** Src: Immediate data  
 Dst: Register direct  
 Extended addressing: Invalid

---

**LDB %EXT,%BA**    *Load BA reg. into EXT reg.*    *1 cycle*

**Function:** EXT ← BA

Loads the content of the BA register into the EXT register. The E flag is set to "1".

**Code:**

Mnemonic	MSB										LSB			
LDB %EXT,%BA	1	1	1	1	1	1	1	0	1	0	1	0	X	1FD4H, (1FD5H)

**Flags:**

E	I	C	Z
↑	–	–	–

**Mode:** Src: Register direct  
 Dst: Register direct  
 Extended addressing: Invalid

**LDB %rr,imm8**

*Load immediate data imm8 into rr reg.*

*1 cycle*

**Function:** rr ← imm8

Loads the 8-bit immediate data imm8 into the rr (XL or YL) register.

**Code:**

Mnemonic	MSB										LSB			
LDB %XL,imm8	0	1	0	1	0	i7	i6	i5	i4	i3	i2	i1	i0	0A00H–0AFFH
LDB %YL,imm8	0	1	0	1	1	i7	i6	i5	i4	i3	i2	i1	i0	0B00H–0BFFH

**Flags:**

E	I	C	Z
↓	–	–	–

**Mode:**

Src: Immediate data

Dst: Register direct

Extended addressing: Valid

**Extended operation:** LDB %EXT,imm8

LDB %XL,imm8' X ← imm16 (upper 8-bit: imm8, lower 8-bit: imm8')

LDB %EXT,imm8

LDB %YL,imm8' Y ← imm16 (upper 8-bit: imm8, lower 8-bit: imm8')

**LDB %rr,%BA**

*Load BA reg. into rr reg.*

*1 cycle*

**Function:** rr ← BA

Loads the content of the BA register into the rr register (XL, XH, YL or YH).

**Code:**

Mnemonic	MSB										LSB			
LDB %XL,%BA	1	1	1	1	1	1	1	0	0	0	0	0	0	1FC0H
LDB %XH,%BA	1	1	1	1	1	1	1	0	0	0	0	0	1	1FC1H
LDB %YL,%BA	1	1	1	1	1	1	1	0	0	0	0	1	0	1FC2H
LDB %YH,%BA	1	1	1	1	1	1	1	0	0	0	0	1	1	1FC3H

**Flags:**

E	I	C	Z
↓	–	–	–

**Mode:**

Src: Register direct

Dst: Register direct

Extended addressing: Invalid

**LDB %sp,%BA**      *Load BA reg. into stack pointer**1 cycle***Function:**  $sp \leftarrow BA$ 

Loads the content of the BA register into the stack pointer sp (SP1 or SP2).

Code:	Mnemonic	MSB											LSB			
	LDB %SP1,%BA	1	1	1	1	1	1	1	1	0	0	0	1	0	X	1FC4H, (1FC5H)
	LDB %SP2,%BA	1	1	1	1	1	1	1	0	0	0	1	1	X	1FC6H, (1FC7H)	

Flags:	E	I	C	Z
	↓	-	-	-

**Mode:** Src: Register direct  
 Dst: Register direct  
 Extended addressing: Invalid

**NOP**      *No operation**1 cycle***Function:** No operation ( $PC \leftarrow PC+1$ )

Expend 1 cycle without doing an operation that otherwise exerts an affect. The PC (program counter) is incremented.

Code:	Mnemonic	MSB											LSB		
	NOP	1	1	1	1	1	1	1	1	1	1	1	1	X	1FEEH, (1FFFH)

Flags:	E	I	C	Z
	↓	-	-	-

**OR %r,%r'**

*Logical OR of r' reg. and r reg.*

*1 cycle*

**Function:**  $r \leftarrow r \vee r'$

Performs a logical OR operation of the content of the r' register (A or B) and the content of the r register (A or B), and stores the result in the r register.

**Code:**

Mnemonic	MSB												LSB	
OR %A,%A	1	1	0	1	1	0	1	1	1	0	0	0	X	1B70H, (1B71H)
OR %A,%B	1	1	0	1	1	0	1	1	1	0	0	1	X	1B72H, (1B73H)
OR %B,%A	1	1	0	1	1	0	1	1	1	0	1	0	X	1B74H, (1B75H)
OR %B,%B	1	1	0	1	1	0	1	1	1	0	1	1	X	1B76H, (1B77H)

**Flags:**

E	I	C	Z
↓	-	-	↑

**Mode:**

Src: Register direct  
 Dst: Register direct  
 Extended addressing: Invalid

**OR %r,imm4**

*Logical OR of immediate data imm4 and r reg.*

*1 cycle*

**Function:**  $r \leftarrow r \vee \text{imm4}$

Performs a logical OR operation of the 4-bit immediate data imm4 and the content of the r register (A or B), and stores the result in the r register.

**Code:**

Mnemonic	MSB												LSB			
OR %A,imm4	1	1	0	1	1	0	1	0	0	i3	i2	i1	i0	1B40H–1B4FH		
OR %B,imm4	1	1	0	1	1	0	1	0	1	i3	i2	i1	i0	1B50H–1B5FH		

**Flags:**

E	I	C	Z
↓	-	-	↑

**Mode:**

Src: Immediate data  
 Dst: Register direct  
 Extended addressing: Invalid

**OR %F,imm4***Logical OR of immediate data imm4 and F reg.**1 cycle***Function:**  $F \leftarrow F \vee \text{imm4}$ 

Performs a logical OR operation of the 4-bit immediate data imm4 and the content of the F (flag) register, and stores the result in the r register. It is possible to set any flag.

Code:	Mnemonic	MSB										LSB			
	OR %F,imm4	1	0	0	0	0	1	0	0	1	i3	i2	i1	i0	1090H–109FH

Flags:	E	I	C	Z
	↑	↑	↑	↑

**Mode:** Src: Immediate data  
 Dst: Register direct  
 Extended addressing: Invalid

**OR %r,[%ir]***Logical OR of location [ir reg.] and r reg.**1 cycle***Function:**  $r \leftarrow r \vee [\text{ir}]$ 

Performs a logical OR operation of the content of the data memory addressed by the ir register (X or Y) and the content of the r register (A or B), and stores the result in the r register.

Code:	Mnemonic	MSB										LSB			
	OR %A,[%X]	1	1	0	1	1	0	1	1	0	0	0	0	0	1B60H
	OR %A,[%Y]	1	1	0	1	1	0	1	1	0	0	0	1	0	1B62H
	OR %B,[%X]	1	1	0	1	1	0	1	1	0	0	1	0	0	1B64H
	OR %B,[%Y]	1	1	0	1	1	0	1	1	0	0	1	1	0	1B66H

Flags:	E	I	C	Z
	↓	–	–	↑

**Mode:** Src: Register indirect  
 Dst: Register direct  
 Extended addressing: Valid

**Extended LDB** %EXT,imm8**operation:** OR %r,[%X]  $r \leftarrow r \vee [00\text{imm8}]$  (00imm8 = 0000H + 00H to FFH)

LDB %EXT,imm8

OR %r,[%Y]  $r \leftarrow r \vee [\text{FFimm8}]$  (FFimm8 = FF00H + 00H to FFH)

**OR %r,[%ir]+**      *Logical OR of location [ir reg.] and r reg. and increment ir reg.*      1 cycle

**Function:**  $r \leftarrow r \vee [ir]$ ,  $ir \leftarrow ir + 1$

Performs a logical OR operation of the content of the data memory addressed by the ir register (X or Y) and the content of the r register (A or B), and stores the result in the r register. Then increments the ir register (X or Y). The flags change due to the operation result of the r register and the increment result of the ir register does not affect the flags.

**Code:**

Mnemonic	MSB												LSB			
OR %A,[%X]+	1	1	0	1	1	0	1	1	0	0	0	0	0	1	1B61H	
OR %A,[%Y]+	1	1	0	1	1	0	1	1	0	0	0	1	1	1B63H		
OR %B,[%X]+	1	1	0	1	1	0	1	1	0	0	1	0	1	1B65H		
OR %B,[%Y]+	1	1	0	1	1	0	1	1	0	0	1	1	1	1B67H		

**Flags:**

E	I	C	Z
↓	-	-	↑

**Mode:** Src: Register indirect  
 Dst: Register direct  
 Extended addressing: Invalid

**OR [%ir],%r**      *Logical OR of r reg. and location [ir reg.]*      2 cycles

**Function:**  $[ir] \leftarrow [ir] \vee r$

Performs a logical OR operation of the content of the r register (A or B) and the content of the data memory addressed by the ir register (X or Y), and stores the result in that address.

**Code:**

Mnemonic	MSB												LSB			
OR [%X],%A	1	1	0	1	1	0	1	1	0	1	0	0	0	1B68H		
OR [%X],%B	1	1	0	1	1	0	1	1	0	1	1	0	0	1B6CH		
OR [%Y],%A	1	1	0	1	1	0	1	1	0	1	0	1	0	1B6AH		
OR [%Y],%B	1	1	0	1	1	0	1	1	0	1	1	1	0	1B6EH		

**Flags:**

E	I	C	Z
↓	-	-	↑

**Mode:** Src: Register direct  
 Dst: Register indirect  
 Extended addressing: Valid

**Extended operation:** LDB %EXT,imm8  
 OR [%X],%r       $[00imm8] \leftarrow [00imm8] \vee r$  (00imm8 = 0000H + 00H to FFH)  
 LDB %EXT,imm8  
 OR [%Y],%r       $[FFimm8] \leftarrow [FFimm8] \vee r$  (FFimm8 = FF00H + 00H to FFH)

**OR [%ir]+,%r**      *Logical OR of r reg. and location [ir reg.] and increment ir reg.*      2 cycles**Function:**  $[ir] \leftarrow [ir] \vee r, ir \leftarrow ir + 1$ 

Performs a logical OR operation of the content of the r register (A or B) and the content of the data memory addressed by the ir register (X or Y), and stores the result in that address. Then increments the ir register (X or Y). The flags change due to the operation result of the data memory and the increment result of the ir register does not affect the flags.

Code:	Mnemonic	MSB												LSB			
	OR [%X]+,%A	1	1	0	1	1	0	1	1	0	1	0	1	0	1	1B69H	
	OR [%X]+,%B	1	1	0	1	1	0	1	1	0	1	1	0	1	1	1B6DH	
	OR [%Y]+,%A	1	1	0	1	1	0	1	1	0	1	0	1	1	1	1B6BH	
	OR [%Y]+,%B	1	1	0	1	1	0	1	1	0	1	1	1	1	1	1B6FH	

Flags:	E	I	C	Z
	↓	-	-	↑

**Mode:** Src: Register direct  
 Dst: Register indirect  
 Extended addressing: Invalid

**OR [%ir],imm4**      *Logical OR of immediate data imm4 and location [ir reg.]*      2 cycles**Function:**  $[ir] \leftarrow [ir] \vee imm4$ 

Performs a logical OR operation of the 4-bit immediate data imm4 and the content of the data memory addressed by the ir register (X or Y), and stores the result in that address.

Code:	Mnemonic	MSB												LSB			
	OR [%X],imm4	1	1	0	1	1	0	0	0	0	i3	i2	i1	i0	1B00H–1B0FH		
	OR [%Y],imm4	1	1	0	1	1	0	0	1	0	i3	i2	i1	i0	1B20H–1B2FH		

Flags:	E	I	C	Z
	↓	-	-	↑

**Mode:** Src: Immediate data  
 Dst: Register indirect  
 Extended addressing: Valid

**Extended operation:** LDB %EXT,imm8  
 OR [%X],imm4       $[00imm8] \leftarrow [00imm8] \vee imm4$  (00imm8 = 0000H + 00H to FFH)  
 LDB %EXT,imm8  
 OR [%Y],imm4       $[FFimm8] \leftarrow [FFimm8] \vee imm4$  (FFimm8 = FF00H + 00H to FFH)

**OR [%ir]+,imm4**      *Logical OR of immediate data imm4 and location [ir reg.] and increment ir reg. 2 cycles*

**Function:** [ir] ← [ir] ∨ imm4, ir ← ir +1

Performs a logical OR operation of the 4-bit immediate data imm4 and the content of the data memory addressed by the ir register (X or Y), and stores the result in that address. Then increments the ir register (X or Y). The flags change due to the operation result of the data memory and the increment result of the ir register does not affect the flags.

**Code:**

Mnemonic	MSB												LSB			
OR [%X]+,imm4	1	1	1	0	1	1	0	0	0	1	i3	i2	i1	i0	1B10H–1B1FH	
OR [%Y]+,imm4	1	1	0	1	1	0	0	1	1	i3	i2	i1	i0	1B30H–1B3FH		

**Flags:**

E	I	C	Z
↓	–	–	↑

**Mode:** Src: Immediate data  
 Dst: Register indirect  
 Extended addressing: Invalid

**POP %r**      *Pop top of stack into r reg. 1 cycle*

**Function:** r ← [SP2], SP2 ← SP2 +1

Loads the 4-bit data that has been stored in the address indicated by the stack pointer SP2 into the r register (A, B or F), then increments the SP2.

**Code:**

Mnemonic	MSB												LSB			
POP %A	1	1	1	1	1	1	1	1	0	1	1	1	1	1FEFH		
POP %B	1	1	1	1	1	1	1	1	0	1	1	1	0	1FEEH		
POP %F	1	1	1	1	1	1	1	1	0	1	1	0	1	1FEDH		

**Flags:**

E	I	C	Z
↓	–	–	–
↑	↑	↑	↑

(r = A, B)  
(r = F)

**Mode:** Register direct  
 Extended addressing: Invalid



**POP %ir***Pop top of stack into ir reg.**1 cycle***Function:**  $ir \leftarrow ([SP1*4+3] \sim [SP1*4]), SP1 \leftarrow SP1 + 1$ 

Loads the 16-bit data that has been stored in the addresses (4 words) indicated by the stack pointer SP1 (SP1 indicates the lowest address) into the ir register (X or Y), then increments the SP1.

**Code:**

Mnemonic	MSB												LSB		
POP %X	1	1	1	1	1	1	1	1	1	0	1	0	0	1	1FE9H
POP %Y	1	1	1	1	1	1	1	1	1	0	1	0	1	X	1FEAH, (1FEBH)

**Flags:**

E	I	C	Z
↓	-	-	-

**Mode:** Register direct  
 Extended addressing: Invalid

**PUSH %r***Push r reg. onto stack**1 cycle***Function:**  $[SP2-1] \leftarrow r, SP2 \leftarrow SP2 - 1$ 

Decrements the stack pointer SP2, then stores the content of the r register (A, B or F) into the address indicated by the SP2.

**Code:**

Mnemonic	MSB												LSB	
PUSH %A	1	1	1	1	1	1	1	1	0	0	1	1	1	1FE7H
PUSH %B	1	1	1	1	1	1	1	1	0	0	1	1	0	1FE6H
PUSH %F	1	1	1	1	1	1	1	1	0	0	1	0	1	1FE5H

**Flags:**

E	I	C	Z
↓	-	-	-

**Mode:** Register direct  
 Extended addressing: Invalid

**PUSH %ir**

*Push ir reg. onto stack*

*1 cycle*

**Function:**  $(([SP1-1]*4+3] \sim [SP1-1]*4]) \leftarrow ir, SP1 \leftarrow SP1 - 1$

Decrements the stack pointer SP1, then stores the content of the ir register (X or Y) into the addresses (4 words) indicated by the SP1 (SP1 indicates the lowest address).

**Code:**

Mnemonic	MSB												LSB		
PUSH %X	1	1	1	1	1	1	1	1	1	0	0	0	0	1	1FE1H
PUSH %Y	1	1	1	1	1	1	1	1	1	0	0	0	1	X	1FE2H, (1FE3H)

**Flags:**

E	I	C	Z
↓	-	-	-

**Mode:** Register direct  
 Extended addressing: Invalid

**RET**

*Return from subroutine*

*1 cycle*

**Function:**  $PC \leftarrow ([SP1*4+3] \sim [SP1*4]), SP1 \leftarrow SP1 + 1$

Loads the 16-bit data (return address) that has been stored in the addresses (4 words) indicated by the stack pointer SP1 (SP1 indicates the lowest address) into the PC to return from the subroutine. The SP1 is incremented.

**Code:**

Mnemonic	MSB												LSB		
RET	1	1	1	1	1	1	1	1	1	1	1	0	X	0	1FF8H, (1FFAH)

**Flags:**

E	I	C	Z
↓	-	-	-

**RETD imm8***Return from subroutine and load imm8 into location [X]**3 cycles***Function:**  $PC \leftarrow ([SP1*4+3] \sim [SP1*4])$ ,  $SP1 \leftarrow SP1 + 1$ ,  $[X] \leftarrow i3-0$ ,  $[X+1] \leftarrow i7-4$ ,  $X \leftarrow X + 2$ 

After executing the RET instruction, stores the 8-bit immediate data imm8 into the data memory (2 words) indicated by the X register (X register specifies the low-order address of the 2 words). The X register is incremented by 2 words.

**Code:**

Mnemonic	MSB										LSB			
RETD imm8	1	0	0	0	1	i7	i6	i5	i4	i3	i2	i1	i0	1100H–11FFH

**Flags:**

E	I	C	Z
↓	–	–	–

**Mode:** Immediate data  
 Extended addressing: Invalid

**RETI***Return from interrupt routine**2 cycles***Function:**  $PC \leftarrow ([SP1*4+3] \sim [SP1*4])$ ,  $SP1 \leftarrow SP1 + 1$ ,  $F \leftarrow [SP2]$ ,  $SP2 \leftarrow SP2 + 1$ 

After executing the RET instruction, loads the 4-bit data that has been stored in the address indicated by the stack pointer SP2 into the F register, then increments the SP2. This instruction is used for returning from interrupt routines.

**Code:**

Mnemonic	MSB										LSB			
RETI	1	1	1	1	1	1	1	1	1	1	0	0	1	1FF9H

**Flags:**

E	I	C	Z
↑	↑	↑	↑

**RETS**

*Return and skip*

*2 cycles*

**Function:**  $PC \leftarrow ([SP1*4+3] \sim [SP1*4]), SP1 \leftarrow SP1 + 1, PC \leftarrow PC + 1$

After executing the RET instruction, increments the PC to skip 1 instruction immediately after the return.

**Code:**

Mnemonic	MSB											LSB			
RETS	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1FFBH

**Flags:**

E	I	C	Z
↓	-	-	-

**RL %r**

*Rotate left r reg. with carry*

*1 cycle*

**Function:**  $\boxed{C} \leftarrow \boxed{3210} \leftarrow r$

Rotates the content of the r register (A or B) including the carry (C) to the left for 1 bit. The content of the C flag moves to bit 0 of the r register and bit 3 moves to the C flag.

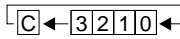
**Code:**

Mnemonic	MSB											LSB		
RL %A	1	0	0	0	0	1	1	1	1	0	0	1	0	10F2H
RL %B	1	0	0	0	0	1	1	1	1	0	1	1	0	10F6H

**Flags:**

E	I	C	Z
↓	-	↑	↑

**Mode:** Register direct  
 Extended addressing: Invalid

**RL [%ir]***Rotate left location [ir reg.] with carry**2 cycles***Function:**  [ir]

Rotates the content of the data memory addressed by the ir register (X or Y) including the carry (C) to the left for 1 bit. The content of the C flag moves to bit 0 of the data memory and bit 3 moves to the C flag.

Code:	Mnemonic	MSB											LSB		
	RL [%X]	1	0	0	0	0	1	1	1	0	1	0	0	0	10E8H
	RL [%Y]	1	0	0	0	0	1	1	1	0	1	0	1	0	10EAH

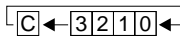
Flags:	E	I	C	Z
	↓	-	↑	↑

**Mode:** Register indirect  
Extended addressing: Valid

**Extended LDB** %EXT,imm8**operation:** RL [%X] Rotates the content of [00imm8] (00imm8 = 0000H + 00H to FFH)

LDB %EXT,imm8

RL [%Y] Rotates the content of [FFimm8] (FFimm8 = FF00H + 00H to FFH)

**RL [%ir]+***Rotate left location [ir reg.] with carry and increment ir reg.**2 cycles***Function:**  [ir], ir ← ir + 1

Rotates the content of the data memory addressed by the ir register (X or Y) including the carry (C) to the left for 1 bit. The content of the C flag moves to bit 0 of the data memory and bit 3 moves to the C flag. Then increments the ir register (X or Y). The increment result of the ir register does not affect the flags.

Code:	Mnemonic	MSB											LSB		
	RL [%X]+	1	0	0	0	0	1	1	1	0	1	0	0	1	10E9H
	RL [%Y]+	1	0	0	0	0	1	1	1	0	1	0	1	1	10EBH

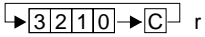
Flags:	E	I	C	Z
	↓	-	↑	↑

**Mode:** Register indirect  
Extended addressing: Invalid

**RR %r**

*Rotate right r reg. with carry*

*1 cycle*

**Function:**  r

Rotates the content of the r register (A or B) including the carry (C) to the right for 1 bit. The content of the C flag moves to bit 3 of the r register and bit 0 moves to the C flag.

**Code:**

Mnemonic	MSB											LSB		
RR %A	1	0	0	0	0	1	1	1	1	0	0	1	1	10F3H
RR %B	1	0	0	0	0	1	1	1	1	0	1	1	1	10F7H

**Flags:**

E	I	C	Z
↓	–	↑	↑

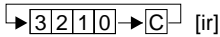
**Mode:**

Register direct  
 Extended addressing: Invalid

**RR [%ir]**

*Rotate right location [ir reg.] with carry*

*2 cycles*

**Function:**  [ir]

Rotates the content of the data memory addressed by the ir register (X or Y) including the carry (C) to the right for 1 bit. The content of the C flag moves to bit 3 of the data memory and bit 0 moves to the C flag.

**Code:**

Mnemonic	MSB											LSB		
RR [%X]	1	0	0	0	0	1	1	1	0	1	1	0	0	10ECH
RR [%Y]	1	0	0	0	0	1	1	1	0	1	1	1	0	10EEH

**Flags:**

E	I	C	Z
↓	–	↑	↑

**Mode:**

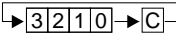
Register indirect  
 Extended addressing: Valid

**Extended operation:** LDB %EXT,imm8

RR [%X] Rotates the content of [00imm8] (00imm8 = 0000H + 00H to FFH)

LDB %EXT,imm8

RR [%Y] Rotates the content of [FFimm8] (FFimm8 = FF00H + 00H to FFH)

**RR [%ir]+***Rotate right location [ir reg.] with carry and increment ir reg.**2 cycles***Function:**  [ir], ir ← ir + 1

Rotates the content of the data memory addressed by the ir register (X or Y) including the carry (C) to the right for 1 bit. The content of the C flag moves to bit 3 of the data memory and bit 0 moves to the C flag. Then increments the ir register (X or Y). The increment result of the ir register does not affect the flags.

Code:	Mnemonic	MSB												LSB		
	RR [%X]+	1	0	0	0	0	0	1	1	1	0	1	1	0	1	10EDH
	RR [%Y]+	1	0	0	0	0	0	1	1	1	0	1	1	1	1	10EFH

Flags:	E	I	C	Z
	↓	-	↑	↑

**Mode:** Register indirect  
 Extended addressing: Invalid

**SBC %r,%r'***Subtract with carry r' reg. from r reg.**1 cycle***Function:**  $r \leftarrow r - r' - C$ 

Subtracts the content of the r' register (A or B) and carry (C) from the r register (A or B).

Code:	Mnemonic	MSB												LSB	
	SBC %A,%A	1	1	0	0	0	1	1	1	1	0	0	0	X	18F0H, (18F1H)
	SBC %A,%B	1	1	0	0	0	1	1	1	1	0	0	1	X	18F2H, (18F3H)
	SBC %B,%A	1	1	0	0	0	1	1	1	1	0	1	0	X	18F4H, (18F5H)
	SBC %B,%B	1	1	0	0	0	1	1	1	1	0	1	1	X	18F6H, (18F7H)

Flags:	E	I	C	Z
	↓	-	↑	↑

**Mode:** Src: Register direct  
 Dst: Register direct  
 Extended addressing: Invalid

**SBC %r,imm4**

*Subtract with carry immediate data imm4 from r reg.*

*1 cycle*

**Function:**  $r \leftarrow r - imm4 - C$

Subtracts the 4-bit immediate data imm4 and carry (C) from the r register (A or B).

**Code:**

Mnemonic	MSB										LSB			
SBC %A,imm4	1	1	0	0	0	1	1	0	0	i3	i2	i1	i0	18C0H–18CFH
SBC %B,imm4	1	1	0	0	0	1	1	0	1	i3	i2	i1	i0	18D0H–18DFH

**Flags:**

E	I	C	Z
↓	–	↑	↑

**Mode:** Src: Immediate data  
 Dst: Register direct  
 Extended addressing: Invalid

**SBC %r,[%ir]**

*Subtract with carry location [ir reg.] from r reg.*

*1 cycle*

**Function:**  $r \leftarrow r - [ir] - C$

Subtracts the content of the data memory addressed by the ir register (X or Y) and carry (C) from the r register (A or B).

**Code:**

Mnemonic	MSB										LSB			
SBC %A,[%X]	1	1	0	0	0	1	1	1	0	0	0	0	0	18E0H
SBC %A,[%Y]	1	1	0	0	0	1	1	1	0	0	0	1	0	18E2H
SBC %B,[%X]	1	1	0	0	0	1	1	1	0	0	1	0	0	18E4H
SBC %B,[%Y]	1	1	0	0	0	1	1	1	0	0	1	1	0	18E6H

**Flags:**

E	I	C	Z
↓	–	↑	↑

**Mode:** Src: Register indirect  
 Dst: Register direct  
 Extended addressing: Valid

**Extended operation:** LDB %EXT,imm8  
 SBC %r,[%X]  $r \leftarrow r - [00imm8] - C$  (00imm8 = 0000H + 00H to FFH)  
 LDB %EXT,imm8  
 SBC %r,[%Y]  $r \leftarrow r - [FFimm8] - C$  (FFimm8 = FF00H + 00H to FFH)



**SBC %r,[%ir]+**      *Subtract with carry location [ir reg.] from r reg. and increment ir reg. 1 cycle***Function:**  $r \leftarrow r - [ir] - C$ ,  $ir \leftarrow ir + 1$ 

Subtracts the content of the data memory addressed by the ir register (X or Y) and carry (C) from the r register (A or B). Then increments the ir register (X or Y). The flags change due to the operation result of the r register and the increment result of the ir register does not affect the flags.

Code:	Mnemonic	MSB												LSB		
	SBC %A,[%X]+	1	1	0	0	0	0	1	1	1	0	0	0	0	1	18E1H
	SBC %A,[%Y]+	1	1	0	0	0	0	1	1	1	0	0	0	1	1	18E3H
	SBC %B,[%X]+	1	1	0	0	0	0	1	1	1	0	0	1	0	1	18E5H
	SBC %B,[%Y]+	1	1	0	0	0	0	1	1	1	0	0	1	1	1	18E7H

Flags:	E	I	C	Z
	↓	-	↑	↑

**Mode:** Src: Register indirect  
 Dst: Register direct  
 Extended addressing: Invalid

**SBC [%ir],%r**      *Subtract with carry r reg. from location [ir reg.] 2 cycles***Function:**  $[ir] \leftarrow [ir] - r - C$ 

Subtracts the content of the r register (A or B) and carry (C) from the data memory addressed by the ir register (X or Y).

Code:	Mnemonic	MSB												LSB	
	SBC [%X],%A	1	1	0	0	0	1	1	1	0	1	0	0	0	18E8H
	SBC [%X],%B	1	1	0	0	0	1	1	1	0	1	1	0	0	18ECH
	SBC [%Y],%A	1	1	0	0	0	1	1	1	0	1	0	1	0	18EAH
	SBC [%Y],%B	1	1	0	0	0	1	1	1	0	1	1	1	0	18EEH

Flags:	E	I	C	Z
	↓	-	↑	↑

**Mode:** Src: Register direct  
 Dst: Register indirect  
 Extended addressing: Valid

**Extended LDB** %EXT,imm8**operation:** SBC [%X],%r       $[00imm8] \leftarrow [00imm8] - r - C$  (00imm8 = 0000H + 00H to FFH)

LDB %EXT,imm8

SBC [%Y],%r       $[FFimm8] \leftarrow [FFimm8] - r - C$  (FFimm8 = FF00H + 00H to FFH)

**SBC [%ir]+,%r**      *Subtract with carry r reg. from location [ir reg.] and increment ir reg. 2 cycles*

**Function:**  $[ir] \leftarrow [ir] - r - C, ir \leftarrow ir + 1$

Subtracts the content of the r register (A or B) and carry (C) from the data memory addressed by the ir register (X or Y). Then increments the ir register (X or Y). The flags change due to the operation result of the data memory and the increment result of the ir register does not affect the flags.

**Code:**

Mnemonic	MSB												LSB			
SBC [%X]+,%A	1	1	0	0	0	0	1	1	1	0	1	0	0	1	18E9H	
SBC [%X]+,%B	1	1	0	0	0	1	1	1	0	1	1	0	1	18EDH		
SBC [%Y]+,%A	1	1	0	0	0	1	1	1	0	1	0	1	1	18EBH		
SBC [%Y]+,%B	1	1	0	0	0	1	1	1	0	1	1	1	1	18EFH		

**Flags:**

E	I	C	Z
↓	–	↑	↑

**Mode:** Src: Register direct  
 Dst: Register indirect  
 Extended addressing: Invalid

**SBC [%ir],imm4**      *Subtract with carry immediate data imm4 from location [ir reg.] 2 cycles*

**Function:**  $[ir] \leftarrow [ir] - imm4 - C$

Subtracts the 4-bit immediate data imm4 and carry (C) from the data memory addressed by the ir register (X or Y).

**Code:**

Mnemonic	MSB												LSB			
SBC [%X],imm4	1	1	0	0	0	1	0	0	0	i3	i2	i1	i0	1880H–188FH		
SBC [%Y],imm4	1	1	0	0	0	1	0	1	0	i3	i2	i1	i0	18A0H–18AFH		

**Flags:**

E	I	C	Z
↓	–	↑	↑

**Mode:** Src: Immediate data  
 Dst: Register indirect  
 Extended addressing: Valid

**Extended operation:** LDB %EXT,imm8  
 SBC [%X],imm4       $[00imm8] \leftarrow [00imm8] - imm4 - C$  (00imm8 = 0000H + 00H to FFH)  
 LDB %EXT,imm8  
 SBC [%Y],imm4       $[FFimm8] \leftarrow [FFimm8] - imm4 - C$  (FFimm8 = FF00H + 00H to FFH)

**SBC [%ir]+,imm4** *Subtract with carry immediate data imm4 from location [ir reg.] and increment ir reg. 2 cycles***Function:**  $[ir] \leftarrow [ir] - imm4 - C$ ,  $ir \leftarrow ir + 1$ 

Subtracts the immediate data imm4 and carry (C) from the data memory addressed by the ir register (X or Y). Then increments the ir register (X or Y). The flags change due to the operation result of the data memory and the increment result of the ir register does not affect the flags.

Code:	Mnemonic	MSB										LSB				
	SBC [%X]+,imm4	1	1	0	0	0	0	1	0	0	1	i3	i2	i1	i0	1890H–189FH
	SBC [%Y]+,imm4	1	1	0	0	0	1	0	1	1	i3	i2	i1	i0	18B0H–18BFH	

Flags:	E	I	C	Z
	↓	–	↑	↑

**Mode:** Src: Immediate data  
 Dst: Register indirect  
 Extended addressing: Invalid

**SBC %B,%A,n4** *Subtract with carry A reg. from B reg. in specified radix 2 cycles***Function:**  $B \leftarrow N's \text{ adjust } (B - A - C)$ 

Subtracts the content of the A register and carry (C) from the B register. The operation result is adjusted with n4 as the radix. The C flag is set according to the radix.

Code:	Mnemonic	MSB										LSB			
	SBC %B,%A,n4	1	0	0	0	0	1	1	0	0	n3	n2	n1	n0	10C0H–10CFH

Flags:	E	I	C	Z
	↓	–	↑	↑

**Mode:** Src: Register direct  
 Dst: Register direct  
 Extended addressing: Invalid

**Note:** n4 should be specified with a value from 1 to 16. When 16 is specified for n4, the low-order 4 bits of the machine code (n3–n0) become 0000B.

**SBC %B,[%ir],n4** Subtract with carry location [ir reg.] from B reg. in specified radix 2 cycles

**Function:**  $B \leftarrow N's \text{ adjust } (B - [ir] - C)$

Subtracts the content of the data memory addressed by the ir register (X or Y) and carry (C) from the B register. The operation result is adjusted with n4 as the radix. The C flag is set according to the radix.

**Code:**

Mnemonic	MSB										LSB				
SBC %B,[%X],n4	1	1	1	1	0	0	1	1	0	0	n3	n2	n1	n0	1CC0H–1CCFH
SBC %B,[%Y],n4	1	1	1	1	0	0	1	1	1	0	n3	n2	n1	n0	1CE0H–1CEFH

**Flags:**

E	I	C	Z
↓	–	↑	↑

**Mode:** Src: Register indirect  
 Dst: Register direct  
 Extended addressing: Valid

**Extended LDB** %EXT,imm8

**operation:** SBC %B,[%X],n4  $B \leftarrow N's \text{ adjust } (B - [00imm8] - C)$  (00imm8 = 0000H + 00H to FFH)

LDB %EXT,imm8

SBC %B,[%Y],n4  $B \leftarrow N's \text{ adjust } (B - [FFimm8] - C)$  (FFimm8 = FF00H + 00H to FFH)

**Note:** n4 should be specified with a value from 1 to 16. When 16 is specified for n4, the low-order 4 bits of the machine code (n3–n0) become 0000B.

**SBC %B,[%ir]+,n4** Subtract with carry location [ir reg.] from B reg. in specified radix and increment ir reg. 2 cycles

**Function:**  $B \leftarrow N's \text{ adjust } (B - [ir] - C)$ ,  $ir \leftarrow ir + 1$

Subtracts the content of the data memory addressed by the ir register (X or Y) and carry (C) from the B register. The operation result is adjusted with n4 as the radix. Then increments the ir register (X or Y). The flags change due to the operation result of the B register and the increment result of the ir register does not affect the flags. The C flag is set according to the radix.

**Code:**

Mnemonic	MSB										LSB				
SBC %B,[%X]+,n4	1	1	1	1	0	0	1	1	0	1	n3	n2	n1	n0	1CD0H–1CDFH
SBC %B,[%Y]+,n4	1	1	1	1	0	0	1	1	1	1	n3	n2	n1	n0	1CF0H–1CFFH

**Flags:**

E	I	C	Z
↓	–	↑	↑

**Mode:** Src: Register indirect  
 Dst: Register direct  
 Extended addressing: Invalid

**Note:** n4 should be specified with a value from 1 to 16. When 16 is specified for n4, the low-order 4 bits of the machine code (n3–n0) become 0000B.

**SBC [%ir],%B,n4** Subtract with carry B reg. from location [ir reg.] in specified radix 2 cycles**Function:**  $[ir] \leftarrow N's \text{ adjust } ([ir] - B - C)$ 

Subtracts the content of the B register and carry (C) from the data memory addressed by the ir register (X or Y). The operation result is adjusted with n4 as the radix. The C flag is set according to the radix.

Code:	Mnemonic	MSB								LSB					
	SBC [%X],%B,n4	1	1	1	0	0	0	1	0	0	n3	n2	n1	n0	1C40H–1C4FH
	SBC [%Y],%B,n4	1	1	1	0	0	0	1	1	0	n3	n2	n1	n0	1C60H–1C6FH

Flags:	E	I	C	Z
	↓	–	↑	↑

**Mode:** Src: Register direct  
 Dst: Register indirect  
 Extended addressing: Valid

**Extended LDB** %EXT,imm8**operation:** SBC [%X],%B,n4 [00imm8]  $\leftarrow N's \text{ adjust } ([00imm8] - B - C)$  (00imm8 = 0000H + 00H to FFH)

LDB %EXT,imm8

SBC [%Y],%B,n4 [FFimm8]  $\leftarrow N's \text{ adjust } ([FFimm8] - B - C)$  (FFimm8 = FF00H + 00H to FFH)

**Note:** n4 should be specified with a value from 1 to 16. When 16 is specified for n4, the low-order 4 bits of the machine code (n3–n0) become 0000B.

**SBC [%ir]+,%B,n4** Subtract with carry B reg. from location [ir reg.] in specified radix and increment ir reg. 2 cycles**Function:**  $[ir] \leftarrow N's \text{ adjust } ([ir] - B - C)$ ,  $ir \leftarrow ir + 1$ 

Subtracts the content of the B register and carry (C) from the data memory addressed by the ir register (X or Y). The operation result is adjusted with n4 as the radix. Then increments the ir register (X or Y). The flags change due to the operation result of the data memory and the increment result of the ir register does not affect the flags. The C flag is set according to the radix.

Code:	Mnemonic	MSB								LSB					
	SBC [%X]+,%B,n4	1	1	1	0	0	0	1	0	1	n3	n2	n1	n0	1C50H–1C5FH
	SBC [%Y]+,%B,n4	1	1	1	0	0	0	1	1	1	n3	n2	n1	n0	1C70H–1C7FH

Flags:	E	I	C	Z
	↓	–	↑	↑

**Mode:** Src: Register direct  
 Dst: Register indirect  
 Extended addressing: Invalid

**Note:** n4 should be specified with a value from 1 to 16. When 16 is specified for n4, the low-order 4 bits of the machine code (n3–n0) become 0000B.

**SBC [%ir],0,n4**      *Subtract carry from location [ir reg.] in specified radix*      2 cycles

**Function:** [ir] ← N's adjust ([ir] - 0 - C)

Subtracts the carry (C) from the data memory addressed by the ir register (X or Y). The operation result is adjusted with n4 as the radix. The C flag is set according to the radix. This instruction is useful for borrow processing of n based counters.

**Code:**

Mnemonic	MSB										LSB				
SBC [%X],0,n4	1	1	1	1	0	0	0	0	0	0	n3	n2	n1	n0	1C00H–1C0FH
SBC [%Y],0,n4	1	1	1	1	0	0	0	0	1	0	n3	n2	n1	n0	1C20H–1C2FH

**Flags:**

E	I	C	Z
↓	–	↑	↑

**Mode:** Src: Register direct  
 Dst: Register indirect  
 Extended addressing: Valid

**Extended LDB** %EXT,imm8

**operation:** SBC [%X],0,n4      [00imm8] ← N's adjust ([00imm8] - 0 - C) (00imm8 = 0000H + 00H to FFH)

LDB %EXT,imm8

SBC [%Y],0,n4      [FFimm8] ← N's adjust ([FFimm8] - 0 - C) (FFimm8 = FF00H + 00H to FFH)

**Note:** n4 should be specified with a value from 1 to 16. When 16 is specified for n4, the low-order 4 bits of the machine code (n3–n0) become 0000B.

**SBC [%ir]+,0,n4**      *Subtract carry from location [ir reg.] in specified radix and increment ir reg.* 2 cycles

**Function:** [ir] ← N's adjust ([ir] - 0 - C), ir ← ir + 1

Subtracts the carry (C) from the data memory addressed by the ir register (X or Y). The operation result is adjusted with n4 as the radix. Then increments the ir register (X or Y). The flags change due to the operation result of the data memory and the increment result of the ir register does not affect the flags. The C flag is set according to the radix. This instruction is useful for borrow processing of n based counters.

**Code:**

Mnemonic	MSB										LSB				
SBC [%X]+,0,n4	1	1	1	1	0	0	0	0	0	1	n3	n2	n1	n0	1C10H–1C1FH
SBC [%Y]+,0,n4	1	1	1	1	0	0	0	0	1	1	n3	n2	n1	n0	1C30H–1C3FH

**Flags:**

E	I	C	Z
↓	–	↑	↑

**Mode:** Src: Register direct  
 Dst: Register indirect  
 Extended addressing: Invalid

**Note:** n4 should be specified with a value from 1 to 16. When 16 is specified for n4, the low-order 4 bits of the machine code (n3–n0) become 0000B.

**SET [addr6],imm2** *Set bit imm2 in location [addr6]**2 cycles***Function:**  $[\text{addr6}] \leftarrow [\text{addr6}] \vee (2^{\text{imm2}})$ 

(addr6 = 0000H–003FH or FFC0H–FFFFH)

Sets the bit specified with the imm2 in the data memory specified with the addr6 to "1".

Code:	Mnemonic	MSB										LSB			
	SET [00addr6],imm2	1	0	1	1	0	i1	i0	a5	a4	a3	a2	a1	a0	1600H–16FFH
	SET [FFaddr6],imm2	1	0	1	1	1	i1	i0	a5	a4	a3	a2	a1	a0	1700H–17FFH

Flags:	E	I	C	Z
	↓	–	–	↑

**Mode:** Src: Immediate data  
 Dst: 6-bit absolute  
 Extended addressing: Invalid

**SLL %r** *Shift left r reg. logical**1 cycle***Function:**  $C \leftarrow [3210] \leftarrow 0 \ r$ 

Shifts the content of the r register (A or B) to the left for 1 bit. Bit 3 of the r register moves to the C flag and bit 0 goes "0".

Code:	Mnemonic	MSB										LSB			
	SLL %A	1	0	0	0	0	1	1	1	1	0	0	0	0	10F0H
	SLL %B	1	0	0	0	0	1	1	1	1	0	1	0	0	10F4H

Flags:	E	I	C	Z
	↓	–	↑	↑

**Mode:** Register direct  
 Extended addressing: Invalid

**SLL [%ir]**

*Shift left location [ir reg.] logical*

*2 cycles*

**Function:**  $C \leftarrow [3210] \leftarrow 0$  [ir]

Shifts the content of the data memory addressed by the ir register (X or Y) to the left for 1 bit. Bit 3 of the r register moves to the C flag and bit 0 goes "0".

**Code:**

Mnemonic	MSB												LSB	
SLL [%X]	1	0	0	0	0	1	1	1	0	0	0	0	0	10E0H
SLL [%Y]	1	0	0	0	0	1	1	1	0	0	0	1	0	10E2H

**Flags:**

E	I	C	Z
↓	-	↑	↑

**Mode:** Register indirect  
 Extended addressing: Valid

**Extended operation:** LDB [%EXT,imm8]  
 SLL [%X] Shifts the content of [00imm8] (00imm8 = 0000H + 00H to FFH)  
 LDB [%EXT,imm8]  
 SLL [%Y] Shifts the content of [FFimm8] (FFimm8 = FF00H + 00H to FFH)

**SLL [%ir]+**

*Shift left location [ir reg.] logical and increment ir reg.*

*2 cycles*

**Function:**  $C \leftarrow [3210] \leftarrow 0$  [ir],  $ir \leftarrow ir + 1$

Shifts the content of the data memory addressed by the ir register (X or Y) to the left for 1 bit. Bit 3 of the r register moves to the C flag and bit 0 goes "0". Then increments the ir register (X or Y). The increment result of the ir register does not affect the flags.

**Code:**

Mnemonic	MSB												LSB	
SLL [%X]+	1	0	0	0	0	1	1	1	0	0	0	0	1	10E1H
SLL [%Y]+	1	0	0	0	0	1	1	1	0	0	0	1	1	10E3H

**Flags:**

E	I	C	Z
↓	-	↑	↑

**Mode:** Register indirect  
 Extended addressing: Invalid



**SLP***Set CPU to SLEEP mode**2 cycles***Function:** Sleep

Sets the CPU to SLEEP status.

The CPU and the peripheral circuits including the oscillation circuit stops operating, thus the power consumption is substantially reduced.

An interrupt from outside the MCU causes it to return from SLEEP status to the normal program execution status.

**Code:**

Mnemonic	MSB											LSB		
SLP	1	1	1	1	1	1	1	1	1	1	1	0	1	1FFDH

**Flags:**

E	I	C	Z
↓	-	-	-

**SRL %r***Shift right r reg. logical**1 cycle***Function:** 0 → 

3	2	1	0
---	---	---	---

 → 

C
---

 r

Shifts the content of the r register (A or B) to the right for 1 bit. Bit 0 of the r register moves to the C flag and bit 3 goes "0".

**Code:**

Mnemonic	MSB											LSB		
SRL %A	1	0	0	0	0	1	1	1	1	0	0	0	1	10F1H
SRL %B	1	0	0	0	0	1	1	1	1	0	1	0	1	10F5H

**Flags:**

E	I	C	Z
↓	-	↑	↑

**Mode:**

Register direct

Extended addressing: Invalid

**SRL [%ir]**

*Shift right location [ir reg.] logical*

*2 cycles*

**Function:** 0 → 

3	2	1	0
---	---	---	---

 → 

C
---

 [ir]

Shifts the content of the data memory addressed by the ir register (X or Y) to the right for 1 bit. Bit 0 of the r register moves to the C flag and bit 3 goes "0".

**Code:**

Mnemonic	MSB												LSB	
SRL [%X]	1	0	0	0	0	1	1	1	0	0	1	0	0	10E4H
SRL [%Y]	1	0	0	0	0	1	1	1	0	0	1	1	0	10E6H

**Flags:**

E	I	C	Z
↓	–	↑	↑

**Mode:** Register indirect  
 Extended addressing: Valid

**Extended operation:** LDB %EXT,imm8  
 SRL [%X] Shifts the content of [00imm8] (00imm8 = 0000H + 00H to FFH)  
 LDB %EXT,imm8  
 SRL [%Y] Shifts the content of [FFimm8] (FFimm8 = FF00H + 00H to FFH)

**SRL [%ir]+**

*Shift right location [ir reg.] logical and increment ir reg.*

*2 cycles*

**Function:** 0 → 

3	2	1	0
---	---	---	---

 → 

C
---

 [ir], ir ← ir + 1

Shifts the content of the data memory addressed by the ir register (X or Y) to the right for 1 bit. Bit 0 of the r register moves to the C flag and bit 3 goes "0". Then increments the ir register (X or Y). The increment result of the ir register does not affect the flags.

**Code:**

Mnemonic	MSB												LSB	
SRL [%X]+	1	0	0	0	0	1	1	1	0	0	1	0	1	10E5H
SRL [%Y]+	1	0	0	0	0	1	1	1	0	0	1	1	1	10E7H

**Flags:**

E	I	C	Z
↓	–	↑	↑

**Mode:** Register indirect  
 Extended addressing: Invalid

**SUB %r,%r'***Subtract r' reg. from r reg.**1 cycle***Function:**  $r \leftarrow r - r'$ 

Subtracts the content of the r' register (A or B) from the r register (A or B).

**Code:**

Mnemonic	MSB											LSB			
SUB %A,%A	1	1	0	0	0	0	0	1	1	1	0	0	0	X	1870H, (1871H)
SUB %A,%B	1	1	0	0	0	0	1	1	1	0	0	1	X	1872H, (1873H)	
SUB %B,%A	1	1	0	0	0	0	1	1	1	0	1	0	X	1874H, (1875H)	
SUB %B,%B	1	1	0	0	0	0	1	1	1	0	1	1	X	1876H, (1877H)	

**Flags:**

E	I	C	Z	
↓	–	↑	↑	( $r \neq r'$ )
↓	–	↓	↑	( $r = r'$ )

**Mode:**

Src: Register direct

Dst: Register direct

Extended addressing: Invalid

**SUB %r,imm4***Subtract immediate data imm4 from r reg.**1 cycle***Function:**  $r \leftarrow r - \text{imm4}$ 

Subtracts the 4-bit immediate data imm4 from the r register (A or B).

**Code:**

Mnemonic	MSB											LSB			
SUB %A,imm4	1	1	0	0	0	0	1	0	0	i3	i2	i1	i0	1840H–184FH	
SUB %B,imm4	1	1	0	0	0	0	1	0	1	i3	i2	i1	i0	1850H–185FH	

**Flags:**

E	I	C	Z
↓	–	↑	↑

**Mode:**

Src: Immediate data

Dst: Register direct

Extended addressing: Invalid

**SUB %r,[%ir]**

*Subtract location [ir reg.] from r reg.*

*1 cycle*

**Function:**  $r \leftarrow r - [ir]$

Subtracts the content of the data memory addressed by the ir register (X or Y) from the r register (A or B).

**Code:**

Mnemonic	MSB												LSB			
SUB %A,[%X]	1	1	0	0	0	0	0	1	1	0	0	0	0	0	0	1860H
SUB %A,[%Y]	1	1	0	0	0	0	0	1	1	0	0	0	0	1	0	1862H
SUB %B,[%X]	1	1	0	0	0	0	0	1	1	0	0	1	0	0	0	1864H
SUB %B,[%Y]	1	1	0	0	0	0	0	1	1	0	0	1	1	0	0	1866H

**Flags:**

E	I	C	Z
↓	–	↑	↑

**Mode:**

Src: Register indirect  
 Dst: Register direct  
 Extended addressing: Valid

**Extended**

LDB %EXT,imm8

**operation:** SUB %r,[%X]  $r \leftarrow r - [00imm8]$  (00imm8 = 0000H + 00H to FFH)

LDB %EXT,imm8

SUB %r,[%Y]  $r \leftarrow r - [FFimm8]$  (FFimm8 = FF00H + 00H to FFH)

**SUB %r,[%ir]+**

*Subtract location [ir reg.] from r reg. and increment ir reg.*

*1 cycle*

**Function:**  $r \leftarrow r - [ir], ir \leftarrow ir + 1$

Subtracts the content of the data memory addressed by the ir register (X or Y) from the r register (A or B). Then increments the ir register (X or Y). The flags change due to the operation result of the r register and the increment result of the ir register does not affect the flags.

**Code:**

Mnemonic	MSB												LSB			
SUB %A,[%X]+	1	1	0	0	0	0	0	1	1	0	0	0	0	0	1	1861H
SUB %A,[%Y]+	1	1	0	0	0	0	0	1	1	0	0	0	0	1	1	1863H
SUB %B,[%X]+	1	1	0	0	0	0	0	1	1	0	0	1	0	1	1	1865H
SUB %B,[%Y]+	1	1	0	0	0	0	0	1	1	0	0	1	1	1	1	1867H

**Flags:**

E	I	C	Z
↓	–	↑	↑

**Mode:**

Src: Register indirect  
 Dst: Register direct  
 Extended addressing: Invalid

**SUB [%ir],%r***Subtract r reg. from location [ir reg.]**2 cycles***Function:**  $[ir] \leftarrow [ir] - r$ 

Subtracts the content of the r register (A or B) from the data memory addressed by the ir register (X or Y).

Code:	Mnemonic	MSB												LSB			
	SUB [%X],%A	1	1	0	0	0	0	0	1	1	0	1	0	0	0	1868H	
	SUB [%X],%B	1	1	0	0	0	0	0	1	1	0	1	1	0	0	186CH	
	SUB [%Y],%A	1	1	0	0	0	0	0	1	1	0	1	0	1	0	186AH	
	SUB [%Y],%B	1	1	0	0	0	0	0	1	1	0	1	1	1	0	186EH	

Flags:	E	I	C	Z
	↓	-	↑	↑

**Mode:** Src: Register direct  
 Dst: Register indirect  
 Extended addressing: Valid

**Extended LDB** %EXT,imm8**operation:** SUB [%X],%r  $[00imm8] \leftarrow [00imm8] - r$  (00imm8 = 0000H + 00H to FFH)

LDB %EXT,imm8

SUB [%Y],%r  $[FFimm8] \leftarrow [FFimm8] - r$  (FFimm8 = FF00H + 00H to FFH)**SUB [%ir]+,%r***Subtract r reg. from location [ir reg.] and increment ir reg.**2 cycles***Function:**  $[ir] \leftarrow [ir] - r, ir \leftarrow ir + 1$ 

Subtracts the content of the r register (A or B) from the data memory addressed by the ir register (X or Y). Then increments the ir register (X or Y). The flags change due to the operation result of the data memory and the increment result of the ir register does not affect the flags.

Code:	Mnemonic	MSB												LSB			
	SUB [%X]+,%A	1	1	0	0	0	0	0	1	1	0	1	0	0	1	1869H	
	SUB [%X]+,%B	1	1	0	0	0	0	0	1	1	0	1	1	0	1	186DH	
	SUB [%Y]+,%A	1	1	0	0	0	0	0	1	1	0	1	0	1	1	186BH	
	SUB [%Y]+,%B	1	1	0	0	0	0	0	1	1	0	1	1	1	1	186FH	

Flags:	E	I	C	Z
	↓	-	↑	↑

**Mode:** Src: Register direct  
 Dst: Register indirect  
 Extended addressing: Invalid

**SUB [%ir],imm4**     *Subtract immediate data imm4 from location [ir reg.]*     2 cycles

**Function:** [ir] ← [ir] - imm4  
 Subtracts the 4-bit immediate data imm4 from the data memory addressed by the ir register (X or Y).

**Code:**

Mnemonic	MSB								LSB					
SUB [%X],imm4	1	1	0	0	0	0	0	0	0	i3	i2	i1	i0	1800H–180FH
SUB [%Y],imm4	1	1	0	0	0	0	0	1	0	i3	i2	i1	i0	1820H–182FH

**Flags:**

E	I	C	Z
↓	–	↑	↑

**Mode:** Src: Immediate data  
 Dst: Register indirect  
 Extended addressing: Valid

**Extended operation:** LDB %EXT,imm8  
 SUB [%X],imm4     [00imm8] ← [00imm8] - imm4 (00imm8 = 0000H + 00H to FFH)  
 LDB %EXT,imm8  
 SUB [%Y],imm4     [FFimm8] ← [FFimm8] - imm4 (FFimm8 = FF00H + 00H to FFH)

**SUB [%ir]+,imm4**     *Subtract immediate data imm4 from location [ir reg.] and increment ir reg.*     2 cycles

**Function:** [ir] ← [ir] - imm4, ir ← ir + 1  
 Subtracts the 4-bit immediate data imm4 from the data memory addressed by the ir register (X or Y). Then increments the ir register (X or Y). The flags change due to the operation result of the data memory and the increment result of the ir register does not affect the flags.

**Code:**

Mnemonic	MSB								LSB					
SUB [%X]+,imm4	1	1	0	0	0	0	0	0	1	i3	i2	i1	i0	1810H–181FH
SUB [%Y]+,imm4	1	1	0	0	0	0	0	1	1	i3	i2	i1	i0	1830H–183FH

**Flags:**

E	I	C	Z
↓	–	↑	↑

**Mode:** Src: Immediate data  
 Dst: Register indirect  
 Extended addressing: Invalid

**TST [addr6],imm2** *Test bit imm2 in location [addr6]**1 cycle***Function:**  $[\text{addr6}] \vee (2^{\text{imm2}})$ 

(addr6 = 0000H–003FH or FFC0H–FFFFH)

Tests the bit specified with the imm2 in the data memory specified with the addr6, and sets/resets the Z flag. It does not change the content of the data memory.

Code:	Mnemonic	MSB										LSB			
	TST [00addr6],imm2	1	0	0	1	0	i1	i0	a5	a4	a3	a2	a1	a0	1200H–12FFH
	TST [FFaddr6],imm2	1	0	0	1	1	i1	i0	a5	a4	a3	a2	a1	a0	1300H–13FFH

Flags:	E	I	C	Z
	↓	–	–	↑

**Mode:** Src: Immediate data  
 Dst: 6-bit absolute  
 Extended addressing: Invalid

**XOR %r,%r'** *Exclusive OR r' reg. and r reg.**1 cycle***Function:**  $r \leftarrow r \vee r'$ 

Performs an exclusive OR operation of the content of the r' register (A or B) and the content of the r register (A or B), and stores the result in the r register.

Code:	Mnemonic	MSB										LSB			
	XOR %A,%A	1	1	0	1	1	1	1	1	1	0	0	0	X	1BF0H, (1BF1H)
	XOR %A,%B	1	1	0	1	1	1	1	1	1	0	0	1	X	1BF2H, (1BF3H)
	XOR %B,%A	1	1	0	1	1	1	1	1	1	0	1	0	X	1BF4H, (1BF5H)
	XOR %B,%B	1	1	0	1	1	1	1	1	1	0	1	1	X	1BF6H, (1BF7H)

Flags:	E	I	C	Z	
	↓	–	–	↑	(r ≠ r')
	↓	–	–	↑	(r = r')

**Mode:** Src: Register direct  
 Dst: Register direct  
 Extended addressing: Invalid

**XOR %r,imm4**

*Exclusive OR immediate data imm4 and r reg.*

*1 cycle*

**Function:**  $r \leftarrow r \vee \text{imm4}$

Performs an exclusive OR operation of the 4-bit immediate data imm4 and the content of the r register (A or B), and stores the result in the r register.

**Code:**

Mnemonic	MSB										LSB			
XOR %A,imm4	1	1	0	1	1	1	1	0	0	i3	i2	i1	i0	1BC0H–1BCFH
XOR %B,imm4	1	1	0	1	1	1	1	0	1	i3	i2	i1	i0	1BD0H–1BDFH

**Flags:**

E	I	C	Z
↓	–	–	↑

**Mode:**

Src: Immediate data  
 Dst: Register direct  
 Extended addressing: Invalid

**XOR %F,imm4**

*Exclusive OR immediate data imm4 and F reg.*

*1 cycle*

**Function:**  $F \leftarrow F \vee \text{imm4}$

Performs an exclusive OR operation of the 4-bit immediate data imm4 and the content of the F (flag) register, and stores the result in the r register. It is possible to set/reset any flag.

**Code:**

Mnemonic	MSB										LSB			
XOR %F,imm4	1	0	0	0	0	1	0	1	0	i3	i2	i1	i0	10A0H–10AFH

**Flags:**

E	I	C	Z
↑	↑	↑	↑

**Mode:**

Src: Immediate data  
 Dst: Register direct  
 Extended addressing: Invalid



**XOR %r,[%ir]***Exclusive OR location [ir reg.] and r reg.**1 cycle***Function:**  $r \leftarrow r \vee [ir]$ 

Performs an exclusive OR operation of the content of the data memory addressed by the ir register (X or Y) and the content of the r register (A or B), and stores the result in the r register.

Code:	Mnemonic	MSB										LSB			
	XOR %A,[%X]	1	1	0	1	1	1	1	1	0	0	0	0	1BE0H	
	XOR %A,[%Y]	1	1	0	1	1	1	1	1	0	0	0	1	1BE2H	
	XOR %B,[%X]	1	1	0	1	1	1	1	1	0	0	1	0	1BE4H	
	XOR %B,[%Y]	1	1	0	1	1	1	1	1	0	0	1	1	1BE6H	

Flags:	E	I	C	Z
	↓	-	-	↑

**Mode:** Src: Register indirect  
 Dst: Register direct  
 Extended addressing: Valid

**Extended LDB** %EXT,imm8**operation:** XOR %r,[%X]  $r \leftarrow r \vee [00imm8]$  (00imm8 = 0000H + 00H to FFH)

LDB %EXT,imm8

XOR %r,[%Y]  $r \leftarrow r \vee [FFimm8]$  (FFimm8 = FF00H + 00H to FFH)**XOR %r,[%ir]+***Exclusive OR location [ir reg.] and r reg. and increment ir reg.**1 cycle***Function:**  $r \leftarrow r \vee [ir], ir \leftarrow ir + 1$ 

Performs an exclusive OR operation of the content of the data memory addressed by the ir register (X or Y) and the content of the r register (A or B), and stores the result in the r register. Then increments the ir register (X or Y). The flags change due to the operation result of the r register and the increment result of the ir register does not affect the flags.

Code:	Mnemonic	MSB										LSB			
	XOR %A,[%X]+	1	1	0	1	1	1	1	1	0	0	0	0	1	1BE1H
	XOR %A,[%Y]+	1	1	0	1	1	1	1	1	0	0	0	1	1	1BE3H
	XOR %B,[%X]+	1	1	0	1	1	1	1	1	0	0	1	0	1	1BE5H
	XOR %B,[%Y]+	1	1	0	1	1	1	1	1	0	0	1	1	1	1BE7H

Flags:	E	I	C	Z
	↓	-	-	↑

**Mode:** Src: Register indirect  
 Dst: Register direct  
 Extended addressing: Invalid

**XOR [%ir],%r** *Exclusive OR r reg. and location [ir reg.]* 2 cycles

**Function:** [ir] ← [ir] ∨ r

Performs an exclusive OR operation of the content of the r register (A or B) and the content of the data memory addressed by the ir register (X or Y), and stores the result in that address.

**Code:**

Mnemonic	MSB												LSB			
XOR [%X],%A	1	1	0	1	1	1	1	1	1	0	1	0	0	0	1BE8H	
XOR [%X],%B	1	1	0	1	1	1	1	1	1	0	1	1	0	0	1BECH	
XOR [%Y],%A	1	1	0	1	1	1	1	1	1	0	1	0	1	0	1BEAH	
XOR [%Y],%B	1	1	0	1	1	1	1	1	1	0	1	1	1	0	1BEEH	

**Flags:**

E	I	C	Z
↓	-	-	↑

**Mode:** Src: Register direct  
 Dst: Register indirect  
 Extended addressing: Valid

**Extended operation:** LDB %EXT,imm8  
 XOR [%X],%r [00imm8] ← [00imm8] ∨ r (00imm8 = 0000H + 00H to FFH)  
 LDB %EXT,imm8  
 XOR [%Y],%r [FFimm8] ← [FFimm8] ∨ r (FFimm8 = FF00H + 00H to FFH)

**XOR [%ir]+,%r** *Exclusive OR r reg. and location [ir reg.] and increment ir reg.* 2 cycles

**Function:** [ir] ← [ir] ∨ r, ir ← ir + 1

Performs an exclusive OR operation of the content of the r register (A or B) and the content of the data memory addressed by the ir register (X or Y), and stores the result in that address. Then increments the ir register (X or Y). The flags change due to the operation result of the data memory and the increment result of the ir register does not affect the flags.

**Code:**

Mnemonic	MSB												LSB			
XOR [%X]+,%A	1	1	0	1	1	1	1	1	1	0	1	0	0	1	1BE9H	
XOR [%X]+,%B	1	1	0	1	1	1	1	1	1	0	1	1	0	1	1BEDH	
XOR [%Y]+,%A	1	1	0	1	1	1	1	1	1	0	1	0	1	1	1BEBH	
XOR [%Y]+,%B	1	1	0	1	1	1	1	1	1	0	1	1	1	1	1BEFH	

**Flags:**

E	I	C	Z
↓	-	-	↑

**Mode:** Src: Register direct  
 Dst: Register indirect  
 Extended addressing: Invalid

**XOR [%ir],imm4** *Exclusive OR immediate data imm4 and location [ir reg.]* 2 cycles**Function:**  $[ir] \leftarrow [ir] \vee imm4$ 

Performs an exclusive OR operation of the 4-bit immediate data imm4 and the content of the data memory addressed by the ir register (X or Y), and stores the result in that address.

Code:	Mnemonic	MSB								LSB					
	XOR [%X],imm4	1	1	0	1	1	1	0	0	0	i3	i2	i1	i0	1B80H–1B8FH
	XOR [%Y],imm4	1	1	0	1	1	1	0	1	0	i3	i2	i1	i0	1BA0H–1BAFH

Flags:	E	I	C	Z
	↓	–	–	↑

**Mode:** Src: Immediate data  
 Dst: Register indirect  
 Extended addressing: Valid

**Extended LDB** %EXT,imm8**operation:** XOR [%X],imm4  $[00imm8] \leftarrow [00imm8] \vee imm4$  (00imm8 = 0000H + 00H to FFH)

LDB %EXT,imm8

XOR [%Y],imm4  $[FFimm8] \leftarrow [FFimm8] \vee imm4$  (FFimm8 = FF00H + 00H to FFH)**XOR [%ir]+,imm4** *Exclusive OR immediate data imm4 and location [ir reg.] and increment ir reg. 2 cycles***Function:**  $[ir] \leftarrow [ir] \vee imm4$ ,  $ir \leftarrow ir + 1$ 

Performs an exclusive OR operation of the 4-bit immediate data imm4 and the content of the data memory addressed by the ir register (X or Y), and stores the result in that address. Then increments the ir register (X or Y). The flags change due to the operation result of the data memory and the increment result of the ir register does not affect the flags.

Code:	Mnemonic	MSB								LSB					
	XOR [%X]+,imm4	1	1	0	1	1	1	0	0	1	i3	i2	i1	i0	1B90H–1B9FH
	XOR [%Y]+,imm4	1	1	0	1	1	1	0	1	1	i3	i2	i1	i0	1BB0H–1BBFH

Flags:	E	I	C	Z
	↓	–	–	↑

**Mode:** Src: Immediate data  
 Dst: Register indirect  
 Extended addressing: Invalid

*Index*

ADC %r,%r' ..... 61	CALR [addr6] ..... 82	LD [%ir]+,imm4 .... 103	SBC %r,imm4 ..... 124
ADC %r,imm4 ..... 61	CALR sign8 ..... 82	LD [%ir],[%ir'] ..... 103	SBC %r,[%ir] ..... 124
ADC %r,[%ir] ..... 62	CALZ imm8 ..... 83	LD [%ir],[%ir']+ ..... 104	SBC %r,[%ir]+ ..... 125
ADC %r,[%ir]+ ..... 62	CLR [addr6],imm2 . 83	LD [%ir]+,[%ir'] ..... 104	SBC [%ir],%r ..... 125
ADC [%ir],%r ..... 63	CMP %r,%r' ..... 84	LD [%ir]+,[%ir']+ ... 105	SBC [%ir]+,%r ..... 126
ADC [%ir]+,%r ..... 63	CMP %r,imm4 ..... 84	LDB %BA,imm8 .... 105	SBC [%ir],imm4 .... 126
ADC [%ir],imm4 ..... 64	CMP %r,[%ir] ..... 85	LDB %BA,[%ir]+ ... 106	SBC [%ir]+,imm4 .. 127
ADC [%ir]+,imm4 ... 64	CMP %r,[%ir]+ ..... 85	LDB %BA,%EXT ... 106	SBC %B,%A,n4 .... 127
ADC %B,%A,n4 .... 65	CMP [%ir],%r ..... 86	LDB %BA,%rr ..... 107	SBC %B,[%ir],n4 .. 128
ADC %B,[%ir],n4 ... 65	CMP [%ir]+,%r ..... 86	LDB %BA,%sp ..... 107	SBC %B,[%ir]+,n4 128
ADC %B,[%ir]+,n4 . 66	CMP [%ir],imm4 ..... 87	LDB [%ir]+,%BA ... 108	SBC [%ir],%B,n4 .. 129
ADC [%ir],%B,n4 ... 66	CMP [%ir]+,imm4 ... 87	LDB [%X]+,imm8 .. 108	SBC [%ir]+,%B,n4 129
ADC [%ir]+,%B,n4 . 67	CMP %ir,imm8 ..... 88	LDB %EXT,imm8 .. 109	SBC [%ir],0,n4 ..... 130
ADC [%ir],0,n4 ..... 67	DEC [addr6] ..... 88	LDB %EXT,%BA ... 109	SBC [%ir]+,0,n4 .... 130
ADC [%ir]+,0,n4 ..... 68	DEC [%ir],n4 ..... 89	LDB %rr,imm8 ..... 110	SET [addr6],imm2 . 131
ADD %r,%r' ..... 68	DEC [%ir]+,n4 ..... 89	LDB %rr,%BA ..... 110	SLL %r ..... 131
ADD %r,imm4 ..... 69	DEC %sp ..... 90	LDB %sp,%BA ..... 111	SLL [%ir] ..... 132
ADD %r,[%ir] ..... 69	EX %A,%B ..... 90	NOP ..... 111	SLL [%ir]+ ..... 132
ADD %r,[%ir]+ ..... 70	EX %r,[%ir] ..... 91	OR %r,%r' ..... 112	SLP ..... 133
ADD [%ir],%r ..... 70	EX %r,[%ir]+ ..... 91	OR %r,imm4 ..... 112	SRL %r ..... 133
ADD [%ir]+,%r ..... 71	HALT ..... 92	OR %F,imm4 ..... 113	SRL [%ir] ..... 134
ADD [%ir],imm4 ..... 71	INC [addr6] ..... 92	OR %r,[%ir] ..... 113	SRL [%ir]+ ..... 134
ADD [%ir]+,imm4 ... 72	INC [%ir],n4 ..... 93	OR %r,[%ir]+ ..... 114	SUB %r,%r' ..... 135
ADD %ir,%BA ..... 72	INC [%ir]+,n4 ..... 93	OR [%ir],%r ..... 114	SUB %r,imm4 ..... 135
ADD %ir,sign8 ..... 73	INC %sp ..... 94	OR [%ir]+,%r ..... 115	SUB %r,[%ir] ..... 136
AND %r,%r' ..... 73	INT imm6 ..... 94	OR [%ir],imm4 ..... 115	SUB %r,[%ir]+ ..... 136
AND %r,imm4 ..... 74	JP %Y ..... 95	OR [%ir]+,imm4 .... 116	SUB [%ir],%r ..... 137
AND %F,imm4 ..... 74	JR %A ..... 95	POP %r ..... 116	SUB [%ir]+,%r ..... 137
AND %r,[%ir] ..... 75	JR %BA ..... 96	POP %ir ..... 117	SUB [%ir],imm4 .... 138
AND %r,[%ir]+ ..... 75	JR [addr6] ..... 96	PUSH %r ..... 117	SUB [%ir]+,imm4 .. 138
AND [%ir],%r ..... 76	JR sign8 ..... 97	PUSH %ir ..... 118	TST [addr6],imm2 . 139
AND [%ir]+,%r ..... 76	JRC sign8 ..... 97	RET ..... 118	XOR %r,%r' ..... 139
AND [%ir],imm4 ..... 77	JRNC sign8 ..... 98	RETD imm8 ..... 119	XOR %r,imm4 ..... 140
AND [%ir]+,imm4 ... 77	JRNZ sign8 ..... 98	RETI ..... 119	XOR %F,imm4 ..... 140
BIT %r,%r' ..... 78	JRZ sign8 ..... 99	RETS ..... 120	XOR %r,[%ir] ..... 141
BIT %r,imm4 ..... 78	LD %r,%r' ..... 99	RL %r ..... 120	XOR %r,[%ir]+ ..... 141
BIT %r,[%ir] ..... 79	LD %r,imm4 ..... 100	RL [%ir] ..... 121	XOR [%ir],%r ..... 142
BIT %r,[%ir]+ ..... 79	LD %r,[%ir] ..... 100	RL [%ir]+ ..... 121	XOR [%ir]+,%r ..... 142
BIT [%ir],%r ..... 80	LD %r,[%ir]+ ..... 101	RR %r ..... 122	XOR [%ir],imm4 .... 143
BIT [%ir]+,%r ..... 80	LD [%ir],%r ..... 101	RR [%ir] ..... 122	XOR [%ir]+,imm4 .. 143
BIT [%ir],imm4 ..... 81	LD [%ir]+,%r ..... 102	RR [%ir]+ ..... 123	
BIT [%ir]+,imm4 ..... 81	LD [%ir],imm4 ..... 102	SBC %r,%r' ..... 123	

# EPSON International Sales Operations

---

## AMERICA

---

### EPSON ELECTRONICS AMERICA, INC.

#### - HEADQUARTERS -

1960 E. Grand Avenue  
El Segundo, CA 90245, U.S.A.  
Phone: +1-310-955-5300 Fax: +1-310-955-5400

#### - SALES OFFICES -

##### West

150 River Oaks Parkway  
San Jose, CA 95134, U.S.A.  
Phone: +1-408-922-0200 Fax: +1-408-922-0238

##### Central

101 Virginia Street, Suite 290  
Crystal Lake, IL 60014, U.S.A.  
Phone: +1-815-455-7630 Fax: +1-815-455-7633

##### Northeast

301 Edgewater Place, Suite 120  
Wakefield, MA 01880, U.S.A.  
Phone: +1-781-246-3600 Fax: +1-781-246-5443

##### Southeast

3010 Royal Blvd. South, Suite 170  
Alpharetta, GA 30005, U.S.A.  
Phone: +1-877-EEA-0020 Fax: +1-770-777-2637

## EUROPE

---

### EPSON EUROPE ELECTRONICS GmbH

#### - HEADQUARTERS -

Riesstrasse 15  
80992 Muenchen, GERMANY  
Phone: +49-(0)89-14005-0 Fax: +49-(0)89-14005-110

#### - GERMANY -

##### SALES OFFICE

Altstadtstrasse 176  
51379 Leverkusen, GERMANY  
Phone: +49-(0)217-15045-0 Fax: +49-(0)217-15045-10

#### - UNITED KINGDOM -

##### UK BRANCH OFFICE

2.4 Doncastle House, Doncastle Road  
Bracknell, Berkshire RG12 8PE, ENGLAND  
Phone: +44-(0)1344-381700 Fax: +44-(0)1344-381701

#### - FRANCE -

##### FRENCH BRANCH OFFICE

1 Avenue de l'Atlantique, LP 915 Les Conquerants  
Z.A. de Courtaboeuf 2, F-91976 Les Ulis Cedex, FRANCE  
Phone: +33-(0)1-64862350 Fax: +33-(0)1-64862355

## ASIA

---

#### - CHINA -

##### EPSON (CHINA) CO., LTD.

28F, Beijing Silver Tower 2# North RD DongSanHuan  
ChaoYang District, Beijing, CHINA  
Phone: 64106655 Fax: 64107320

##### SHANGHAI BRANCH

4F, Bldg., 27, No. 69, Gui Jing Road  
Caohejing, Shanghai, CHINA  
Phone: 21-6485-5552 Fax: 21-6485-0775

#### - HONG KONG, CHINA -

##### EPSON HONG KONG LTD.

20/F., Harbour Centre, 25 Harbour Road  
Wanchai, HONG KONG  
Phone: +852-2585-4600 Fax: +852-2827-4346  
Telex: 65542 EPSCO HX

#### - TAIWAN, R.O.C. -

##### EPSON TAIWAN TECHNOLOGY & TRADING LTD.

10F, No. 287, Nanking East Road, Sec. 3  
Taipei, TAIWAN, R.O.C.  
Phone: 02-2717-7360 Fax: 02-2712-9164  
Telex: 24444 EPSONTB

##### HSINCHU OFFICE

13F-3, No. 295, Kuang-Fu Road, Sec. 2  
HsinChu 300, TAIWAN, R.O.C.  
Phone: 03-573-9900 Fax: 03-573-9169

#### - SINGAPORE -

##### EPSON SINGAPORE PTE., LTD.

No. 1 Temasek Avenue, #36-00  
Millenia Tower, SINGAPORE 039192  
Phone: +65-337-7911 Fax: +65-334-2716

#### - KOREA -

##### SEIKO EPSON CORPORATION KOREA OFFICE

50F, KLI 63 Bldg., 60 Yoido-Dong  
Youngdeungpo-Ku, Seoul, 150-010, KOREA  
Phone: 02-784-6027 Fax: 02-767-3677

#### - JAPAN -

##### SEIKO EPSON CORPORATION

##### ELECTRONIC DEVICES MARKETING DIVISION

##### Electronic Device Marketing Department

##### IC Marketing & Engineering Group

421-8, Hino, Hino-shi, Tokyo 191-8501, JAPAN  
Phone: +81-(0)42-587-5816 Fax: +81-(0)42-587-5624

##### ED International Marketing Department I (Europe & U.S.A.)

421-8, Hino, Hino-shi, Tokyo 191-8501, JAPAN  
Phone: +81-(0)42-587-5812 Fax: +81-(0)42-587-5564

##### ED International Marketing Department II (Asia)

421-8, Hino, Hino-shi, Tokyo 191-8501, JAPAN  
Phone: +81-(0)42-587-5814 Fax: +81-(0)42-587-5110



In pursuit of **“Saving” Technology**, Epson electronic devices.  
Our lineup of semiconductors, liquid crystal displays and quartz devices  
assists in creating the products of our customers' dreams.  
**Epson IS energy savings.**

# EPSON

---

**SEIKO EPSON CORPORATION**  
**ELECTRONIC DEVICES MARKETING DIVISION**

■ Electronic devices information on Epson WWW server

<http://www.epson.co.jp/device/>

First issue JULY 1995, Printed SEPTEMBER 1999 in Japan ® A