NEC

User's Manual

μPD170×× Series

4-bit Single-chip Microcontroller

Common items

Document No. U13262EJ2V0UM00 (2nd edition)
(Previous No. IEU-1363)
Date Published May 1998 N CP(K)

Printed in Japan

[MEMO]

NOTES FOR CMOS DEVICES

① PRECAUTION AGAINST ESD FOR SEMICONDUCTORS

Note:

Strong electric field, when exposed to a MOS device, can cause destruction of the gate oxide and ultimately degrade the device operation. Steps must be taken to stop generation of static electricity as much as possible, and quickly dissipate it once, when it has occurred. Environmental control must be adequate. When it is dry, humidifier should be used. It is recommended to avoid using insulators that easily build static electricity. Semiconductor devices must be stored and transported in an anti-static container, static shielding bag or conductive material. All test and measurement tools including work bench and floor should be grounded. The operator should be grounded using wrist strap. Semiconductor devices must not be touched with bare hands. Similar precautions need to be taken for PW boards with semiconductor devices on it.

② HANDLING OF UNUSED INPUT PINS FOR CMOS

Note:

No connection for CMOS device inputs can be cause of malfunction. If no connection is provided to the input pins, it is possible that an internal input level may be generated due to noise, etc., hence causing malfunction. CMOS devices behave differently than Bipolar or NMOS devices. Input levels of CMOS devices must be fixed high or low by using a pull-up or pull-down circuitry. Each unused pin should be connected to V_{DD} or GND with a resistor, if it is considered to have a possibility of being an output pin. All handling related to the unused pins must be judged device by device and related specifications governing the devices.

③ STATUS BEFORE INITIALIZATION OF MOS DEVICES

Note:

Power-on does not necessarily define initial status of MOS device. Production process of MOS does not define the initial operation status of the device. Immediately after the power source is turned ON, the devices with reset function have not yet been initialized. Hence, power-on does not guarantee out-pin levels, I/O settings or contents of registers. Device is not initialized until the reset signal is received. Reset operation must be executed immediately after power-on for devices having reset function.

Purchase of NEC I^2C components conveys a license under the Philips I^2C Patent Rights to use these components in an I^2C system, provided that the system conforms to the I^2C Standard Specification as defined by Philips.

The export of this product from Japan is regulated by the Japanese government. To export this product may be prohibited without governmental license, the need for which must be judged by the customer. The export or re-export of this product from a country other than Japan may also be prohibited without a license from that country. Please call an NEC sales representative.

M7 96.5

Regional Information

Some information contained in this document may vary from country to country. Before using any NEC product in your application, please contact the NEC office in your country to obtain a list of authorized representatives and distributors. They will verify:

- Device availability
- Ordering information
- Product release schedule
- Availability of related technical literature
- Development environment specifications (for example, specifications for third-party tools and components, host computers, power plugs, AC supply voltages, and so forth)
- Network requirements

In addition, trademarks, registered trademarks, export restrictions, and other legal issues may also vary from country to country.

NEC Electronics Inc. (U.S.)
Santa Clara, California
Tel: 408-588-6000
800-366-9782
Fax: 408-588-6130
800-729-9288

NEC Electronics (Germany) GmbH
Duesseldorf, Germany
Tel: 0211-65 03 02
Fax: 0211-65 03 490

NEC Electronics (UK) Ltd.
Milton Keynes, UK
Tel: 01908-691-133
Fax: 01908-670-290

NEC Electronics Italiana s.r.1.
Milano, Italy
Tel: 02-66 75 41
Fax: 02-66 75 42 99

NEC Electronics (Germany) GmbH
Benelux Office
Eindhoven, The Netherlands
Tel: 040-2445845
Fax: 040-2444580

NEC Electronics (France) S.A.
Velizy-Villacoublay, France
Tel: 01-30-67 58 00
Fax: 01-30-67 58 99

NEC Electronics (France) S.A.
Spain Office
Madrid, Spain
Tel: 01-504-2787
Fax: 01-504-2860

NEC Electronics (Germany) GmbH
Scandinavia Office
Taeby, Sweden
Tel: 08-63 80 820
Fax: 08-63 80 388

NEC Electronics Hong Kong Ltd.
Hong Kong
Tel: 2886-9318
Fax: 2886-9022/9044

NEC Electronics Hong Kong Ltd.
Seoul Branch
Seoul, Korea
Tel: 02-528-0303
Fax: 02-528-4411

NEC Electronics Singapore Pte. Ltd.
United Square, Singapore 1130
Tel: 65-253-8311
Fax: 65-250-3583

NEC Electronics Taiwan Ltd.
Taipei, Taiwan
Tel: 02-719-2377
Fax: 02-719-5951

NEC do Brasil S.A.
Cumbica-Guarulhos-SP, Brasil
Tel: 011-6465-6810
Fax: 011-6465-6829

J98. 2

MAJOR REVISIONS IN THIS VERSION

Page	Contents
Throughout	Assembler changed (AS17K → RA17K)
Throughout	In-circuit emulator IE-17K-ET added
p. 8	**Related Documents** in **INTRODUCTION** changed
in previous edition	**μPD170×× Product Development** and **List of Functions** in **CHAPTER 1 GENERAL** deleted
p. 79	**6.7.3 Notes on using general register pointer** added
p. 102	Diagram of the relationship between program status word (PSWORD) and status flip-flop in **Figure 8-1. Configuration of ALU Block** added
p. 104	Operation and description of instructions added to **Table 8-1. ALU Processing Instructions**
p. 106	The following descriptions added to **8.2.3 Status flip-flop functions**: (1) Z flag (2) CY flag (3) CMP flag (4) BCD flag
p. 108	**Table 8-2. Results for Binary 4-bit and BCD Operations** changed
p. 110	**Table 8-3. Arithmetic Operation Instructions** added
p. 121	**Table 8-4. Logical Operation Instructions** added
p. 124	**Table 8-6. Bit Testing Instructions** added
p. 126	**Table 8-7. Compare Instructions** added
p. 138	**Example 3** added to **9.4.2 Symbol definition of register file and reserved words**
p. 160	Description added to **12.2.6 Interrupt enable flip-flop (INTE)**
p. 183	Remarks added to: **13.4.3 Releasing halt status by key input** **13.4.4 Releasing halt status by timer carry (basic timer 0 carry)**
p. 183, 184	Remark and Caution added to **13.4.5 Releasing halt status by interrupt**
in previous edition	**12.6 Current Dissipation in Halt and Clock Stop Modes** in previous edition deleted
in previous edition	**CHAPTER 14 ONE-TIME PROM MODEL** in previous edition deleted
p. 198	Description added to **14.4 Power-ON Reset**
p. 282	**15.5.9 (3) SYSCAL entry** added
p. 289, 290	**A.1 Hardware** and **A.2 Software** changed
p. 293	**C.1 Instruction Index (by function)** added
p. 295	**APPENDIX D REVISION HISTORY** added

The mark ★ shows major revised points in this edition.

INTRODUCTION

Targeted reader

This manual is intended for the users who understand the functions of the μPD170×× series microcontrollers and design application systems using these microcontrollers.

Objective

This manual describes the functions common to all the models in the μPD170×× series, and will serve as a reference manual when you develop a program for a μPD170×× series microcontrollers.

How to read this manual

It is assumed that the readers of this manual possess general knowledge about electric engineering, logic circuits, and microcomputers.
Since the number of registers and memory capacity differ depending on the model of the microcontroller, the maximum number and permissible range are described in this manual. For the exact values, refer to the Data Sheet for each microcontroller.
The hardware peripherals are not described in this manual. For the hardware peripherals, refer to the Data Sheet for each microcontroller.

- To understand the overall functions of the μPD170×× series,
  → Read this manual using the Contents.

- To understand the function of an instruction whose mnemonic is known,
  → Use the **APPENDIX C INSTRUCTION INDEX**.

- To understand the function of the instruction whose mnemonic is not known but whose function is known,
  → Refer to **15.3 Instruction List** by referring to **15.5 Instruction Functions**.

- To learn the electrical specifications of the μPD170×× series,
  → Refer to the Data Sheet for the respective models.

Legend

Data significance	: Higher digit on left, lower digit on right
Active low	: $\overline{\times\times\times}$ (bar over pin and signal names)
Memory map address	: Top-low, bottom-high
Note	: Description of **Note** in the text.
Caution	: Information requiring particular attention
Remark	: Supplementary explanation
Number	: Binary ... ××××B
	Decimal number ... ×××× or ××××D
	Hexadecimal number ... ××××H

★ **Related Documents** Also use the following documents.
The Data Sheet of each device, and the User's Manuals of the SE board and device files are also available.

Document Name		Document No.	
		Japanese	English
17K Series/DTS Standard Models Selection Guide		U10317J	U10317E
RA17K User's Manual		U10305J	U10305E
IE-17K/IE-17K-ET CLICE/CLICE-ET User's Manual		U10063J	U10063E
SIMPLEHOST™ *emIC-17K*™/RA17K Compatible	Introduction	U10445J	U10445E
User's Manual	Reference	U10496J	U10496E
17K Series Project Manager User's Manual	Reference	U12810J	EEU-1527
MAKE/CNV17K User's Manual		U10596J	U10596E
emIC-17K User's Manual	Reference	U12829J	U12829E
LK17K User's Manual		U12518J	U12518E
DOC17K User's Manual		EEU-5006	EEU-1536

TABLE OF CONTENTS

LIST OF FIGURES (1/3)

LIST OF FIGURES (2/3)

LIST OF FIGURES (3/3)

LIST OF TABLES

CHAPTER 1 GENERAL

μPD170×× is a lineup of models in the 17K Series 4-bit microcontrollers. These models have functions for TV and AM/FM radio applications, such as PLL circuit necessary for station selection and voltage synthesizer circuit.

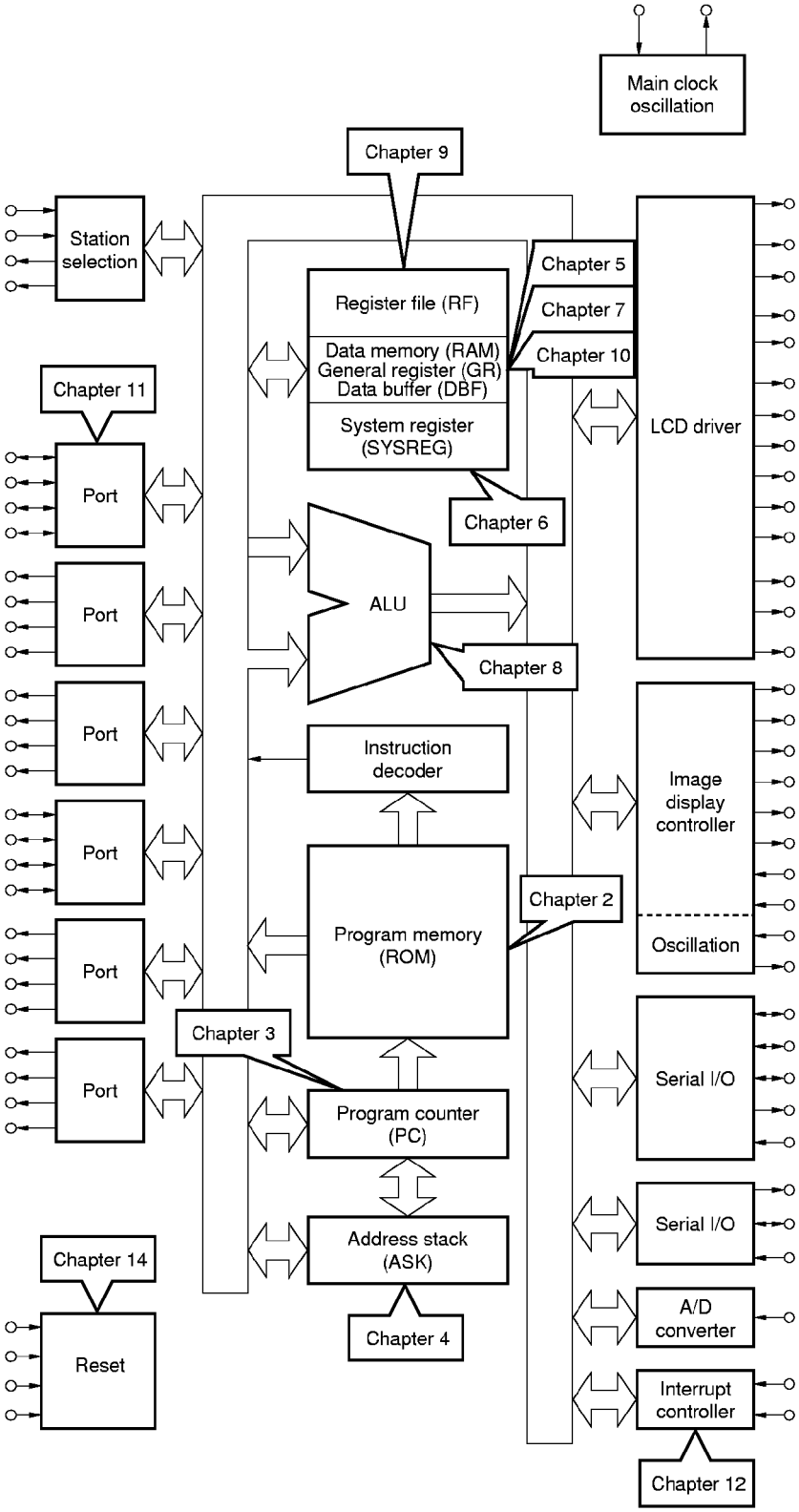
The μPD170×× series microcontrollers have the CPU commonly employed in the 17K series with registers connecting and controlling the station selection circuit and synthesizer circuit.

Each model has a different station selection circuit. Since this manual describes common features for the μPD170×× series, for details on the station selection circuit and features peculiar to each model, refer to the Data Sheet for the model.

μPD17P0×× is a one-time PROM model of μPD170××, which is convenient for evaluation of a developed system or small-scale production.

1.1 Internal Configuration of μPD170×× Subseries

The following block diagram indicates which chapter describes each functional block. Note that this diagram does not necessarily show the actual internal block for the μPD170×× series.



CHAPTER 2 PROGRAM MEMORY (ROM)

The program memory stores the "program", which is executed by the CPU (central processing unit), and predetermined "constant data".

As the program memory, μPD170×× is provided with a mask ROM (read-only memory), and μPD17P0××, with an EPROM (electrically erasable ROM).

2.1 Program Memory Configuration

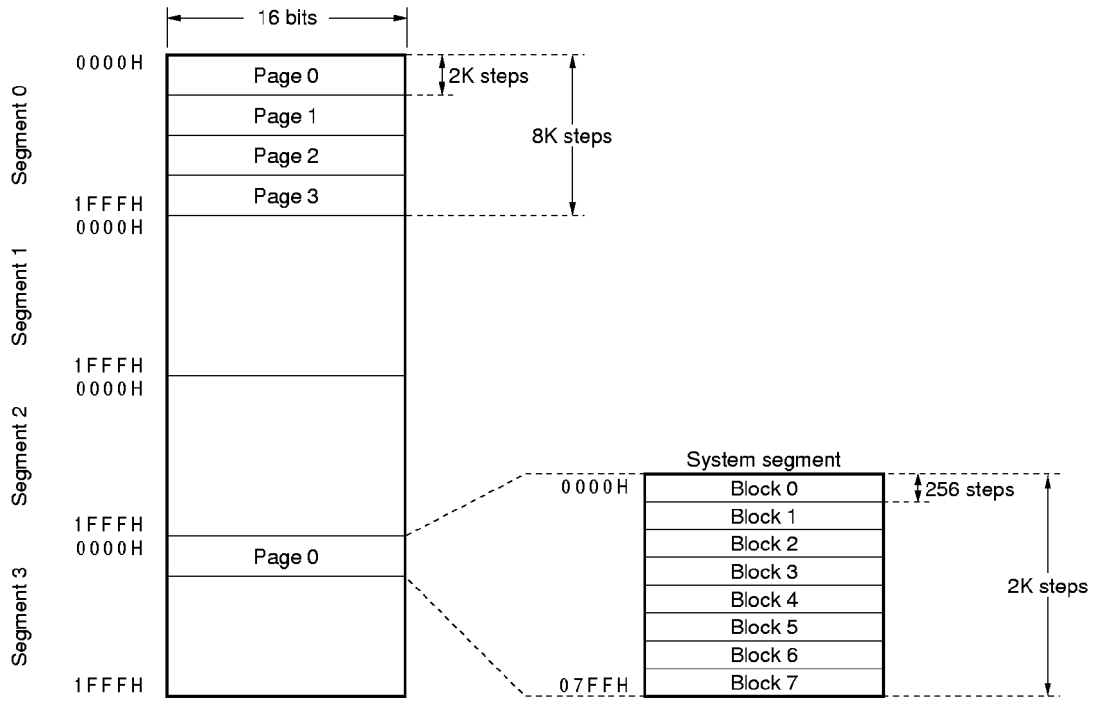
As shown in Figure 2-1, the program memory (ROM) consists of several steps, with each step made up of 16 bits. Each step is assigned as "address".

Each 2K steps in the program memory constitute a "page".

Four pages form a "segment". Therefore, one page has addresses 0000H through 1FFFH.

One of the segments, called the "system segment", consists of several blocks, with each block consisting of 256 steps.

Figure 2-1. Configuration of Program Memory (ROM)



2.2 Program Memory Functions

Broadly speaking, the program memory has the following three functions:

(1) To store programs
(2) To store constant data
(3) To store data for peripheral functions

A program is a collection of "instructions" according to which the CPU (Central Processing Unit: a functional block that actually controls the microcontroller) operates. The CPU sequentially executes processing in accordance with the "instructions" written in a program. Specifically, the CPU sequentially reads the "instructions" of the program stored in the program memory and executes processing according to each "instruction".

The "instructions" are all "one-word instruction" 16 bit long; therefore, one instruction is stored at one address in the program memory.

The constant data is predetermined data. The program memory contents, including the constant data, can be read to the data buffer (DBF) on the data memory (RAM) by executing special instruction MOVT. Reading constant data from the program memory is called "table reference".

Since the program memory is a read-only memory, its contents cannot be rewritten by an instruction. This is why the program memory is also referred to as "ROM" (Read-Only Memory).

2.3 Program Flow

The program operation flow is controlled by a program counter (PC) that specifies an address in the program memory.

The program stored in the program memory is usually executed on an address-by-address basis, starting from address 0000H. However, if a different program is to be executed, when a certain condition is satisfied, the program execution flow must be changed (branched). In such a case, a branch (BR) instruction is used.

When the same portion of the program is to be executed over and over again, the execution efficiency will be degraded, unless the execution sequence is altered, because the program is usually executed from address 0000H. To enhance the execution efficiency, that portion of the program to be executed repeatedly should be stored at one place and this portion should be called by special instruction CALL. This portion of the program is called a "subroutine". As opposed to the subroutine, the portion of the program that is usually executed is called the "main routine".

Some programs should be executed only when a certain condition is satisfied, regardless of the flow of the main routine. In this case, the interrupt function is used, which cause the execution to branch to a predetermined address (called a vector address), regardless of the program current flow, when a specified event occurs.

2.4 Branching Program

The program is branched by the branch (BR) instructions.

The branch (BR) instructions are classified into two categories: the direct branch instruction (BR addr), which causes the execution to directly branch to a program memory address (addr) specified by the operand of the instruction, and the indirect branch instruction (BR @AR), which causes the execution to branch to a program memory address specified by the contents of an address register (AR) to be described shortly.

For details, also refer to **CHAPTER 3 PROGRAM COUNTER (PC)**.

2.4.1 Direct branch

With the direct branch instruction, the branch destination program memory address is specified by the low-order 2 bits in the op code for an instruction and 11 bits in the operand for the instruction, totaling 13 bits. Therefore, any address in a segment, address 0000H to 1FFFH, can be specified as the branch destination address by the direct branch instruction. Note that the execution cannot be branched from one segment to another.

2.4.2 Indirect branch

With the indirect branch instruction, the branch destination address is set by the program in the address register, as shown in (2) in Figure 2-2. Consequently, the address range in which the branch destination can be specified differs, depending on the number of bits in the address register.

The indirect branch instruction allows branching from one segment to another.

For details, refer to **6.3 Address Register (AR)**.

2.4.3 Notes on debugging

As shown in (1) in Figure 2-2, the operation code for a direct branch instruction differs, depending on the page to which the execution is to be branched.

For example, the operation code for the instruction that branches the execution in page 0 is "0CH", while that for the instruction that branches the execution in page 1 is "0DH". The operation code for the instruction that branches the execution to page 2 is "0EH", and that for the instruction that branches the execution to page 3 is "0FH".

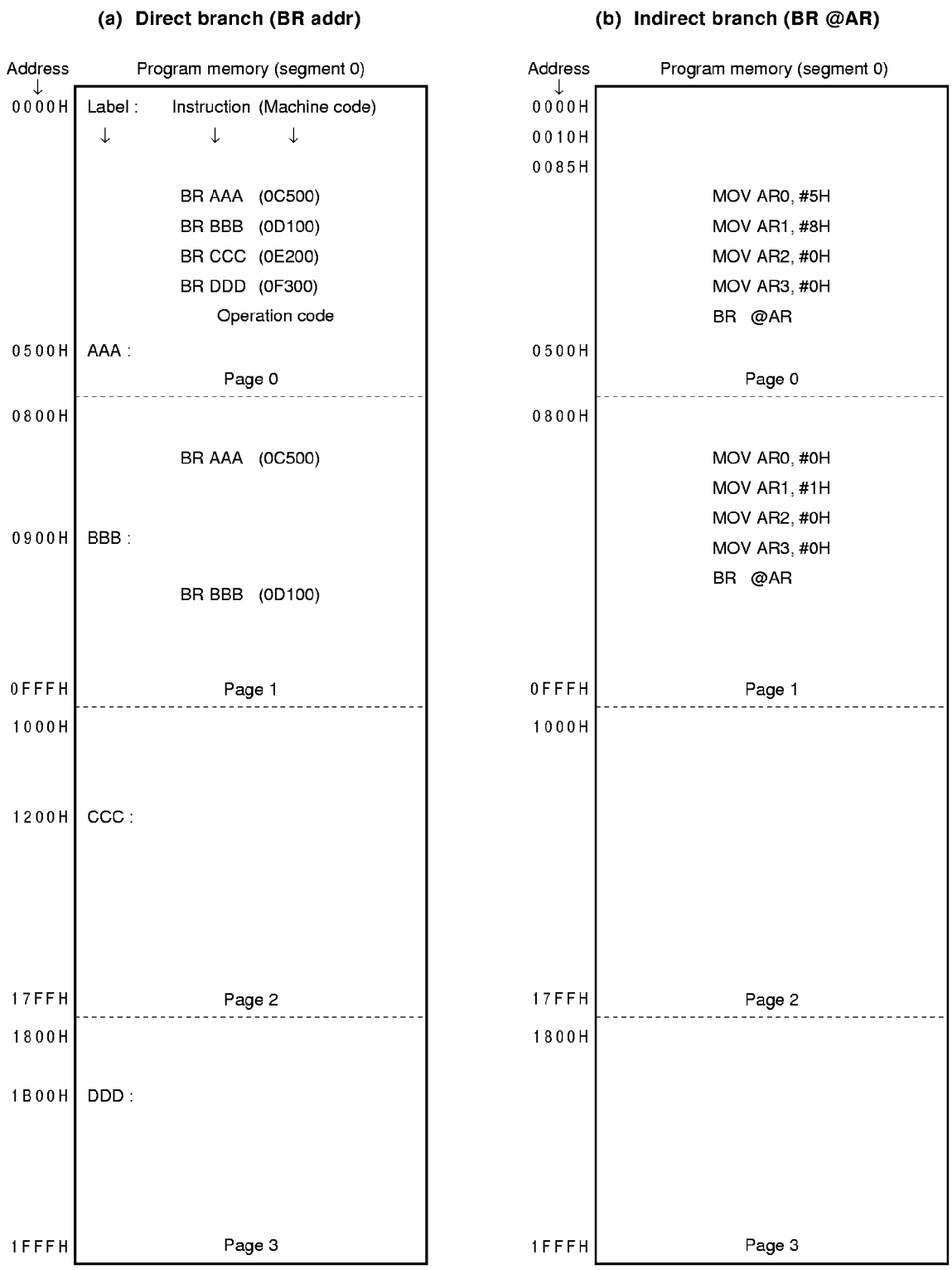
This is because the low-order 2 bits in the operation code are used to specify the branch destination address, as the operand for the direct branch instruction, "addr", has only 11 bits.

When these operation codes are assembled by the 17K Series Assembler (RA17K), the jump destination specified by a label is automatically referenced and converted by the Assembler.

However, when performing patching while debugging the program, the programmer must understand the page to which the execution is to be branched, and convert the operation code.

For example, when patching address 0900H for BBB in (1) in Figure 2-2 into address 0910H, input "0D110" as the machine code for the "BR BBB" instruction.

Figure 2-2. Operations of Branch Instructions and Machine Codes



2.5 Subroutine

The subroutine is used with the subroutine call (CALL) and subroutine return (RET or RETSK) instructions.

The subroutine call instructions are classified into two categories: the direct subroutine call instruction (CALL addr), which directly calls a program memory address (addr) specified by the operand of the instruction, and the indirect subroutine call instruction (CALL @AR), which calls a program memory address specified by the contents of the address register.

In addition, a system call instruction (SYSCAL entry), that branches the execution to the system segment, is also available.

To return from a subroutine, the RET and RETSK instructions are used. By executing these instructions, the execution is returned to a program memory address next to the one at which the subroutine call (CALL) instruction was executed. At this time, the RETSK instruction executes the first instruction after the return as no operation (NOP) instruction.

For details, refer to **CHAPTER 3 PROGRAM COUNTER (PC)**.

2.5.1 Direct subroutine call

The direct subroutine call instruction specifies the program memory address to be called by 11 bits in the operand for an instruction. Therefore, when the direct subroutine call instruction is used, the called address, i.e., the first address in the subroutine, must be in page 0 (address 0000H to 07FFH), as in (1) in Figure 2-3. The subroutine whose call address is not in page 0 cannot be called.

However, the direct subroutine call instruction and the subroutine return (RET or RETSK) instruction can be in pages other than page 0.

The direct subroutine call instruction cannot call a subroutine from one segment to another.

Examples 1. When return instruction is in page 1

As shown in Figure 2-4, the return address and return instruction can be in any page, as long as the first address for the subroutine is in page 0.

As long as the first address for the subroutine is in page 0, the CALL instruction can be used without being restricted by the concept of the page. However, if the first address for the subroutine cannot be placed in page 0, follow the action described in Example 2 below.

2. When first address is in page 1

As shown in Figure 2-4, use a branch instruction (BR) in page 0, and call the actual subroutine (SUB1) through this BR instruction.

2.5.2 Indirect subroutine call

The indirect subroutine call instruction (CALL @AR) specifies the address to which the execution is to be branched by using the address register (AR), as shown in (2) in Figure 2-3. Therefore, the range for the program memory address, in which the execution can be branched by this instruction, varies depending on the number of bits in the address register.

For details, refer to **6.3 Address Register (AR)**.

The indirect subroutine call instruction can call a subroutine from one segment to another.

Figure 2-3. Subroutine Call Instructions Operations

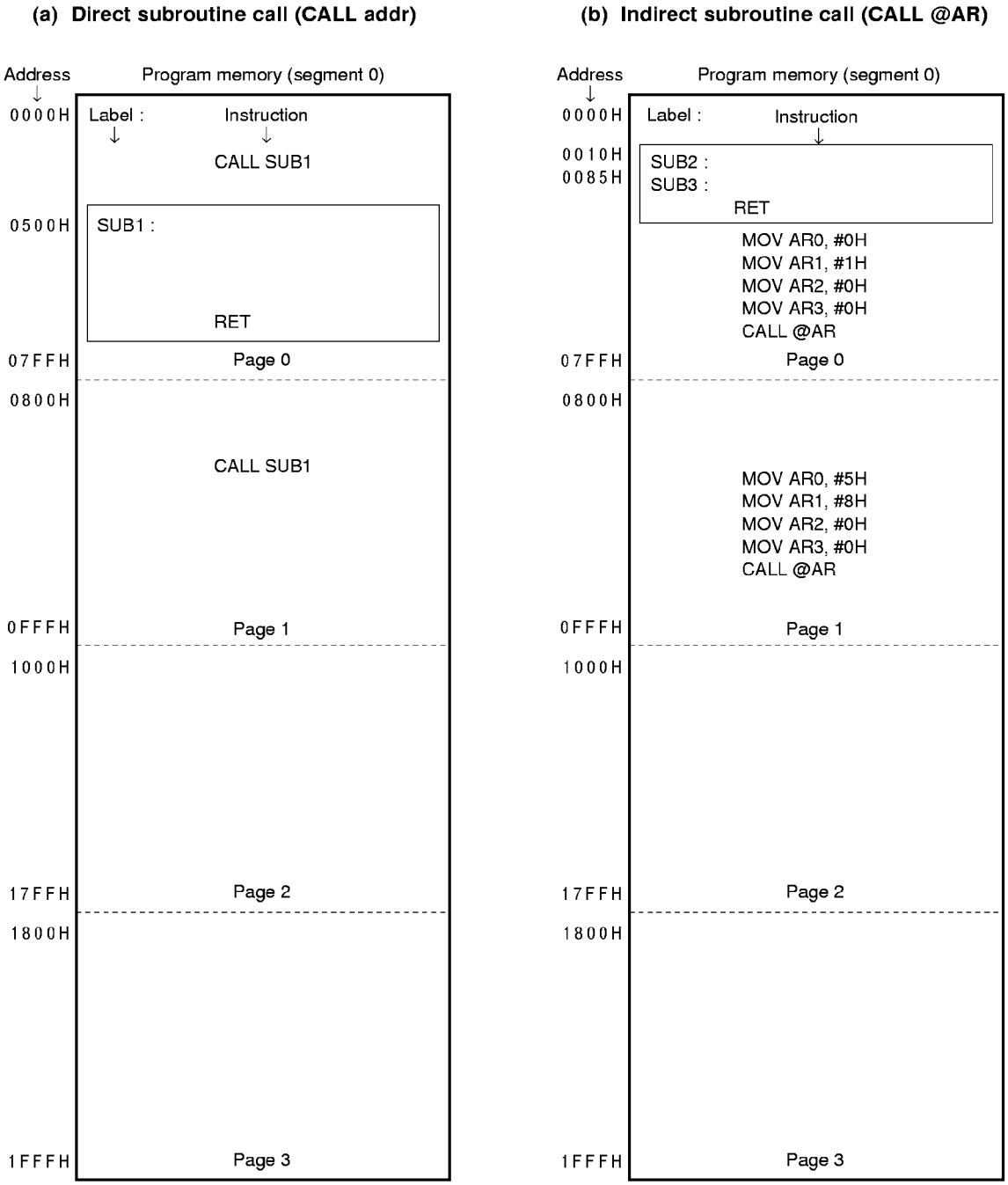
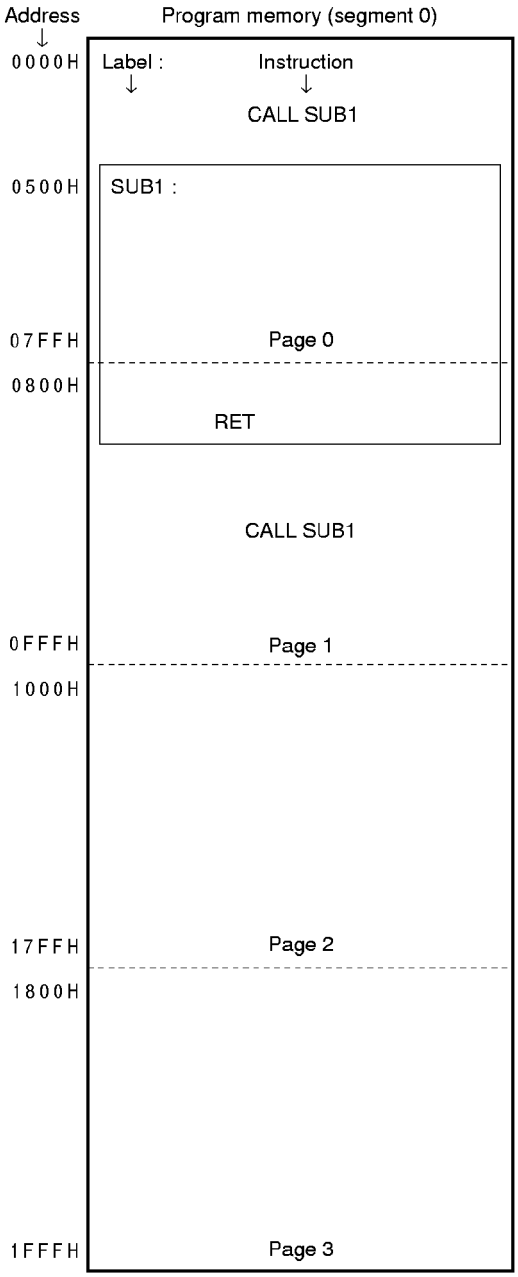
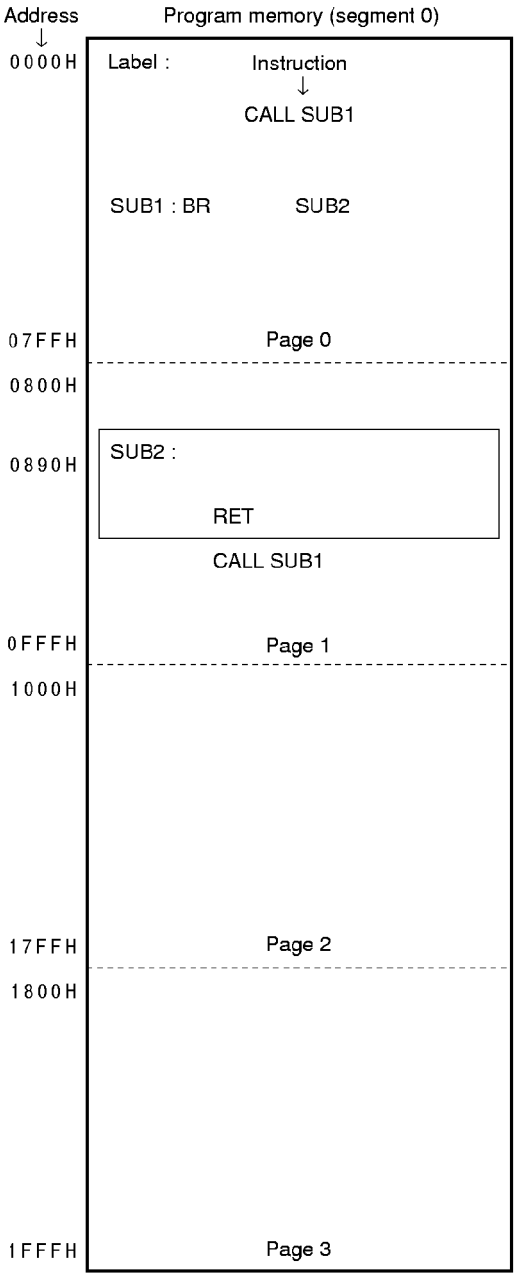


Figure 2-4. Using Subroutine Call Instructions

(1) When subroutine return instruction is in page 1



(2) When first address for subroutine is in page 1



2.6 System Call

The system call instruction (SYSCAL entry) allows the execution to branch from each segment to a subroutine with single instruction in the system segment.

The operand "entry" for of this instruction can specify the program memory address to which the execution is to be branched. The high-order 3 bits of the 7 bits for the operand specify a block, while the low-order 4 bits specify an address. Therefore, only the first 16 steps for each block (block 0 to 7 in page 0 of the system segment) can be specified. For details, refer to **3.2.7 System call**.

The shaded portion in Figure 2-5 indicates the program memory range to which the execution can be branched by the system call instruction. The SYSCAL instruction or subroutine return instruction (RET or RETSK) can be anywhere.

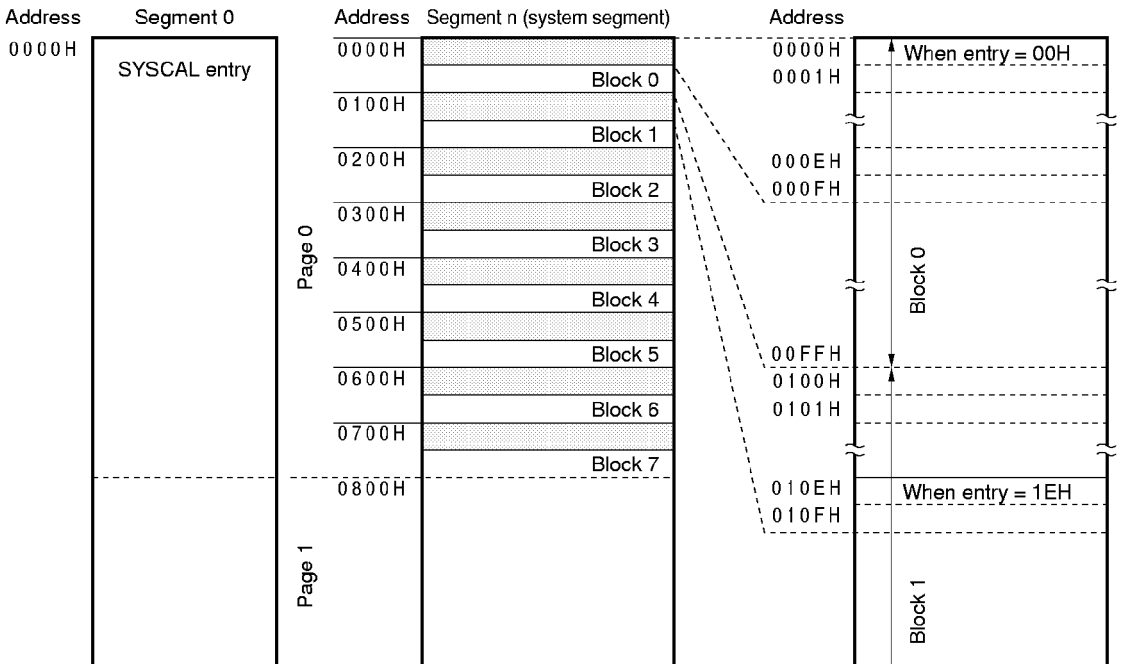
Examples 1. When SYSCAL instruction is in segment 0

As shown in Figure 2-5, the execution branches to the system segment specified by the operand "entry", when the SYSCAL instruction in segment 0 is executed.

If the operand is 00H (entry = 00H) at this time, the execution branches to address 00H (0000H) in block 0.

If the operand is 1EH (entry = 1EH), the execution branches to address 0EH (010EH) in block 1.

Figure 2-5. Using System Call Instruction



Examples 2. Program using system call instruction

```
  ;
  ; module 1 (segment 0)
  :

EXTERN LAB ENTRY0
EXTERN LAB ENTRY1
EXTERN LAB ENTRY2
EXTERN LAB ENTRY3

SYSCAL.DL. (ENTRY0 SHR 4 AND 0070H) OR (ENTRY0 AND 000FH)
SYSCAL.DL. (ENTRY1 SHR 4 AND 0070H) OR (ENTRY1 AND 000FH)
SYSCAL.DL. (ENTRY2 SHR 4 AND 0070H) OR (ENTRY2 AND 000FH)
SYSCAL.DL. (ENTRY3 SHR 4 AND 0070H) OR (ENTRY3 AND 000FH)

;
; module 2 (segment 1)
:
CSEG1
PUBLIC ENTRY0, ENTRY1, ENTRY2, ENTRY3

ORG 2000H

ENTRY0:
  BR SUB0
ENTRY1:
  BR SUB1

ORG 2100H

ENTRY2:
  BR SUB2
ENTRY3:
  BR SUB3
```

2.7 Table Referencing

Table referencing is used to reference the constant data in the program memory. When the MOVT DBF, @AR instruction is executed, the program memory address contents, specified by the address register, are stored in the data buffer (DBF).

Since the program memory contents consist of 16 bits, the constant data stored in the data buffer by the MOVT instruction is 16 bits (4 words) long. The program memory address that can be referenced by the MOVT instruction is in the range the address register of each model can specify.

When table referencing is performed, one level of the stack is used.

For details, refer to **6.3 Address Register (AR)** and **10.4 Data Buffer and Table Reference**.

2.8 Notes on Using Operand for Branch and Subroutine Call Instructions

An error occurs, if a program memory address is directly (in numeral) specified as the operand for the branch (BR) and subroutine call (CALL) instructions, as shown in Example 1 below, when the 17K Series Assembler (RA17K) is used.

This feature to generate an error is incorporated in the assembler to reduce the causes of bugs that may occur, when the program is edited.

Use label as the operand for the branch (BR) and subroutine call (CALL) instructions.

Examples 1. Error occurs

```
; <1>
        BR      0005H    ; Error occurs during assembly
; <2>
        CALL    00F0H    ;
```

2. Error does not occur

```
; <3>
  LOOP1:                 ; BR or CALL instruction is
        BR      LOOP1    ; executed to label in program
; <4>
  SUB1:                  ;
        CALL    SUB1     ;
; <5>
  LOOP2 LAB     0005H    ; 0005H is assigned to LOOP2
        BR      LOOP2    ; as label type
; <6>
        BR. LD. 0005H    ; Converts operand value into label type.
                         ; This method, should not be used often, to reduce causes of bugs
```

For details, refer to **RA17K Assembler User's Manual (U10305E)**.

CHAPTER 3 PROGRAM COUNTER (PC)

The program counter specifies an address in the program memory.

3.1 Program Counter Configuration

The program counter is a 13-bit binary counter and a segment register (SGR) of up to 3 bits, as shown in Figure 3-1.

Figure 3-1. Configuration of Program Counter

MSB LSB

SGR2	SGR1	SGR0	PC_{12}	PC_{11}	PC_{10}	PC_9	PC_8	PC_7	PC_6	PC_5	PC_4	PC_3	PC_2	PC_1	PC_0

3.2 Program Counter Functions

The program counter selects an address containing an instruction to be actually executed or constant data to be used from several instructions or constant data written in the program memory.

Usually, the program counter contents are incremented by one, each time an instruction has been fetched. When the branch (BR), subroutine call (CALL), return (RET, RETSK, RETI), or table reference (MOVT) instruction has been executed, and when an interrupt has been accepted, a specified address value is stored in the program counter and the instruction at that address is executed.

3.2.1 through **3.2.7** describe the program counter operations, when each of the above instructions is executed.

3.2.1 When branch (BR) instruction is executed

Two kinds of branch instructions are available: the direct branch instruction (BR addr), which directly specifies the branch destination, and the indirect branch instruction (BR @AR), which indirectly specifies the branch destination by the contents of the address register (AR) to be described shortly.

When the direct branch instruction (BR addr) is executed, the value specified by the low-order 11 bits in the operand for the instruction is stored in the program counter as is, as shown in Figure 3-2. Since the low-order 2 bits in the operation code (the high-order 5 bits) for the instruction are added to bits 12 (b_{12}) and 11 (b_{11}) in the program counter, the range in which the execution can be branched is the same segment (address 0000H through 1FFFH) of the program memory that can be specified by a total of 13 bits.

The operation code for the instruction is determined by the Assembler (RA17K) that automatically searches for the branch destination position.

When the indirect branch instruction (BR @AR) is executed, the contents of the address register (AR) in the system register are written to the program counter to specify the branch destination, as shown in Figure 3-2. The low-order 13 bits in the address register are written to the program counter, and the high-order bits (3 bits max.) are written to the segment register. Therefore, the branch range differs, depending on the number of bits in the address register for the model used.

3.2.2 When subroutine call (CALL) or subroutine return (RET, RETSK) instruction is executed

Two kinds of subroutine call instructions are available: the direct subroutine call (CALL addr), which directly specifies the call destination, and the indirect subroutine call (CALL @AR) instruction, which indirectly specifies the call destination by the address register (AR) contents.

When the direct subroutine call instruction (CALL addr) is executed, the value specified by the instruction operand is stored in the program counter as is, as shown in Figure 3-2. Since the operand is 11 bits long, the high-order 2 bits in the program counter (b_{11} and b_{12}) are fixed to 0. Consequently, the range in which the execution can be branched, by the direct subroutine call instruction, is address 0000H to 07FFH on page of the memory program.

When the indirect subroutine call instruction (CALL @AR) is executed, the address register contents in the system register are written to the program counter, to specify the branch destination. Therefore, the range in which the execution can be branched by the indirect subroutine call instruction differs, depending on the number of bits in the address register.

When the subroutine return instruction (RET, RETSK) is executed, the address stack register (ASR) contents, specified by the stack pointer (SP), i.e., the return address, are written to the program counter.

For the details on the address stack, refer to **CHAPTER 4 ADDRESS STACK**.

3.2.3 When table reference (MOVT) instruction is executed

When the table reference (MOVT DBF, @AR) instruction is executed, the address register (AR) contents are stored in the program counter, as shown in Figure 3-2, and the specified program memory contents are read to the data buffer. Therefore, the range in which table referencing can be executed differs, depending on the number of bits in the address register.

After the program memory contents have been read to the data buffer, the address, next to the one at which the table reference instruction is executed, is written to the program counter, and the subsequent program is executed. Note that one stack level is used at this time. Exercise care not to exceed the permitted stack level, when using the table referencing instruction in a subroutine or interrupt processing.

Also note that, to execute one table reference instruction, two instruction cycles are required.

3.2.4 When interrupt is accepted and when interrupt return (RETI) instruction is executed

When an interrupt has been accepted, an vector address (branch destination address), specified by the interrupt, is stored in the program counter, as shown in Figure 3-2.

When the interrupt return (RETI) instruction has been executed, the return address, written to the address stack specified by the stack pointer, is restored to the program counter.

For details, refer to **CHAPTER 12 INTERRUPT FUNCTION**.

3.2.5 When skip instruction is executed

When the skip instruction (SKT, SKF, SKE, etc.) has been executed, the address, next to the one containing the skip instruction, is stored in the program counter, regardless of the skip condition contents. When the subroutine return skip (RETSK) instruction is executed, the address stack register (ASR) contents, specified by the stack pointer, are stored in the program counter.

If the instruction executed causes the execution to skip (such as RETSK), the subsequent instruction is executed as a no-operation (NOP) instruction; therefore, the number of instructions to be executed is the same as when the skip instruction has been executed, regardless of whether or not the instruction next to the skip instruction is skipped.

3.2.6 On reset

When power-ON reset (V_{DD} = low to high) or CE reset (CE = low to high) has been executed, the program counter contents are reset to 0000H, and the segment register (SGR) contents in the program counter are reset to 0. Consequently, the program is executed from address 0 in segment 0.

When the clock stop instruction (STOP s) is executed, the program is stopped at the address for this instruction. When the clock stop instruction is released (CE = low to high), the program is executed from address 0 in segment 0.

Also refer to **CHAPTER 14 RESET FUNCTION**.

3.2.7 On system call instruction execution

When the system call (SYSCAL entry) instruction is executed, the operand "entry" for the instruction is stored in the program counter. Since the operand is 7 bits long, the segment value for the system segment is written to the segment register (SGR) in the program counter and 0 is written to the remaining 6 bits, to specify an address for the system segment.

Figure 3-2. Program Counter Setting When Each Instruction Is Executed

Instruction \ Program counter		Contents of program counter (PC)															
		SGR2	SGR1	SGR0	b_{12}	b_{11}	b_{10}	b_9	b_8	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
BR addr	Page 0	Not changed			0	0	← Operand of instruction (addr) →										
	Page 1				0	1											
	Page 2				1	0											
	Page 3				1	1											
CALL addr					0	0	← Operand of instruction (addr) →										
BR @AR CALL @AR MOVT DBF, @AR		← Contents of address register (b_{13}-b_0 of AR) →															
RET RETSK RETI		← Contents of address stack register (ASR) specified by stack (Return address) →															
When interrupt is accepted		0	0	0	← Vector address of each interrupt →												
Power-ON reset, CE reset		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
SYSCAL entry		Value of system segment			0	0	Operand of instruction			0	0	0	0	Operand of instruction			

3.3 Segment Register (SGR)

The segment register specifies a segment of the program memory.

SGR0-SGR2 in Figure 3-2 show the segment register operations, when each instruction is executed.

The segment register is set when the SYSCAL instruction is executed, and when the indirect branch or direct subroutine call instruction is executed.

The segment register is reset to 0 on power-ON reset or CE reset.

3.4 Notes on Using Program Counter

The program counter contents are incremented by one, each time an instruction is fetched.

If a branch (BR) or return (RET, RETSK, or RETI) instruction is at the end address (1FFFH) in the program memory, the next address specified by the program counter is 0000H. Consequently, the microcontroller may malfunction.

Moreover, if the program memory capacity is small; for example, if the end address is 0FEFH and its contents are an instruction other than the branch or return instruction, the next address specified by the program counter is 0FF0H, which also causes the microcontroller to malfunction.

Therefore, write the branch instruction (BR) to the last address for each segment. However, if the return address has been written to that address, it can remain untouched.

If an instruction causes the program counter value to exceed the permitted range, the Assembler (RA17K) generates a warning with an error.

CHAPTER 4 ADDRESS STACK

The address stack is a register that saves the program return address, when a subroutine is called or when an interrupt is accepted, or a table reference instruction is executed.

In addition to the address stack, an interrupt stack is also provided to which the system register contents are saved, when an interrupt has been accepted. For details on the interrupt stack, refer to **12.2.8 Interrupt stack**.

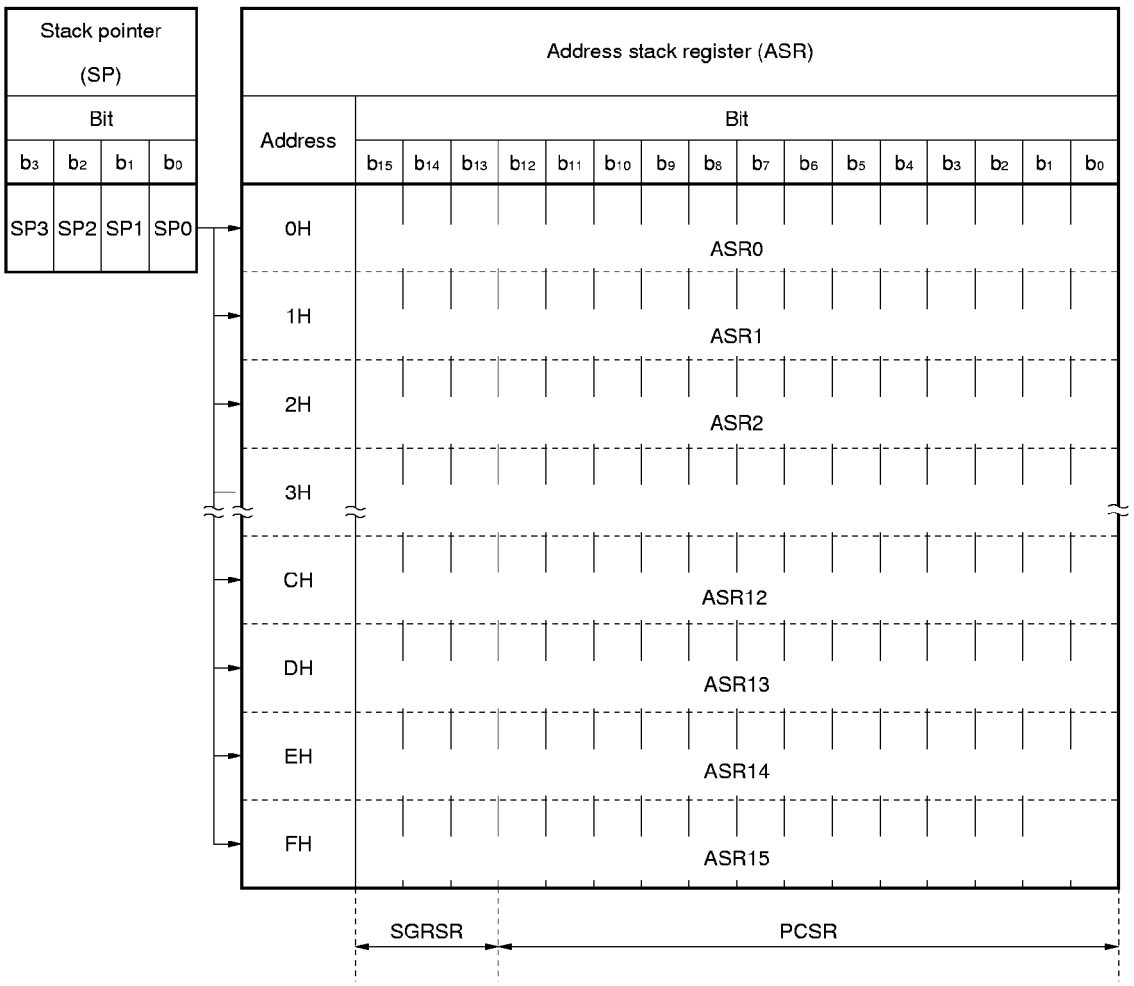
4.1 Address Stack Configuration

As shown in Figure 4-1, the address stack is made up of a stack pointer and address stack registers. The stack pointer is a 4-bit binary counter.

Up to 16 address stack registers, ASR0-ASRn ($n \leq 15$), are available, with each register consisting of up to 16 bits.

The low-order 13 bits in each address stack register form a program counter stack (PCSR) and the high-order 3 bits form a segment register stack (SGRSR), as shown in Figure 4-1.

Figure 4-1. Configuration of Stack



4.2 Address Stack Functions

The address stack saves the return address from a subroutine or interrupt routine or when a subroutine has been called or when an interrupt has been accepted or a table reference instruction is executed.

When the subroutine call (CALL addr, CALL @AR) instruction has been executed, the program memory address next to the one executing the subroutine call instruction, i.e., the return address, is saved to an address stack register ASR0-ASRn ($n \leq 15$). When the subroutine return instruction (RET, RETSK) is executed later, the return address, saved to the address stack register, is restored to the program counter.

The address stack is also used when the table reference instruction (MOVT DEF, @AR) is executed.

The address stack can be manipulated by stack manipulation instructions (POP AR and PUSH AR).

4.3 and **4.4** describe the functions for the stack pointer and address stack registers.

4.3 Stack Pointer (SP)

The stack pointer is a register that selects one of up to 16 address stack registers ASR0-ASRn ($n \leq 15$).

4.3.1 Stack pointer configuration

The stack pointer is a register consisting of up to 4 bits flags, as shown in Figure 4-2.

The stack pointer is in a control register in the register file. For details on the control register, refer to **CHAPTER 9 REGISTER FILE (RF)**.

Figure 4-2. Configuration of Stack Pointer

Name	Flag Symbol				Address	Read/Write
	b_3	b_2	b_1	b_0		
Stack pointer SP	SP3	SP2	SP1	SP0	Depends on model	R/W

				Specifies address of address stack register (ASR)
0	0	0	0	Address 0 (ASR0)
0	0	0	1	Address 1 (ASR1)
0	0	1	0	Address 2 (ASR2)
0	0	1	1	Address 3 (ASR3)
0	1	0	0	Address 4 (ASR4)
0	1	0	1	Address 5 (ASR5)
0	1	1	0	Address 6 (ASR6)
0	1	1	1	Address 7 (ASR7)
1	0	0	0	Address 8 (ASR8)
1	0	0	1	Address 9 (ASR9)
1	0	1	0	Address 10 (ASR10)
1	0	1	1	Address 11 (ASR11)
1	1	0	0	Address 12 (ASR12)
1	1	0	1	Address 13 (ASR13)
1	1	1	0	Address 14 (ASR14)
1	1	1	1	Address 15 (ASR15)

4.3.2 Stack pointer operation

The stack pointer contents are decremented by one, as shown in Table 4-1, during the first instruction cycle in the subroutine call (CALL addr, CALL @AR), system call (SYSCAL), or table reference instruction (MOVT DBF, @AR), or when an interrupt has been accepted, and incremented by one during the second instruction cycle of the subroutine return instruction (RET, RETSK), table reference instruction (MOVT DBF, @AR), stack manipulation (POP AR), or interrupt return (RETI) instruction.

Table 4-1. Operations of Stack Pointer (SP)

Instruction	Stack Pointer Value
CALL addr CALL @AR SYSCAL, entry MOVT DBF, @AR PUSH AR When interrupt is accepted	SP – 1
RET RETSK MOVT DBF, @AR POP AR RETI	SP + 1

For the operation of each instruction, refer to **4.5 Stack Operations, When Subroutine, Table Reference, or Interrupt Is Executed**.

Since the stack pointer is a binary counter, that can be up to 4 bits long, its value ranges from 0H to nH ($n \leq F$). However, there are fewer stack registers for some models than the maximum value of the stack pointer. In this case, if the stack pointer specifies a value corresponding to no stack register, the microcontrollers malfunction. For details, refer to **4.6 Nesting Level for Address Stack and PUSH and POP Instructions**.

Because the stack pointer is located on the register file, its value can be read or data can be written to it by manipulating the stack with the PEEK or POKE instruction. Although the stack pointer value is changed at this time, the address stack register contents are not affected.

4.4 Address Stack Registers

The address stack register is used to save the return address when a subroutine call instruction or table reference instruction is executed. When an interrupt is accepted, the return address of the program and the contents of the program status word (PSWORD) are automatically saved to the stack.

The address stack registers save the value of the program counter (PC) plus 1, i.e., the return address when the first instruction cycle of a subroutine call (CALL addr, CALL @AR), system call (SYSCAL entry), or table reference (MOVT DBF, @AR) instruction is executed. When a stack manipulation instruction (PUSH AR) is executed, the contents of the address register (AR) are saved to an address stack register. The address stack register that stores data is specified by the value of the stack pointer (SP) minus 1 when any of the above instructions is executed.

When the second instruction cycle of a subroutine return (RET, RETSK), interrupt return (RETI), or table reference (MOVT DBF, @AR) instruction is executed, the contents of the address stack register specified by the stack pointer are restored to the program counter, and the value of the stack pointer is incremented by 1. When a stack manipulation instruction (POP AR) is executed, the value of the address stack register specified by the stack pointer is transferred to the address register, and the value of the stack pointer is incremented by 1.

For the operation of each instruction, refer to **4.5 Stack Operations, When Each Subroutine, Table reference, or Interrupt Is Executed**.

The address stack register number is up to 16 (ASR0-ASRn where $n \leq 15$) and differs depending on the microcontroller model. If a subroutine is called or an interrupt occurs exceeding the maximum stack level for a microcontroller, the microcontroller malfunctions. For details, refer to **4.6 Nesting Level for Address Stack and PUSH and POP Instructions**, and **12.6 Nesting**.

4.5 Stack Operations, When Subroutine, Table Reference, or Interrupt Is Executed

4.5.1 through **4.5.4** describe the address stack operations.

4.5.1 When subroutine call (CALL) or return (RET, RETSK) instruction is executed

Table 4-2 below shows the operations for the stack pointer, address stack registers, and program counter, when the subroutine call or return instruction has been executed.

Table 4-2. Operation When Subroutine Call or Return Instruction Is Executed

Instruction	Operation
CALL addr	<1> Increments value of program counter (PC) by 1 <2> Decrements value of stack pointer (SP) by 1 <3> Saves value of program counter (PC) to address stack register (ASR) specified by stack pointer (SP) <4> Transfers value specified by operand (addr) of instruction to program counter
RET RETSK	<1> Restores value of address stack register (ASR) specified by stack pointer (SP) to program counter (PC) <2> Increments value of stack pointer (SP) by 1

When the RETSK instruction has been executed, the instruction to be executed first, after the program execution has returned from the subroutine to the main routine, is treated as a no-operation (NOP) instruction.

Figure 4-3 shows an operation example. In this example, the CALL instruction at address 100H in the main routine calls the subroutine at address 30H, and another CALL instruction at address 35H calls the subroutine at address 50H.

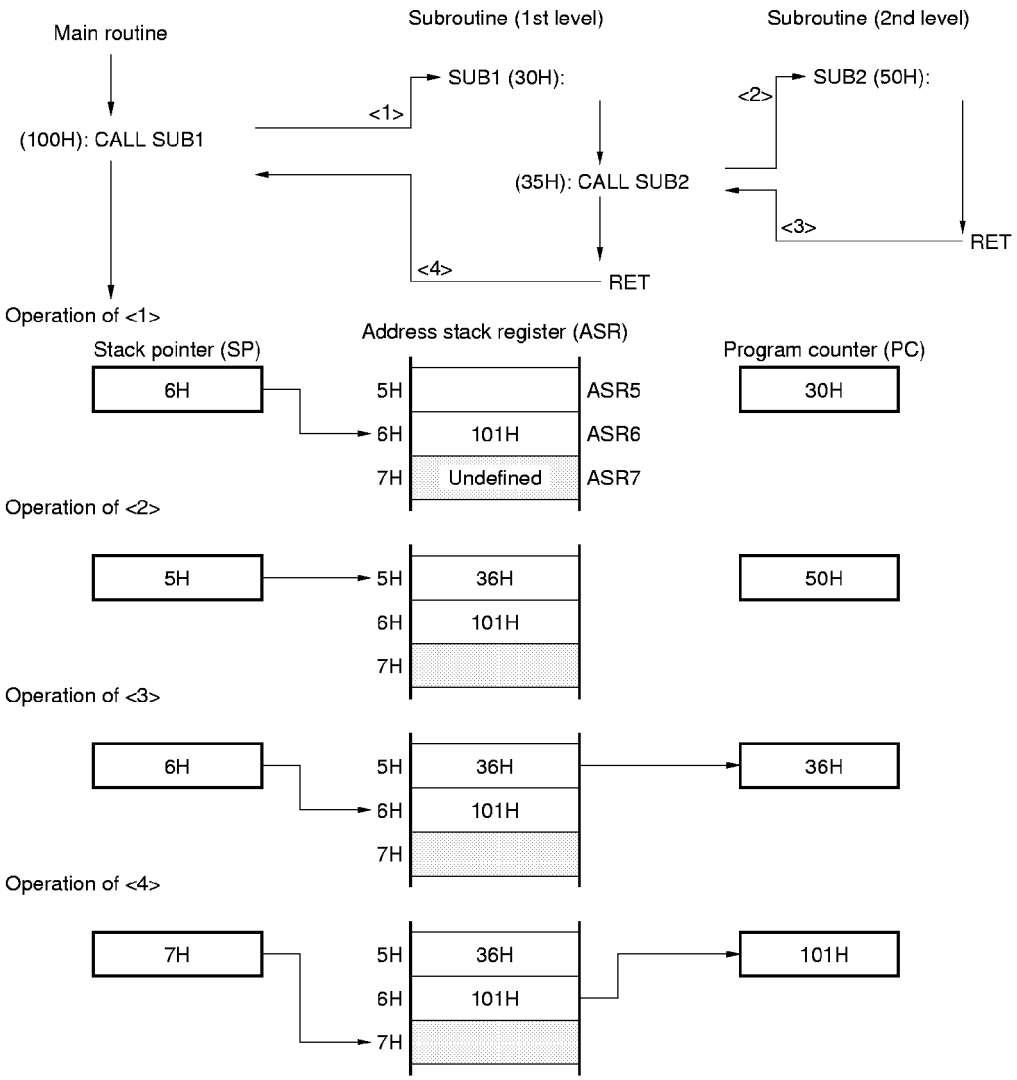
The subroutine, starting from address 30H, is called a subroutine for the "first level", while the subroutine, starting from address 50H, is called a subroutine for the "second level". The arrows in the figure show the program flow.

In this example, assume that the stack pointer value is 7H, before the instruction at address 100H is executed. Consequently, when the CALL instruction at address 100H is executed, the program counter contents become 101H, and the stack pointer value is decremented by one to 6H.

Next, address 101H, which is the return address from the first-level subroutine, is saved to the address stack register at address 6H, and the operand for the CALL instruction, 30H, is transferred to the program counter.

When the CALL instruction at address 35H is executed, the stack pointer value is decremented by one to 5H. The return address from the second-level subroutine, 36H, is saved to the address stack register at address 5H (ASR5), and the operand for the CALL instruction, 50H, is transferred to the program counter. When the RET instruction is executed in the second-level subroutine, the address stack register contents (36H) at 5H (ASR5), which is specified by the stack pointer, are restored to the program counter. The stack pointer value is, accordingly, incremented by one to 6H. When the RET instruction for the first-level subroutine is subsequently executed, the return address for the main routine, 101H, is restored to the program counter, and the stack pointer value is incremented by one to 7H.

Figure 4-3. Stack Operation Examples When Subroutines Are Called



4.5.2 Table reference instruction (MOVT DBF, @AR)

Table 4-3 shows the operations to be performed, when the table reference instruction has been executed

Table 4-3. Operation When Table Reference Instruction Is Executed

Instruction	Cycle	Operation
MOVT DBF, @AR	First	<1> Increments value of program counter (PC) by 1 <2> Decrements value of stack pointer (SP) by 1 <3> Saves value of program counter (PC) to address stack register (ASR) specified by stack pointer (SP) <4> Transfers value of address register (AR) to program counter (PC)
	Second	<5> Transfers contents of program memory (ROM) specified by program counter (PC) to data buffer (DBF) <6> Restores value of address stack register (ASR) specified by stack pointer (SP) to program counter (PC) <7> Increments value of stack pointer (SP) by 1

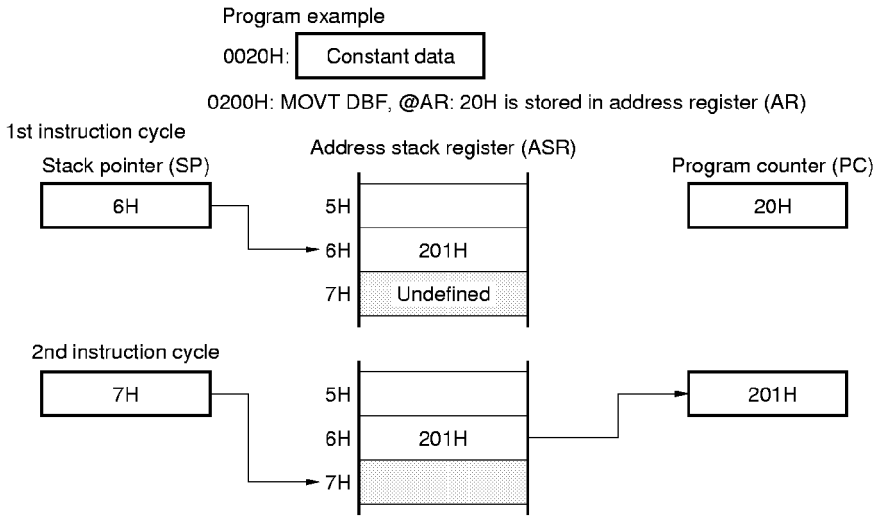
Figure 4-4 shows an operation example. Assume that, in this example, the table reference instruction is at address 200H, that the program memory address, in which the constant data to be referenced is stored, is 20H, and that the stack pointer value, immediately before the "MOVT DBF, @AR" instruction at address 200H is executed, is 7H.

When the "MOVT DBF, @AR" instruction at address 200H is executed, the stack pointer value is decremented by one to 6H during the first instruction cycle and address 201H, which is next to the address storing the "MOVT DBF, @AR" instruction, is saved to the address stack register at address 6H. The program memory address 20H, in which the constant data is stored, is transferred to the program counter.

This address, 20H, is specified by the address register.

During the second cycle in the instruction, the constant data at address 20H, which is the program counter contents, is transferred to the data buffer, and the contents for the address stack register, 201H, are restored to the program counter. The stack pointer value is incremented by one to 7H.

Figure 4-4. Stack Operation Example When Table Reference Instruction Is Executed



4.5.3 System call instruction (SYSCAL) and return instruction (RETI, RETSK)

Table 4-4 indicates the operations of the stack pointer (SP), address stack register (ASR), and program counter (PC), when the system call or return instruction has been executed.

Table 4-4. Operations When System Call Instruction Is Executed

Instruction	Operation
SYSCAL entry	<1> Increments value of program counter (PC) by 1. <2> Decrements value of stack pointer (SP) by 1 <3> Saves values of program counter (PC) and segment register (SGR) to address stack register (ASR) specified by stack pointer (SP) <4> Sets segment register (SGR) to 1 <5> Transfers value specified by operand (entry) for instruction to bits b_{10}-b_8 and b_3-b_0 for program counter (PC)
RET, RETSK	<1> Restores value of address stack register (ASR), specified by stack pointer (SP), to program counter (PC) and segment register (SGR) <2> Increments value of stack pointer (SP) by 1 <3> Only when RETSK is executed, processes first instruction after restoration as no-operation (NOP) instruction and proceeds to next instruction (skip operation)

The stack operations, when the system call instruction has been executed, are the same as those when the subroutine call instruction is executed, except that the segment register is set to 1, when the system call instruction has been executed.

Note, however, that the subroutine call instruction specifies a different program memory address to be called.

4.5.4 When interrupt is accepted or when return (RETI) instruction is executed

Table 4-5 shows the stack operations, when an interrupt has been accepted or when the return instruction has been executed.

Table 4-5. Operation of Stack When Interrupt Is Accepted and Return Instruction Is Executed

Instruction	Operation
When interrupt is accepted	<1> Increments value of program counter (PC) by 1 However, if branch (BR) or subroutine call (CALL) instruction is executed when interrupt is accepted, address of program memory (ROM) to which execution branches or from which subroutine is called is loaded to PC <2> Decrements value of stack pointer (SP) by 1 <3> Saves value of program counter (PC) and segement register (SGR) to address stack register specified by stack pointer (SP) <4> Saves BCD, CMP, CY, Z, and IXE flags of PSWORD and BANK to interrupt stack register <5> Transfers vector address to program counter (PC), and resets segment register (SGR)
RETI	<1> Restores value of interrupt stack register to BCD, CMP, CY, Z, and IXE flags of PSWORD and BANK <2> Restores value of address stack register specified by stack pointer (SP) to program counter (PC) and segment register (SGR) <3> Increments stack pointer (SP) by 1

Figure 4-5 shows an operation example. Assume that the stack pointer value is 7H and that an interrupt is accepted, while the instruction at address 300H is executed.

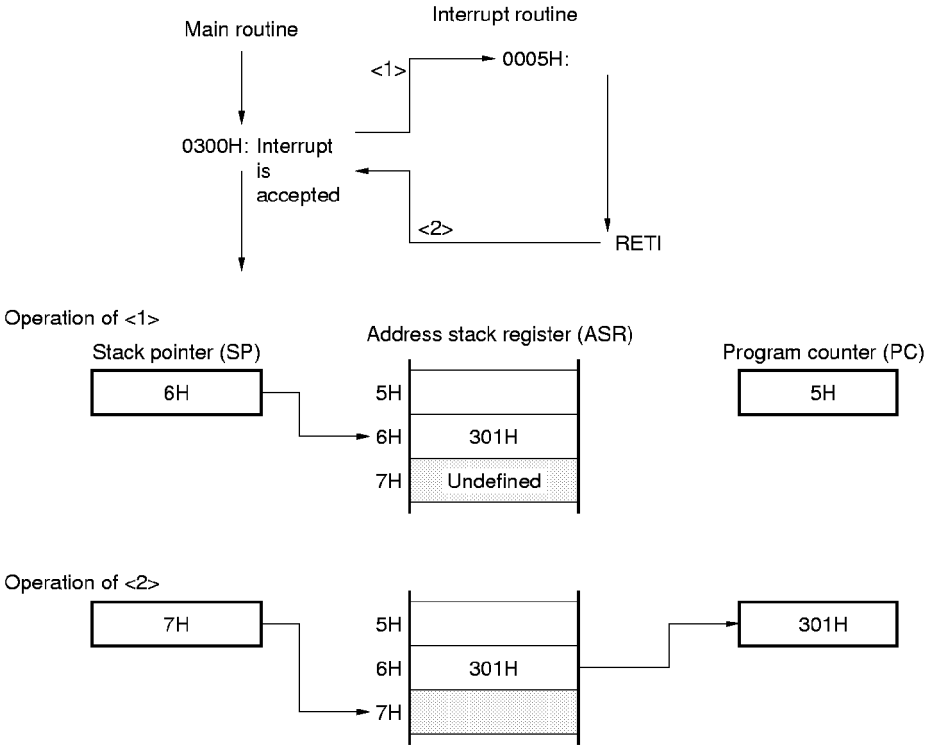
After the instruction at address 300H has been executed, the stack pointer value is decremented by one to 6H. The address stack register at address 6H (ASR6), address 301H, which would have been executed next, is stored, and the low-order 2 bits in the bank register and 1 bit in the index enable flag are saved to the interrupt stack register. The interrupt vector address for the INT_0 pin, 0005H, is transferred to the program counter, and the instruction at address 0005H is executed.

When the return (RETI) instruction is executed in the interrupt processing routine, the interrupt stack register contents are restored to the bank register and index enable flag. The address stack register contents, 301H, are restored to the program counter, and the stack pointer is incremented by 1 to 7H.

For details on interrupt operations, refer to **CHAPTER 12 INTERRUPT FUNCTIONS**.

★ Interrupt sources and the vector address differ depending on the model. Refer to the Data Sheet of each mode.

Figure 4-5. Stack Operation Example When Interrupt Occurs



4.6 ASR7 Nesting Level for Stack and PUSH AR and POP AR Instructions

The stack pointer operates as a 3-bit binary counter, whose contents are simply incremented or decremented by one, when the subroutine call or return instruction has been executed.

Therefore, while the stack pointer value is 0H, if the CALL, SYSCAL, or MOVT instruction is executed or if an interrupt is accepted the stack pointer value is decremented by one to 7H, and the return address and address register value from the subroutine or the interrupt processing routine are written to ASR7, which is the address 7H in the address stack registers. Since ASR7 does not exist in fact, the return address and the address register value cannot be written.

If the return instruction is executed, when the stack pointer value is 7H, therefore, the ASR7 contents at address stack register address 7H are transferred to the program counter and segment register.

To prevent this, the contents, read from ASR7 at address 7H, are "undefined" and the program flow cannot be restored normally.

In this case, save the address stack register value by using the PUSH or POP instruction.

Table 4-6 shows the operations for PUSH and POP instructions.

Table 4-6. Operations of PUSH and POP Instructions

Instruction	Operation
POP	<1> Transfers value of address stack register specified by stack pointer (SP) to address register (AR) <2> Increments value of stack pointer (SP) by 1
PUSH	<1> Decrements value of stack pointer (SP) by 1 <2> Transfers value of address register (AR) to address stack register specified by stack pointer (SP)

Figure 4-6 shows an operation example. In this example, a CALL instruction, that calls the seventh-level subroutine, which starts from address 30H, exists at address 10H of the sixth-level subroutine, and another CALL instruction that calls the eighth-level subroutine, which starts from address 50H, exists at address 35H.

The arrows in the figure indicate the program execution flow.

In this example, the value for the stack pointer, immediately before address 10H is executed, is 1H. When the CALL instruction at address 10H has been executed, the stack pointer value is decremented by one to 0H, and the return address from the seventh-level subroutine, 11H, is saved to the address stack register at address 0H. The operand for the CALL instruction, 30H, is transferred to the program counter.

When the POP instruction in the subroutine for the seventh-level is executed, the stack pointer value is incremented by one to 1H. Consequently, the address stack register contents at address 0H, 11H, are transferred to the address register.

When the CALL instruction at address 35H is executed, the stack pointer value is decremented by one to 0H, and the return address from the eighth-level subroutine, 41H, is saved to the address stack register at 0H. The operand for the CALL instruction, 50H, is transferred to the program counter.

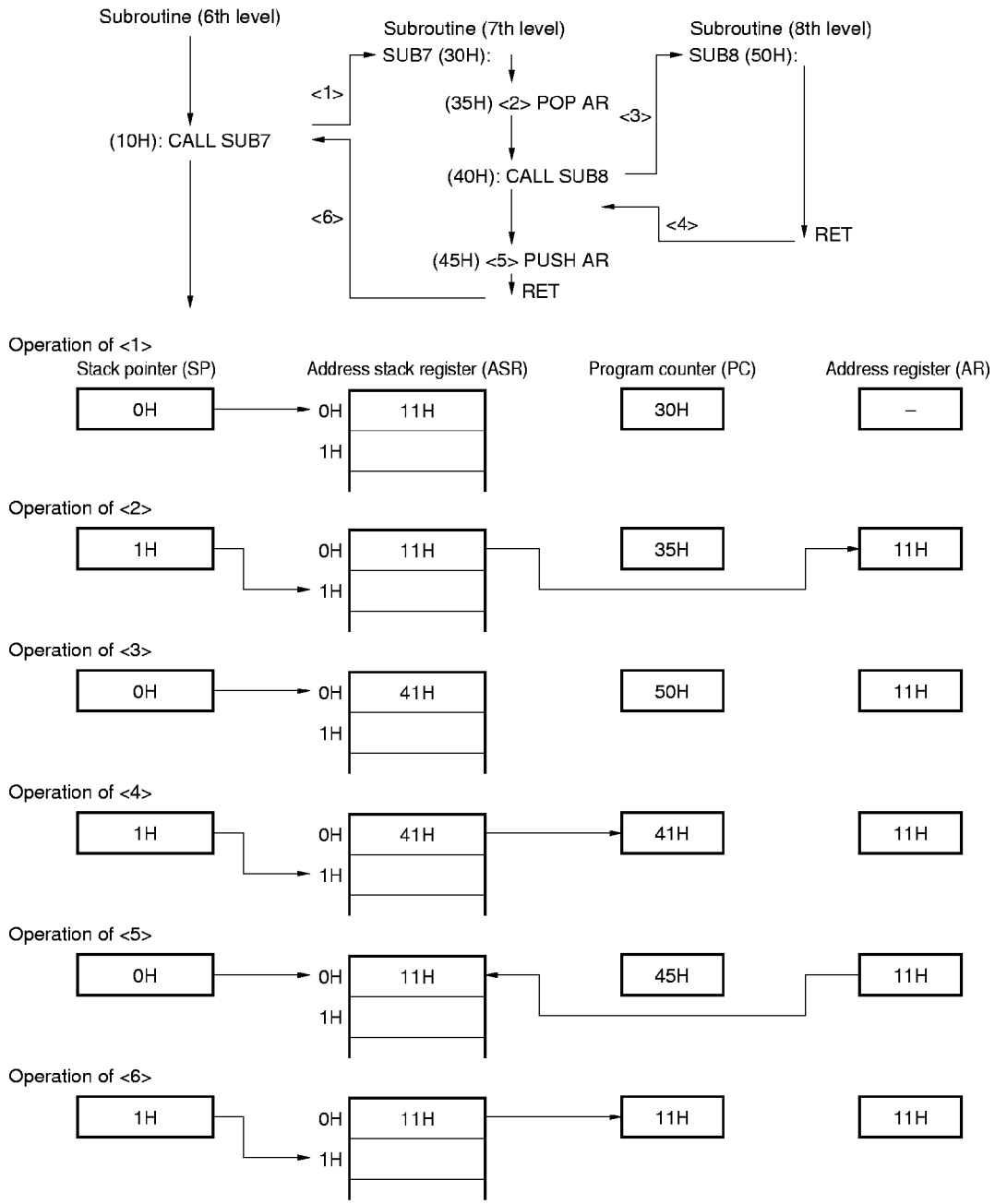
When the RET instruction in the eighth-level subroutine is executed, the address stack register contents 0H, 41H, are restored to the program counter, and the stack pointer is incremented by one to 1H.

When the PUSH instruction in the seventh-level subroutine is executed, the stack pointer is decremented by one to 0H. The contents of the address register, which are address 11, i.e., the return address to the sixth-level subroutine, are transferred to the address stack register at address 0H.

When the RET instruction in the seventh-level subroutine is executed, the address stack register contents at 0H, 11H, are restored to the program counter. The stack pointer is, accordingly, incremented by one to 1H.

In this way, the nesting levels for the stack can be set to 8 levels.

Figure 4-6. Stack Operation Example When PUSH and POP Instructions Are Executed



CHAPTER 5 DATA MEMORY (RAM)

The data memory is to store data for arithmetic and control operations. The data in the data memory can always be read or the data can be written to the memory by instructions.

5.1 Data Memory Configuration

As shown in Figure 5-1, the data memory is divided into up to 16 divisions in units of "banks". Each bank is assigned a number. Therefore, there are bank 0 to bank n ($n \leq 15$).

Each bank is assigned an address, which stores 4-bit data. The high-order 3 bits in an address are called a "row address", while the low-order 4 bits are called a "column address". For example, the data memory address, whose row address is 1H and whose column address is 0AH, is 1AH. One address consists of 4 bits, or one "nibble".

The data memory is also divided into the following five blocks, in terms of functions:

(1) System register (SYSREG)

★ The system register consists of 12 nibbles assigned to addresses 74H through 7FH in the data memory. The system register is assigned, regardless of the bank. That is, any bank has the same system register at addresses 74H through 7FH.

(2) Data buffer (DBF)

The data buffer consists of 4 nibbles at addresses 0CH through 0FH in bank 0 for the data memory.

(3) Port data register (port register)

This register is configured of a part of each bank in the data memory.

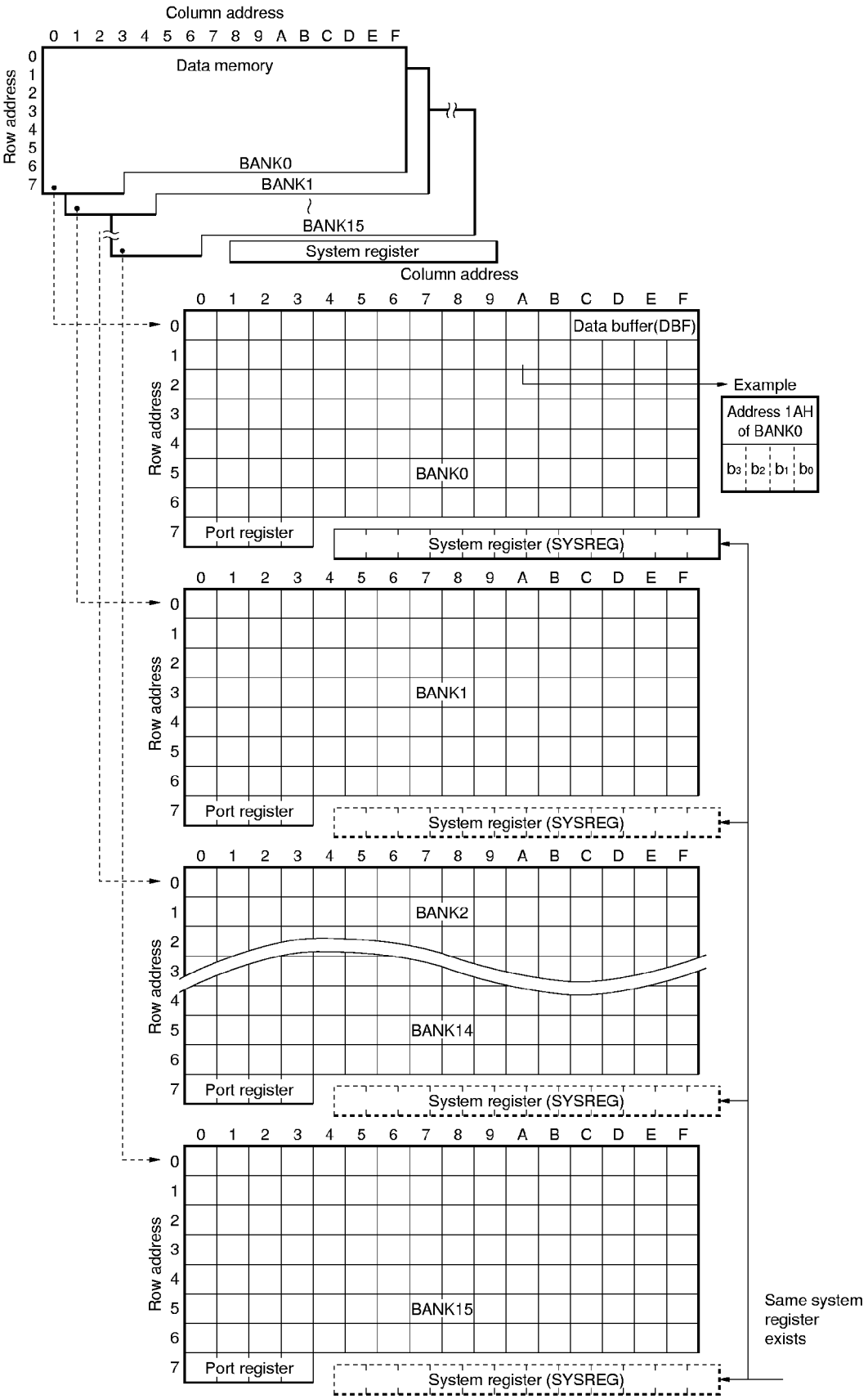
(4) General-purpose data memory

This is a portion of the data memory, other than the system register and port register, and consists of 112 nibbles.

(5) Unassigned data memory

The data memory area in the port register, to which no actual port is assigned, is fixed to 0.

Figure 5-1. Configuration of Data Memory



5.2 Notes on Specifying Data Memory Address

When using the 17K Series Assembler (RA17K), an error occurs if a data memory address is directly described as a numeral in the operand for a data memory manipulation instruction, as shown in Example 1 below.

This feature of the assembler is to reduce the causes of bugs, when the program is edited.

Examples 1. Error occurs

```
; <1>
   MOV    2FH, #0001B    ; Directly specifies address 2FH
; <2>
   MOV  0.2FH, #0001B    ; Directly specifies address 2FH of BANK0
```

Error does not occur

```
; <3>
   M02F MEM 0.2FH        ; Defines symbol in M02F with address 2FH of BANK0 as memory type
   MOV M02F, #0001B      ;
; <4>
   MOV .MD.2FH, #0001B ; Converts address 2FH into memory type by .MD.
                         ; This method, however, must be avoided to reduce causes of bugs.
```

It is therefore necessary to define data memory addresses as symbols in advance, using the assembler directive MEM (symbol definition directive).

To define a data memory address as a symbol, it is also necessary to define a bank, as shown in Example 2.

This is used by the data memory map creation function of the Assembler.

At this time, however, if the data in the following example 2 memory address, which is defined as a symbol in the BANK2 as shown, is used in the BANK1 range for the program, the BANK1 data memory address is manipulated.

2.

```
M1    MEM 0.15H    ;  ┐
M2    MEM 1.15H    ;  │ Symbol definition directive
M3    MEM 2.15H    ;  ┘
```

Bank Row address Column address

```
BANK1              ;   Assembler macroinstruction  BANK ← 1
MOV   M1, #0000B   ;  ┐ M1, M2, and M3 are defined as symbols by another bank in <1>,
MOV   M2, #0000B   ;  │ but are treated as BANK1 on program.  These three instructions
MOV   M3, #0000B   ;  ┘ write 0 to data memory at address 15H in BANK1
```

[MEMO]

The system register directly controls the CPU and is located on the data memory.

6.1 System Register Configuration

Figure 6-1 shows the system register location on the data memory. As shown in this figure, the system register is located at addresses 74H through 7FH in the data memory, independent from the bank. Therefore, the same system register exists at addresses 74H through 7FH for any memory bank.

Since the system register is on the data memory, it can be manipulated by any data memory manipulation instruction.

It is also possible to specify the system register as a general register.

Figure 6-2 shows the system register configuration.

As shown in this figure, the system register consists of the following seven kinds of registers:

Address register (AR)
Window register (WR)
Bank register (BANK)
Index register (IX)
Data memory row address pointer (MP)
General register pointer (RP)
Program status word (PSWORD)

Figure 6-1. System Register Location on Data Memory Location

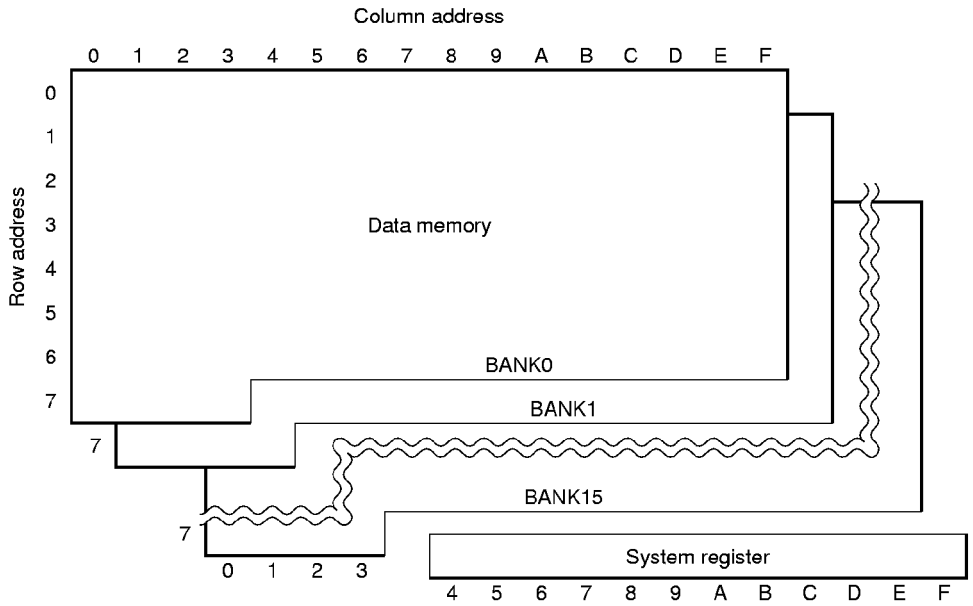


Figure 6-2. Configuration of System Register

Address	74H	75H	76H	77H	78H	79H	7AH	7BH	7CH	7DH	7EH	7FH
Name	System register											
	Address register (AR)				Window register (WR)	Bank register (BANK)	Index register (IX)			Genral register pointer (RP)		Progrem status word (PSWORD)
							Data memory row address pointer (MP)					
Symbol	AR3	AR2	AR1	AR0	WR	BANK	IXH	IXM	IXL	RPH	RPL	PSW
							MPH	MPL				
Bit	b3 b2 b1 b0	b3 b2 b1 b0	b3 b2 b1 b0	b3 b2 b1 b0	b3 b2 b1 b0	b3 b2 b1 b0	b3 b2 b1 b0	b3 b2 b1 b0	b3 b2 b1 b0	b3 b2 b1 b0	b3 b2 b1 b0	b3 b2 b1 b0
Data							MPE (IX)					BCD CMP CY Z IXE
							(MP)					

6.2 System Register Functions

6.2.1 Each register functions

The functions for each constituent register in the system register are as follows. The functions for each register are described in more detail in **6.4** through **6.9**.

(1) Address register (AR)
Indirectly specifies an address in the program memory.

(2) Window register (WR)
Transfers data with the register file.

(3) Bank register (BANK)
Specifies a bank in the data memory.

(4) Index register (IX)
Qualifies an address in the data memory.

(5) Data memory row address pointer (MP)
Specifies a row address during general register indirect transfer.

(6) General register pointer (RP)
Specifies a bank and row address in the general register.

(7) Program status word (PSWORD)
Sets the conditions of arithmetic operation and transfer instructions.

6.2.2 System register manipulation instruction

Since the system register is located on the data memory, it can be controlled by all data memory manipulation instructions. In addition, the address register and index register can be manipulated by the following dedicated instructions:

INC AR: Increments the address register (AR) contents by one. The address register has 14 valid bits. When the address register contents are incremented, when the current contents are 1FFFH, the register contents become 0000H. The address register contents cannot be incremented to a total exceeding 8K steps.

INC IX : Increments the index register by (IX) contents by one. The index register has 11 valid bits. When the index register contents are incremented, when the current contents are incremented, when the current contents are 7FFH, the register contents become 000H.

6.3 Address Register (AR)

6.3.1 Address register configuration

Figure 6-3 shows the address register configuration.

As shown in this figure, the address register consists of 16 bits in the system register: 74H through 77H (AR3 through AR0).

★ **Figure 6-3. Configuration of Address Register**

Address	74H				75H				76H				77H			
Name	Address register (AR)															
Symbol	AR3				AR2				AR1				AR0			
Bit	b_3	b_2	b_1	b_0	b_3	b_2	b_1	b_0	b_3	b_2	b_1	b_0	b_3	b_2	b_1	b_0
Data	MSB															LSB

6.3.2 Address register functions

The address register specifies a program memory address, when the indirect branch instruction (BR @AR), indirect subroutine call instruction (CALL @AR), table reference instruction (MOVT DBF, @AR), or stack manipulation instruction (PUSH AR, POP AR) has been executed.

6.3.3 through **6.3.4** describe the address register operations, when each of these instructions has been executed.

A sole use instruction (INC AR), that can increment the contents of the address register by one, is available. When this instruction is used, the address register data can be incremented in 13-bit units. When the "INC AR" instruction is executed, while the address register contents are 1FFFH, the address register is incremented to **0000H**.

Note that an address exceeding the program memory range must not be set.

6.3.3 Table reference instruction (MOVT DBF, @AR)

When the "MOVT DBF, @AR" instruction is executed, the constant data (16 bits) in a program memory address, specified by the address register contents, are read to the data buffer (DBF: addresses 0CH through 0FH in BANK0) on the data memory.

The program memory addresses, from which constant data can be read to the data buffer, can be specified in the address register range for each model.

For details, also refer to **10.4 Data Buffer and Table Reference**.

Example

```
Address   Label
0000H     DATA1  : 16-bit constant data
                   MOV   AR0, #0FH   ; Writes 0FH to AR0
                   MOV   AR1, #0H    ; Writes 0H to AR1
                   MOV   AR2, #0H    ; Writes 0H to AR2
                   MOV   AR3, #0H    ; Writes 0H to AR3
                   MOVT  DBF, @AR    ; Roads constant data at program memory address
                                     ; 000FH to data buffer
```

6.3.4 Stack manipulation instruction (PUSH AR, POP AR)

By executing the "PUSH AR" instruction, the stack pointer is decremented by one and the address register (AR) contents are stored to the address stack register specified by the stack pointer.

When the "POP AR" instruction is executed, the address stack register contents, specified by the stack pointer, are transferred to the stack register, and the address stack register is incremented by one.

For details, refer to **CHAPTER 4 ADDRESS STACK**.

6.3.5 Indirect branch instruction (BR @AR)

When the "BR @AR" instruction is executed, the program execution branches to a program memory address specified by the address register contents.

Example

```
MOV   AR0, #0FH     ; Writes 0FH to AR0
MOV   AR1, #0H      ; Writes 0H to AR1
MOV   AR2, #0H      ; Writes 0H to AR2
MOV   AR3, #0H      ; Writes 0H to AR3
BR    @AR           ; Program branches to 000FH
```

6.3.6 Indirect subroutine call instruction (CALL @AR)

When the "CALL @AR" instruction is executed, the subroutine at the program memory, specified by the address register contents, can be called.

Examples 1.

```
Address   Label
000FH     SUB:
                  Subroutine processing
                  RET
          MOV  AR0, #0FH     ; Writes 0FH to AR0
          MOV  AR1, #0H      ; Writes 0H to AR1
          MOV  AR2, #0H      ; Writes 0H to AR2
          MOV  AR3, #0H      ; Writes 0H to AR3
          CALL @AR           ; Calls subroutine at address 000FH
```

In this example, the address from which a subroutine is indirectly called is specified by the "MOV" instruction.

By this method, however, the program memory efficiency is degraded, if the subroutine is frequently called.

Therefore, it is recommended to use the "POP", "PUSH", and table reference instructions, as shown in Example 2 below.

2.

```
SUBENTRY:
          DI
          POP   AR
          MOVT  DBF, @AR
          INC   AR
          PUSH  AR
          EI
          PUT   AR, DBF
          BR    @AR
SUB1      :
SUB2      :
MAIN      :
          CALL  SUBENTRY
          DW    .DL.SUB1
          CALL  SUBENTRY
          DW    .DL.SUB2
```

6.3.7 Address register and data buffer

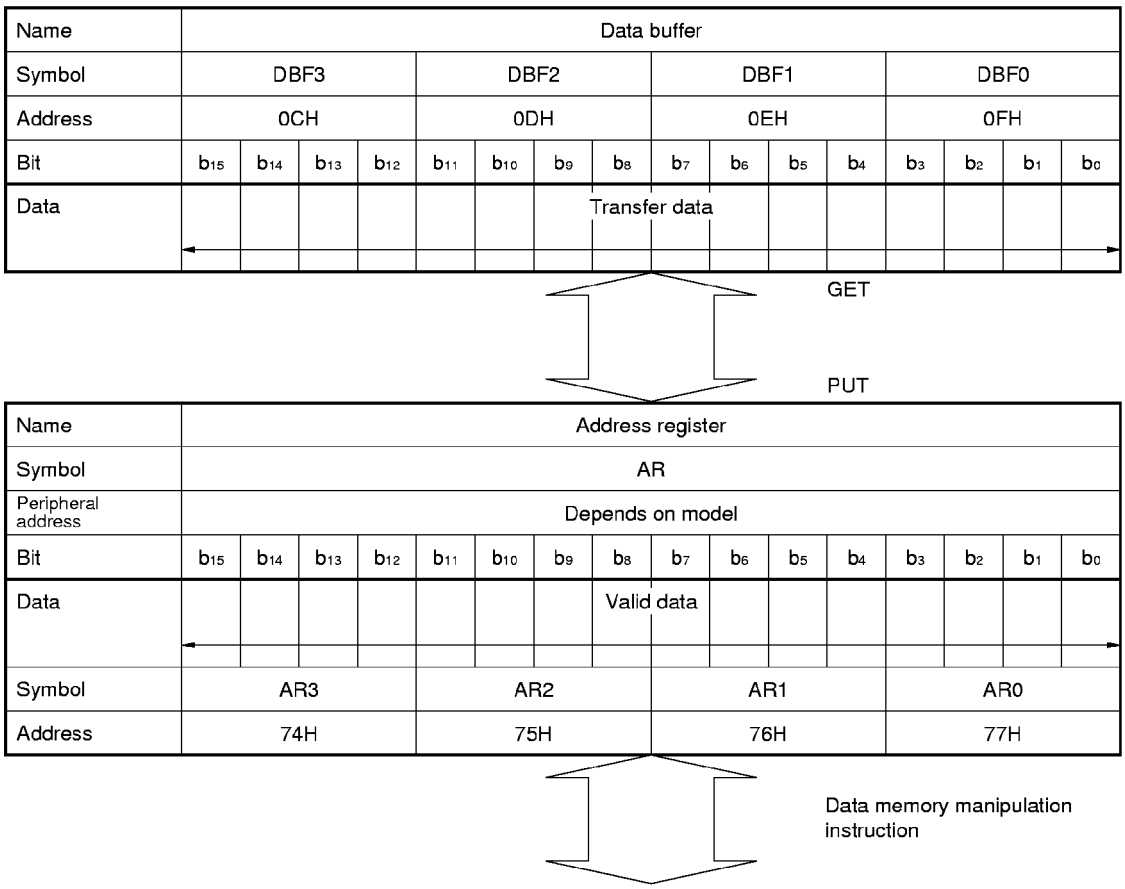
The address register can be directly manipulated by the data memory manipulation instruction. It can also transfer data through the data buffer as a part of the hardware peripherals.

Data can be read from or written to the address register through the data buffer by using the "PUT" and "GET" instructions, in addition to the data memory manipulation instruction.

Figure 6-4 shows the relations between the address register and data buffer.

For details on the data buffer, refer to **CHAPTER 10 DATA BUFFER (DBF)**.

Figure 6-4. Data Transfer between Address Register and Data Buffer



When data is directly read from or written to address register

6.4 Window Register (WR)

6.4.1 Window register configuration

Figure 6-5 shows the window register configuration. As shown in this figure, the window register consists of 4 bits in 78H in the system register.

Figure 6-5. Configuration of Window Register

Address	78H			
Name	Window register (WR)			
Symbol	WR			
Bit	b_3	b_2	b_1	b_0
Data	MSB			LSB

6.4.2 Window register functions

The window register is used to transfer data with the register file (RF).

To transfer data between the window register and register file, sole use instructions, "PEEK WR, rf" and "POKE rf, WR", are used.

6.4.3 and **6.4.4** describe the window register operation, when each of these instruction is executed.

For details, refer to **CHAPTER 9 REGISTER FILE (RF)**.

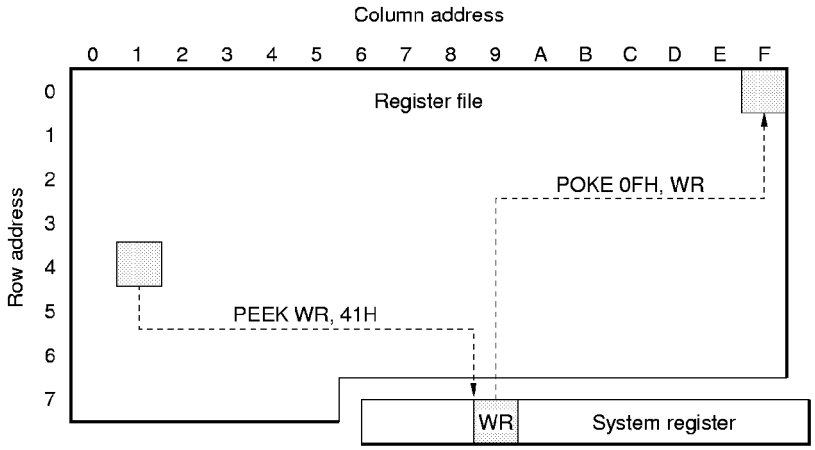
6.4.3 PEEK WR, rf instruction

As shown in Figure 6-6, the register file contents, addressed by rf, are transferred to the window register, when the PEEK WR, rf instruction (rf: address of register file) is executed.

6.4.4 POKE rf, WR instruction

As shown in Figure 6-6, the window register contents are transferred to the register file addressed by rf, when the POKE rf, WR instruction (rf: register file address) is executed.

Figure 6-6. Operations for PEEK and POKE Instructions



6.5 Bank Register (BANK)

6.5.1 Bank register configuration

Figure 6-7 shows the bank register configuration.

As shown in this figure, the bank register consists of 4 bits in 79H (BANK) for the system register.

Figure 6-7. Configuration of Bank Register

Address	79H			
Name	Bank register (BANK)			
Symbol	BANK			
Bit	b_3	b_2	b_1	b_0
Data	MSB			LSB

6.5.2 Bank register function

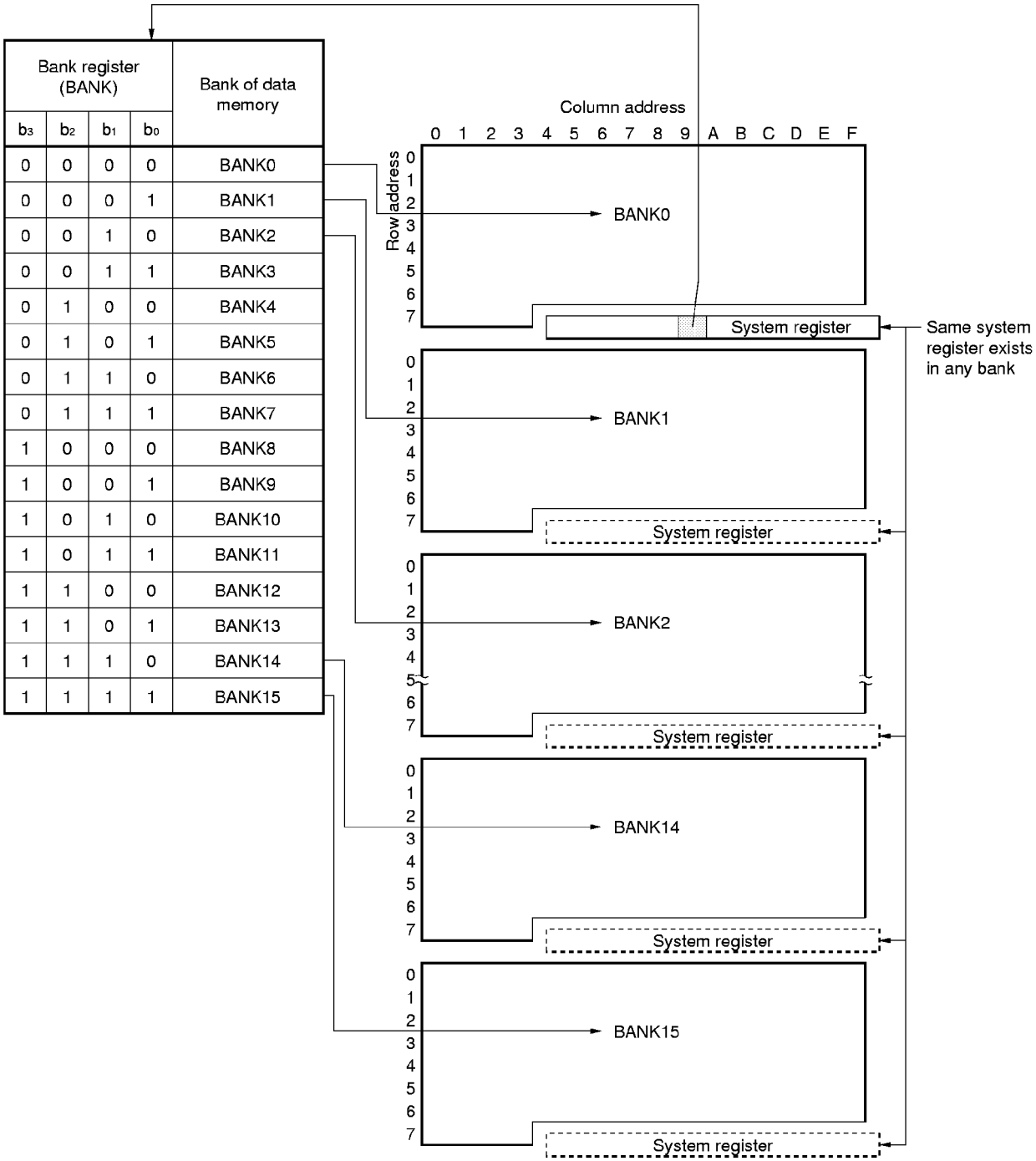
The bank register selects a bank in the data memory.

As shown in Figure 6-8, the data memory is divided into up to 16 banks, and the data memory area in the bank, specified by the bank register, is manipulated by a data memory manipulation instruction.

Therefore, to manipulate the data memory area in BANK1, when BANK0 is currently selected, it is necessary to write 0001H to the bank register, in order to select BANK1.

At this time, the bank concept does not apply to the system register located at addresses 74H through 7FH in the data memory, and the system register in any bank serves as is. Consequently, 0 is written to the bank register (BANK: address 78H), regardless of whether the "MOV BANK, #0" instruction is executed in BANK1 or BANK2. To manipulate the bank register, therefore, the bank specified at that time is independent.

Figure 6-8. Specifying Data Memory Bank



★ As an instruction that specifies a bank, the 17K Series Assembler (RA17K) offers a macroinstruction "BANKn" (0 ≤ n ≤ 4).

Here is an example showing how to manipulate the bank and data memory:

Example

```
M000    MEM 0.00H       ; Defines symbol
M100    MEM 1.00H       ;
BANK0                   ; Same as MOV BANK, #0000B
MOV     M000, #0101B    ; Writes 0101B to address 00H in BANK0
BANK1                   ; Same as MOV BANK, #0001B
MOV     M100, #0101B    ; Writes 0101B to address 00H in BANK1
MOV     M000, #0101B    ; Writes 0101B to address 00H in BANK1
                        ; This means that data memory M000 is defined in BANK0 by
                        ; means in symbol definition, but the bank selected at that time is
                        ; assumed when program is executed.
```

6.6 Index Register (IX) and Data Memory Row Address Pointer (MP: Memory Pointer)

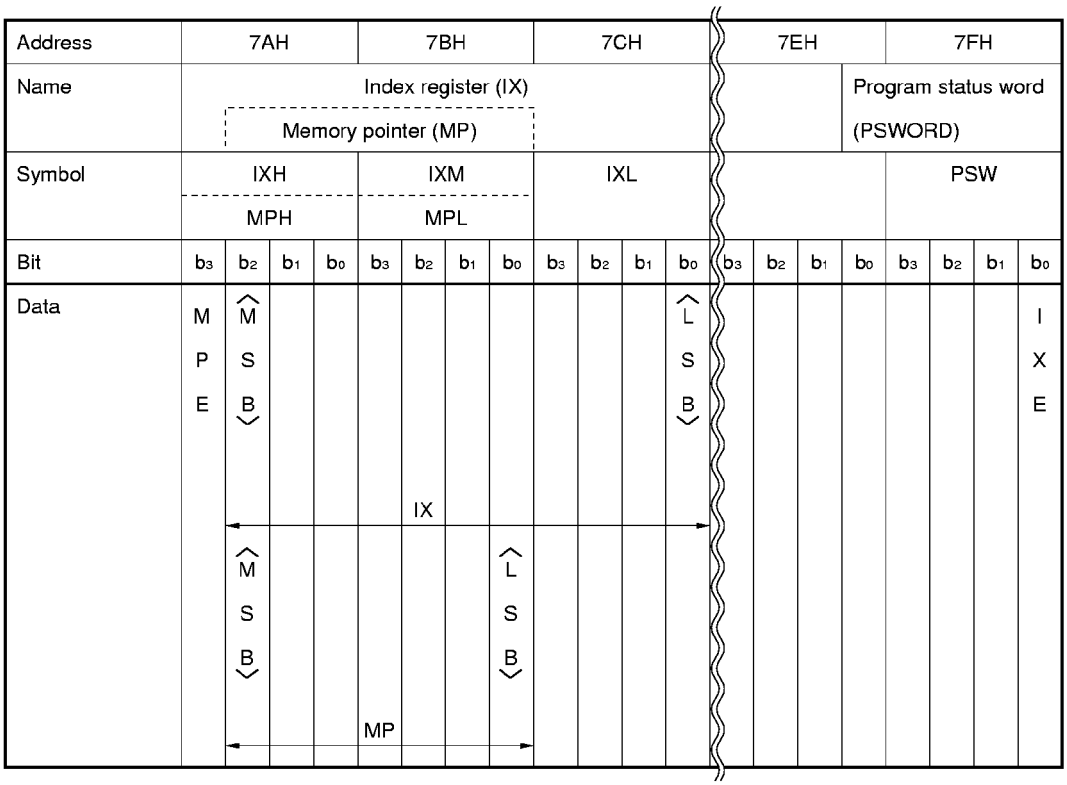
6.6.1 Configurations for index register and data memory row address pointer

Figure 6-9 shows the configurations for the index register and data memory row address pointer. As shown in this figure, the index register consists of a total of 12 bits, of which the low-order 3 bits (IXH) are for 7AH in the system register, and the others are for 7BH and 7CH (IXM, IXL), and an index enable (IXE) flag, which is at the least significant bit for 7FH (PSW).

The data memory row address pointer (memory pointer) consists of a total of 7 bits, of which the low-order 3 bits are for 7AH (MPH) and the others are for 7BH (MPL), plus a data memory row address pointer enable flag (MPE) at the most significant bit for 7AH (MPH).

This means that the high-order 7 bits in the index register are shared by the data memory row address pointer.

Figure 6-9. Configurations for Index Register and Data Memory Row Address Pointer



6.6.2 Index register and data memory row address pointer functions

The following paragraphs (1) and (2) describe the functions of the index register and data memory row address pointer:

(1) Index register

When a data memory manipulation instruction is executed, the index register modifies with its contents the bank and address of the data memory specified by the instruction.

However, the address modification by the index register is valid only when the index enable flag (IXE) is set.

To modify an address, the bank and address of the data memory are ORed with the contents of the index register, and the instruction is executed to the data memory at the address (called real address) specified by the result of the OR operation.

The index register modifies an address with all the data memory manipulation instructions.

The instructions that cannot be used for address modification are as follows:

MOVT	DBF, @AR	BR	addr	INC	AR	EI
PEEK	WR, rf	BR	@AR	INC	IX	DI
POKE	rf, WR	CALL	addr	RORC	r	
GET	DBF, p	CALL	@AR	STOP	s	
PUT	p, DBF	RET		HALT	h	
PUSH	AR	RETSK		NOP		
POP	AR	RETI				

(2) Data memory row address pointer

The data memory row address pointer modifies with its contents the address at the indirect transfer destination when a general register indirect transfer instruction (MOV @r, m or MOV m, @r) is executed.

However, address modification by the data memory row address pointer is valid only when the data memory row address pointer enable flag (memory pointer enable flag: MPE) is set to 1.

In addition, the address specified by an instruction other than the general register indirect transfer instruction is not modified.

To modify an address, the bank and row address at the indirect transfer destination are replaced with the contents of the data memory row address pointer.

Figure 6-1 illustrates data memory address modification and indirect transfer address modification by the index register and data memory row address pointer.

6.6.3 through **6.6.6** describe the operations to modify a data memory address by the index register and data memory row address pointer.

Table 6-1. Data Memory Address Modification by Index Register and Data Memory Row Address Pointer

IXE	MPE	General Register Address Specified by r			Data Memory Address Specified by m			Indirect Transfer Address Specified by @r		
		Bank	Row Address	Column Address	Bank	Row Address	Column Address	Bank	Row Address	Column Address
		b_3 b_2 b_1 b_0	b_2 b_1 b_0	b_3 b_2 b_1 b_0	b_3 b_2 b_1 b_0	b_2 b_1 b_0	b_3 b_2 b_1 b_0	b_3 b_2 b_1 b_0	b_2 b_1 b_0	b_3 b_2 b_1 b_0
0	0	RP		r	BANK		m	BANK	m_R	(r)
0	1		ditto			ditto		MP		(r)
1	0		ditto		BANK	Logical OR IX	m	BANK	m_R Logical OR IXH IXM	(r)
1	1		ditto			ditto		MP		(r)

		Instructions modified		
Add/Sub	ADD ADDC SUB SUBC	r	m	
			m, #n4	
Logical	AND OR XOR	r	m	
			m, #n4	
Compare	SKE SKGE SKLT SKNE		m, #n4	
Judge-ment	SKT SKF		m, #n	
Transfer	LD ST	r	m	
	MOV		m, #n4	
		@r	m	Indirect transfer address

BANK : Bank register
IX : Index register
IXE : Index enable flag
IXH : Bits 10-8 of index register
IXM : Bits 7-4 of index register
IXL : Bits 3-0 of index register
m : Data memory address specified by m_R, m_C
m_R : Data memory row address (high)
MP : Data memory row address pointer
MPE : Memory pointer enable flag
r : General register column address
RP : General register pointer
(×) : Contents addressed by ×
× : Direct address such as m, r
: Register such as BANK

★ **Remark** The settings of IXE and MPE differ depending on the model. Refer to the Data Sheet for each model.

6.6.3 When MPE = 0, IXE = 0 (no data memory modification)

As indicated in Table 6-1, the data memory address is not affected by the index register and data memory row address pointer.

(1) Data memory manipulation instruction

Examples 1. If general register is at row address 0

```
R003    MEM 0.03H
M061    MEM 0.61H
ADD     R003, M061
```

When the above instructions are executed, the contents of general register R003 and those of data memory M061 are added, and the result is stored to general register R003, as shown in Figure 6-10.

(2) General register indirect transfer

Examples 2. If general register is at row address 0

```
R005     MEM 0.05H
M034     MEM 0.34H
MOV      R005, #8          ; R005 ← 8
MOV      @R005, M034       ; Register indirect transfer
```

When the above instructions are executed, the contents of data memory M034 are transferred to address 38H of the data memory, as shown in Figure 6-10.
Therefore, the "MOV @r, m" instruction transfers the contents of the data memory specified by m to the data memory at an indirect address specified by @r of the same row address as m.
The indirect transfer address is the contents of the general register with a row address same as m (row address 3 in the above example) and a column address specified by r (8 in the above example). Therefore, it is 38H in the above example.

Examples 3. If general register is at row address 0

```
R00B    MEM 0.0BH
M034    MEM 0.34H
MOV     R00B, #0EH      ; R00B ← 0EH
MOV     M034, @R00B     ; Register indirect transfer
```

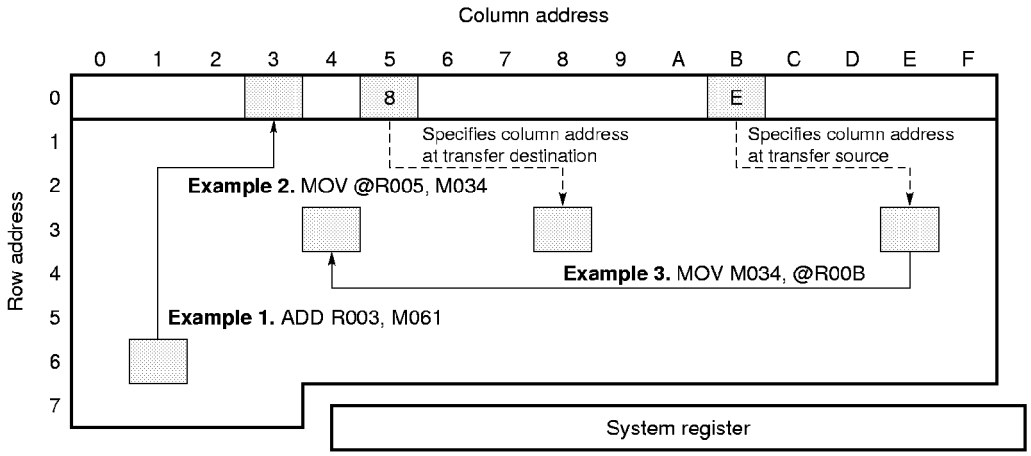
When the above instructions are executed, the contents of the data memory at address 3EH are transferred to data memory M034 as shown in Figure 6-10.

Therefore, the "MOV m, @r" instruction transfers the contents of the data memory at an indirect address specified by @r of a row address same as m to the data memory addressed by m.

The indirect transfer address is the contents of the general register with a row address same as m (row address 3 in the above example) and a column address specified by r (0EH in the above example). Therefore, it is 3EH in the above example.

Comparing this with Example 2, the source address of the data memory whose contents are to be transferred and the destination address are exchanged.

Figure 6-10. Example of Operation When MPE = 0, IXE = 0



Generation of address in Example 1

ADD R003, M061

	Bank	Row Address	Column Address
Data memory address M	0000	110	0001
General register address R	0000	000	0011

Generation of address in Example 2

MOV @R005, M034

	Bank	Row Address	Column Address
Data memory address M	0000	011	0100
General register address R	0000	000	0101
Indirect transfer address @R	0000	011	1000
	Same as M		Content of R

6.6.4 When MPE = 1, IXE = 0 (diagonal indirect transfer)

As shown in Table 6-1, the bank and row address of the indirect transfer address specified by @r are the value of the data memory row address pointer only when a general register indirect transfer instruction (MOV @r, m or MOV m, @r) is executed.

Examples 1. If general register is at row address 0

```
R005    MEM 0.05H
M034    MEM 0.34H
MOV     MPL, #0110B     ; MP ← 6
MOV     MPH, #1000B     ; MPE ← 1
MOV     R005, #8        ; R005 ← 8
MOV     @R005, M034     ; Register indirect transfer
```

When the above instructions are executed, the contents of data memory M034 are transferred to data memory address 68H as shown in Figure 6-11.

When the "MOV @r,m" instruction is executed when MPE = 1, the contents of the data memory specified by m are transferred to the column address specified by @r having a row address specified by the memory pointer.

At this time, the indirect address specified by @r is the contents of the general register with a bank and row address being the value of the data memory row address pointer (row address 6 in the above example) and a column address specified by r.

It is, therefore, 68H in the above example.

When this is compared with **Example 2** in **6.6.3**, the bank and row address of the indirect address specified by @r are specified by the data memory row address pointer in the above example, while the bank and row address of the indirect address in **Example 2** in **6.6.3** are the same as m.

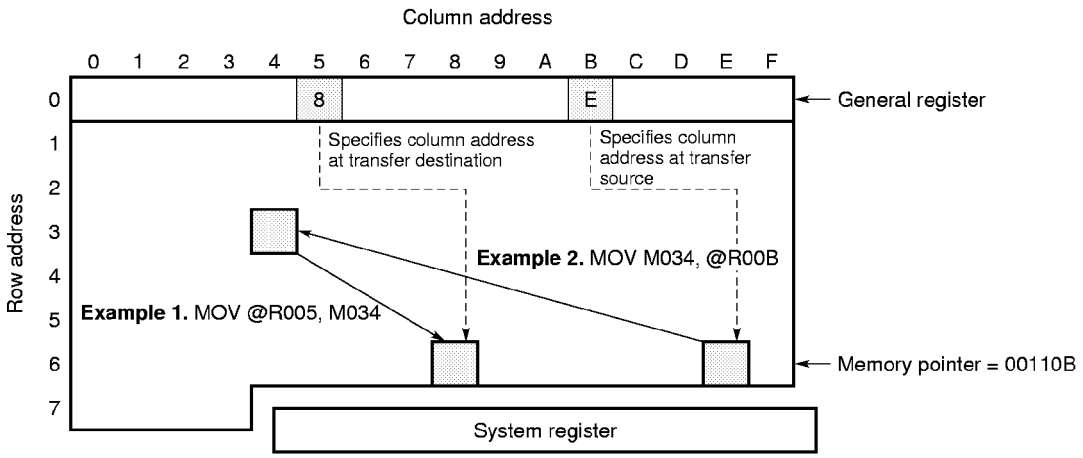
Therefore, general register indirect transfer can be diagonally carried out by setting MPE to 1.

Examples 2. If general register is at row address 0

```
R00B    MEM 0.0BH
M034    MEM 0.34H
MOV     MPL, #0110B     ; MP ← 6
MOV     MPH, #1000B     ; MPE ← 1
MOV     R00B, #0EH      ; R00B ← 0EH
MOV     M034, @R00B     ; Register indirect transfer
```

When the above instructions are executed, the contents of the data memory at address 6EH are transferred to data memory M034, as shown in Figure 6-11.

Figure 6-11. Example of Operation When MPE = 1, IXE = 0



Generation of address in Example 1

MOV @R005, M034

	Bank	Row Address	Column Address
Data memory address M	0000	011	0100
General register address R	0000	000	0101
Indirect transfer address @R	0000	110	1000
	Content of MP		Content of R

Generation of address in Example 2

MOV M034, @R00B

	Bank	Row Address	Column Address
Data memory address M	0000	011	0100
General register address R	0000	000	0111
Indirect transfer address @R	0000	110	1110
	Content of MP		Content of R

6.6.5 When MPE = 0, IXE = 1 (data memory address index modification)

When a data memory manipulation instruction is executed as indicated in Table 6-1, all the banks and addresses of the data memory directly specified by the operand "m" of the instruction are modified by the index register.

When a general register indirect transfer instruction (MOV @r, m or MOV m, @r) is executed, the bank and row address of the indirect transfer address specified by @r are also modified by the index register.

To modify an address, the contents of the data memory address and those of the index register are ORed, and the instruction is executed to the data memory address (called an real address) specified by the result of the OR operation.

Examples 1. If general register is at row address 0

```
R003    MEM 0.03H
M061    MEM 0.61H
MOV     IXL, #0010B              ; IX ← 00000010010B
MOV     IXM, #0001B              ;
MOV     IXH, #0000B              ; MPE ← 0
OR      PSW, #.DF.IXE AND 0FH    ; IXE ← 1
ADD     R003, M061
```

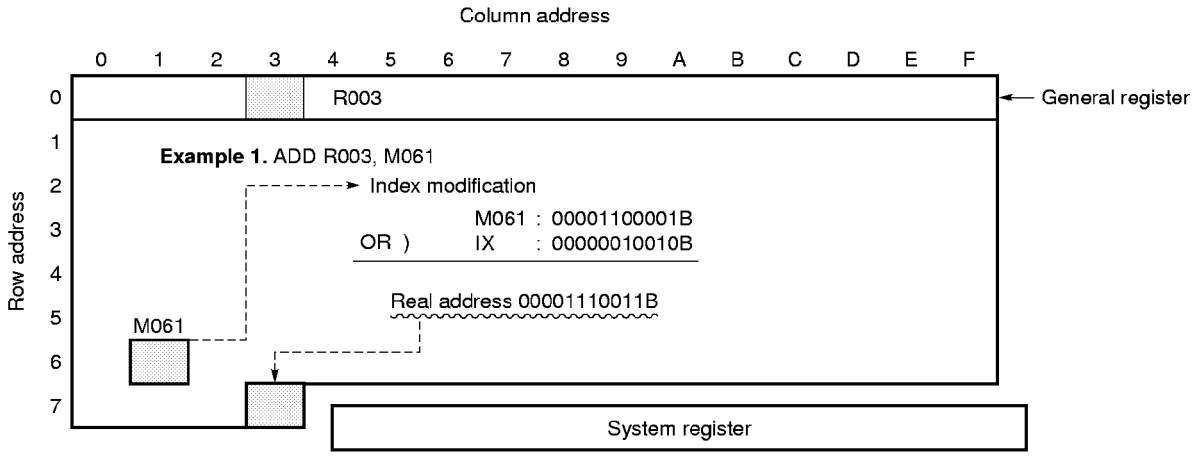
When the instructions in this example are executed, the contents of the data memory at address 73H (real address) and the contents of general register R003 (address 03H) are added, and the result is stored to general register R003 as shown in Figure 6-12.

Therefore, when the "ADD r, m" instruction is executed, the data memory address specified by "m" (address 61H in the above example) is modified by the index register.

To modify the address, address 61H, which is the address of data memory M061 (00001100001B in binary), is ORed with the value of the index register (00000010010B in the above example), and the result 00001110011B is treated as the real address (address 73H), and the instruction is executed to this real address.

Comparing this with **Example** in **6.6.3** (when IXE = 0), the address of the data memory directly specified by the operand "m" of the instruction is modified (ORed) by the index register.

Figure 6-12. Example of Operation When MPE = 0, IXE = 1



Generation of address in Example 1

ADD R003, M061

		Bank	Row Address	Column Address
Data memory address M		0000	110	0001
General register address R		0000	000	0011
Index modification	M061	0000	110	0001
		BANK	m	
	IX	0000	001	0010
		IXH	IXM	IXL
	Real addr. (ORed)	0000	111	0011

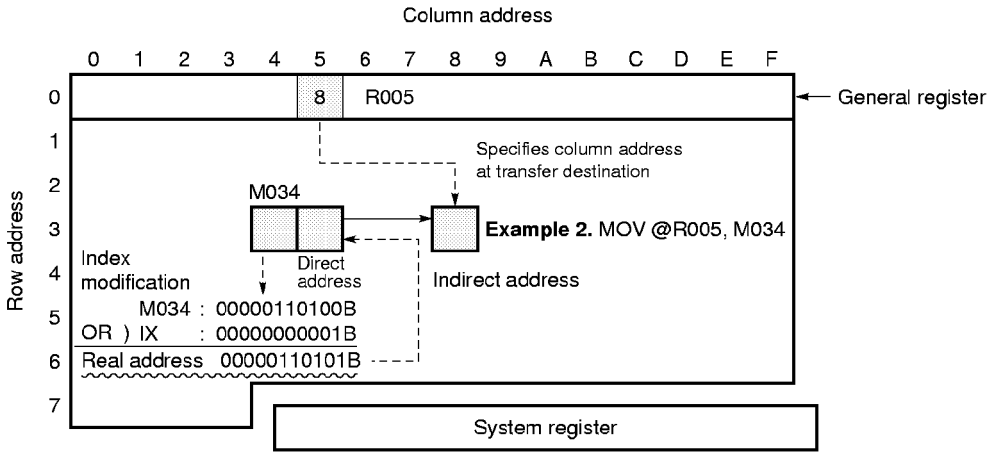
Instruction is executed to this address.

Examples 2. General register indirect transfer

```
If general register is in BANK0 at row address 0
R005    MEM 0.05H
M034    MEM 0.34H
MOV     IXL, #0001B              ; IX ← 00000000001B
MOV     IXM, #0000B              ;
MOV     IXH, #0000B              ; MPE ← 0
OR      PSW, #.DF.IXE AND 0FH    ; IXE ← 1
MOV     R005, #8                 ; R005 ← 8
MOV     @R005, M034              ; Register indirect transfer
```

When the above instructions are executed, the contents of the data memory at address 35H are transferred to the address 38H of the data memory as shown in Figure 6-13.
Therefore, if the "MOV @r, m" instruction is executed when IXE = 1, the data memory address (direct address) specified by "m" is modified with the contents of the index register, and the bank and row address of the indirect address specified by "@r" are also modified by the index register. All the bank, row, and column address of the address specified by "m" are modified, and the bank and row address of the indirect address specified by "@r" are modified.
In the above example, therefore, the direct address is 35H and the indirect address is 38H. When this is compared with **Example 3** in **6.6.3** when IXE = 0, the bank, row, and column address of the direct address specified by "m" are modified by the index register and general register indirect transfer is executed to the row address same as the modified data memory address in the above example, while the direct address is not modified in **Example 3** in **6.6.3**.

Figure 6-13. Example of General Register Indirect Transfer Operation When MPE = 0, IXE = 1



Examples 3. To clear contents of all data memory to 0

```
M000        MEM 0.00H
            MOV   IXL, #0                ; IX ← 0
            MOV   IXM, #0                ;
            MOV   IXH, #0                ; MPE ← 0
LOOP:
            OR    PSW, #.DF.IXE AND 0FH  ; IXE ← 1
            MOV   M000, #0               ; Clears data memory specified by IX to 0
            INC   IX                     ; IX ← IX + 1
            AND   PSW, #1110B            ; IXE ← 0: Since IXE is
                                         ; at address 7FH, it is not modified by IX
            SKE   IXM, #0111B            ; Row address 7?
            BR    LOOP                   ; LOOP if not 7 (row address is not cleared)
```

Examples 4. Processing of array

Suppose 8-bit data A is defined one-dimensionally as shown in Figure 6-14. To execute the following operation, the instructions below should be executed:

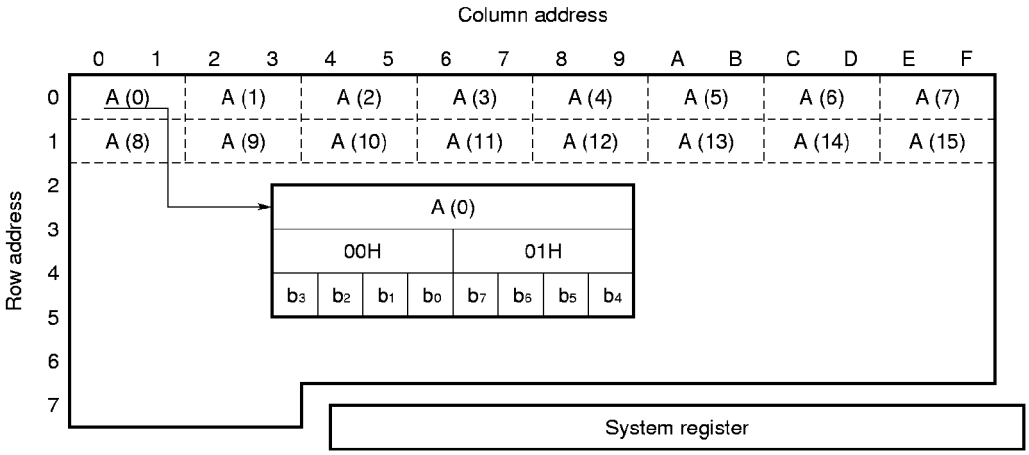
A (N) = A (N) + 4 (0 ≤ N ≤ 15)

Where general register is at row address 7

```
M000    MEM 0.00H
M001    MEM 0.01H
MOV     IXH, #0                 ; IX ← 2N
MOV     IXM, #N SHR 3           ; Since array element is 8 bits, data memory address to
MOV     IXL, #N SHL 1 AND 0FH   ; be modified is shifted
OR      PSW, #.DF.IXE AND 0FH   ; IXE ← 1
ADD     M000, #4                ; Adds 4 to data memory M000
ADDC    M001, #0                ; and M001 that are modified by IX, i.e., adds 4 to 8-bit
                                ; array specified by A(N)
```

To specify N of array A(N) as indicated in the above example, specify a value 2 times that of N to the index register.

Figure 6-14. Example of Operation When MPE = 0, IXE = 1 (array processing)



6.6.6 When MPE = 1, IXE = 1

All the addresses for the data memory, directly specified by operand "m" when a data memory manipulation instruction is executed, are qualified by the index register, as shown in Table 6-1.

★ When a general register indirect transfer instruction (MOV @r, m or MOV m, @r) has been executed, the direct address specified by "m" is qualified by the index register, and the indirect address, specified by "@r", is specified by the data memory row address pointer.

Example When the row address for general register in BANK0 is 0

```
R005    MEM 0.05H
R034    MEM 0.34H
MOV     IXL, #0001B              ; (IX) ← 00010000001B
MOV     IXM, #1000B              ; (MP) ← 0001000B
MOV     IXH, #0000B              ;
MOV     R005, #8                 ; R005 ← 8
OR      IXH, #1000B              ; MPE ← 1
★ OR    PSW, #.DF.IXE AND 0FH    ; IXE ← 1
MOV     @R005, M034              ; Register indirect transfer
```

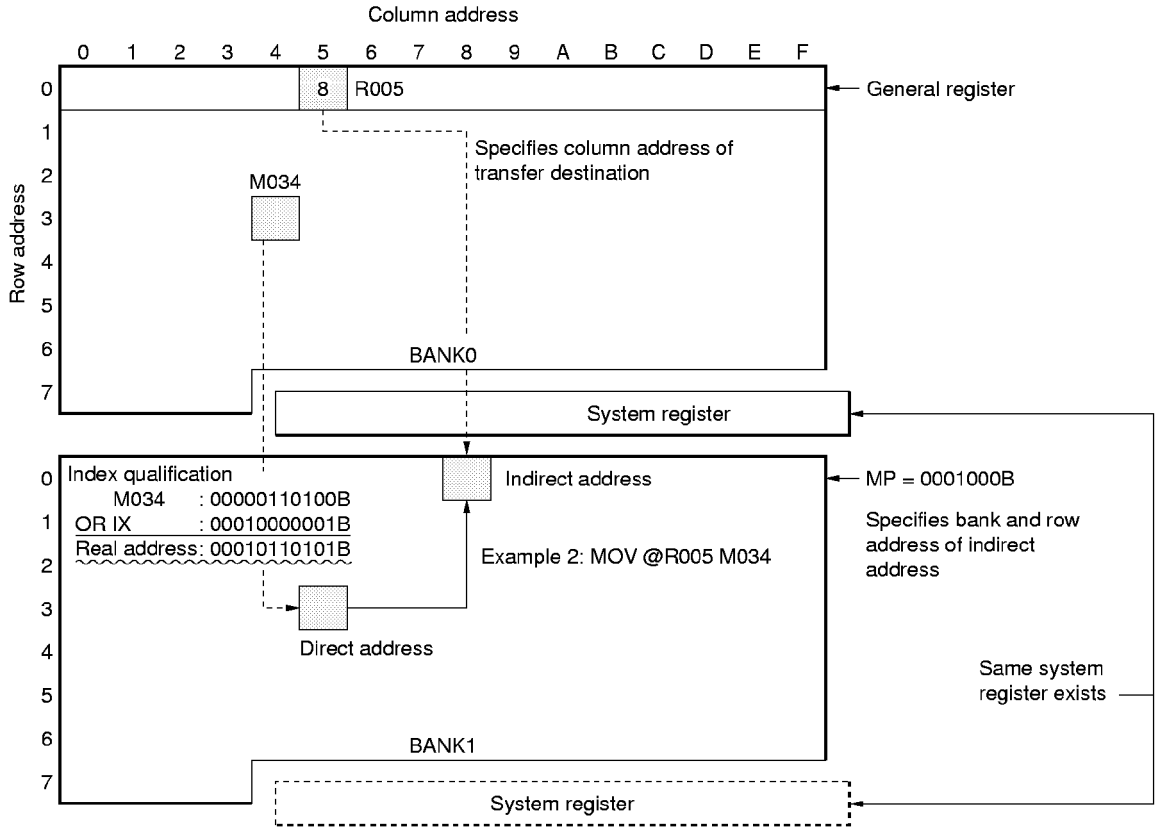
When the above instructions are executed, the data memory address 35H contents in BANK1 are transferred to address 08H in BANK1, as shown in Figure 6-15.

When the "MOV @r, m" instruction is executed with MPE = 1 and IXE = 1, therefore, the data memory address (direct address) specified by "m" is qualified with the index register contents, and the indirect address, specified by "@r", is specified by the data memory row address pointer contents.

To qualify the direct address, all the bank in the data memory address specified by "m", row address, and column address are ORed with the index register contents, and the indirect address, specified by @r, is the bank and row addresses, which are contained in the data memory row address pointer.

Therefore, in the above example, the direct address is 35H in BANK1, and the indirect address is 08H in BANK1. The difference between this example and **Example 2** in **6.6.5**, where MPE = 0, IXE = 1, is that the bank and row addresses, for the indirect address specified by "@r", are specified by the data memory row address pointer contents (in Example 2 in (5), the indirect address is qualified by the index register).

Figure 6-15. General Register Indirect Transfer Example When MPE = 1, IXE = 1



6.7 General Register Pointer (RP)

6.7.1 General register pointer configuration

Figure 6-16 shows the general register pointer configuration. As shown in this figure, the general register pointer consists of a total of 7 bits: 4 bits in address 7DH (RPH) for the system register and the high-order 3 bits in address 7EH (RPL).

Figure 6-16. Configuration of General Register Pointer

Address	7DH				7EH			
Name	General register pointer (RP)							
Symbol	RPH				RPL			
Bit	b_3	b_2	b_1	b_0	b_3	b_2	b_1	b_0
Data	M S B						L S B	B C D

6.7.2 General register pointer functions

The general register pointer specifies a general register (GR) on the data memory.

The general register can specify 16 nibbles, which are at the same row address on the data memory. Therefore, as shown in Figure 6-17, the general register pointer specifies which row address is to be used.

The row address in the data memory, that can be specified as the general register, differs depending on the general register pointer (RP) for each model.

When the data memory is specified as a general register, an arithmetic operation or data transfer can be executed between the general register and data memory.

For example, when an instruction, such as ADD r, m or LD r, m, is executed, addition or transfer is executed between a general register, addressed by operand "r" of the instruction, and the data memory, addresses by "m".

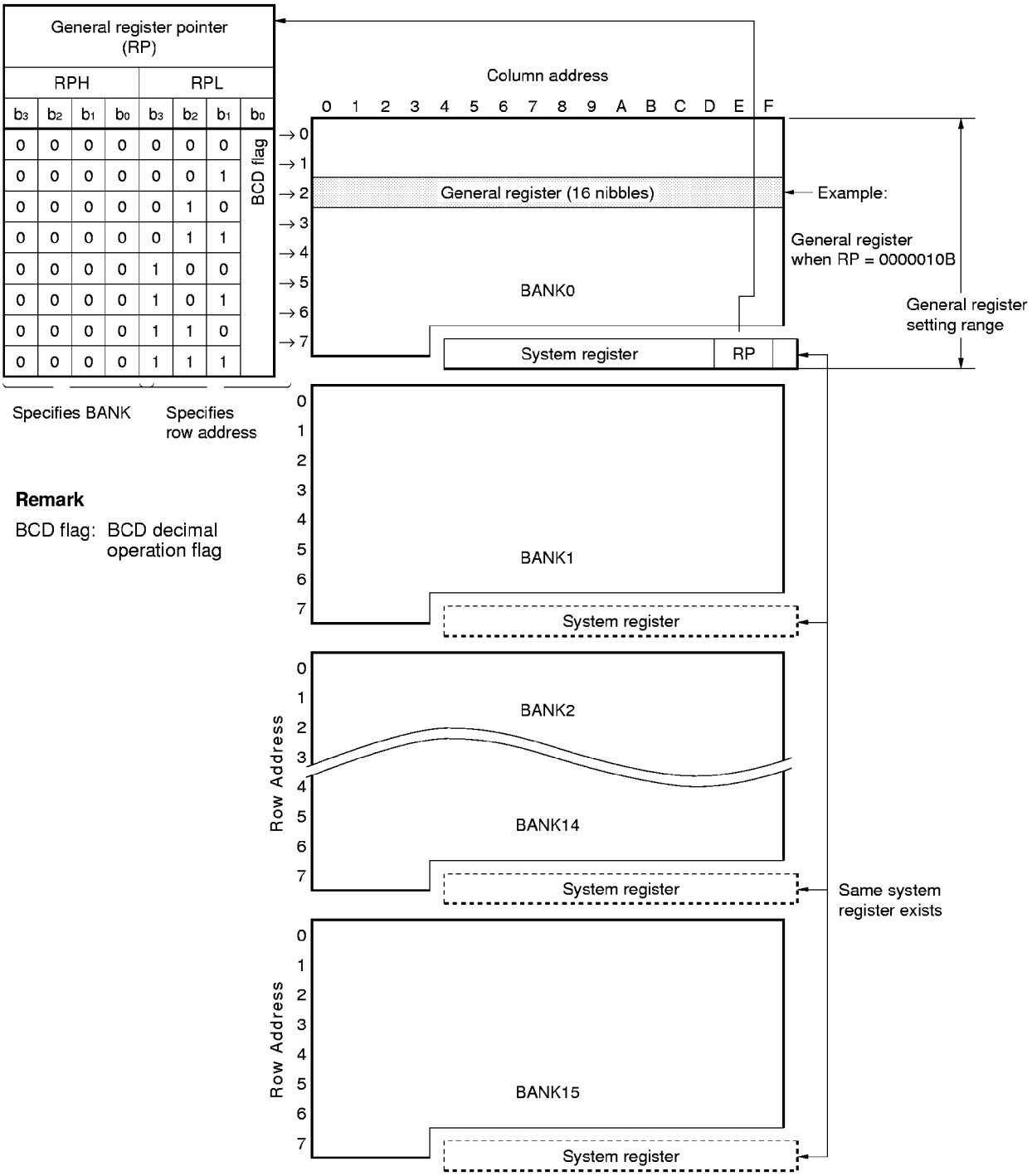
For details, refer to **CHAPTER 7 GENERAL REGISTER (GR)**.

★ **6.7.3 Notes on using general register pointer**

The lowest-order bit of address 7EH (RPL) to which the general register pointer is assigned is allocated to the BCD flag of the program status word.

Therefore, the value of the BCD flag is changed when RPL is rewritten.

Figure 6-17. Configuration of General Register



General register pointer (RP)							
RPH				RPL			
b_3	b_2	b_1	b_0	b_3	b_2	b_1	b_0
0	0	0	0	0	0	0	BCD flag
0	0	0	0	0	0	1	
0	0	0	0	0	1	0	
0	0	0	0	0	1	1	
0	0	0	0	1	0	0	
0	0	0	0	1	0	1	
0	0	0	0	1	1	0	
0	0	0	0	1	1	1	

Remark

BCD flag: BCD decimal operation flag

6.8 Program Status Word (PSWORD)

6.8.1 Program status word configuration

Figure 6-18 shows the program status word configuration.

As shown in this figure, the program status word consists of a total of 5 bits: the least significant bit in address 7EH (RPL) for the system register and 4 bits in 7FH (PSW). Each of the 5 bits in the program status word has its own function as a binary-coded decimal flag (BCD), compare flag (CMP), carry flag (CY), zero flag (Z), and index enable flag (IXE), respectively.

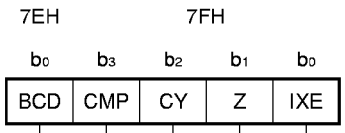
Figure 6-18. Configuration of Program Status Word

Address	7EH				7FH			
Name	(RP)			Program status word (PSWORD)				
Symbol	RPL				PSW			
Bit	b_3	b_2	b_1	b_0	b_3	b_2	b_1	b_0
Data				BCD	CMP	CY	Z	IXE

6.8.2 Program status word function

Each flag of the program status word sets the condition for an arithmetic operation or transfer instruction in the ALU (Arithmetic Logic Unit), or to indicate an operation result. Figure 6-19 shows the program status word functions.

Figure 6-19. Functions of Program Status Word



Index enable flag	Index modification is enabled when this flag is set.
Zero flag	Reset if result of arithmetic operation is other than "0". Set condition differs depending on contents of CMP flag. (1) When CMP = 0 Set if operation result is "0" (2) When CMP = 1 Set if Z = 1 and operation result is "0"
Carry flag	Set (1) if carry occurs as result of executing addition instruction, or if borrow occurs as result of executing subtraction instruction. Remains reset (0) if neither carry nor borrow occurs. Also set (1) if least significant bit of general register is "1" when RORC instruction is executed, and reset (0) if LSB is "0".
Compare flag	Result of arithmetic operation is not stored in data memory while this flag is set (1). CMP flag is reset (0) automatically when SKT or SKF instruction is executed.
BCD flag	All arithmetic operations are performed in decimal (BCD) when this flag is set (1), and in binary when this flag is reset (0).

6.8.3 Index enable flag (IXE)

The IXE flag is used to modify an address of the data memory when a data memory manipulation instruction is executed.

When this flag is set to 1, the contents of the data memory address specified by the instruction are ORed with the contents of the index register (IX), and the instruction is executed to the data memory addressed by the result of the OR operation (real address).

For details, refer to **6.6 Index Register (IX) and Data Memory Row Address Pointer (MP: Memory Pointer)**.

6.8.4 Zero (Z) and compare (CMP) flags

The Z flag indicates that the result of an arithmetic operation executed is 0, and the CMP flag made setting so that the result of an arithmetic operation is not stored in the data memory or general register.

The conditions under which the Z flag is set or reset differ depending on the status of the CMP flag, as shown in Table 6-2.

Table 6-2. Status of Compare Flag (CMP) and Set and Reset Conditions of Zero Flag (Z)

Condition	Status of Z Flag	
	When CMP Is 0	When CMP Is 1
On reset	Reset	Reset with CMP
When "0" is directly written to Z flag by data memory manipulation instruction	Reset	Reset
When "1" is directly written to Z flag by data memory manipulation instruction	Set	Set
If result of arithmetic operation is other than"0"	Reset	Reset
If result of arithmetic operation is "0"	Set	Retains previous status of Z flag

The Z and CMP flags are used to compare the contents of a general register with those of the data memory. The status of the Z flag is not changed by an operation other than an arithmetic operation, and the status of the CMP flag is not changed by an operation other than bit testing.

6.8.5 Carry flag (CY)

The CY flag indicates occurrence of a carry or borrow after an addition or subtraction instruction is executed.

The CY flag is set to 1 if a carry or borrow occurs as a result of the arithmetic operation; it is reset to 0 if neither a carry nor a borrow occurs.

When the "RORC r" instruction, which shifts the contents of a general register specified by r 1 bit to the right, is executed, the value of the CY flag immediately before the instruction is executed is shifted to the most significant bit position of the general register, and the least significant bit is shifted to the CY flag.

The CY flag is convenient for skipping the next instruction if a carry or borrow occurs.

★ The status of this flag is not changed by an operation other than arithmetic operation or rotation processing.

6.8.6 Binary coded decimal flag (BCD)

The BCD flag is used to execute a BCD operation.

When this flag is set to 1, all arithmetic operations are executed in BCD format. When it is reset to 0, the operations are executed in binary and 4-bit units.

★ This flag does not affect the logical operation, bit judgment, comparison, and rotation processing.

6.8.7 Notes on executing arithmetic operation

When executing an arithmetic operation (addition or subtraction) to the program status word (PSWORD), note that the "result" of the arithmetic operation is stored in the PSWORD, as indicated by the following example:

Example
```
MOV     PSW, #0001B
ADD     PSW, #1111B
```

When the above instructions are executed, a carry occurs. Consequently, the CY flag, which is bit 2 of the PSW, would be set to 1. Actually, however, 0000B is stored to the PSW because the result of the operation is 0000B.

6.9 Notes on Using System Registers

6.9.1 Reserved words of system registers

Because the system registers are located on the data memory, all the data memory manipulation instructions can ★ be used to manipulate the system registers. When using the 17K series assembler (RA17K), however, a data memory address must be defined as a symbol in advance because a data memory address cannot be directly written as the operand of an instruction.

Although the system registers are part of the data memory, they are defined as symbols as "reserved words" by ★ the assembler (RA17K) because they have dedicated functions, unlike the ordinary data memory areas.

The reserved words of the system registers are assigned to addresses 74H through 7FH, and are defined by symbols (such as AR3, AR2, and PSW) shown in **Figure 6-2 Configuration of System Registers**.

When these reserved words are used, it is not necessary to define a symbol, as shown in the following **Example 2**.

For the reserved words, refer to the Data Sheet of each model.

```
Examples 1.  MOV    34H, #0101B     ; If data memory address 34H or 76H is
             MOV    76H, #1010B     ; written as operand, error occurs.
             M037   MEM 0.37H       ; Data memory address of general-purpose
             MOV    M037, #0101B    ; data memory must be defined as symbol by MEM directive

          2. MOV    AR1, #1010B     ; Symbol needs not to be defined if reserved word AR1 (address 6H)
                                     ; is used.
                                     ; Reserved word AR1 is defined in device file as "AR1 MEM 0.76H"
```

★ When the assembler (RA17K) is used, the following macro instructions are embedded in the assembler as flag type symbol manipulation instructions:

```
SETn    : Sets flag to "1"
CLRn    : Resets flag to "0"
SKTn    : Skips if all flags are "1"
SKFn    : Skips if all flags are "0"
NOTn    : Inverts flag
INITFLG : Initializes flag
```

Therefore, by using these macro instructions, the data memory can be manipulated as flags as shown in **Example 3** below.

★ Since each bit (flag) of the program status word and memory pointer enable flag has its own function, a reserved word (MPE, BCD, CMP, CY, Z, or IXE) is defined for each bit.

By using this flag type reserved word, therefore, an embedded macro instruction can be used as is as shown in **Example 4**.

Examples 3.
```
F0003 FLG 0.00.3      ; Flag type symbol definition
      SET1 F0003      ; Embedded macro
```

Macro expansion
```
OR    .MF.F0003 SHR 4, #.DF.F0003 AND 0FH
                ; Sets bit 3 at address 00H in BANK0
```

4.
```
      SET1 BCD        ; Embedded macro
```

Macro expansion
```
OR    .MF.BCD SHR 4, #.DF.BCD AND 0FH
                ; Sets BCD flag
                ; BCD is defined by "BCD FLG 0.7EH.0"
```

```
CLR2 Z, CY      ; Flag of same address
```

Macro expansion
```
AND   .MF.Z SHR 4, #.DF. (NOT (Z OR CY) AND 0FH)
```

```
CLR2 Z, BCD     ; Flag of different addresses
```

Macro expansion
```
AND   .MF.Z SHR 4, #.DF. (NOT Z AND 0FH)
AND   .MF.BCD SHR 4, #.DF. (NOT BCD AND 0FH)
```

6.9.2 Handling system register fixed to "0"

Data of the system registers fixed to "0" (refer to **Figure 6-2. Configuration of System Registers** calls for your attention when the device, emulator, or assembler operates, as described in **(1)**, **(2)**, and **(3)** below.

(1) When device operates

The data fixed to "0" is not affected even when a write instruction is executed to it. When this data is read, "0" is always read.

★ **(2) When using 17K series in-circuit emulator (IE-17K or IE-17K-ET)**

An error occurs if an instruction that writes "1" is executed to the data fixed to "0".
Therefore, if the following instructions are executed, an error occurs on the in-circuit emulator:

Examples 1.
```
MOV     BANK, #0100B    ; Writes 1 to bit 3 fixed to 0
```

2.
```
MOV     IXL, #1111B     ;
MOV     IXM, #1111B     ;
MOV     IXH, #0001B     ;
ADD     IXL, #1         ;
ADDC    IXM, #0         ;
ADDC    IXH, #0         ;
```

However, an error does not occur even if the "INC AR" or "INC IX" instruction is executed when all the valid bits are "1" as shown in Example 2. This is because the "INC" instruction, which is executed when all the valid bits of the address register and index register are "1", clears all the valid bits to "0".
Even if "1" is written to the data fixed to "0" of the address register as shown in **Examples 1** and **2** above, an error does not occur.

★ **(3) When using 17K series assembler (RA17K)**

An error is not output even if there is an instruction that writes "1" to data fixed to "0". Therefore, when "MOV BANK, #0100B" instruction shown in **Example 1** is used, the assembler does not cause an error, but an emulator error occurs when the instruction is executed on the in-circuit emulator.

★ The assembler (RA17K) does not causes an error because it cannot detect the data memory address subject to manipulation by an instruction while register indirect transfer is executed.
The assembler causes an error only on the following occasion:

When value greater than 1 is used as "n" of embedded macro instruction "BANKn"

This is because it is judged that the bank register of the system registers is to be explicitly manipulated when the BANKn instruction is used.

[MEMO]

CHAPTER 7 GENERAL REGISTER (GR)

The general registers are located on the data memory. They perform direct arithmetic operations and transfer operations with the data memory.

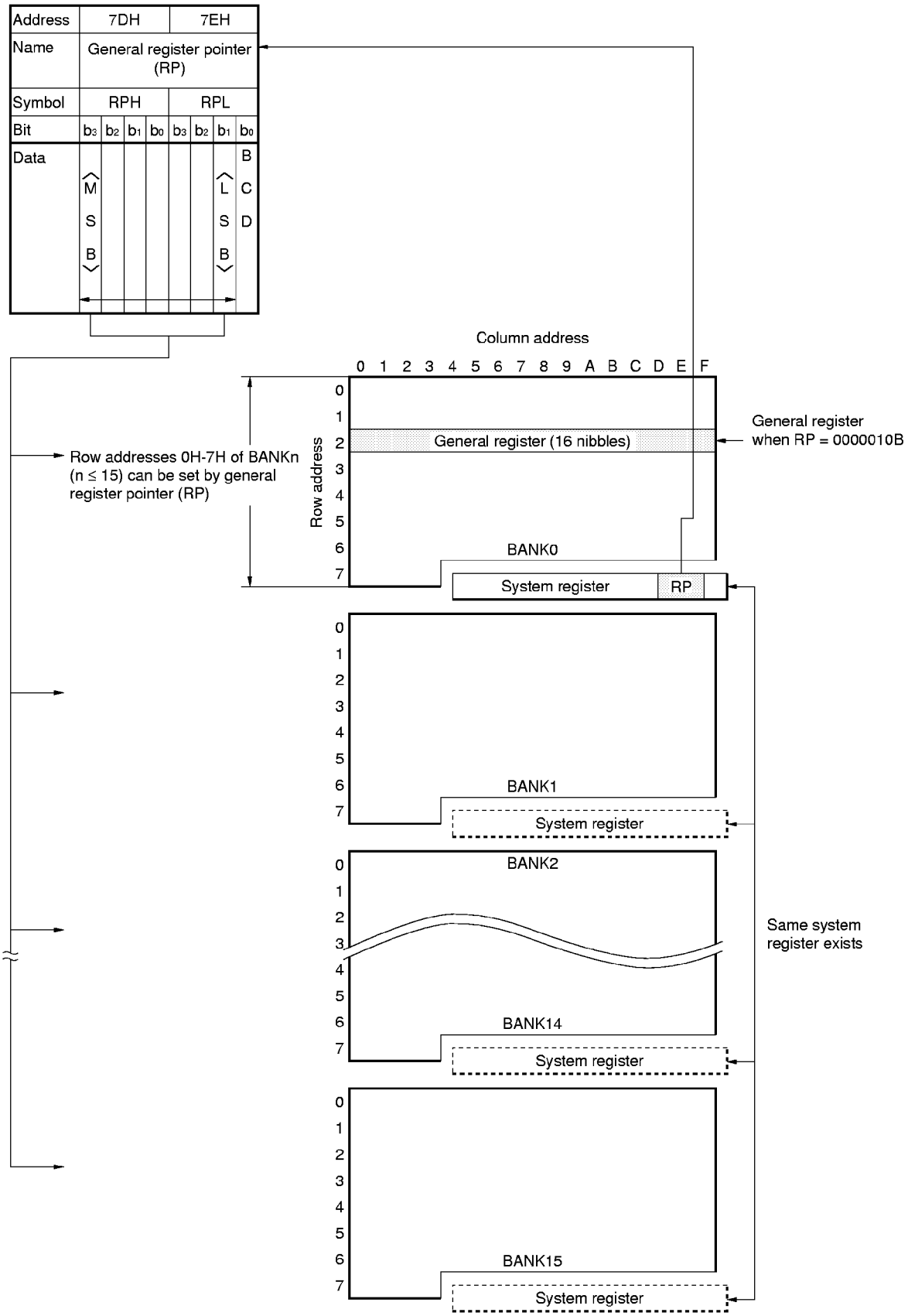
7.1 General Register Configuration

The general register configuration is shown in Figure 7-1. As shown in this figure, sixteen nibbles in the row addresses for the data memory (16 words × 4 bits) can be used as a general register area.

Which row, among row addresses to be used, is specified by the general register pointer. Specified row address is set to the general register pointer in the system register.

For details, refer to **6.7 General Register Pointer (RP)**.

Figure 7-1. Configuration of General Register



7.2 General Register Functions

By using the general register, arithmetic operations and transfer operations can be executed between the data memory and general memory with a single instruction.

To put this in another way, since the general register is on the data memory, arithmetic operations and data transfer between two data memory addresses can be executed with a single instruction.

Moreover, the general register can be controlled by the data memory manipulation instruction in the same manner as the data memory, since the general register is on the data memory.

For details on data memory manipulation instructions, refer to **7.4 Address Generation and Operation for General Register and Data Memory by Each Instruction**.

7.3 Notes on General Register Use

7.3.1 through **7.3.3** describes the points to be noted in using the general register.

7.3.1 Address specification for general register

★ When using the 17K Series Assembler (RA17K), an error occurs, if a general register address is directly described as the operand for an instruction, as shown below.

This assembler feature reduces the bugs causes, when the program is edited.

★ Therefore, a general register address should be defined as a symbol in advance.

Example Error occurs

```
LD      04H, 32H    ; General register address or data memory address is directly specified
                    ; as a numeral
```

Error does not occur

```
R004    MEM 0.04H   ; Defines address 04H as a symbol, in R004 as memory type
M032    MEM 0.32H   ;
LD      R004, M032  ;
```

7.3.2 Row address in general

Since the row address in the general register is determined by the general register pointer, the bank for the address and row address, specified by operand "r" for an instruction, are ignored.

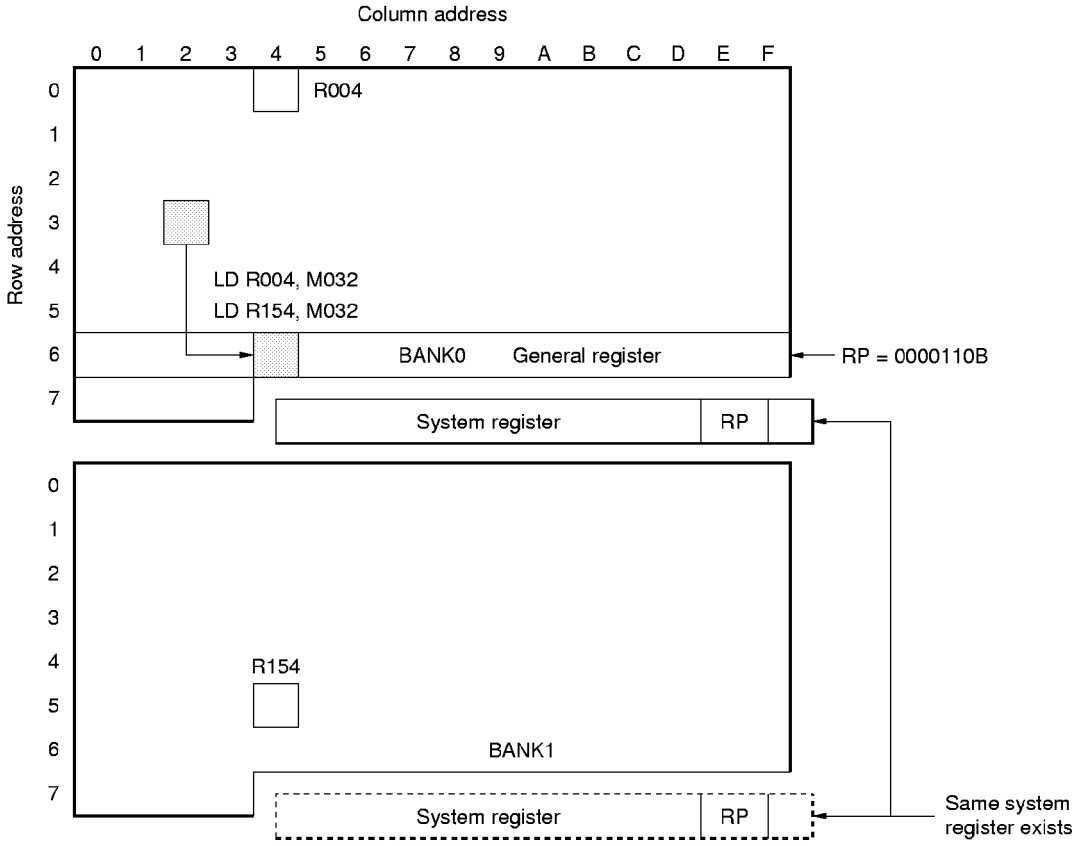
If the following example program is executed, both <1> and <2> transfer the data memory M032 contents (address 32H in BANK0) to address 64H in BANK0, as shown in Figure 7-2.

That is, instructions <1> and <2> ignore the bank and row addresses for R004 and R154, which specify a general register address, and only address 4H in the column address is valid.

Example Specifying row address for general register, where BANK0 is specified

```
R004    MEM 0.04H   ;
R154    MEM 1.54H   ;
M032    MEM 0.32H   ;
MOV     RPH, #0000B ;
MOV     RPL, #0110B ; RP ← 0000110B
; <1>
  LD    R004, M032
; <2>
  LD    R154, M032
```

Figure 7-2. Example of Specifying General Register Row Address



7.3.3 Operation between general register and immediate data

There is no instruction that executes an arithmetic operation between a general register and immediate data. To execute an operation between a data memory area, specified as a general register, and immediate data, the data memory area must be treated as a data memory area, instead of as a general register. For example,

Example **Example showing operation between general register and immediate data, where BANK0 is specified**

```
 R065     MEM 0.65H   ;
 M105     MEM 1.05H   ;
; <1>
   MOV    RPH, #0001B ; Sets general register at row address 6H in BANK0
   MOV    RPL, #0100B ;
★  BANK1               ; Assembler (RA7K) macroinstruction
; <2>
   ADD R065, #3        ;
; <3>
   ADD M105, #3        ;
```

In the above Example <2>, immediate data 3 is added to a data memory area at address 65H in BANK1. In <3>, 3 is added to a data memory area at address 0.5H.
Although the general register is set at row address 6H for BANK0 in <1>, the instruction <2> operand, R065, is treated as data memory, rather than as a general register.
Therefore, to add data to the general register at address 6H in BANK0 with instruction <2>, the following program must be used:

```
★ BANK0               ; Assembler (RA17K) macroinstruction
  ADD    R065, #3
```

7.4 Address Generation and Operation for General Register and Data Memory by Each Instruction

Table 7-1 lists the instructions that execute arithmetic operation or data transfer between a general register and a data memory area.

To specify an address with these instructions, taking instruction ADD r, m for example, general register R is specified by the register pointer contents and the operand r value for the instruction, as shown in Figure 7-3.

Data memory address "M" is specified by the bank register contents and operand m for the instruction.

Therefore, this instruction adds the general register "(R)" contents to the data memory contents "(M)", and stores the result in general register R.

This general register address generation is executed by the other instructions listed in Table 7-1. Examples 1 through 3 show instructions operation examples.

Table 7-1. Instructions Manipulating General Register and Data Memory

Group	Instruction	Operation
Addition	ADD r, m	(R) ← (R) + (M)
	ADDC r, m	(R) ← (R) + (M) + (CY)
Subtraction	SUB r, m	(R) ← (R) – (M)
	SUBC r, m	(R) ← (R) – (M) – (CY)
Logical operation	AND r, m	(R) ← (R) AND (M)
	OR r, m	(R) ← (R) OR (M)
	XOR r, m	(R) ← (R) XOR (M)
Transfer	LD r, m	(R) ← (M)
	ST m, r	(M) ← (R)
	MOV @r, m	[MP, (R)] ← (M) or, [m, (R)] ← (M)
	MOV m, @r	M ← [MP, (R)] or, M ← [H, (R)]
Shift	RORC r	Right shift with (CY)

Figure 7-3. Address Specification for General Register and Data Memory

Instruction	Address Contents	Generated Address										
		Symbol	Bank				Row address			Column address			
			b_3	b_2	b_1	b_0	b_2	b_1	b_0	b_3	b_2	b_1	b_0
ADD r, m	Address of general register specified by r	R	(RP)							r			
	Address of data memory specified by m	M	(BANK)				m						

Examples 1. Operation between data memory and general register

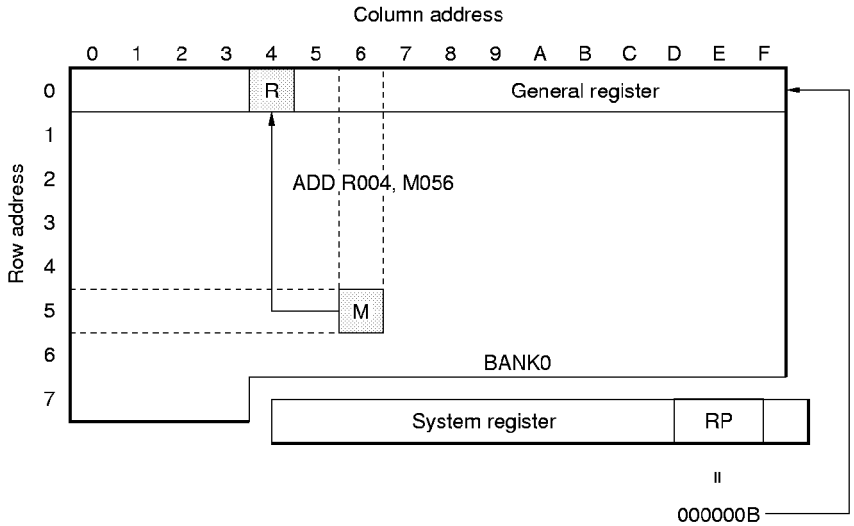
(1) When the bank for the data memory is equal to that for the general register

Assuming that a general register is at row address 0H in BANK0

```
R004    MEM 0.04H   ; Symbol definition
M056    MEM 0.56H   ;
ADD     R004, M056  ; Addition of contents of data memory and general register
```

When the above instructions are executed, the general register R004 contents (address 04H of BANK0) are added to those for data memory M056 (address 56H), and the result is stored in general register R004 (04H), as shown in Figure 7-4.

Figure 7-4. Example Showing Operation between Data Memory and General Register (1)



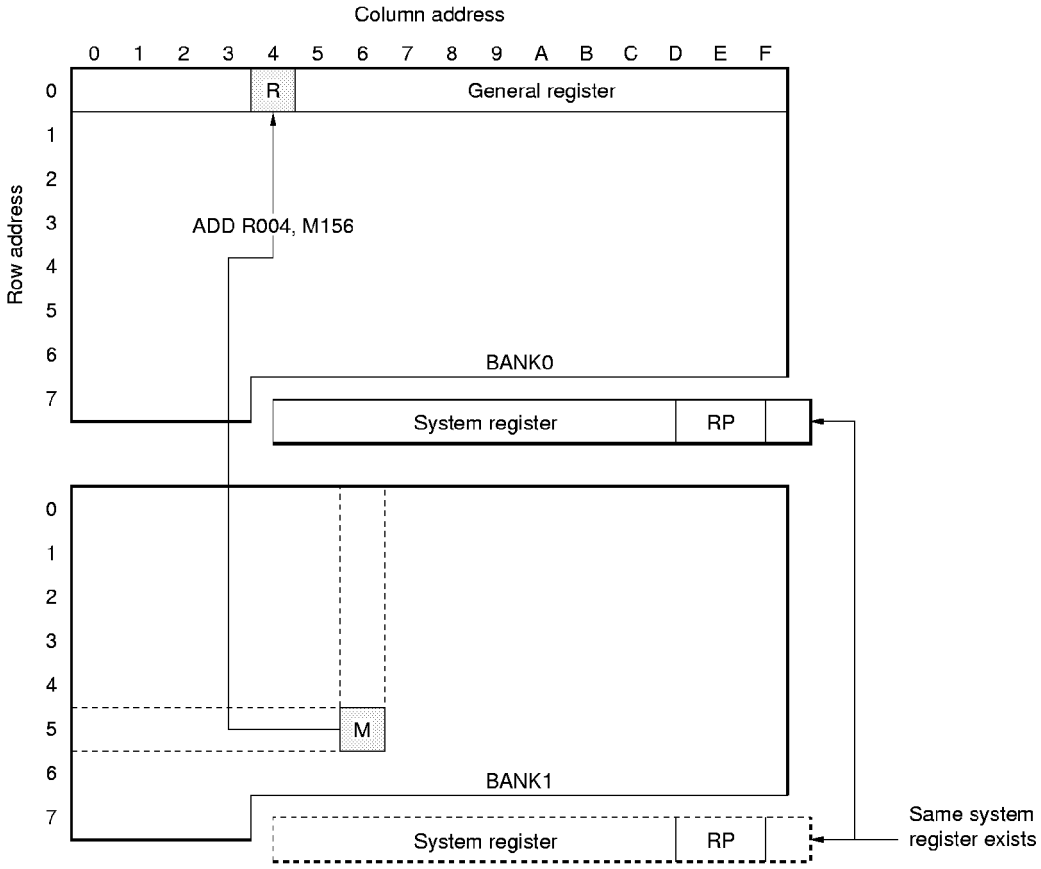
(2) When the data memory bank is different from that for the general register

Assuming the general register is at row address 0H in BANK0

```
R004    MEM 0.04H    ; Symbol definition
M156    MEM 1.56H    ;
BANK1                ; Assembler (RA17K) macroinstruction
ADD     R004, M156   ; Addition of contents of data memory and general register
```

When the above instructions are executed, the general register R004 contents (at address 04H in BANK0) are added to those for the data memory M156 (address 56H in BANK1), and the result is stored in general register R004 (04H), as shown in Figure 7-5.
Although the selected bank is BANK1, the data memories in BANK1 and BANK0 are added with a single instruction, because the general register is in BANK0.

Figure 7-5. Example Showing Operation between Data Memory and General Register (2)



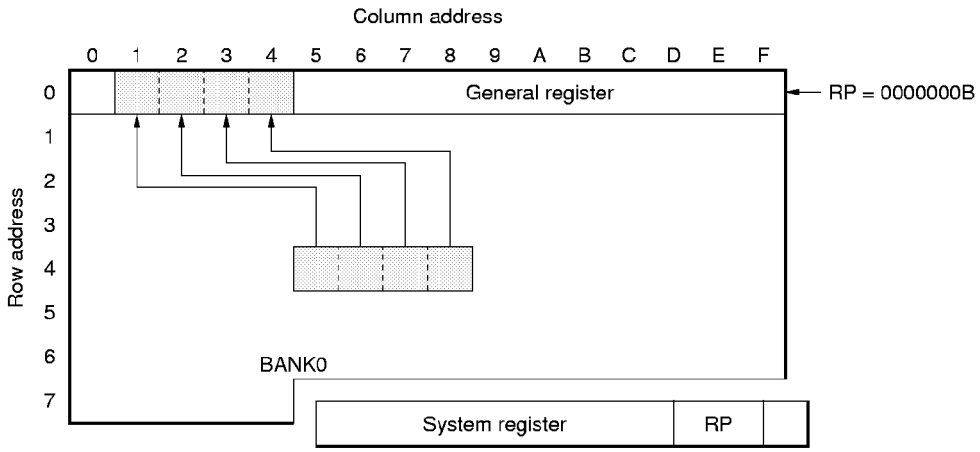
Examples 2. Transfer to general register

Assuming that a general register is at row address 0H in BANK0

```
R001    MEM  0.01H  ; Symbol definition
R002    MEM  0.02H  ;
R003    MEM  0.03H  ;
R004    MEM  0.04H  ;
M045    MEM  0.45H  ;
M046    MEM  0.46H  ;
M047    MEM  0.47H  ;
M048    MEM  0.48H  ;
LD      R001, M045
LD      R002, M046
LD      R003, M047
LD      R004, M048
```

This program transfers the contents for data memory areas M045, M046, M047, and M048 (addresses 45H, 46H, 47H, and 48H) to general registers R001, R002, R003, and R004 (addresses 01H, 02H, 03H, and 04H), respectively.

Figure 7-6. Example Showing Data Transfer to General Register



Examples 3. Indirect transfer to general register

Assuming that the row address for a general register is 0H in BANK0

```
R004    MEM 0.04H     ;
M052    MEM 0.52H     ;
MOV     R004, #8      ; (R004) ← 8
MOV     @R004, M052 ; General register indirect transfer
```

When the above instructions are executed, the data memory area M052 contents (address 52H) are transferred to another data memory area (in this example, address 58H), as shown in Figure 7-7.

The "MOV @r, m" instruction is called a general register indirect transfer instruction. It transfers the contents in a data memory area, addressed by m, to another data memory area, specified by @r (called an indirect address).

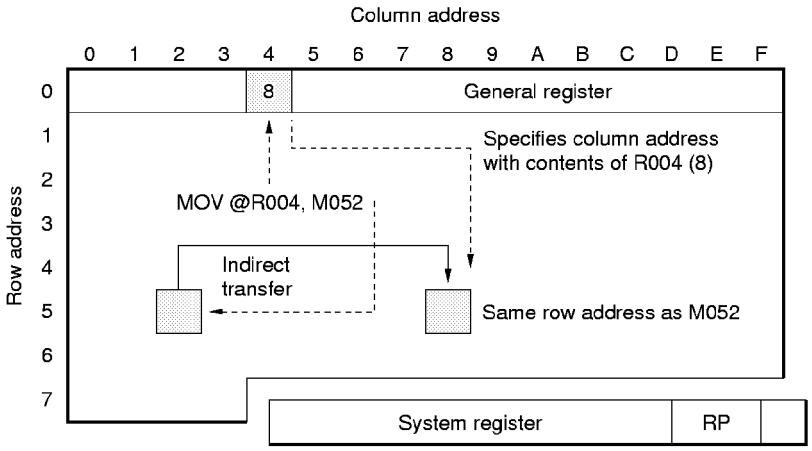
★ At this time, the data memory address (indirect address) for indirect transfer specified by @r is as follows:

- Row address : The same row address as data memory specified by m
- Column address : Contents of the general register specified by r

In **Example 3** above, the row address is 5H (row address of address 52H), and the column address is 8H (contents of address 08 is 8), therefore, data memory address is address 58H.

For details on the general register indirect transfer, refer to **6.6 Index Register (IX) and Data Memory Row Address Pointer (MP: Memory Pointer)**.

Figure 7-7. General Register Indirect Transfer Example



Examples 4. To change a row address in general register

Assuming that general register is at row address 0H in BANK0

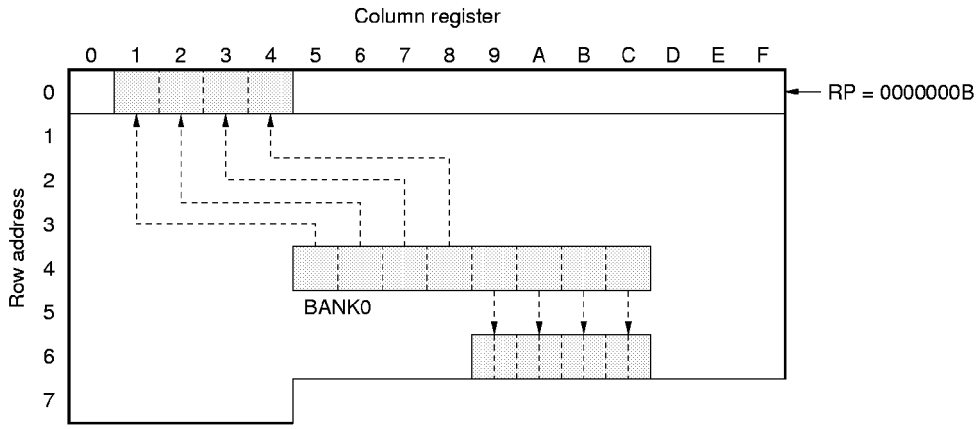
```
R001    MEM 0.01H   ; Symbol definition
R002    MEM 0.02H   ;
R003    MEM 0.03H   ;
R004    MEM 0.04H   ;
R005    MEM 0.05H   ;
R006    MEM 0.06H   ;
R007    MEM 0.07H   ;
R008    MEM 0.08H   ;
M045    MEM 0.45H   ;
M046    MEM 0.46H   ;
M047    MEM 0.47H   ;
M048    MEM 0.48H   ;
M049    MEM 0.49H   ;
M04A    MEM 0.4AH   ;
M04B    MEM 0.4BH   ;
M04C    MEM 0.4CH   ;
LD      R001, M045
LD      R002, M046
LD      R003, M047
LD      R004, M048
; <1>
  MOV   RPH, #0000B ; Transfers 0000110B to general register pointer, i.e., sets row
  MOV   RPL, #0110B ; address 6H in BANK0
  LD    R005, M049
  LD    R006, M04A
  LD    R007, M04B
  LD    R008, M04C
```

The above program is to transfer the contents in 8-nibble data memory M045-M04C on BANK0 to a different row address in BANK0, 4 nibbles at a time. At this time, if the general register is fixed, for example, when it exists only at row address 0 in BANK0, an instruction is necessary to enable the above program <1> to transfer all 8 nibbles to the general register and then store the nibbles in the data memory again, as shown in the following program.

However, as shown in the above program, the operation can be performed with only the LD instruction, if the row address for the general register is changed by the general register pointer.

```
M065    MEM 0.65H   ; Symbol definition
M066    MEM 0.66H   ;
M067    MEM 0.67H   ;
M068    MEM 0.68H   ;
LD      R005, M049
LD      R006, M04A
LD      R007, M04B
LD      R008, M04C
BANK1               ; Assembler (RA17K) macroinstruction
                    ; BANK ← 1
ST      M065, R005
ST      M066, R006
ST      M067, R007
ST      M068, R008
```

Figure 7-8. Example Showing Changing Row Address in General Register



CHAPTER 8 ARITHMETIC LOGIC UNIT (ALU)

The ALU performs arithmetic operations, logical operations, bit testings, compare, and rotations of 4-bit data.

8.1 ALU Block Configuration

Figure 8-1 shows the configuration of the ALU block.

As shown, the ALU block consists of an ALU, which processes 4-bit data, temporary registers A and B, which are peripheral circuits of the ALU, status flip-flops controlling the status of the ALU, and a decimal correction circuit that is used when a BCD operation is performed.

★ The status flip-flops include a zero flag FF, carry flag FF, compare flag FF, and BCD flag FF, as shown in Figure 8-1.

The status flip-flops correspond to the zero (Z), carry (CY), compare (CMP), and BCD (BCD) flags of the program status word (PSWORD: addresses 7EH and 7FH) of the system registers on a one-to-one basis.

8.2 ALU Block Function

The ALU performs arithmetic operation, logical operation, bit testing, compare, or rotation processing, depending on the instructions written to the program. Table 8-1 lists the operation, testing, and rotation instructions.

By executing each of the instructions listed in this table, operation in 4-bit units, testing, rotation processing, or 1-digit decimal operation can be executed with a single instruction.

8.2.1 ALU function

Arithmetic operations include addition and subtraction. An arithmetic operation can be executed between the contents of a general register and those of the data memory, or between the contents of the data memory and immediate data. In addition, an arithmetic operation can be executed in binary number in 4-bit units, or in decimal number in 1-digit units (BCD operation).

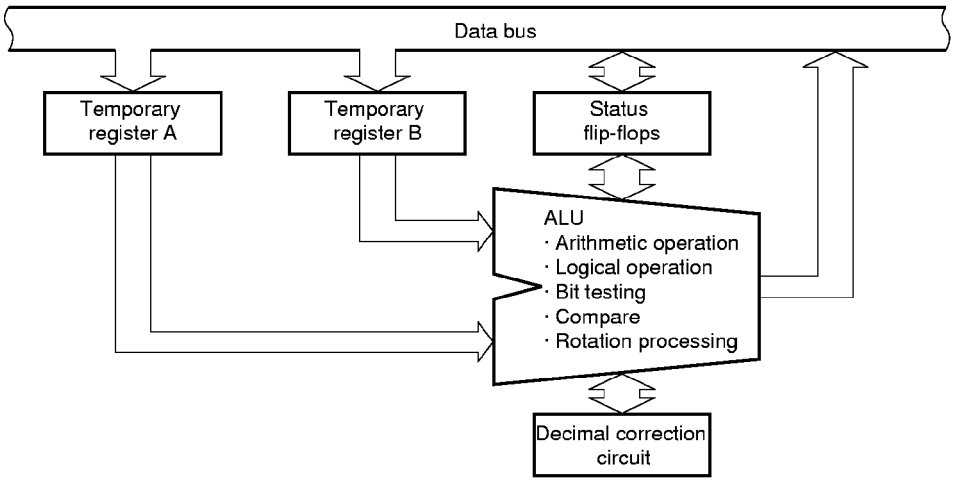
Logical operations include logical product (AND), logical sum (OR), and exclusive logical sum (XOR). A logical operation can be executed between the contents of a general register and those of the data memory, or between the contents of the data memory and immediate data.

Bit testing is to test whether one of the bits of the 4-bit data in the data memory is "0" or "1".

Comparison is to compare the contents of the data memory with immediate data to judge whether one data is "equal to", "not equal to", "greater than", or "less than" the other.

Rotation processing is to shift the 4-bit data of a general register 1 bit toward the least significant bit direction (rotation to the right).

Figure 8-1. Configuration of ALU Block



★

Address	7EH	7FH			
Name	Program status word (PSWORD)				
Bit	b_0	b_3	b_2	b_1	b_0
Flag	BCD	CMP	CY	Z	IXE

Status Flip-flop			
BCD flag FF	CMP flag FF	CY flag FF	Z flag FF

Functional Outline
Indicates result of arithmetic operation is 0
Stores carry or borrow resulting from arithmetic operation
Specifies whether result of arithmetic operation is stored
Specifies whether decimal correction is performed for arithmetic operation

[MEMO]

★ **Table 8-1. ALU Processing Instructions (1/2)**

ALU Function		Instruction	Operation	Remarks
Arithmetic	Addition	ADD r, m	(r) ← (r) + (m)	Adds general register and data memory contents, and stores result to general register
		ADD m, #n4	(m) ← (m) + n4	Adds data memory and immediate data contents, and stores result to data memory
		ADDC r, m	(r) ← (r) + (m) + CY	Adds general register and data memory contents with CY flag, and stores result to general register
		ADDC m, #n4	(m) ← (m) + n4 + CY	Adds data memory and immediate data contents with CY flag, and stores result to data memory
	Subtraction	SUB r, m	(r) ← (r) – (m)	Subtracts data memory contents from general register contents, and stores result to general register
		SUB m, #n4	(m) ← (m) – n4	Subtracts immediate data from data memory contents, and stores result to data memory
		SUBC r, m	(r) ← (r) – (m) – CY	Subtracts data memory contents from general register contents with CY flag, and stores result to general register
		SUBC m, #n4	(m) ← (m) – n4 – CY	Subtracts immediate data and CY flag from data memory contents, and stores result to data memory
Logical	OR	OR r, m	(r) ← (r) ∨ (m)	ORs general register and data memory contents, and stores result to general register
		OR m, #n4	(m) ← (m) ∨ n4	ORs data memory contents and immediate data, and stores result to data memory
	AND	AND r, m	(r) ← (r) ∧ (m)	ANDs general register and data memory contents, and stores result to general register
		AND m, #n4	(m) ← (m) ∧ n4	ANDs data memory contents and immediate data, and stores result to data memory
	XOR	XOR r, m	(r) ← (r) ⊽ (m)	XORs general register and data memory contents, and stores result to general register
		XOR m, #n4	(m) ← (m) ⊽ n4	XORs data memory contents and immediate data, and stores result to data memory
Bit testing	True	SKT m, #n	CMP ← 0, if (m) ∧ n = n, then skip	Skips if all bits of data memory contents specified by n are True (1). Result is not stored
	False	SKF m, #n	CMP ← 0, if (m) ∧ n = 0, then skip	Skips if all bits of data memory contents specified by n are False (0). Result is not stored
Compare	Equal to	SKE m, #n4	(m) – n4, skip if zero	Skips if data memory contents are equal to immediate data. Result is not stored
	Not equal to	SKNE m, #n4	(m) – n4, skip if not zero	Skips if data memory contents are not equal to immediate data. Result is not stored
	Greater than	SKGE m, #n4	(m) – n4, skip if not borrow	Skips if data memory contents are greater than immediate data. Result is not stored
	Less than	SKLT m, #n4	(m) – n4, skip if borrow	Skips if data memory contents are less than immediate data. Result is not stored
Rotation	Right rotation	RORC r	→ CY → (r)$_{b3}$ → (r)$_{b1}$ → (r)$_{b2}$ → (r)$_{b0}$ →	Rotates general register contents to right with CY flag, and stores result to general register

★ **Table 8-1. ALU Processing Instructions (2/2)**

<table>
<tr><th>ALU Function</th><th>Difference in Operation Because of Program Status Word (PSWORD)</th></tr>
<tr><td>Arithmetic operation</td><td>
<table>
<tr><th>Value of BCD Flag</th><th>Value of CMP Flag</th><th>Operation</th><th>CY Flag</th><th>Z Flag</th><th>Modification when IXE = 1</th></tr>
<tr><td>0</td><td>0</td><td>Binary operation. Result is stored.</td><td rowspan="4">Set when carry or borrow occurs; otherwise, reset</td><td>Set if operation result is 0000B; otherwise, reset</td><td rowspan="4">Executed</td></tr>
<tr><td>0</td><td>1</td><td>Binary operation. Result is not stored.</td><td>Retains status if operation result is 0000B; otherwise, reset</td></tr>
<tr><td>1</td><td>0</td><td>BCD operation. Result is stored.</td><td>Set if operation result is 0000B; otherwise, reset</td></tr>
<tr><td>1</td><td>1</td><td>BCD operation. Result is not stored.</td><td>Retains status if operation result is 0000B; otherwise, reset</td></tr>
</table>
</td></tr>
<tr><td>Logical operation</td><td>
<table>
<tr><td>Don't care (retained)</td><td>Don't care (retained)</td><td>Not affected</td><td>Don't care (retained)</td><td>Don't care (retained)</td><td>Executed</td></tr>
</table>
</td></tr>
<tr><td>Bit testing</td><td>
<table>
<tr><td>Don't care (retained)</td><td>Reset</td><td>Not affected</td><td>Don't care (retained)</td><td>Don't care (retained)</td><td>Executed</td></tr>
</table>
</td></tr>
<tr><td>Comparison</td><td>
<table>
<tr><td>Don't care (retained)</td><td>Don't care (retained)</td><td>Not affected</td><td>Don't care (retained)</td><td>Don't care (retained)</td><td>Executed</td></tr>
</table>
</td></tr>
<tr><td>Rotation</td><td>
<table>
<tr><td>Don't care (retained)</td><td>Don't care (retained)</td><td>Not affected</td><td>Value of b_0 of general register</td><td>Don't care (retained)</td><td>Executed</td></tr>
</table>
</td></tr>
</table>

8.2.2 Functions of temporary registers A and B

The temporary registers A and B are necessary for processing 4-bit data at one time, and temporarily store data to be processed and data processing.

8.2.3 Status flip-flop functions

The status flip-flops control the operations of the ALU and store the status of the processed data. Since these flip-flops correspond to the flags of the program status word (PSWORD) on a one-to-one basis, they can be manipulated by manipulating the system register. Each flag of the program status word has the following functions:

★ **(1) Z flag**

This flag is set to 1 if the result of an arithmetic operation is 0000B; otherwise, it is reset to 0.
However, the condition under which this flag is set to 1 differs depending on the status of the CMP flag, as follows:

(i) When CMP flag = 0

The Z flag is set to 1 if the result of an operation is 0000B; otherwise, it is reset to 0.

(ii) When CMP flag = 1

The Z flag retains the previous status if the result of an operation is 0000B; otherwise, it is reset to 0.
The flag is not affected by an operation other than arithmetic operations.

★ **(2) CY flag**

This flag is set to 1 if a carry or borrow occurs as a result of an arithmetic operation; otherwise, it is reset to 0.
If an arithmetic operation executed involves a carry or borrow, the content of the CY flag is reflected on the least significant bit of the execution result.
When rotation processing (RORC instruction) is executed, the content of the CY flag at that time is loaded to the most significant bit (b_3) position of a general register, and the content of the least significant bit of the general register is loaded to the CY flag.
The CY flag is not affected by any operation other than arithmetic operation and rotation processing.

★ **(3) CMP flag**

The result of an arithmetic operation executed when the CMP flag is set to 1 is not stored in a general register or data memory.
If a bit test instruction is executed, the CMP flag is reset to 0.
This flag does not affect the compare and logical operations, and rotation processing.

★ **(4) BCD flag**

When the BCD flag is set to 1, the results of all the arithmetic operations executed are corrected to decimal.
When this flag is reset to 0, operation is performed in binary 4-bit.
The BCD flag does not affect the logical operation, bit test, compare, and rotation processing.

The values of these flags can be changed by directly manipulating the program status word. At this time, the value of the corresponding status flip-flop is changed accordingly.

8.2.4 Binary 4-bit operation

An arithmetic operation is executed in binary and in 4-bit units when the BCD flag is 0.

8.2.5 BCD operation

When the BCD flag is 1, the arithmetic operation is performed in decimal format. The differences between the binary 4-bit operation and BCD operation are shown in Table 8-2. If the result of a decimal correction operation is more than 20, of if the result of a decimal subtraction is other than –10 to +9, data for more than 1010B (0AH) is stored in the data memory (shaded part in Table 8-2).

★ **Table 8-2. Results for Binary 4-bit and BCD Operations**

Result	Binary 4-bit Addition		BCD Addition	
	CY	Result	CY	Result
0	0	0000	0	0000
1	0	0001	0	0001
2	0	0010	0	0010
3	0	0011	0	0011
4	0	0100	0	0100
5	0	0101	0	0101
6	0	0110	0	0110
7	0	0111	0	0111
8	0	1000	0	1000
9	0	1001	0	1001
10	0	1010	1	0000
11	0	1011	1	0001
12	0	1100	1	0010
13	0	1101	1	0011
14	0	1110	1	0100
15	0	1111	1	0101
16	1	0000	1	0110
17	1	0001	1	0111
18	1	0010	1	1000
19	1	0011	1	1001
20	1	0100	1	1110
21	1	0101	1	1111
22	1	0110	1	1100
23	1	0111	1	1101
24	1	1000	1	1110
25	1	1001	1	1111
26	1	1010	1	1100
27	1	1011	1	1101
28	1	1100	1	1010
29	1	1101	1	1011
30	1	1110	1	1100
31	1	1111	1	1101

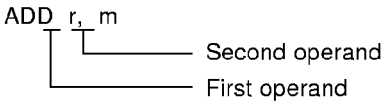
Result	Binary 4-bit Addition		BCD Addition	
	CY	Result	CY	Result
0	0	0000	0	0000
1	0	0001	0	0001
2	0	0010	0	0010
3	0	0011	0	0011
4	0	0100	0	0100
5	0	0101	0	0101
6	0	0110	0	0110
7	0	0111	0	0111
8	0	1000	0	1000
9	0	1001	0	1001
10	0	1010	1	1100
11	0	1011	1	1101
12	0	1100	1	1110
13	0	1101	1	1111
14	0	1110	1	1100
15	0	1111	1	1101
–16	1	0000	1	1110
–15	1	0001	1	1111
–14	1	0010	1	1100
–13	1	0011	1	1101
–12	1	0100	1	1110
–11	1	0101	1	1111
–10	1	0110	1	0000
–9	1	0111	1	0001
–8	1	1000	1	0010
–7	1	1001	1	0011
–6	1	1010	1	0100
–5	1	1011	1	0101
–4	1	1100	1	0110
–3	1	1101	1	0111
–2	1	1110	1	1000
–1	1	1111	1	1001

8.2.6 ALU block processing sequence

When an arithmetic operation, logical operation, bit testing, compare, or rotation processing instruction is executed on the program, the data to be operated, tested, or processed and processing data are temporarily stored in temporary registers A and B.

The data to be processed is the contents of a general register or data memory addressed by the first operand of the instruction, and is 4-bit data. The processing data is the contents of the data memory addressed by the second or immediate data directly specified by the second operand, and is 4-bit data.

Take the following instruction for example:



The data to be processed is the contents of a general register addressed by r, and the processing data is the contents of the data memory addressed by m.

ADD m, #n4

The data to be processed by this instruction is the contents of the data memory addressed by m, and the processing data is immediate data specified by #n4.

RORC r

With the following rotation processing instruction, only the data to be processed is necessary because the processing method is determined, and the data to be processed is the contents of a general register addressed by r:

The data stored in temporary registers A and B are operated arithmetically or logically, tested, compared, or rotated according to the instruction executed. If an arithmetic operation, logical operation, or rotation processing instruction has been executed, the data processed by the ALU is stored in a general register or the data memory addressed by the first operand of the instruction, and the operation is finished. If a bit testing or compare instruction is executed, the next instruction on the program is skipped (i.e., executed as an NOP instruction) depending on the result of the processing performed by the ALU, and the operation is finished.

Bear in mind the following points when using the ALU block:

(1) Arithmetic operations are affected by the CMP and BCD flags of the program status word.
(2) Logical operations are not affected by the CMP and BCD flags of the program status word, and do not affect the Z and CY flags.
(3) The bit test instruction resets the CMP flag of the program status word.
(4) Arithmetic and logical operations, bit test, compare, and rotation processing are modified by the index register if the IXE flag of the program status word is set to 1.

8.3 Arithmetic Operation (Binary 4-bit addition/subtraction and BCD addition/subtraction)

As shown in Table 8-3, the arithmetic operations are broadly classified into four types: addition, subtraction, addition with carry, and subtraction with borrow. These operations are performed by "ADD", "ADDC", "SUB", and "SUBC" instructions, respectively.

These instructions are also classified into addition or subtraction between a general register and data memory, and that between the data memory and immediate data. Whether the operation is executed between a general register and data memory, or between the data memory and immediate data is determined by the value written as the operand of the instruction. If the operand is "r, m", addition or subtraction is executed between a general register and the data memory; if the operand is "m, #n4", the operation is between the data memory and immediate data.

The arithmetic operation instruction is affected by the status flip-flops, that is, the program status word (PSWORD) of the system registers. The BCD flag of the program status word specifies whether the operation is executed in binary and 4-bit units or in BCD, and the CMP flag specifies that the result of the operation is not stored anywhere.

8.3.1 through **8.3.4** describe the relations between each arithmetic operation instruction and the program status word.

★ **Table 8-3. Arithmetic Operation Instructions**

Arithmetic operation	Add	Without carry	General register and data memory	ADD r, m
		ADD	Data memory and immediate data	ADD m, #n4
		Add w/carry	General register and data memory	ADDC r, m
		ADDC	Data memory and immediate data	ADDC m, #n4
	Subtract	Without borrow	General register and data memory	SUB r, m
		SUB	Data memory and immediate data	SUB m, #n4
		Subtract w/borrow	General register and data memory	SUBC r, m
		SUBC	Data memory and immediate data	SUBC m, #n4

8.3.1 Addition/subtraction when CMP = 0, BCD = 0

Addition or subtraction is executed in binary and 4-bit units, and the result is stored in a specified general register or data memory address.

The CY flag is set to 1 if the result of the operation exceeds 1111B (if a carry occurs) or is less than 0000B (a borrow occurs); otherwise, it is reset to 0.

If the result of the operation is 0000B, the Z flag is set to 1, regardless of whether a carry or borrow occurs; if the result is other than 0000B, the Z flag is reset to 0.

Examples 1.

```
MOV    R1, #1111B     ; Transfers 1111B to general register R1
MOV    M1, #0001B     ; Transfers 0001B to data memory M1
ADD    R1, M1         ; Adds R1 to M1
```

At this time, R1 + M1 is calculated as follows:

```
  1111B ..... Contents of R1
+ 0001B ..... Contents of M1
-----------
1 0000B
Carry
```

Therefore, the addition result, 0000B, is written to R1, and the CY flag is set to 1. The M1 contents do not change.
In addition, the Z flag is set to 1, because the result is 0000B.
If the carry is not output, when the R1 and M1 contents are added, the CY flag is reset to 0.

2.

```
MOV    M1, #1010B     ; Transfers 1010B to data memory M1
ADD    M1, #0101B     ; Adds immediate data 0101B to M1
```

At this time, M + 0101B is calculated as follows:

```
  1010B ..... M1 contents
+ 0101B ..... Immediate data
-----------
0 1111B
Carry
```

Therefore, 1111B is written to M1, and the CY and Z flags are reset.

3.

```
  MOV    R1, #1000B     ; Writes 1000B to general register R1
  MOV    M1, #1111B     ; Writes 1111B to data memory M1
; <1>
  ADD    M1, #0001B     ; Adds immediate data 0001B to M1
; <2>
  ADDC   R1, M1         ; Adds R1 to M1 with carry
```

In <1> above, the calculation is executed as follows:

```
    1111B ..... M1 contents
  + 0001B ..... Immediate data
  ------------
  1 0000B
  ~~
  Carry
```

Therefore, 0000B is written to R1 and the CY and Z flags are set to 1. In <2> above, the calculation is executed as follows:

```
    1000B ..... R1 contents
    0000B ..... M1 contents
  +      1 ..... CY flag contents
  ------------
  0 1001B
  ~~
  Carry
```

When the ADDC instruction is executed, therefore, the addition is executed, including the CY flag content at that time, and the CY flag is rewritten by the resultant carry output.

4.

```
MOV    R1, #0000B     ; Writes 0000B to general register R1
MOV    M1, #1000B     ; Writes 1000B to data memory M1
SUB    R1, M1         ; Subtracts M1 from R1
```

At this time, R1 – M1 is calculated as follows:

```
    0000B ..... R1 contents
  – 1000B ..... M1 contents
  ------------
  1 1000B
  ~~
  Borrow
```

Therefore, the result, 1000B, is written to R1. At this time, the CY flag is set to 1, because a borrow has occurred.
The carry, that occurs as a result of executing an addition instruction, and the borrow, that occurs as a result of executing a subtraction instruction, are governed by the same CY flag.

5.

```
 MOV    R1, #0000B    ;
 MOV    M1, #0000B    ;
; <1>
 SUB    M1, #0001B    ;
; <2>
 SUBC   R1, M1        ;
```

At this time, <1> and <2> are calculated as follows:

<1>

```
    0000B ..... M1 contents
  – 0001B ..... Immediate data
  ------------
  1 1111B
  ^^
  Borrow
```

<2>

```
    0000B ..... R1 contents
    1111B ..... M1 contents
  –      1 ..... CY flag contents
  ------------
  1 0000B
  ^^
  Borrow
```

Therefore, the results are R1 = 0000B, M1 = 1111B, CY flag = 1, and Z flag = 1.

8.3.2 Addition/subtraction when CMP = 1, BCD = 0

Addition or subtraction is executed in binary and 4-bit units.

However, the result of the operation is not stored in a general register or data memory address because the CMP flag is set to 1.

If a carry or borrow occurs as a result of the operation, the CY flag is set to 1; otherwise, the flag is reset to 0.

The Z flag retains the previous status if the result of the operation is 0000B; otherwise, it is reset to 0.

Examples 1.

```
 MOV    PSW, #1000B  ; Sets CMP flag (writes to program status word)
 MOV    R1, #1111B   ;
 MOV    M1, #1111B   ;
; <1>
 ADD    R1, M1       ;
; <2>
 SUB    R1, M1       ;
 MOV    PSW, #1010B  ; Sets CMP and Z flags
; <3>
 SUB    R1, M1       ;
```

At this time, <1> is calculated as follows:

```
   1111B ..... R1 contents
+  1111B ..... M1 contents
-------
1 1110B
^^
Carry
```

The operation result is not stored in R1, because the CMP flag is set to 1.
The CY flag is set to 1, because a carry occurs.
The Z flag is reset, because the result is not 0000B.
In <2>, the CY flag is reset to 0, because the R1 and M1 contents are the same as <1>. The Z flag retains the current status, 0, though the result is 0000B.
In <3>, the calculation is executed in the same manner as in <2>, but the Z flag retains 1, because it has been set to 1 in advance.
If the CMP flag is set to 1, the operation result is not stored, and only the statuses for CY and Z flags change. This is convenient for comparing data, which is 5 bits or longer.

2.

```
 MOV    PSW, #1010B       ; Sets CMP and Z flags to 1
; <1>
 SUB    M1, #0001B (1H)   ;
; <2>
 SUBC   M2, #0010B (2H)   ;
; <3>
 SUBC   M3, #0011B (3H)   ;
```

At this time, the operation result is not stored because the CMP flag is set to 1. Therefore, the M1, M2, and M3 contents remain unchanged, even if <1>, <2>, and <3> have been executed.
In addition, because the Z flag is set to 1 at first, the Z flag remains set to 1, if all the <1>, <2>, and <3> results are 0000B. The Z flag is reset to 0, if even one of the results is not 0000B.
The CY flag is set if the 12-bit contents for M3, M2, and M1 are less than 001100100001B (321H).
Consequently, by testing the Z and CY flags after <1>, <2> and <3> have been executed, the 12-bit data for M3, M2, and M1 can be compared with the 12-bit data for 321H, as follows:

```
If Z = 1, CY = 0; M3, M2, M1 = 321H
                  ↑
              Always 0
If Z = 0, CY = 0; M3, M2, M1 > 321H
If Z = 0, CY = 1; M3, M2, M1 < 321H
```

It is also possible to compare the general register contents with those for data memory area, by using the SUB r, m and SUBC r, m instructions in Example 2.

8.3.3 Addition/subtraction when CMP = 0, BCD = 1

A BCD operation is executed.

The result of the operation is stored in a specified general register or data memory address. The CY flag is set to 1 if the result exceeds 1001B (9D) or is less than 0000B (0D), and is reset to 0 if the result is in the range of 0000B (0D) to 1001B (9D).

The Z flag is set to 1 if the result is 0000B (0D); otherwise, it is reset to 0.

The BCD operation is executed by converting the result of an operation executed in binary into decimal number by using the decimal correction circuit. For details on this binary-to-decimal conversion, refer to **Table 8-2 Results for Binary 4-bit and BCD Operations**.

To execute a BCD operation correctly, therefore, keep in mind the following points:

(1) The result of addition must be 0D to 19D.

(2) The result of subtraction must be 0D to 9D or –10 to –1D.

The value range of 0D to 19D is determined by giving consideration to the CY flag, and is in binary:

0,0000B to 1,0011B
CY CY

Likewise, the range of –10D to –1D is:

1,0110B to 1,1111B
CY CY

If a BCD operation is executed without the above conditions (1) and (2) satisfied, the CY flag is set to 1, and data greater than 1010B (0AH) is output as a result.

Examples 1.

```
        MOV     M1, #0111B (7)      ;
        MOV     RPL, #0001B         ; Sets BCD flag (BCD flag is assigned to b0 in RPL for
                                    ; system register)
        MOV     PSW, #0000B         ; Resets CMP, CY, and Z flags
; <1>
        ADD     M1, #1001B (9)      ; 7 + 9
; <2>
        SUB     M1, #0111B (7)      ; 6 – 7
```

At this time, <1> is calculated as follows:

```
  0111B ..... M1 contents
+ 1001B ..... Immediate data
-------
1 0000B ..... Binary addition result
Carry
      ↓ Converted by binary-to-decimal adjustment in Table 7-2
1 0110B ..... Data stored in M1
Carry
```

Therefore, the CY flag is set and 0110B (6) is stored in M1. Assuming that the CY flag significance is 10, this means that a decimal operation of 7 + 9 = 16 has been executed.
In <2>, the calculation is executed as follows:

```
  0110B ..... M1 contents
- 0111B ..... Immediate data
-------
1 1111B ..... Binary subtraction result
Borrow
      ↓ Binary-to-decimal adjustment
1 1001B ..... Data stored in M1
```

Since 6 is stored in M1 in <1>, 6-7 has been performed with the result of 9. Therefore, the CY flag is set.

2.

```
 MOV   M1, #0101B (5)    ;
 MOV   M2, #0110B (6)    ;
 MOV   M3, #0111B (7)    ;
 MOV   RPL, #0001B       ; Sets BCD flag to 1
 MOV   PSW, #0000B       ; Resets CMP, CY, and Z flags to 0
; <1>
 SUB   M1, #0111B (7)    ;
; <2>
 SUBC  M2, #0110B (6)    ;
; <3>
 SUBC  M3, #0101B (5)    ;
```

At this time, the calculation is carried out as follows in <1>, <2>, and <3>.

<1>

```
   0101B ..... M1 contents
 – 0111B ..... Immediate data
 ---------
 1 1110B
 Borrow
      ↓ Binary-to-decimal adjustment
 1 1000B (8) ..... Data stored in M1
 Borrow
```

<2>

```
   0110B ..... M2 contents
 – 0110B ..... Immediate data
 ---------
 1 1111B ..... CY flag
 Borrow
      ↓ Binary-to-decimal adjustment
 1 1001B (9) ..... Data stored in M2
 Borrow
```

<3>

```
   0111B ..... M3 contents
 – 0101B ..... Immediate data
 ---------
 0 0001B ..... CY flag
 Borrow
      ↓ Binary-to-decimal adjustment
 0 0001B (1) ..... Data stored in M3
```

Therefore, immediate data 567 is subtracted from 765 stored in M3, M2, and M1, and the result is 198.

3.

```
 MOV   M1, #1001B      ;
 MOV   RPL, #0001B     ; Sets BCD flag to 1
 MOV   PSW, #0000B     ; Resets CMP, CY, and Z flags to 0
; <1>
 ADD   M1, #1010B      ;
; <2>
 ADDC  M1, #1010B      ;
```

At this time, <1> is calculated as follows:

```
  1001B (9) ..... M1 contents
+ 1010B (10) ..... Immediate data
-----------
1 0011B ..... CY flag
Carry
      ↓ Binary-to-decimal adjustment
1 1001B ..... Result
Carry
```

Therefore, 9 + 10 = 9 is executed. If the CY flag is taken into consideration, a decimal operation of 9 + 10 = 19 has been performed.
However, in <2>, the calculation is carried out as follows:

```
  1001B (9) ..... M1 contents
+ 1010B (10) ..... Immediate data
-----------
1 0100B ..... CY flag
Carry
      ↓ Binary-to-decimal adjustment
1 1110B ..... Result
Carry
```

The operation result exceeds 19, because the CY flag is set to 1, and accurate decimal operation cannot be performed.

8.3.4 Addition/subtraction when CMP = 1, BCD = 1

A BCD operation is performed.

The result of the operation is not stored in a general register or data memory address.

Therefore, the operation to be performed when the CMP flag is 1 and that performed when the BCD flag is 1 are performed at the same time.

Examples 1.

```
MOV    RPL,    #0001B ; Sets BCD flag to (1)
MOV    PSW,    #1010B ; Sets CMP and Z flags to 1 and resets CY flag to (0)
SUB    M1,     #0001B ; <1>
SUBC   M2,     #0010B ; <2>
SUBC   M3,     #0011B ; <3>
```

At this time, the contents of the 12 bits of M3, M2, and M1 can be compared with immediate data 321 in decimal number.

2.

```
 MOV    RPL,    #0001B ; Sets BCD flag to 1
 MOV    PSW,    #1010B ; Sets CMP and Z flags to 1, and resets CY flag to 0
; <1>
 SUB    M1,     #0001B ;
; <2>
 SUBC   M2,     #0010B ;
; <3>
 SUBC   M3,     #0011B ;
```

At this time, 12-bit contents in M3, M2, and M1 can be compared with immediate data 321 in decimal number by <1>, <2>, and <3>.

8.3.5 Notes on using arithmetic operation instruction

When an arithmetic operation is executed to the program status word (PSWORD), note that the result of the operation is stored in the program status word.

The CY and Z flags of the program status word are usually set or reset as a result of an arithmetic operation. If an arithmetic operation is executed to the program status word, however, the result is stored to the program status word, making it impossible to test occurrence of a carry or borrow, or whether the result is zero.

When the CMP flag is set to 1, however, the result is not stored in the program status word, and the CY and Z flags are set or reset as usual.

Examples 1.

```
MOV    PSW, #0110B
MOV    PSW, #1010B
```

At this time, the calculation is carried out as follows:

```
  0110B ..... PSW contents
+ 1010B ..... Immediate data
-----------
1 0000B
~~
Carry
```

Although the CY and Z flags must be set, the result 0000B is stored in the PSW, because the CMP flag is 0.

2.

```
MOV    PSW, #1010B
ADD    PSW, #1000B
```

At this time, the calculation is carried out as follows:

```
  1010B ..... PSW contents
+ 1000B ..... Immediate data
-----------
1 0010B
~~
Carry
```

Because the CMP flag is set to 1, the result 0010B is not stored in the PSW. Consequently, the CY flag is set to 1 and the Z flag is reset to 0, and 1100B is stored in the PSW.

8.4 Logical Operation

As logical operations, logical sum (OR), logical product (AND), and exclusive logical OR (XOR) can be executed as shown in Table 8-4.

The logical operations are classified into these three types and are implemented by the "OR", "AND", and "XOR" instructions.

These instructions are also classified into an operation executed between a general register and data memory, and that between the data memory and immediate data. Whether the operation is executed between a general register and data memory, or between the data memory and immediate data is determined depending on the value written as the operand of the instruction, i.e., whether "r, m" or "m, #n4" is described as the operand, like the arithmetic operation instruction.

The logical operation is not affected by the BCD and CMP flags of the program status word (PSWORD). It does not affect the CY and Z flags. However, the operation is subject to modification by the index register if the index enable flag (IXE) is set to 1.

★ **Table 8-4. Logical Operation Instructions**

Logical operation	Logical sum OR	General register and data memory OR r, m
		Data memory and immediate data OR m, #n4
	Logical product AND	General register and data memory AND r, m
		Data memory and immediate data AND m, #n4
	Exclusive Logical product XOR	General register and data memory XOR r, m
		Data memory and immediate data XOR m, #n4

Table 8-5. Logical Operation Truth Table

Logical product C = A AND B			Logical sum C = A OR B			Exclusive logical sum C = A XOR B		
A	B	C	A	B	C	A	B	C
0	0	0	0	0	0	0	0	0
0	1	0	0	1	1	0	1	1
1	0	0	1	0	1	1	0	1
1	1	1	1	1	1	1	1	0

8.4.1 Logical sum (Logical OR)

The logical sum instruction ORs 4-bit data, according to the truth table shown in Table 8-5.

Example

```
    MOV     R1, #1010B      ;
    MOV     M1, #1001B      ;
; <1>
    OR      R1, M1  ;
; <2>
    OR      M1, #1100B      ;
```

At this time, <1> is calculated as follows:

```
    1010B ..... R1 contents
OR  1001B ..... M1 contents
-----------
    1011B ..... Result
```

Therefore, 1011B is stored in R1.
In <2>, the calculation is executed as follows:

```
    1001B ..... M1 contents
OR  1100B ..... Immediate data
-----------
    1101B
```

Therefore, 1101B is stored in M1.

The logical sum instruction is convenient for setting the contents of a data memory area to 1 in 1, 2, 3, or 4 bit units.

8.4.2 Logical product (Logical AND)

The logical product instruction ANDs 4-bit data, according to the truth table shown in Table 8-5.

Example

```
 MOV    R1, #1010B    ;
 MOV    M1, #1001B    ;
; <1>
 AND    R1, M1  ;
; <2>
 AND    M1, #1100B    ;
```

At this time, <1> is calculated as follows:

```
    1010B..... R1 contents
AND 1001B..... M1 contents
-----------
    1000B.....
```

Therefore, 1000B is stored in R1.
In <2>, the calculation is executed as follows:

```
      1001B..... M1 contents
AND   1100B..... Immediate data
-------------------------------
      1000B
```

Therefore, 1000B is stored in M1.

The logical product instruction is convenient for resetting the data memory area contents to 0 in 1, 2, 3, or 4 bit units.

8.4.3 Logical exclusive sum (Logical exclusive OR)

The logical exclusive sum instruction exclusive-ORs 4-bit data, according to the truth table shown in Table 8-5.

Example

```
  MOV    R1, #1010B    ;
  MOV    M1, #1001B    ;
; <1>
  XOR    R1, M1  ;
; <2>
  XOR    M1, #1100B    ;
```

At this time, <1> is calculated as follows:

```
      1010B..... R1 contents
XOR   1001B..... M1 contents
-------------------------------
      0011B
```

Therefore, 0011B is stored in R1.
In <2>, the calculation is executed as follows:

```
      1001B..... M1 contents
XOR   1100B..... Immediate data
-------------------------------
      0101B
```

Therefore, 0101B is stored in M1.

The exclusive logical sum instruction is convenient for inverting data memory area contents in 1, 2, 3, or 4 units

8.5 Bit Testing

As shown in Table 8-6, bit testing can be classified into True bit (1) testing and False bit (0) testing.

These testings are made respectively by the "SKT" and "SKF" instructions.

These instructions can be executed to only the data memory.

Bit testing is not affected by the BCD flag of the program status word (PSWORD). It does not affect the CY and Z flags. However, the CMP flag is reset to 0 when the "SKT" or "SKF" instruction is executed. Modification is made by the index register if the instruction is executed while the index enable flag (IXE) is set to 1. For details on modification by the index register, refer to **CHAPTER 6 SYSTEM REGISTER (SYSREG)**.

8.5.1 and **8.5.2** describe True bit (1) testing and False bit (0) testing, respectively.

★ **Table 8-6. Bit Test Instructions**

Bit testing	True bit (1) testing	SKT m, #n
	False bit (0) testing	SKF m, #n

8.5.1 True bit (1) testing

The True bit (1) test instruction, "SKT m, #n", tests whether bit(s) specified by n of the 4 bits of a data memory address is "True (1)". If all the bits specified by n is "True (1)", the next instruction is skipped.

Example

```
MOV   M1,   #1011B
SKT   M1,   #1011B ; <1>
BR    A
BR    B
SKT   M1,   #1101B ; <2>
BR    C
BR    D
```

In <1>, execution branches to B because all the bits 3, 1, and 0 of M1 are True (1).
In <2>, the bits 3, 2, and 0 of M1 are tested, and execution branches to C because bit 2 is False (0).

8.5.2 False bit (0) testing

The False bit (0) test instruction, "SKF m, #n", tests whether bit(s) specified by n of the 4 bits of a data memory address is "False (0)". If all the bits specified by n is "False (0)", the next instruction is skipped.

Example

```
MOV   M1,   #1001B
SKF   M1,   #0110B ; <1>
BR    A             ;
BR    B             ;
SKF   M1,   #1110B ; <2>
BR    C             ;
BR    D             ;
```

In <1>, execution branches to B because both the bits 2 and 1 of M1 are False (0).
In <2>, the bits 3, 2, and 0 of M1 are tested, and execution branches to C because bit 3 of M1 is True (1).

8.6 Compare

As shown in Table 8-7, the compare operations are divided into four types: "equal to", "not equal to", "greater than", and "less than".

To make these comparisons, the "SKE", "SKNE", "SKGE", and "SKLT" instructions are used.

These instructions can be used only to compare the contents of a data memory address with immediate data. To compare the contents of a general register and those of a data memory address, use a subtraction instruction with the CMP and Z flags of the program status word (PSWORD) (refer to **8.3 Arithmetic Operation (Binary 4-bit addition/subtraction and BCD addition/subtraction)**).

Comparison is not affected by the BCD and CMP flags of the program status word. It does not affect the CY and Z flags.

When the index enable flag (IXE flag) is set to 1, modification is performed by the index register. For modification by the index register, refer to **CHAPTER 6 SYSTEM REGISTER (SYSREG)**.

8.6.1 through **8.6.4** describe comparison of "equal to", "not equal to", "greater than", and "less than", respectively.

★ **Table 8-7. Compare Instructions**

Compare	Equal to SKE m, #n4
	Not equal to SKNE m, #n4
	Greater than SKGE m, #n4
	Less than SKLT m, #n4

8.6.1 Comparison of "Equal to"

The "SKE m, #n4" instruction tests whether the contents of a specified data memory address are "equal to" specified immediate data.

If the data memory contents are "equal to" the immediate data, the instruction next to this instruction is skipped.

Example

```
MOV   M1,   #1010B
SKE   M1,   #1010B ; <1>
BR    A
BR    B
 ;
SKE   M1,   #1000B ; <2>
BR    C
BR    D
```

In <1>, execution branches to B because the contents of M1 are equal to immediate data 1010B. In <2>, however, execution branches to C because the contents of M1 are not equal to immediate data 1000B.

8.6.2 Comparison of "Not equal to"

The "SKNE m, #n4" instruction tests whether the contents of a specified data memory address are "not equal to" specified immediate data.

If the data memory contents are "not equal to" the immediate data, the instruction next to this instruction is skipped.

Example

```
MOV   M1,   #1010B
SKNE  M1,   #1000B ; <1>
BR    A
BR    B
 ;
SKNE  M1,   #1010B ; <2>
BR    C
BR    D
```

In <1>, execution branches to B because the contents of M1 are not equal to immediate data 1000B. In <2>, however, execution branches to C because the contents of M1 are equal to immediate data 1010B.

8.6.3 Comparison of "Greater than"

The "SKGE m, #n4" instruction tests whether the contents of a specified data memory address are "greater than" specified immediate data.

If the data memory contents are "greater than" or "equal to" the immediate data, the instruction next to this instruction is skipped.

Example

```
MOV   M1,   #1000B
SKGE  M1,   #0111B ; <1>
BR    A
BR    B
 ;
SKGE  M1,   #1000B ; <2>
BR    C
BR    D
 ;
SKGE  M1,   #1001B ; <3>
BR    E
BR    F
```

Because the contents of M1 are 1000B, <1> is judged to be "Greater than", <2>, "Equal to", and <3>, "Less than", and execution branches to B, D, and E, respectively.

8.6.4 Comparison of "Less than"

The "SKLT m, #n4" instruction tests whether the contents of a specified data memory are "less than" specified immediate data.

If the data memory contents are "less than" the immediate data, the instruction next to this instruction is skipped.

Example

```
MOV   M1,   #1000B
SKLT  M1,   #1001B ; <1>
BR    A
BR    B
 ;
SKLT  M1,   #1000B ; <2>
BR    C
BR    D
 ;
SKLT  M1,   #0111B ; <3>
BR    E
BR    F
```

Because the contents of M1 are 1000B, <1> is judged to be "Less than", <2>, "Equal to", and <3>, "Greater than", and execution branches to B, C, and E, respectively.

8.7 Rotation Processing

Rotation processing can be classified into right rotation and left rotation.

To execute the right rotation processing, the "RORC" instruction is used.

This instruction can be executed only to a general register.

The rotation processing by the "RORC" instruction is not affected by the BCD and CMP flags of the program status word (PSWORD). It does not affect the Z flag.

The "RORC" instruction does not modify (increment/decrement) addresses by using the index register (IX) even if the index enable flag (IXE flag) is set to 1.

8.7.1 and **8.7.2** below describe the respective rotation processing.

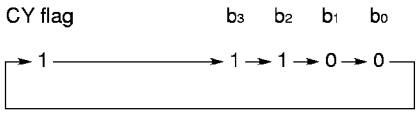
8.7.1 Right rotation processing

The right rotation processing instruction "RORC r" rotates the contents of a specified general register 1 bit toward the least significant bit direction.

At this time, the content of the CY flag is written to the most significant bit (bit 3) position of the general register, and the content of the least significant bit (bit 0) is written to the CY flag.

Examples 1.
```
MOV   PSW,   #0100B ; Sets CY flag to 1
MOV   R1,    #1001B
RORC  R1
```

At this time, the processing is performed as follows:



Therefore, right rotation is executed from the CY flag as shown above.

Examples 2.
```
MOV   PSW,   #0000B ; Resets CY flag to 0
MOV   R1,    #1000B ; MSB
MOV   R2,    #0100B
MOV   R3,    #0010B ; LSB
RORC  R1
RORC  R2
RORC  R3
```

The above program rotates the 13-bit data of R1, R2, and R3 to the right.

8.7.2 Left rotation processing

The left rotation processing can be performed by using the addition instruction "ADDC r, m" as follows:

```
Example  MOV   PSW,   #0000B ; Resets CY flag to 0
         MOV   R1,    #1000B ; MSB
         MOV   R2,    #0100B
         MOV   R3,    #0010B ; LSB
         ADDC  R3, R3
         ADDC  R2, R2
         ADDC  R1, R1
         SKF   CY
         OR    R3,    #0001B
```

The above program rotates the 13-bit data of R1, R2, and R3 to the left.

CHAPTER 9 REGISTER FILE (RF)

The register file is a register area that can be manipulated by the "PEEK" and "POKE" instructions.
The register file mainly sets the hardware conditions peripheral.

9.1 Register File Configuration

The register file consists of a control register and a data memory area, as shown in Figure 9-1. Addresses 40H through 7FH overlap with a data memory area. Therefore, these register file addresses are addresses 40H through 7FH in the bank currently selected for the data memory.

Therefore, if BANK0 is currently selected, register file addresses 40H through 7FH are for BANK0. These addresses can be manipulated as both data memory addresses and register file addresses.

Addresses 00H through 3FH in the register file form a control register area that sets various conditions for the hardware peripherals.

These areas constitute a 128-nibble (128 words × 4 bits) register file, as shown in Figure 9-2.

Figure 9-1. Relations between Register File and Data Memory

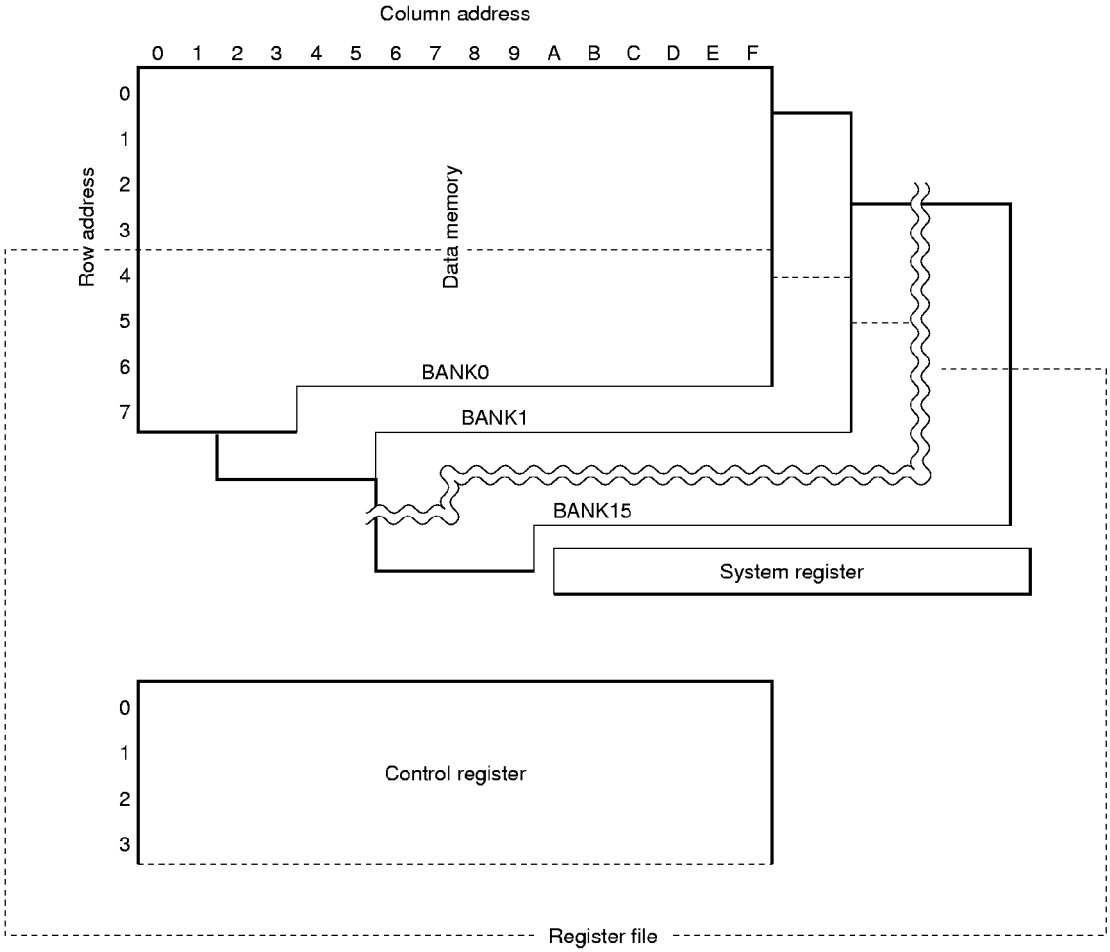
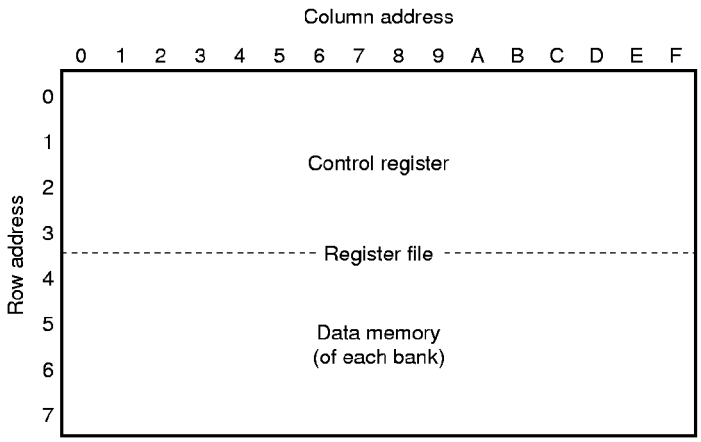


Figure 9-2. Configuration of Register File



9.2 Register File Functions

9.2.1 Register file functions

The register file control registers mainly set the peripheral hardware conditions.

The rest of the register file (addresses 40H through 7FH) is overlapped with the data memory. Therefore, it can be operated in the same manner as the data memory, except that they can be manipulated by the "PEEK" and "POKE" instructions.

9.2.2 Register file manipulation instruction

Data is written to or read from the register file via the window register of the system registers (WR: address 78H). To write or read data, the following dedicated instructions are used:

PEEK WR, rf: Reads data of register file addressed by rf to WR
POKE rf, WR: Writes data of WR to register file addressed by rf

★ **Example**

```
        M030    MEM 0.30H   ; Uses address 30H of data memory as WR saving area
        M032    MEM 0.32H   ; Uses address 32H of data memory as WR manipulation area
        RF11    MEM 0.91H   ; Symbol definition
        RF33    MEM 0.B3H   ; Symbols at addresses 00H-3FH of register file must be defined as
        RF70    MEM 0.70H   ; 80H-BFH of BANK0. For details, refer to 9.4 Notes on Using Register
        RF73    MEM 0.73H   ; File
        BANK0
<1>     PEEK    WR, RF11    ;

        CLR1    MPE         ; Indicates example for saving contents of WR to general-purpose data
        CLR1    IXE         ; memory (addresses 00H-3FH). As example, saving data to data memory
        OR      RPL, #0110B ; address 30H without address modification is indicated.
<2>     LD      M030, WR    ;

<3>     POKE    RF73, WR    ; Data can be directly transferred between data memory at addresses
<4>     PEEK    WR, RF70    ; 40H-7FH and control register by WR, PEEK, and POKE instructions
<5>     POKE    RF33, WR    ;

<6>     ST      WR, M032    ;
```

Figure 9-3 shows an example of operation.

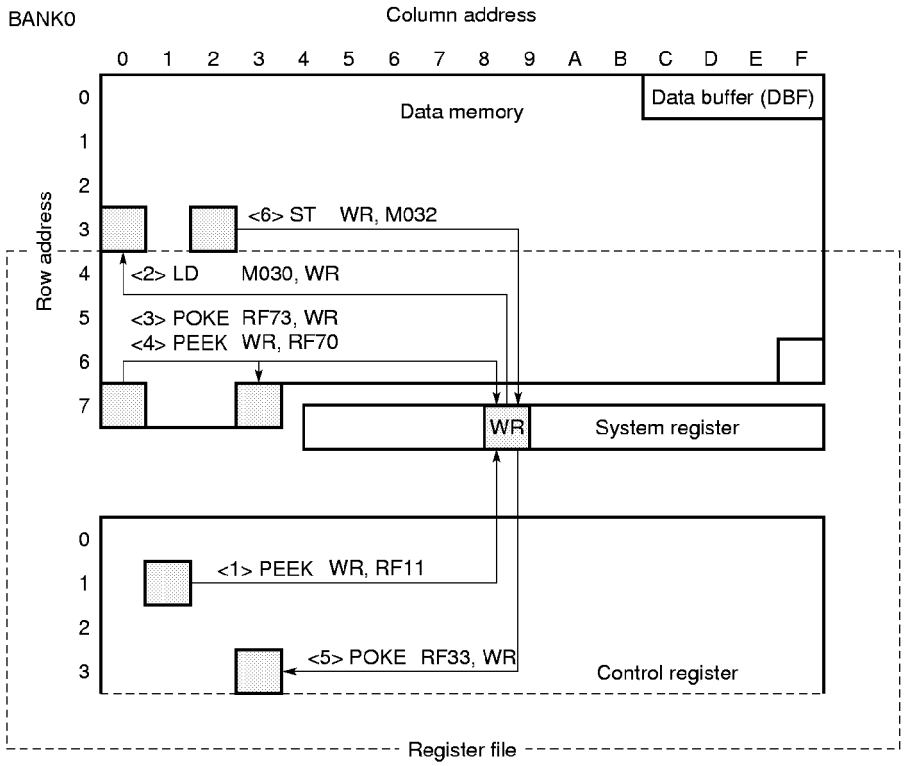
As shown in this figure, the control register (addresses 00H-3FH) reads or writes the contents of the register file addressed by "rf" from or to the window register when the "PEEK WR, rf" or "POKE rf, WR" instruction is executed.

Since addresses 40H through 7FH of the register file overlap the data memory, the "PEEK WR, rf" or "POKE rf, WR" instruction is executed to data memory address "rf" in the bank selected at that time.

Addresses 40H through 7FH of the register file can also be manipulated by a memory manipulation instruction.

The control register can be manipulated in 1-bit units by using a macro instruction (refer to **9.4.2 Symbol definition of register file and reserved word**).

★ **Figure 9-3. Accessing Example of Register File with PEEK or POKE Instruction**



9.3 Control Register

9.3.1 Control register configuration

The control register sets the conditions for hardware peripherals.

The control register consists of 64 words × 4 bits at addresses 00H through 3FH in the register file.

Of these control register words, those actually used differ, depending on the microcontroller model.

Each control register has 1 nibble of attribute and may be read/write (R/W), read-only (R), write-only (W), or reset, ★ when read (R & Reset). Note, however that some of the read/write (R/W) flags are always "0" when they are read.

Nothing is changed when data is written to the read-only register (R or R & Reset).

An "undefined value" is read when the write-only register (W) is read.

Of the 4-bit data for 1 nibble, the bit fixed to "0" is always "0" when read, and retains "0", even when it is written.

An undefined value is read when the unused register is read, and nothing is changed when data is written to this register.

To manipulate the unused register, write-only register (W), and read-only register (R), care must be exercised in ★ using the Assembler (RA17K). For details, refer to **9.4 Notes on Using Register File**.

9.3.2 Hardware peripheral control functions for control register

The control functions, register to control the hardware peripherals, are described in the Data Sheet for each model.

9.4 Notes on Using Register File

9.4.1 Notes on manipulating control registers (read-only and unused registers)

When you manipulate the read-only (R) and unused registers of the control registers (addresses 00H through 3FH of the register file), you must pay attention when the device operates, as described in (1), (2), and (3) below when ★ you use the 17K Series assembler (RA17K) and the in-circuit emulators (IE-17K, IE-17K-ET).

(1) When device operates

Nothing is changed even when data is written to a read-only register.

If an unused register is read, an "undefined value" is read. Nothing is changed even when data is written to this register.

★ **(2) When using assembler (RA17K)**

An "error" occurs when an instruction that writes data is executed to access a read-only register.

An "error" also occurs when an instruction that reads or writes data is executed to an unused register.

★ **(3) When using an 17K series in-circuit emulator (IE-17K or IE-17K-ET) (patch processing, etc.)**

An "error" does not occur even when data is written to a read-only register.

When an unused register is read, an "undefined value" is read, and nothing is changed even when data is written to this register, but an "error" does not occur.

9.4.2 Symbol definition of register file and reserved words

★ If a register file address is directly written in numeric value as operand "rf" of the "PEEK WR, rf" or "POKE rf, WR" instruction when the 17K series assembler (RA17K) is used, an "error" occurs.

It is therefore necessary to define the address of the register file as a symbol as shown in Example 1 below.

Examples 1. Error occurs

```
PEEK   WR, 02H      ;
POKE   21H, WR      ;
```

Error does not occur

```
RF71   MEM0.71H     ; Symbol definition
PEEK   WR, RF71     ;
```

At this time, pay attention to the following point:

- To define a control register as a symbol of data memory address type, it must be defined as the addresses 80H through BFH of BANK0.

This is because the control register is manipulated via the window register, and an error must occur when the control register is manipulated by an instruction other than "PEEK" and "POKE".

However, the register file (addresses 40H through 7FH) that overlap the data memory can be defined as a symbol without changing the address.

Here is an example:

★ **Examples 2.**

```
RF71   MEM1.71H     ; Register file overlapping data memory
RF02   MEM0.82H     ; Control register

PEEK   WR, RF71     ; RF71 is data memory at address "71H"
PEEK   WR, RF02     ; RF02 is control register at address 02H
```

★ When the assembler (RA17K) is used, the following macro instructions are included in the assembler as flag type symbol manipulation instructions:

SETn	: Sets flag to "1"
CLRn	: Clears flag to "0"
SKTn	: Skips if all flags are "1"
SKFn	: Skips if all flags are "0"
NOTn	: Inverts flag
INITFLG	: Initializes flag
★ INITFLGX	: Initializes flag

Therefore, by using these macro instructions, the contents of the register file can be manipulated in 1-bit units, as shown in the following Example 3.

Because many flags of the control registers are manipulated in 1-bit units, "reserved words" are defined on the ★ assembler (RA17K) as flag type symbols.

However, no flag type reserved word is available for the stack pointer. The reserved word for the stack pointer is defined as data memory type, "SP". Therefore, the flag manipulation instruction cannot be used with a reserved word.

★ **Examples 3.**
```
INITFLG  WDTRES      ; Initialize
(SET1    WDTRES      ; Sets flag)
```

Macro expansion
```
PEEK   WR, .MF.WDTRES SHR4
OR     WR, #.DF.WDTRES AND 0FH
POKE   .MF.WDTRES SHR4, WR
```

★ 9.4.3 Notes on using assembler (RA17K) macroinstructions

The following points (1) and (2) call for specific attention, when using the Assembler macroinstructions to access the control registers:

(1) Flag manipulation macroinstructions cannot be used to manipulate the stack pointer

As described in **9.4.2**, no flag type reserved word is defined for the stack pointer. Therefore, a flag manipulation instruction cannot be used with a reserved word.

(2) Flag manipulation macroinstructions cannot be used to manipulate write-only register

The flag manipulation macroinstruction cannot be used to manipulate the write-only register.

If the "SETn" macroinstruction is used to manipulate a write-only register, the register file contents are once read to the window register.

At this time, the value read to the window register becomes undefined (an undefined value is read from the write-only register), and an undefined value is written to a bit not specified by the "SETn" instruction.

★ At this time, the Assembler (RA17K) generates an error.

CHAPTER 10 DATA BUFFER (DBF)

The data buffer is used to transfer data with the hardware peripherals and to read data for table reference.

10.1 Data Buffer Configuration

As shown in Figure 10-1, the data buffer (DBF) is assigned to addresses 0CH through 0FH in BANK0 for the data memory, and consists of 4 bits × 4 words, or a total of 16 bits.

Since the data buffer is on the data memory, it can be manipulated by all the data memory manipulation instructions.

Figure 10-1. Data Buffer Location

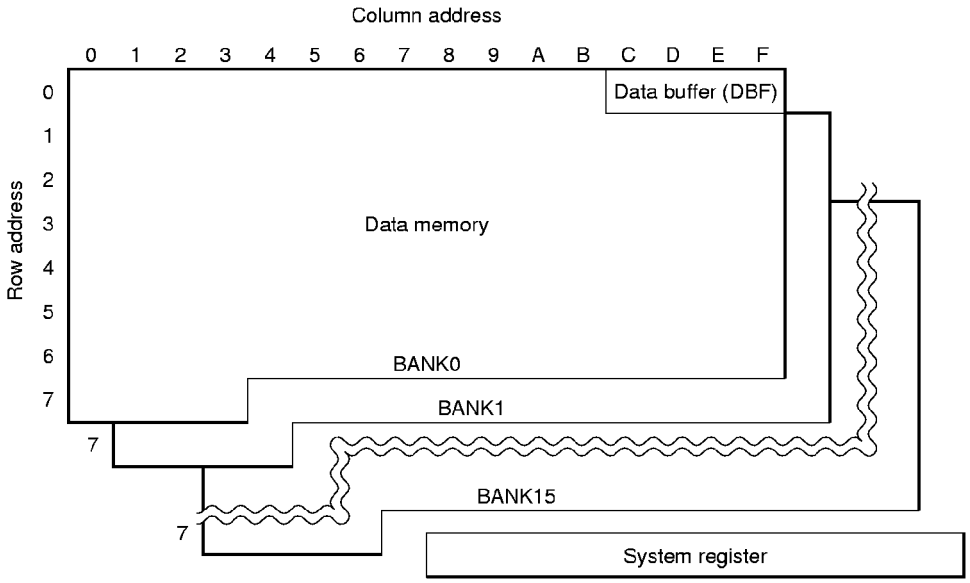


Figure 10-2 shows the data buffer configuration. As shown, the LSB for the data buffer is bit b_0 for address 0FH in the data memory, and the MSB is bit b_3 for address 0CH.

Figure 10-2. Configuration of Data Buffer

Data Memory	Address	0CH				0DH				0EH				0FH			
	Bit	b_3	b_2	b_1	b_0	b_3	b_2	b_1	b_0	b_3	b_2	b_1	b_0	b_3	b_2	b_1	b_0
Data buffer	Bit	b_{15}	b_{14}	b_{13}	b_{12}	b_{11}	b_{10}	b_9	b_8	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
	Symbol	DBF3				DBF2				DBF1				DBF0			
	Data	MSB															LSB
		Data															

10.2 Data Buffer Functions

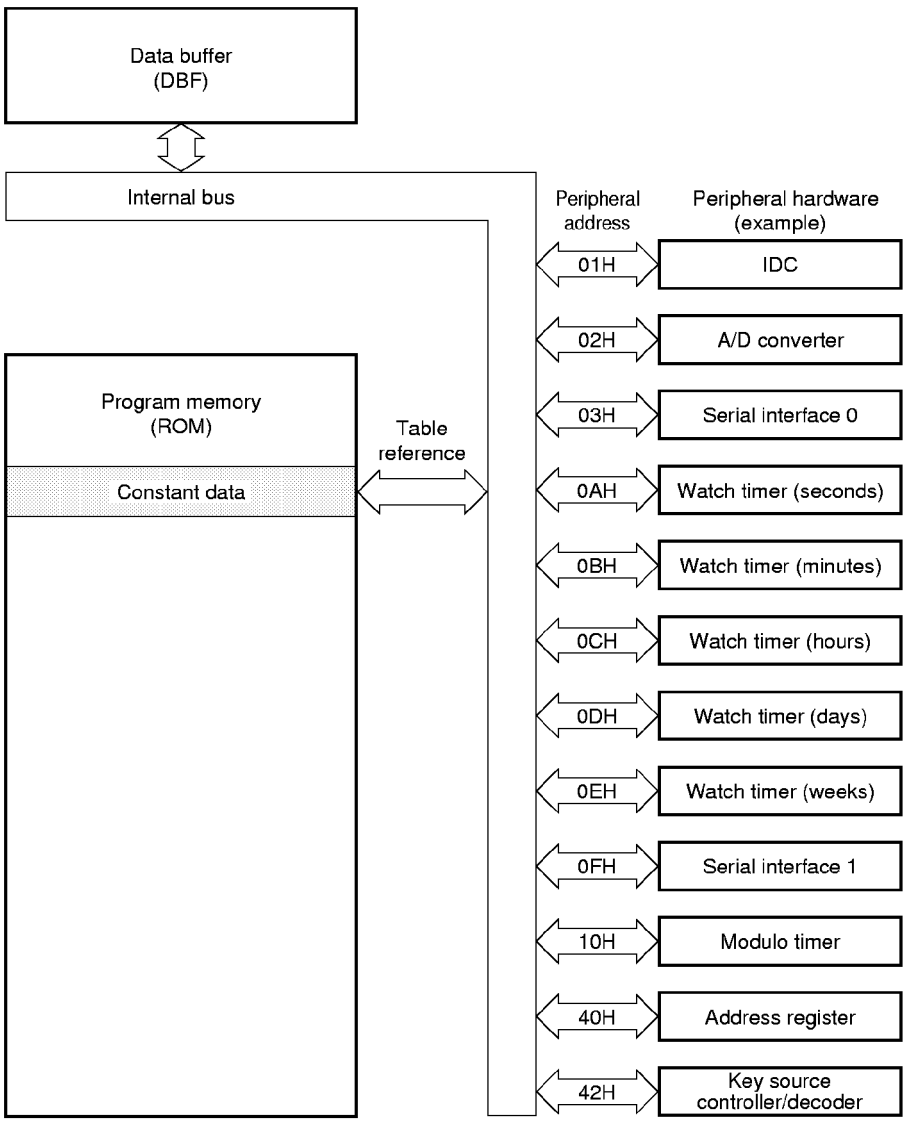
The data buffer has the following two functions:

(1) Reads constant data on the program memory (table reference)
(2) Transfers data with hardware peripherals

Figure 10-3 shows the relations between the data buffer, hardware peripherals, and table reference.

For details on table reference, refer to **10.4 Data Buffer and Table Reference**, and for relations with hardware peripherals, refer to **10.5 Data Buffer and Hardware Peripherals**.

Figure 10-3. Relations between Data Buffer, Hardware Peripherals and Table Reference (Example)



10.3 Notes on Using Data Buffer

10.3.1 When manipulating addresses for write-only and read-only registers and an unused address

When transferring data through the data buffer to the hardware peripherals, pay attention to the following points concerning the unused peripheral address, write-only peripheral register (PUT only), and read-only peripheral register (GET only):

(1) Device operation

An "undefined value" is read from the write-only register when it is read.

The read-only register contents are not changed, even when an attempt has been made to write data to this register.

When the unused register is read, an "undefined value" is read. The unused register contents are not changed, even when an attempt has been made to write data to this register.

★ **(2) When using Assembler (RA17K)**

An "error" occurs, when an instruction is executed to read the write-only register, to write the read-only register, or to read/write the unused register.

★ **(3) When using Emulator (IE-17K, IE-17K-ET) (manipulation for batch processing)**

When the write-only register is read, an "undefined value" is read, but no "error" occurs.

When the read-only register is written, the register contents are not changed and no "error" occurs.

When the unused register is read, an "undefined value" is read. When this register is written, its contents are not changed and no "error" occurs.

10.3.2 Specification of peripheral register address

★ When using the 17K series Assembler (RA7K), an "error" does not occur, if a peripheral address "p" is directly specified (in numeral) by the "PUT p, DBF" or "GET DBF, p" instruction, as shown in **Example 1** below.

Using this method, however, is not desirable, in order to reduce the number of bugs in the program.

It is therefore recommended to define a symbol for the peripheral device, as shown in **Example 2**, by using the symbol definition directive in the Assembler.

★ To simplify symbol definition, peripheral addresses are defined in advance in the Assembler (RA17K) as "reserved words".

By using the reserved words, therefore, the program can be created without defining symbols, as shown in **Example 3**.

For the reserved words, refer to the Data Sheet for each model.

Examples 1.
```
PUT   02H, DBF      ; Error does not occur, even when peripheral address 02H or 03H is
GET   DBF, 03H      ; specified. Using this is not desirable, in order to reduce bugs
```

2.
```
SIO0DATA DAT 03H    ; Assigns 03H to SIO0DATA by symbol definition directive
PUT   SIO0DATA, DBF ;
```

3.
```
PUT   SIO0SFR, DBF  ; Symbol need not be defined, if reserved word "SIO0SFR" is used
```

10.4 Data Buffer and Table Reference

10.4.1 Table reference operation

By using the "MOVT DBF, @AR" instruction, constant data on the program memory can be read to the data buffer.

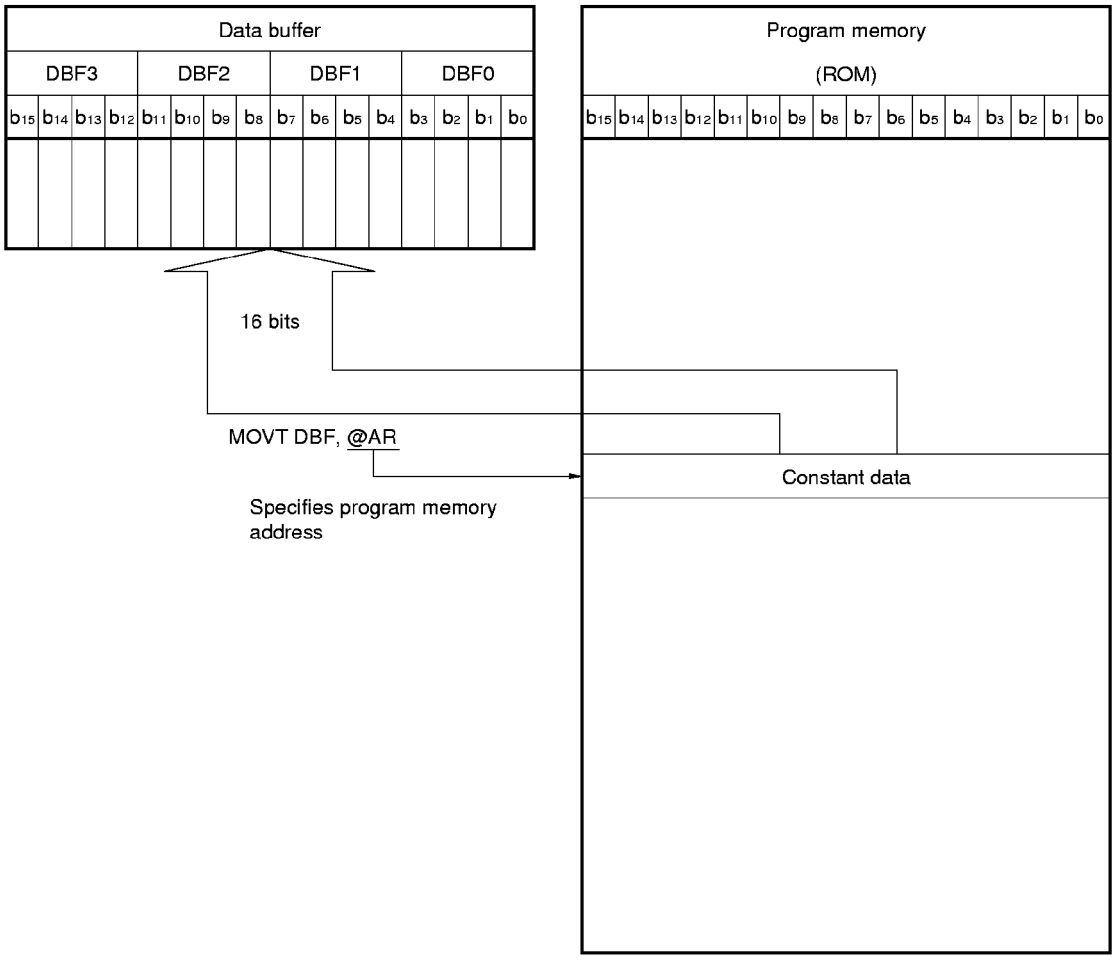
Therefore, by writing, for example, display data and constant data to the program memory in advance and performing table reference as necessary, the need for creating a complicated data conversion program is eliminated.

The MOVT instruction function is as illustrated below.

Example

```
MOVT    DBF, @AR      ; Reads the program memory contents specified by the address
                        register contents to the data buffer, as shown in Figure 10-4
```

Figure 10-4. Example of Table Reference



When the table reference instruction is executed, one stack level is used.

The program memory address, to which table reference can be executed, differs depending on the number of bits in the address register.

For details, refer to **CHAPTER 4 ADDRESS STACK** and **6.3 Address Register (AR)**.

10.4.2 Table reference program example

The following examples show table reference programs:

Examples 1.

```
M000    MEM 0.00H     ;
P0A     MEM 0.70H     ;
P0B     MEM 0.71H     ;
P0C     MEM 0.72H     ;

START:                ; Program address 0000H
    BR      MAIN

DATA:
    DW      0001H     ; Constant data
    DW      0002H     ;
    DW      0004H     ;
    DW      0008H     ;
    DW      0010H     ;
    DW      0020H     ;
    DW      0040H     ;
    DW      0080H     ;
    DW      0100H     ;
    DW      0200H     ;
    DW      0400H     ;
    DW      0800H     ;
MAIN:
    BANK0             ; Macroinstruction
    SET4    P0ABIO3, P0ABIO2, P0ABIO1, P0ABIO0
    SET4    P0BBIO3, P0BBIO2, P0BBIO1, P0BBIO0
    SET1    P0CGIO
    MOV     RPH, #0000B   ; Sets general register to row address 7H in BANK0
    MOV     RPL, #0100B   ;
    MOV     AR3, # (.DL.DATA SHR 12 AND 0FH)
    MOV     AR2, # (.DL.DATA SHR 8 AND 0FH)
    MOV     AR1, # (.DL.DATA SHR 4 AND 0FH)
    MOV     AR0, # (.DL.DATA SHR 0 AND 0FH)
                          ; Sets 0001H in address register (AR)
```

```
LOOP:
; <1>
    MOVT  DBF, @AR      ; Transfers ROM value, specified by AR contents, to data buffer
; <2>
    LD    P0A, DBF2     ; Transfers data buffer value to data register in Port0A (70H),
    LD    P0B, DBF1     ; Port0B (71H), and Port0C (72H)
    LD    P0C, DBF0     ;
    ADD   M000, #1      ; Increments address register contents by 1
    ADD   AR0, M000
    ADDC  AR1, #0
    ADDC  AR2, #0
    ADDC  AR3, #0
    SKNE  AR0, #0CH     ; Writes 0 to AR0, when AR0 value becomes 0CH
    MOV   AR0, #0       ;
    BR    LOOP
```

When this program is executed, the constant data stored in addresses 0001H through 000CH in the program memory are sequentially read to the data buffer by <1> and output to ports 0A, 0B, and 0C by <2>.

Since the constant data is shifted to the left on a bit-by-bit basis at this time, the high-level signal is sequentially output to ports 0A, 0B, and 0C as a result.

In this example, the start address for the program memory, that stores the constant data, is set in the address register by the "MOV" instruction.

If the "MOV" instruction is used in this way, the start address for each constant set of data must be set in the address register, when there are many kinds of constant data to be stored.

Therefore, if the number of steps increases, because the "MOV" instruction is used many times, or in order to use a common routine for control, the program shown in **Example 2** is convenient.

Examples 2.

```
        M000    MEM 0.00H       ;
START:
        BR      MAIN            ;
DATAFETCH:
        DI                      ;
        POP     AR              ; Reads address stack register contents to address register.
                                ; At this time, stack pointer is shifted by constant data address,
        ADD     AR0, M000       ; specified by contents M000, specifying return address for
        ADDC    AR1, #0         ; main routine
        ADDC    AR2, #0         ;
        ADDC    AR3, #0         ;
        MOVT    DBF, @AR        ; Reads constant data
        EI
        RET                     ; Returns to main routine
DATA1:
        CALL    DATAFETCH       ; Calls common processing routine
        DW      0123H           ; At this time, DATA + 1 address is saved to address stack
        DW      4567H           ; register
                :
        DW      89ABH           ;
DATA2:
        CALL    DATAFETCH       ; Calls common processing routine
        DW      1357H           ; At this time, DATA2 + 1 address is saved to address stack
        DW      2468H           ; register
                :
        DW      9BDFH           ;
MAIN:
        BANK0                   ; Macroinstruction
        SET4    P0ABIO3, P0ABIO2, P0ABO1, P0ABIO0
        SET4    P0BBIO3, P0BBIO2, P0BBO1, P0BBIO0
        SET1    P0CGIO
        MOV     RPH, #0000B     ; Sets general register to row address 7H in BANK0
        MOV     RPL, #0100B     ;
LOOP:
        CALL    DATA1           ; Reads value for constant data DATA1, specified by M000 contents
        LD      P0A, DBF2       ;
        LD      P0B, DBF1       ; Transfers data buffer value to each port register in Port0A
        LD      P0C, DBF0       ; (70H), Port0B (71H) and Port0C (72H)
        CALL    DATA2           ; Reads value for constant data DATA2, specified by M000
        LD      P0A, DBF2       ; contents
        LD      P0B, DBF1       ; Transfers data buffer value to each port register in Port0A (70H),
        LD      P0C, DBF0       ; Port0B (71H), and Port0C (72H)
        ADD     M000, #1        ;
        SKNE    M000, #0CH      ; Writes 0 to AR0, when M000 contents become 0CH
        MOV     M000, #0        ;
        BR      LOOP
```

In this example, two stack levels are necessary, because the "CALL" instruction is executed two times, and the "POP" and "MOVT" instructions are executed.

The "CALL" instruction can be executed only once, as shown in **Example 3** below. In this case, two stack levels are also necessary for the "MOVT" instruction.

Examples 3.

```
DATAFETCH:
    DI                  ;
    POP     AR          ; Reads address stack register contents to address register
    MOVT    DBF, @AR    ; Transfers constant data storage address to data buffer
    INC     AR          ; Stores return address for main routine
    PUSH    AR          ;
    PUT     AR, DBF     ; Transfers constant data storage address to address register
    ADD     AR0, M000   ; Shifts by constant data address specified by M000 contents
    ADDC    AR1, #0     ;
    ADDC    AR2, #0     ;
    ADDC    AR3, #0     ;
    MOVT    DBF, @AR    ; Reads constant data
    EI
    RET                 ; Returns to main routine
DATA1:
    DW      0123H       ; Constant data
                        ;
DATA2:
    DW      1357H       ; Constant data
                        ;
MAIN:

LOOP:
    CALL    DATAFETCH   ;
    DW      .DL.DATA1   ;
    LD      P0A, DBF2   ;

    CALL    DATA2       ;
    DW      .DL.DATA2   ;
    LD      P0A, DBF2   ;

    BR      LOOP
```

10.5 Data Buffer and Hardware Peripherals

10.5.1 Controlling hardware peripherals

The central processing unit (CPU) controls hardware peripherals by setting data in or reading data from the hardware peripherals through the data buffer.

Each of the hardware peripherals has a register for data transfer (called a peripheral register), to which an address (peripheral address) is assigned.

By executing the sole use instructions "GET" and "PUT" to these peripheral registers, data can be transferred between the data buffer and hardware peripherals.

The "GET" and "PUT" instruction functions are as follows:

```
GET   DBF, p  ; Reads data for peripheral register addressed by p, to data buffer
PUT   p, DBF  ; Sets data for the data buffer in peripheral register addressed by p
```

The peripheral registers are classified into read-write (PUT/GET), write-only (PUT), and read-only (GET) registers.

If the "GET" instruction is executed to the write-only (PUT only) register, and undefined value is read.

However, if the "PUT" instruction is executed to the read-only (GET) register, the register contents are not affected.

★ Care must be exercised in using the 17K series Assembler (RA17K) or Emulator (IE-17K, IE-17K-ET). For details, refer to **10.3 Notes on Using Data Buffer**.

For the peripheral registers, refer to the Data Sheet for each model.

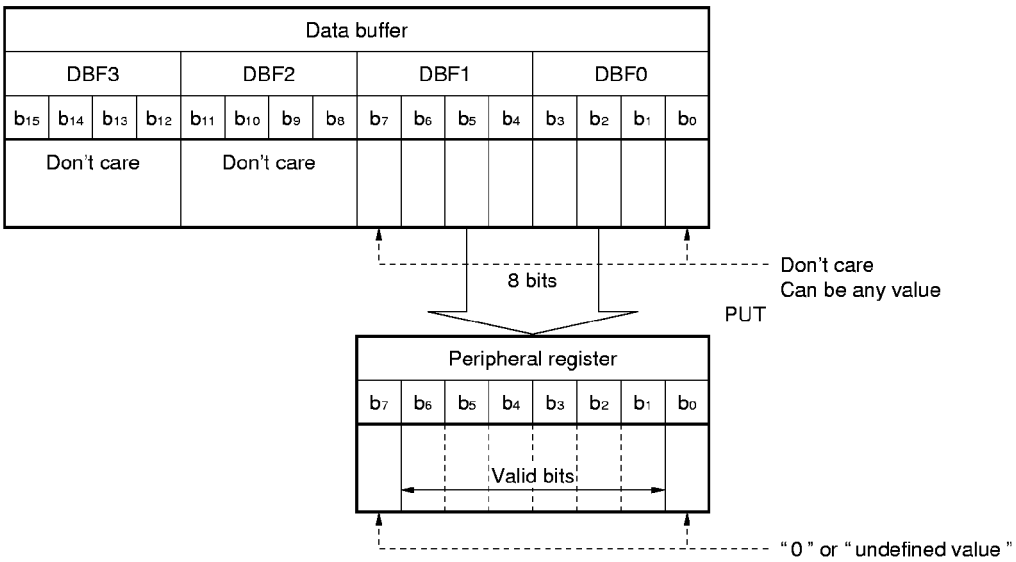
10.5.2 Data length when transferring data with peripheral register

Data is transferred between the data buffer and a hardware peripheral in 8- or 16-bit units. The PUT and GET instructions can be executed in one instruction execution time, regardless of whether or not the data is 16 bits long.

If the actual data bit length for a hardware peripheral is less than 8 bits, say, 7 bits, and if data transfer is carried out in 8 bit units, 1 excess bit results. This excess bit is treated as a "don't care (can be any value)" bit, when data is written, and as an undefined value, when data is read.

Figure10-5 shows an operation example, when the "PUT" instruction is executed (there are 6 valid bits in the peripheral register, bits b_1 through b_6).

Figure 10-5. Example Showing Data Transfer between Data Buffer and Hardware Peripheral

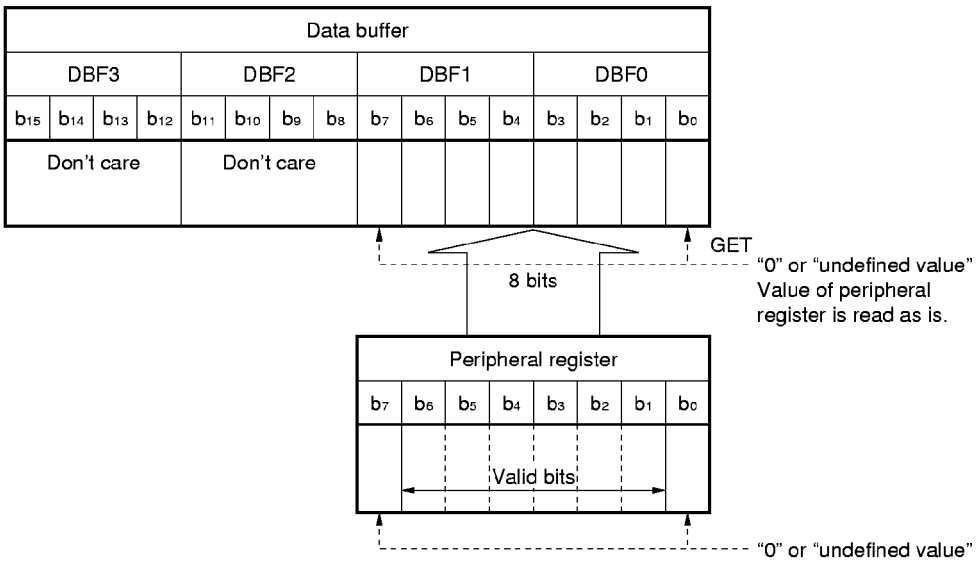


When 8-bit data is written to the peripheral register, the high-order 8 bits in the data buffer (contents of DBF3 and DBF2) are don't care bits.

Of the 8-bit data, the data buffer bits that correspond to the excess bits for the hardware peripheral are treated as don't care bits.

Figure 10-6 shows an operation example, when the GET instruction is executed.

Figure 10-6. Example Showing Data Transfer between Data Buffer and Hardware Peripheral



When 8-bit data is read, the values for the high-order 8 bits in the data buffer (contents of DBF3 and DBF2) do not change.

Of the 8-bit data for the data buffer, the bits that correspond to the excess bits in the peripheral register are "0" or "undefined". Whether the bits are "0" or "undefined" is determined in advance by the peripheral register.

CHAPTER 11 GENERAL-PURPOSE PORTS

The general-purpose ports output signals to external circuits and read signals from external circuits.

11.1 General-Purpose Port Configuration

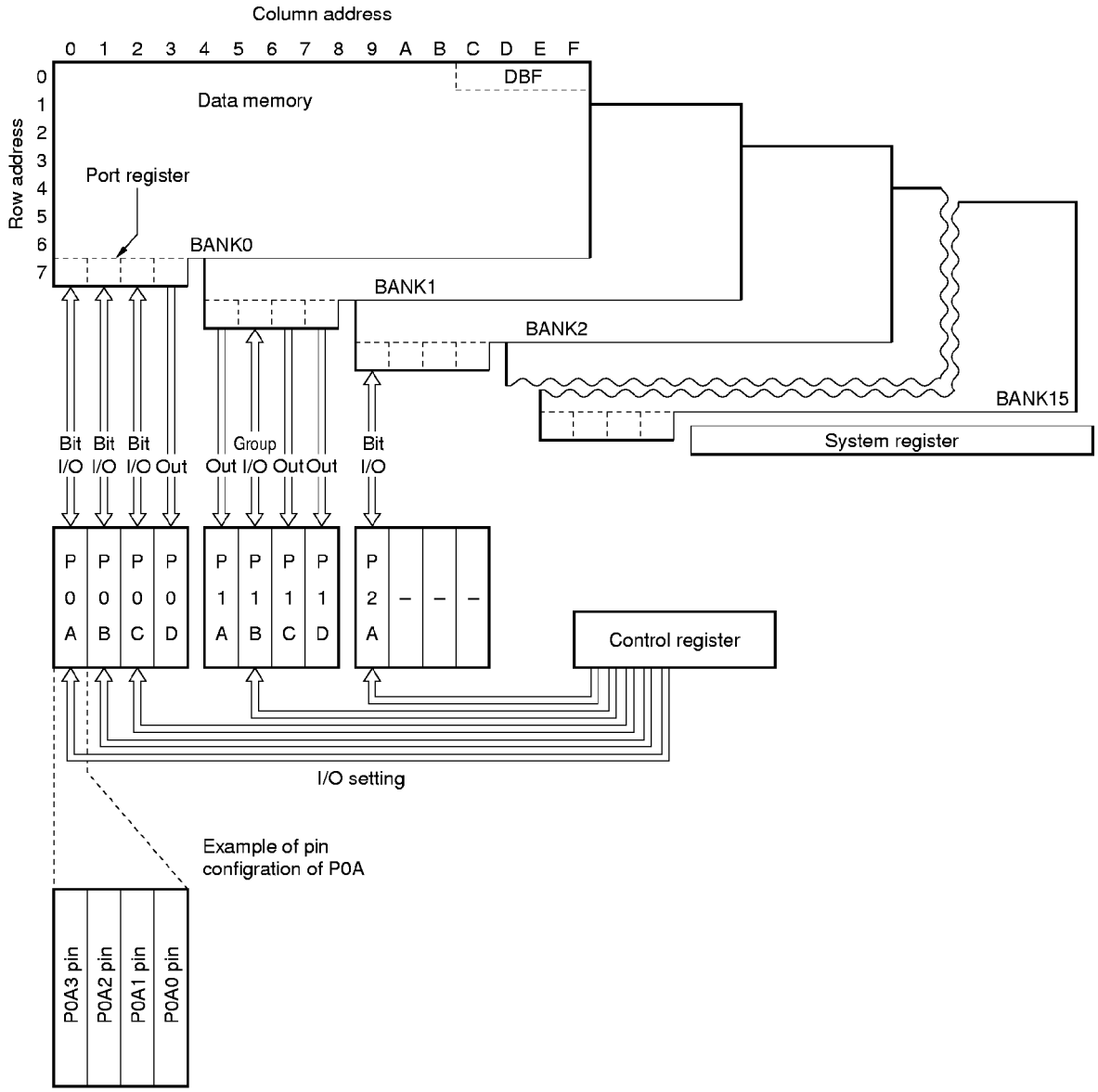
As shown in Figure 11-1, the general-purpose port writes data it inputs or outputs to addresses 70H through 73H (port register) for each bank in the data memory.

Each port has several pins (for example, Port0A consists of $P0A_3$ through $P0A_0$ pins).

The general-purpose ports are classified into I/O ports, input ports, and output ports.

The I/O ports are classified into bit I/O ports, which can be specified for input for output in 1-bit units (1-pin units), and group I/O ports, which can be specified for input or output in 4-bit units (4-pin units).

Figure 11-1. Block Diagram of General-Purpose Port



11.2 Function of General-Purpose Ports

The general-purpose output ports and the general-purpose I/O ports set in the output mode output a high or low level from the corresponding pins when data are set to the corresponding port register.

The general-purpose input ports and the general-purpose I/O ports set in the input mode detect the level of the signals input to the corresponding pins by reading the contents of the corresponding port register.

The general-purpose I/O ports are set in the input or output mode by the corresponding control register.

In other words, these ports can be set in the input or output mode by software.

Since general-purpose I/O ports are set to the general-purpose input port after a power-on reset, the pins that are also used for other hardware peripheral are specified independently by the corresponding control register.

11.2.1 General-purpose port data register (port register)

A port register sets output data of and reads the input data of the corresponding general-purpose port.

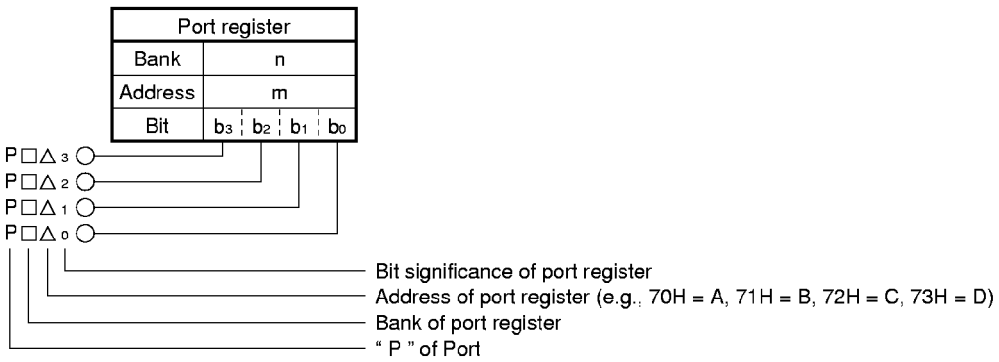
Because the port registers are mapped on the data memory, they can be manipulated by any data memory manipulation instruction.

Figure 11-2 shows the relation between a port register and the corresponding port pins.

By setting data to the port register corresponding to the port pins set in the general-purpose output port mode, the output of each pin is set.

By reading the contents of the port register corresponding to the port pins set in the general-purpose input port mode, the input status of each pin is detected.

Figure 11-2. Relation between Port Register and Pins



Reserved words are defined for the port registers by the assembler.

Because these reserved words are defined in flag (bit) units, the assembler embedded macro instructions can be used.

Note that data memory type reserved words are not defined for the port registers.

[MEMO]

CHAPTER 12 INTERRUPT FUNCTIONS

The interrupt function stops any on-going processing and executes a program which is to be executed when generating specific data, if a specified hardware peripheral outputs predetermined data.

Therefore, when a request is issued from a hardware peripheral, the program execution is stopped and branched to a program starting with an address (vector address) specified in advance.

12.1 Interrupt Block Configuration

As shown in Figure 12-1, the interrupt block consists of interrupt request blocks that control interrupt requests, interrupt enable flip-flop (INTE) that enables the interrupt, stack pointer that is controlled when an interrupt has been accepted, address stack register, program counter, and interrupt stack.

An interrupt request is issued from a hardware peripheral.

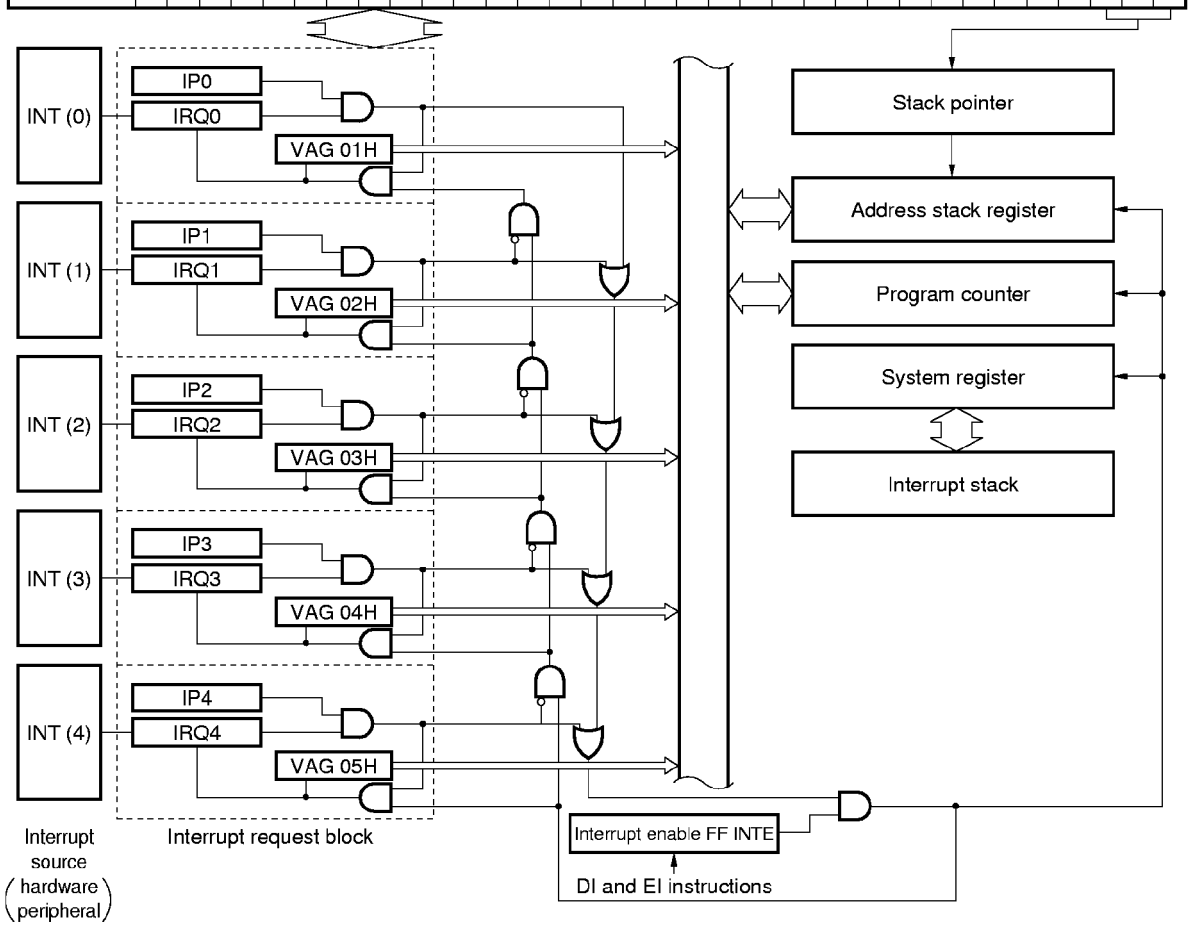
The interrupt request processing block for each hardware peripheral consists of interrupt request flag (IRQ×××) that detects an interrupt request, interrupt permission flag (IP×××) that enables each interrupt, and vector address generator (VAG) that specifies a vector address (branch destination address), when an interrupt has been accepted.

The interrupt request flag (IRQ×××) and interrupt permission flag (IP×××) are shown in the interrupt processing block in Figure 12-1.

Actually, however, they are in the interrupt request register and interrupt permission register in the control register.

Figure 12-1. Configuration Example of Interrupt Block

Control register																															
Name	Interrupt request 1 (INTREQ1)				Interrupt request 2 (INTREQ2)				Interrupt request 3 (INTREQ3)				Interrupt request 4 (INTREQ4)				Interrupt request 5 (INTREQ5)				Interrupt permission 1 (INTPM1)				Interrupt permission 2 (INTPM2)					Stack pointer SP			
Address	3BH				3CH				3DH				3EH				3FH				2EH				2FH					01H			
Bit	b_3	b_2	b_1	b_0	b_3	b_2	b_1	b_0	b_3	b_2	b_1	b_0	b_3	b_2	b_1	b_0	b_3	b_2	b_1	b_0	b_3	b_2	b_1	b_0	b_3	b_2	b_1	b_0		b_3	b_2	b_1	b_0
Flag symbol	0	0	0	IRQ4	0	0	0	IRQ3	0	0	0	IRQ2	0	0	0	IRQ1	0	0	0	IRQ0	0	0	0	IP4	IP3	IP2	IP1	IP0		SP3	SP2	SP1	SP0



12.2 Interrupt Functions

An interrupt function stops the on-going program and executes a sole use processing program when a hardware peripheral enters a certain status.

At this time, the interrupt signal from the hardware peripheral is called an “interrupt request”, and generation of the interrupt signal is called “interrupt request issuance”. The sole use interrupt processing routine is called an “interrupt processing routine”.

When an interrupt has been accepted, the program memory address contents, determined for each interrupt source (vector address), are read and the program execution is branched. Therefore, each interrupt processing routine is started from this vector address.

The interrupt functions are classified into processing before an interrupt is accepted and processing after the interrupt has been accepted. Therefore, the functions are divided into accepting an interrupt in response to an interrupt request from a hardware peripheral and, when the interrupt has been accepted, branching the execution to a vector address and returning the execution to the program executed before the interrupt has been accepted.

12.2.1 through **12.2.5** describe the functions of each block.

12.2.1 Hardware peripheral

A condition, under which an interrupt request is to be issued, can be set to each hardware peripheral.

For example, an external interrupt pin can be set so that an interrupt request is issued, when the rising or falling edge of the signal is applied to the pin.

For details on the interrupt request issuance conditions for each hardware peripheral, refer to the Data Sheet for each model.

12.2.2 Interrupt request processing block

An interrupt request processing block is available for each hardware peripheral. It detects the presence or absence of each interrupt request, enables the interrupt, and generates a vector address when the interrupt has been accepted.

12.2.3 and **12.2.4** describe each flag for the interrupt request processing block.

12.2.3 Configuration and function of interrupt request flag (IRQ×××)

Each interrupt request flag (IRQ×××) is set to "1" when an interrupt request is issued from the corresponding peripheral hardware unit, and is reset to "0" when the interrupt is acknowledged.

Detecting these interrupt request flags (IRQ×××) when no interrupt is enabled will allow the state of each interrupt request to be detected.

Directly writing "1" to an interrupt request flag via the window register is also equivalent to an interrupt request being issued.

Once this flag has been set to "1", it is not reset until the corresponding interrupt is acknowledged or an interrupt request reset macro is executed.

If more than one interrupt request is issued at the same time, the interrupt request flag corresponding to the interrupt that has not been acknowledged is not reset.

The configuration and function of the interrupt request flag are shown below.

Figure 12-2. Configuration Example of Interrupt Request Flag

Name	Interrupt request register 1				Interrupt request register 2			
Address	Depends on model							
R/W								
Bit	b_3	b_2	b_1	b_0	b_3	b_2	b_1	b_0
Flag symbol	0	0	0	IRQ4	0	0	0	IRQ3

Fixed to 0 Fixed to 0

IRQ3	Detects issuance status of interrupt request for INT (3)
0	Interrupt not requested
1	Interrupt requested

IRQ4	Detects issuance status of interrupt request for INT (4)
0	Interrupt not requested
1	Interrupt requested

12.2.4 Configuration and functions of Interrupt permission flag (IP×××)

Each interrupt permission flag enables the interrupt of the corresponding peripheral hardware unit.

★ All the following three conditions must be satisfied in order that an interrupt may be acknowledged:

- The interrupt is enabled by the corresponding interrupt permission flag.
- The interrupt request is issued by the corresponding interrupt request flag.
- The "EI" instruction (that enables all the interrupts) is executed.

Since the interrupt permission flag is in the interrupt permission register in the control flag, it can be read or written through the window register (WR).

Once this flag has been set, it will not be reset until "0" is written to it through the window register.

The interrupt permission register configuration and functions are as follows.

Figure 12-3. Configuration Example of Interrupt Permission Flag

Name	Interrupt permission register 1				Interrupt permission register 2			
Address	Depends on model							
R/W								
Bit	b_3	b_2	b_1	b_0	b_3	b_2	b_1	b_0
Flag symbol	0	0	0	IP4	IP3	IP2	IP1	IP0

Fixed to 0

IP0	Setting interrupt for INT (0)
0	Disables
1	Enables

IP1	Setting interrupt for INT (1)
0	Disables
1	Enables

IP2	Setting interrupt for INT (2)
0	Disables
1	Enables

IP3	Setting interrupt for INT (3)
0	Disables
1	Enables

IP4	Setting interrupt for INT (4)
0	Disables
1	Enables

12.2.5 Stack pointer, address stack register, and program counter

The address stack register saves the return address to which execution is to be returned from an interrupt processing routine.

The stack pointer specifies the address of the address stack register.

When an interrupt is acknowledged, therefore, the value of the stack pointer is decremented by one and the value of the program counter at that time is saved to the address stack register specified by the stack pointer.

When the dedicated return instruction "RETI" is executed after the processing of the interrupt processing routine has been executed, the contents of the address stack register specified by the stack pointer are restored to the program counter, and the value of the stack pointer is incremented by one.

For further information, also refer to **CHAPTER 4 ADDRESS STACK**.

12.2.6 Interrupt enable flip-flop (INTE)

The interrupt enable flip-flop enables all the interrupts.

When this flip-flop is set, all the interrupts are enabled. When it is reset, all the interrupts are disabled.

This flip-flop is set or reset by using dedicated instructions "EI (to set)" and "DI (to reset)".

The "EI" instruction sets this flip-flop when the instruction next to the "EI" instruction is executed, and the "DI" instruction resets the flip-flop while the "DI" instruction is executed.

When an interrupt is acknowledged, this flip-flop is automatically reset.

Nothing is affected even if the "DI" instruction is executed in the DI state, or if the "EI" instruction is executed in the EI state.

★ This flip-flop is reset on power-ON reset, CE reset, and on execution of the clock stop instruction.

12.2.7 Vector address generator (VAG)

VAG generates a branch destination address (vector address) of the program memory when a peripheral hardware interrupt is acknowledged.

12.2.8 Interrupt stack

The interrupt stack is configured as shown in Figure 12-4. It saves the system register contents, when an interrupt request has been accepted.

When the interrupt request has been accepted and the system register contents have been saved, the system register is reset to 0.

Figure 12-4. Configuration Example of Interrupt Stack

Stack Name		Interrupt Stack (INTSK)								
		Bank stack (BANKSK)				Status stack (PSWSK)				
Bit		b_3	b_2	b_1	b_0	b_4	b_3	b_2	b_1	b_0
Address	0H	–	BANKSK0			BCD SK0	Z SK0	CY SK0	CMP SK0	IXE SK0
	1H	–	BANKSK1			BCD SK1	Z SK1	CY SK1	CMP SK1	IXE SK1

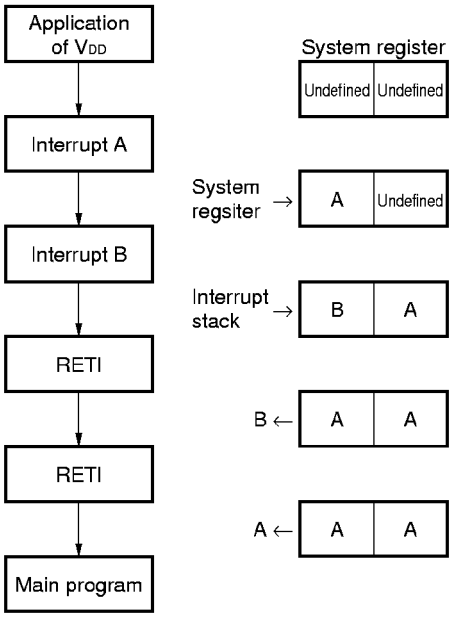
The interrupt stack can save up to the maximum stack level for each model. Therefore, the maximum levels for nesting, which are to accept another interrupt in an interrupt processing routine, can be saved to the interrupt stack.

Data is saved to the interrupt stack each time an interrupt has been accepted. Each time the interrupt return instruction (RETI) has been executed, the data is restored to the system register, as shown in (a) in Figure 12-5.

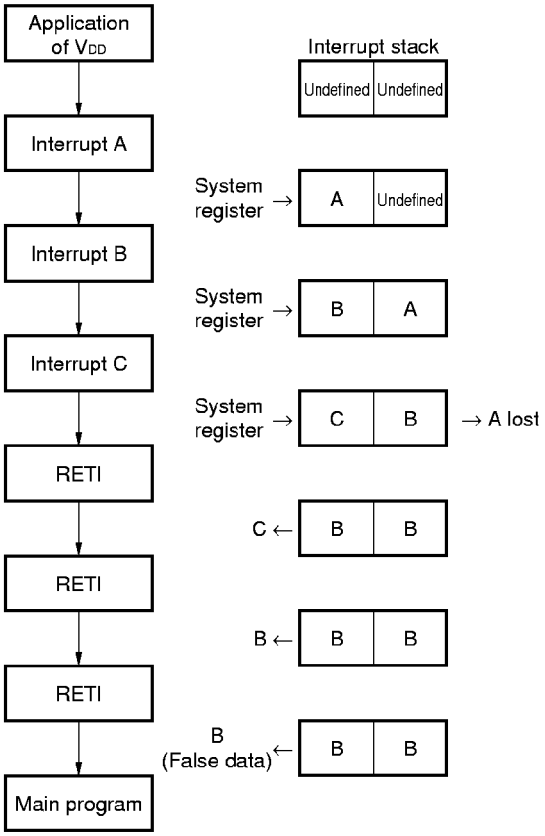
However, if an interrupt exceeding the maximum stack level is accepted, the first data is discarded, as shown in (b) in Figure 12-5 (B). To prevent the data from being discarded, it must be saved by program.

Figure 12-5. Example of Interrupt Stack Operation (when maximum stack level = 2)

(a) When interrupt level is 2



(b) When interrupt level exceeds 2



12.3 Acknowledging Interrupts

12.3.1 Acknowledging interrupts and priority

An interrupt is acknowledged in the following procedure:

(1) Each peripheral hardware unit outputs an interrupt request signal to the corresponding interrupt control block if a given interrupt condition is satisfied (e.g., if a valid signal is input to the external interrupt pin).

(2) When the interrupt control block has received the interrupt request signal from the peripheral hardware unit, it sets the corresponding interrupt request flag (IRQ×××) to "1".

(3) If the interrupt permission flag corresponding to the interrupt request flag (IP×××) is set to "1" when the interrupt request flag is set to "1", the interrupt control block outputs "1".

(4) The signal output by the interrupt control block is ANDed with the output of the interrupt enable flip-flop, and an interrupt acknowledge signal is output.
This interrupt enable flip-flop is set to "1" by the "EI" instruction and reset to "0" by the "DI" instruction.
If the interrupt control block outputs "1" while the interrupt enable flip-flop is "1", the interrupt is acknowledged.

As shown in Figure 12-1, the interrupt acknowledge signal is input to each interrupt control block when the interrupt has been acknowledged.

The interrupt request flag is reset to "0" by the signal input to the interrupt control block, and a vector address corresponding to the interrupt is output.

If the interrupt control block outputs "1" at this time, the interrupt acknowledge signal is not transferred to the next stage. If two or more interrupt requests are issued at the same time, the interrupts are acknowledged according to the predetermined priority. This priority is called hardware priority.

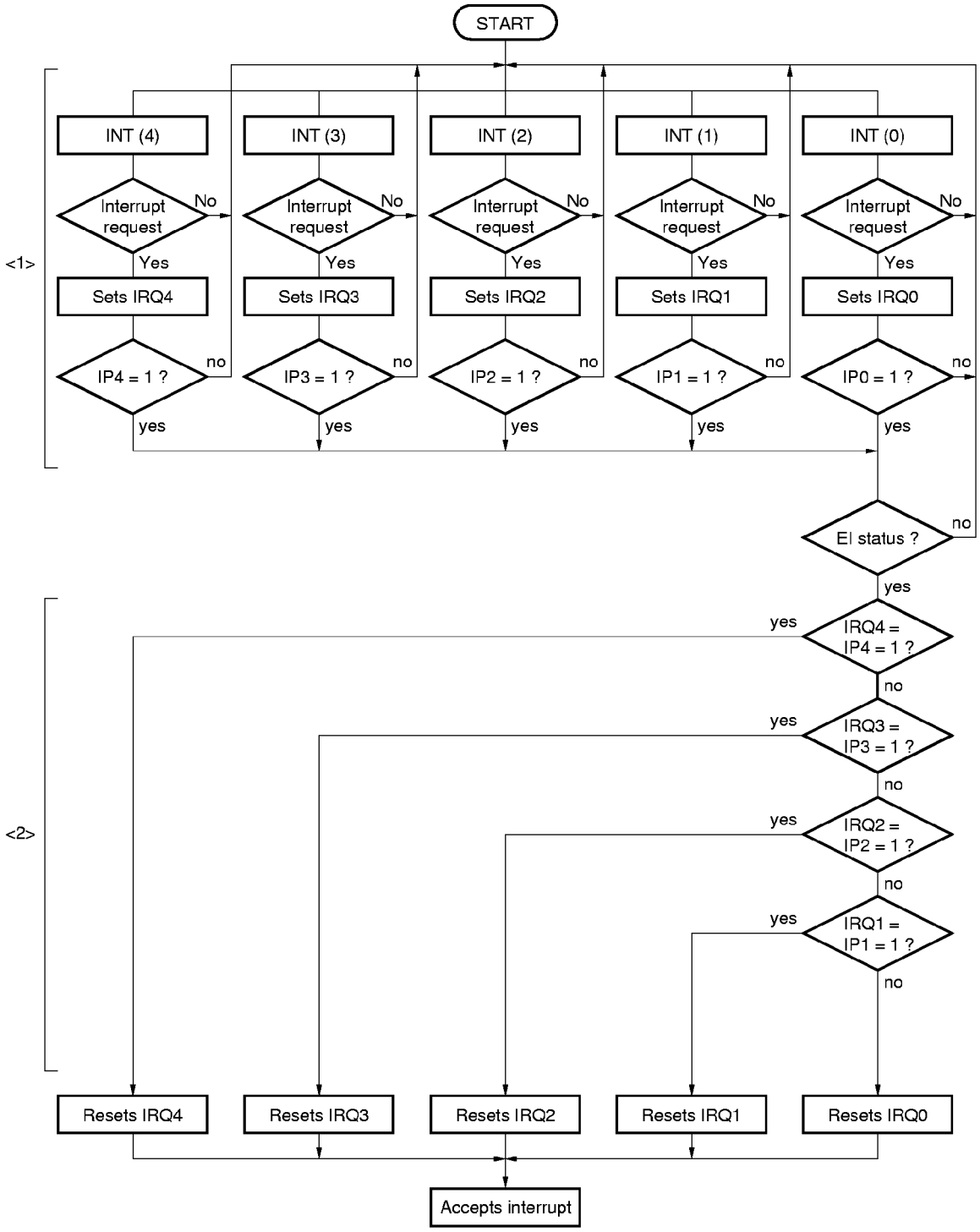
In the flowchart in Figure 12-6, processing <1> is always performed in parallel. If two or more interrupt requests are generated at the same time, each interrupt request flag (IRQ×××) is simultaneously set.

In processing <2>, however, interrupt servicing can be performed in any sequence by setting or resetting each interrupt permission flag (IP×××) by program.

At this time, the interrupt having an interrupt permission flag is called a maskable interrupt. Because a maskable interrupt, even one with a high hardware priority, can be disabled, its priority is called software priority.

For the interrupt permission flag, refer to **12.2.4 Configuration and function of interrupt permission flag (IP×××)**.

Figure 12-6. Accepting Interrupt



12.3.2 Timing chart for acknowledging interrupt

Figure 12-7 shows the timing chart illustrating acknowledging interrupts.

(1) in this figure illustrates how one interrupt is acknowledged.

(a) in (1) shows the case where the interrupt request flag is lastly set to "1", and (b) in (1) shows the case where the interrupt permission flag is lastly set to "1".

In either case, the interrupt is acknowledged when all the interrupt request flag, interrupt enable flip-flop, and interrupt permission flag are set to "1".

If the last flag or flip-flop that was set to "1" satisfies the first instruction cycle of the "MOVT DBF, @AR" instruction or a given skip condition, the interrupt is acknowledged after the second instruction cycle of the "MOVT DBF, @AR" instruction or the instruction that is skipped (NOP) has been executed.

The interrupt enable flip-flop is set in the instruction cycle next to the one in which the "EI" instruction is executed.

(2) in Figure 12-7 illustrates how more than one interrupt is used.

In this case, the interrupts are sequentially acknowledged according to the hardware priority if all the interrupt permission flags are set. The hardware priority can be changed by manipulating the interrupt permission flag by program.

"Interrupt cycle" shown in Figure 12-7 is a special cycle in which the interrupt request flag is reset, a vector address is specified, and the contents of the program counter are saved after an interrupt has been acknowledged, and lasts for the one instruction execution time.

Because the interrupt request flag is set to "1" by an interrupt request from the peripheral hardware, regardless of the "EI" instruction and interrupt permission flag, whether an interrupt request has been issued or not can be checked by checking the interrupt request flag (IRQxxx) by program.

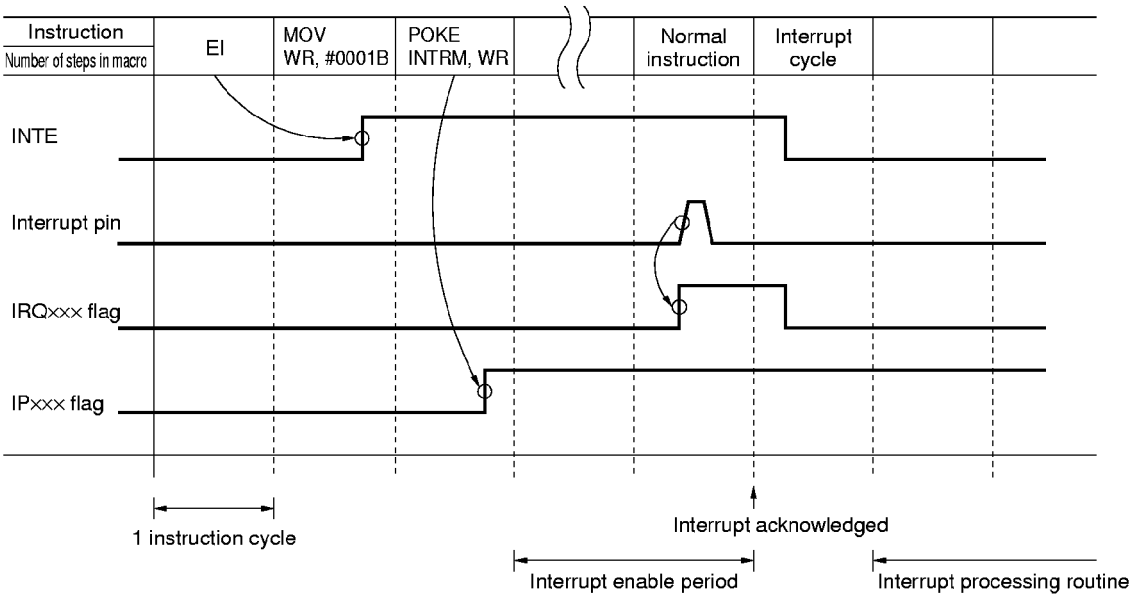
For details, refer to **12.4 Operation After Interrupt Has Been Acknowledged**.

Figure 12-7. Timing Chart of Acknowledging Interrupt (1/3)

(1) When one interrupt (e.g., rising of external interrupt pin) is used

(a) If interrupt is not masked by interrupt permission flag (IP×××)

<1> If "MOVT" instruction or normal instruction that satisfies skip condition is not executed when interrupt is acknowledged



<2> If "MOVT" instruction or "instruction satisfying skip condition" is executed when interrupt is acknowledged

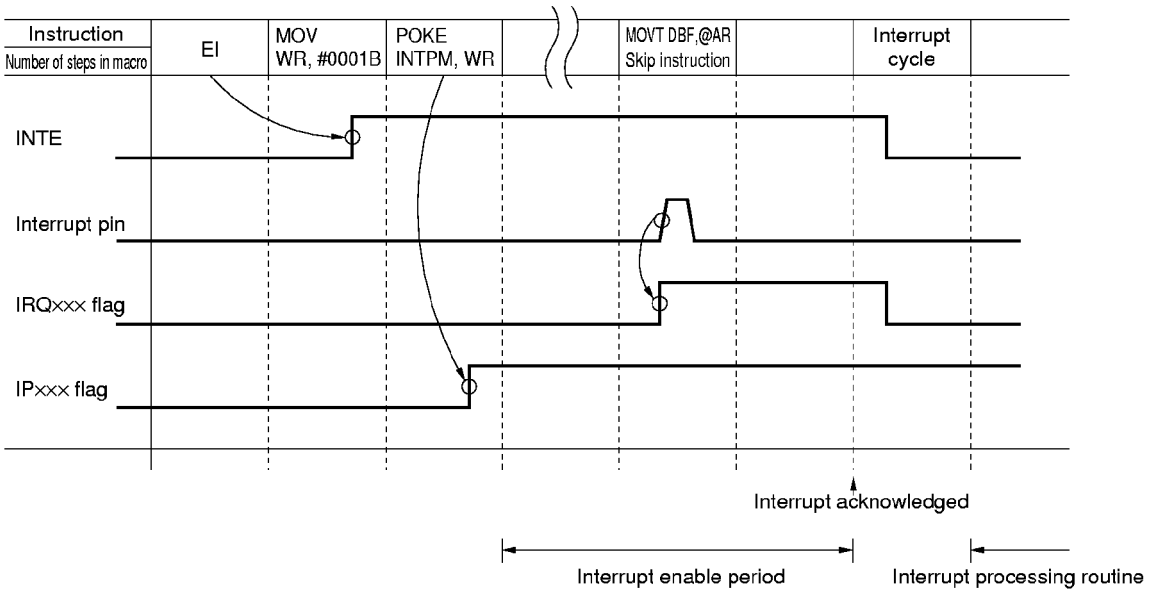


Figure 12-7. Timing Chart of Acknowledging Interrupt (2/3)

(b) If interrupt is kept pending by interrupt permission flag

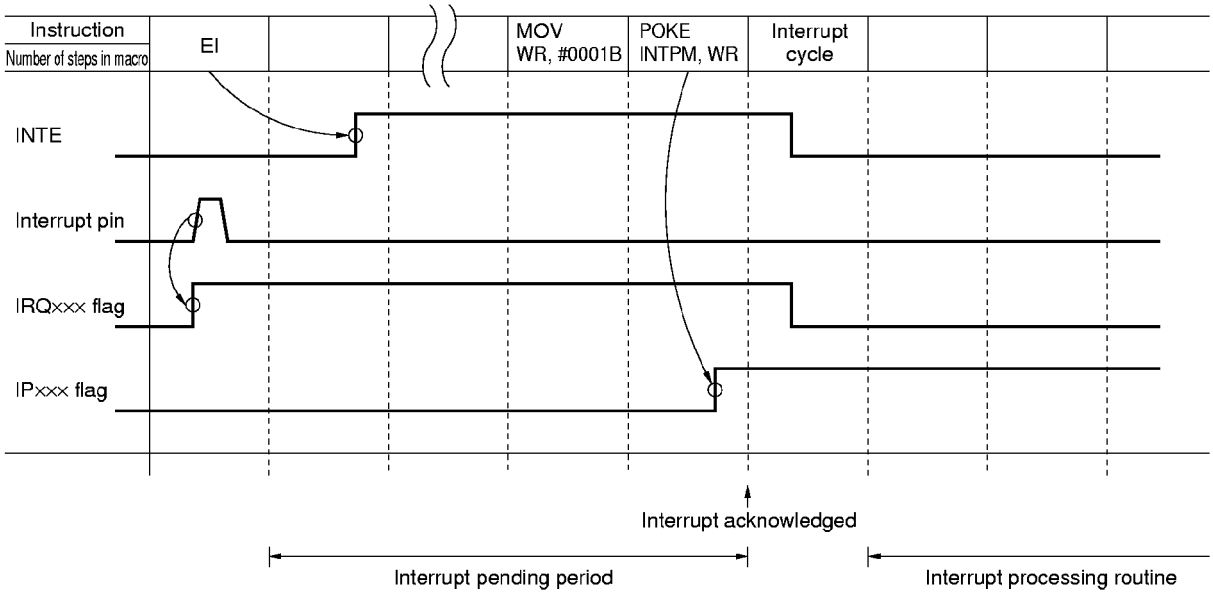
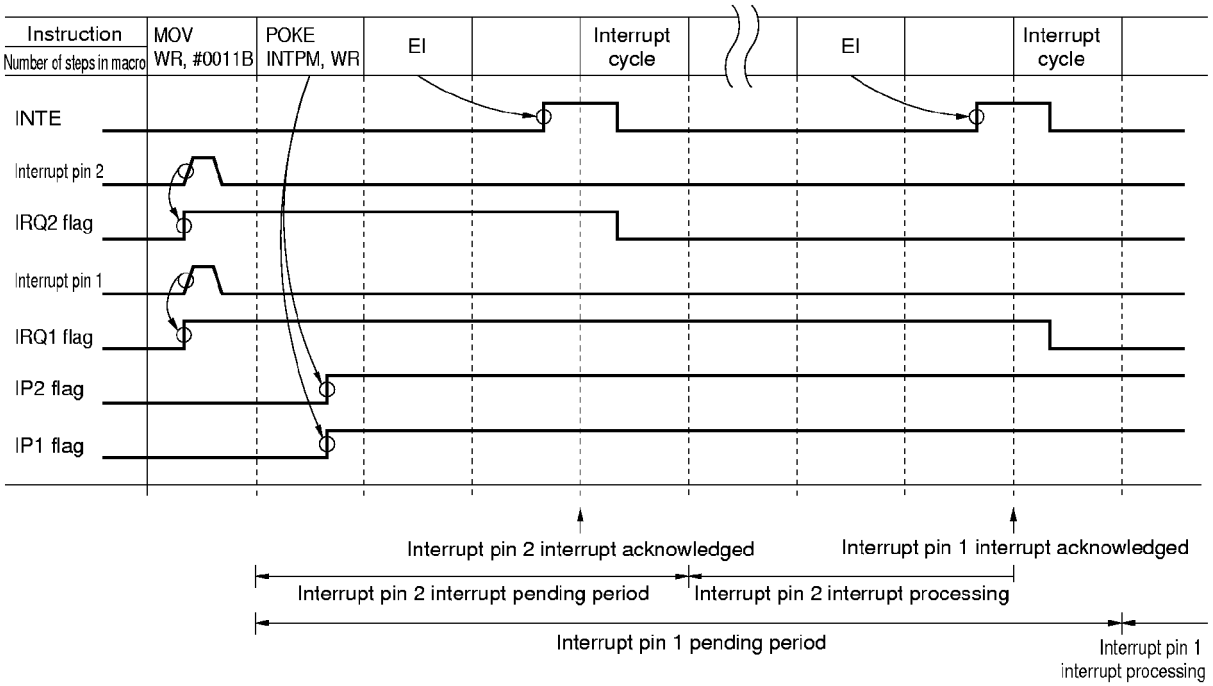


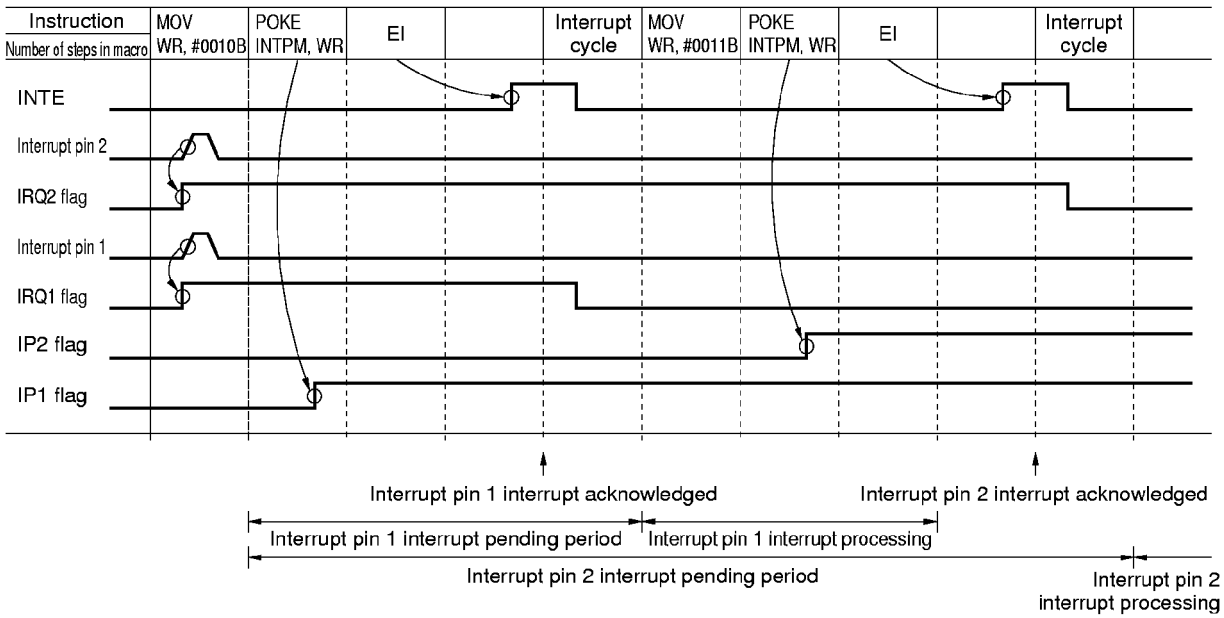
Figure 12-7. Timing Chart of Acknowledging Interrupt (3/3)

(2) When two interrupts (Example: Two types of interrupt pins 1 and 2 (2: Hardware priority)) are used

(a) Hardware priority



(b) Software priority



12.4 Operation After Interrupt Has been Acknowledged

When an interrupt has been acknowledged, the following processing is sequentially executed.

(1) The interrupt enable flip-flop and the interrupt request flag corresponding to the acknowledged interrupt are reset to "0", disabling the interrupts.

(2) The contents of the stack pointer are decremented by one.

(3) The contents of the program counter are saved to the address stack register specified by the stack pointer. The contents saved at this time are the next program memory address that is used after the interrupt has been acknowledged. For example, if a branch instruction is executed, the contents saved are the branch destination address; if a subroutine call instruction is executed, they are the called address. Because the interrupt is acknowledged after the next instruction is executed as a "NOP" instruction if a skip condition is satisfied by a skip instruction, the saved contents are the skipped address.

(4) The least significant bit of the bank register (BANK) is saved to the interrupt stack.

(5) The contents of the vector address generator corresponding to the acknowledged interrupt are transferred to the program counter. In other words, execution branches to an interrupt processing routine.

The processing (1) through (5) above is executed in one special instruction cycle in which the normal instruction is not executed. This instruction cycle is called an interrupt cycle.

In other words, one instruction cycle time is necessary since an interrupt has been acknowledged until execution branches to the corresponding vector address.

12.5 Interrupt processing Routine

An interrupt is accepted immediately, when an interrupt request has been issued independently from the program being executed at that time, as long as it is in the program area, where the interrupt is enabled.

Therefore, to return the execution to the original program, after the interrupt processing has been executed, the status must be restored as if the interrupt processing had not been executed.

For example, if an arithmetic operation is executed during interrupt processing, the carry flag (CY) content may change from that before the interrupt has been accepted. This leads to misjudgment, after the execution has been restored to the original program.

It is therefore necessary to save and restore the system register and control register that may be manipulated by the interrupt processing routine.

To enable another interrupt while one interrupt processing is being accomplished (nesting), refer to **12.6 Nesting**.

12.5.1 Saving

Among the system registers, only the bank (BANK) register and index enable flag are automatically saved by the hardware. Save the other system registers through a program, if necessary.

As shown in Figure 12-8, the POKE and PEEK instructions are convenient for saving or restoring the system registers.

The system registers can also be saved or restored by using the transfer instruction (LD r,m or ST m,r). However, specifying data memory addresses, to which the registers are to be saved, is difficult unless the row address for the general register is constant, when the interrupt has been accepted.

The reason is that, when the transfer instruction is used to save the general register, the saved address is not constant, unless the general register address is constant. Consequently, at least the general register must be fixed in the interrupt enable routine.

By contrast, since the address for the register file, that is controlled by the PEEK and POKE instructions, can be specified independently from the general register contents. Since addresses 40H through 7FH for the register file overlap the bank in the data memory selected at that time, the system register can be saved merely by specifying the bank.

In Figure 12-8, the window register and register pointers (RPH and RPL) are saved by the PEEK and POKE instructions. The general register is re-set to row address 07H in BANK1. The other system registers are saved by the next ST instruction.

Figure 12-9 illustrates how the saving operation is performed in the example program in Figure 12-8.

12.5.2 Restoration processing

To restore registers, an operation in reverse to the saving processing should be performed, as shown in Figure 12-8.

When an interrupt has been accepted, the interrupt was naturally enabled (EI status). It is therefore necessary to execute the EI instruction before executing the RETI instruction, as shown in Figure 12-8.

The EI instruction sets the interrupt enable flip-flop to 1 after the next RETI instruction has been executed. Consequently, the interrupt is enabled after the execution has been returned to the program executed before the interrupt was accepted.

When the sole use instruction "RETI" is executed, the following processing is automatically executed in sequence, and the processing, during which the interrupt was accepted, is resumed.

(1) The address stack register contents, specified by the stack pointer, are restored to the program counter.

(2) The interrupt stack contents are restored to the low-order 3 bits in the bank register (BANK: address 79H) and program status word (IXE, BCD, CMP, CY, and Z flags).

(3) The stack pointer contents are incremented by one.

The above steps (1) through (3) are executed in one instruction cycle, during which the RETI instruction is executed.

The only difference between the RETI instruction and subroutine return instructions RET and RETSK is the operation to restore the bank register and program status word in (2) above.

Figure 12-8. Saving and Restoring in Interrupt Processing Routine

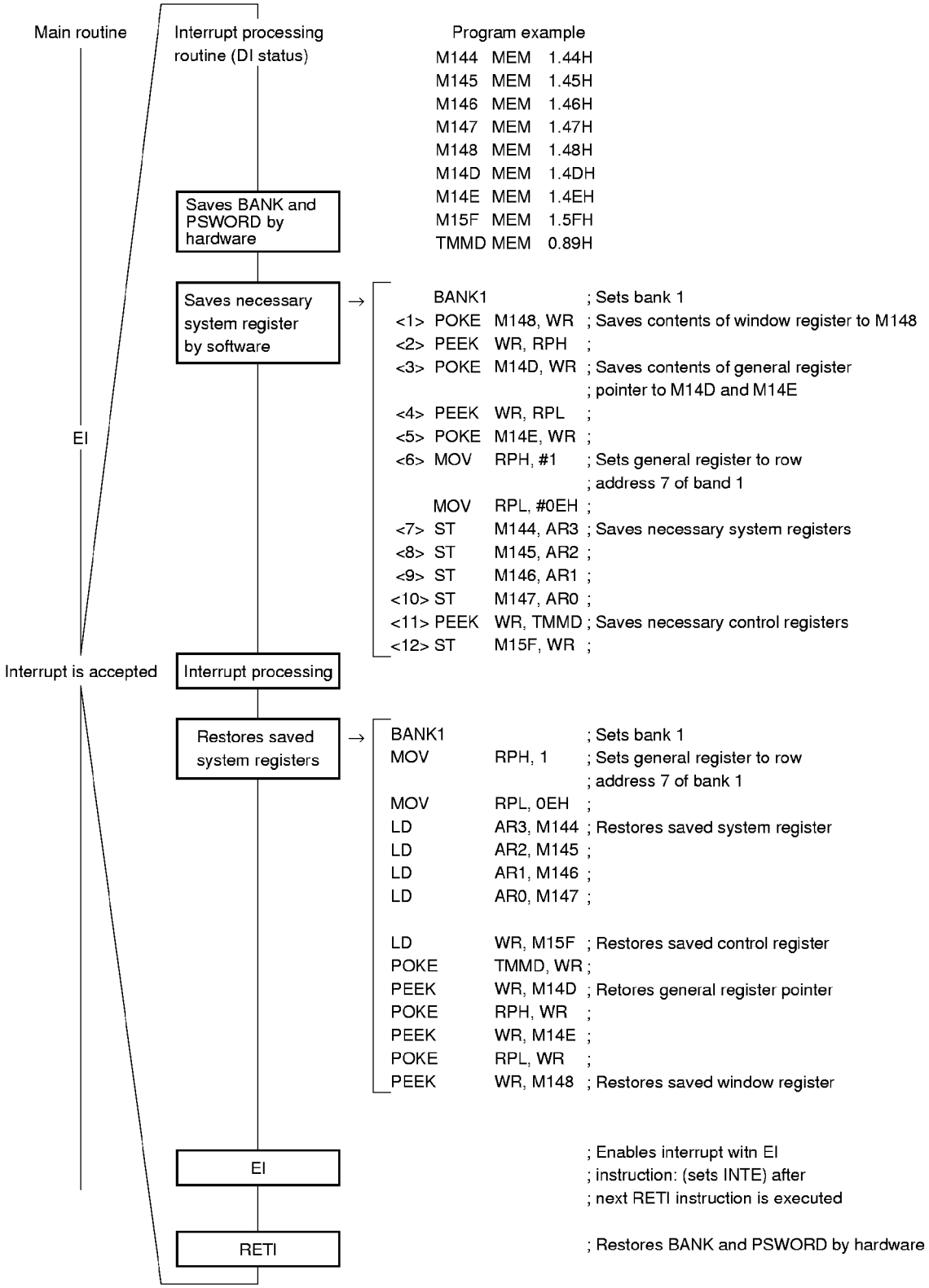
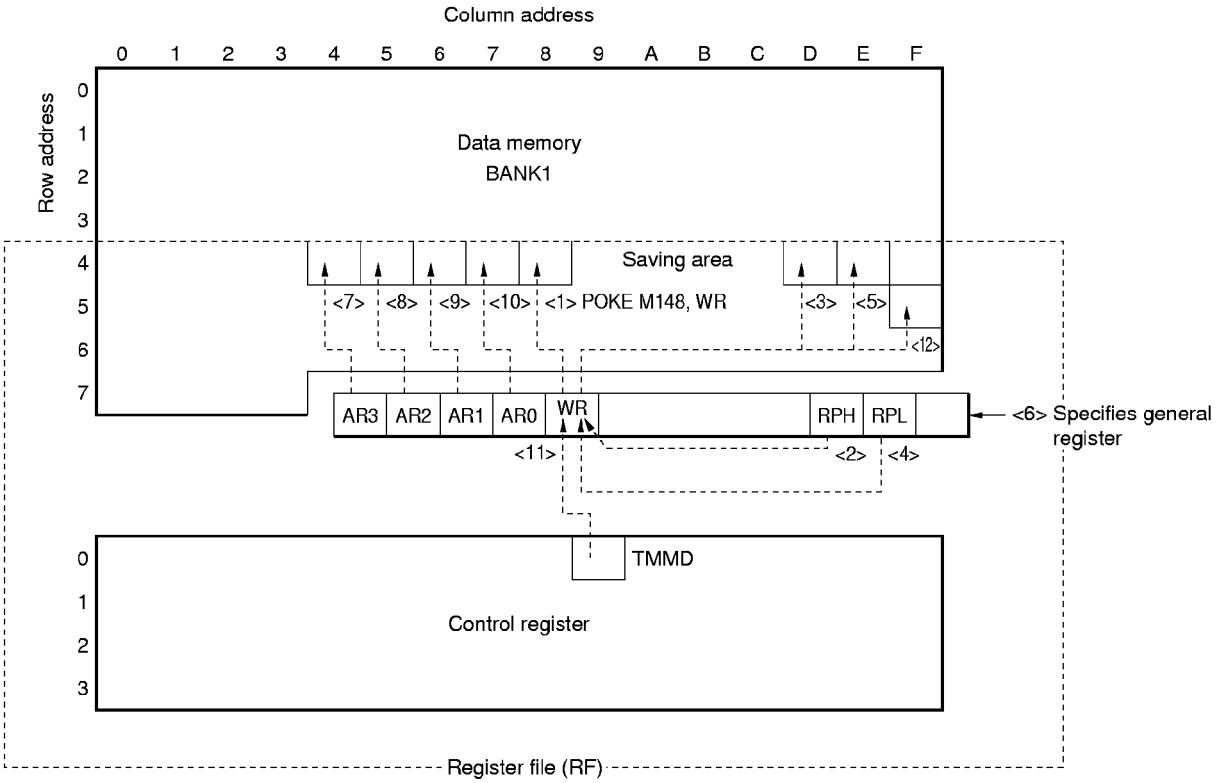


Figure 12-9. Saving System Register and Control Register When PEEK and POKE Instructions Are Used



Remark <1> through <12> corresponds to numbers in program in Figure 12-8.

12.5.3 Notes on interrupt processing routine

Pay close attention to the following points, concerning the interrupt processing routine:

(1) The bank register and program status word

The bank register and program status word are reset to "0", after they are have been saved to the interrupt stack.

(2) The other system registers saved through software

Data for the other system registers, saved through software, are not reset, even after they have been saved.

12.6 Nesting

Nesting is a technique to process interrupts C and D that are generated from separate sources, while the interrupt processing for sources A and B is being executed, as shown in Figure 12-10.

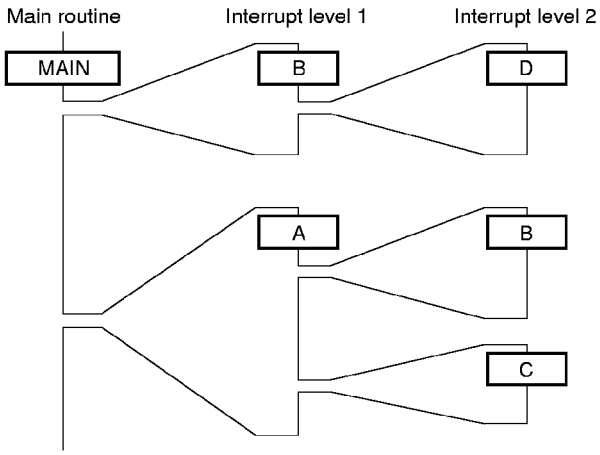
At this time, the interrupt depth is called a level.

To use nesting, consider the following points:

(1) Interrupt source priority
(2) Interrupt level limit, due to interrupt stack

12.6.1 and **12.6.2** describe (1) and (2) above in detail.

Figure 12-10. Example of Nesting



12.6.1 Interrupt source priority

In order to use a nesting, it is necessary to set the priorities for the interrupt sources.

For example, when interrupt sources are A, B, C, and D, the priorities can be:

A = B = C = D

or,

A < B < C < D

Note, however, that if all the sources are given the same priority (A = B = C = D), the main routine always accepts interrupts A, B, C, and D. Yet, if interrupt C has been accepted, all the other interrupts A, B, and D are disabled, and nesting cannot be implemented.

When the priorities are set as A < B < C < D, interrupt C takes precedence over A and B, even while A or B is being processed.

Similarly, D takes precedence over C.

The priority can be set by the hardware or software, by using the interrupt permission flag, as described in **12.3 Acknowledging Interrupts**.

★ Priorities for nesting interrupts must be specified in the following cases:

- Interrupt source A: Issues an interrupt request every 10 ms.
  Interrupt service time: 4 ms
- Interrupt source B: Issues an interrupt request every 2 ms.
  Interrupt service time: 1 ms

Also, assume that A and B are both assigned the same priority.

If interrupt processing for A is executed while B is being processed, B is not processed several times.

Since an interrupt is generally used for processing with a high emergency status, it is necessary to determine the priorities for these two interrupts, for example, A < B, so that A is disabled, while B is being processed, and that B is accepted, even while A is being processed.

If nesting is used for purposes not emergent, priorities are not necessarily determined. However, if the number of interrupt sources were to exceed the nesting limit, as described in **12.6.2 Interrupt limit by interrupt stack**, it is necessary to determine appropriate priorities, to prevent exceeding the interrupt level.

12.6.2 Interrupt limit by interrupt stack

The bank register contents in the system registers and program status word are saved to the interrupt stack.

Usually, the interrupt stack operates as shown in (a) in Figure 12-11.

The bank register and program status word are reset as soon as their contents are saved to the interrupt stack.

If nesting were to exceed the maximum level for the interrupt stack, the bank register contents and program status word are not correctly restored, as shown in (b) in Figure 12-11.

However, if nesting is performed in a main routine, where the interrupt is enabled, so that the bank register and program status word are always fixed and that the priorities for the interrupts are clear, the maximum stack level can be exceeded by using the subroutine return instruction (RET) as shown in Figure 12-12.

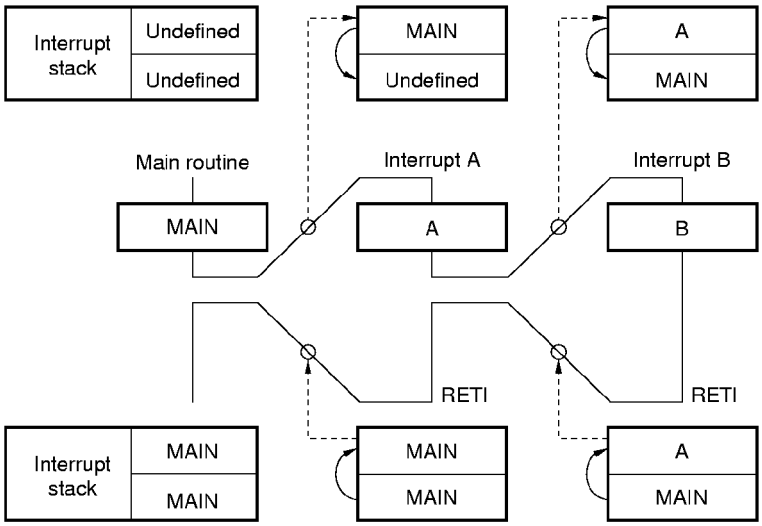
However, at this time, the device operations and those for the emulator differ, as shown in Figures 12-12 and 12-13.

★ As shown, the interrupt stack for the device is a “sweeping category”, while that for the emulator (IE-17K, IE-17K-ET) is a “rotating category”.

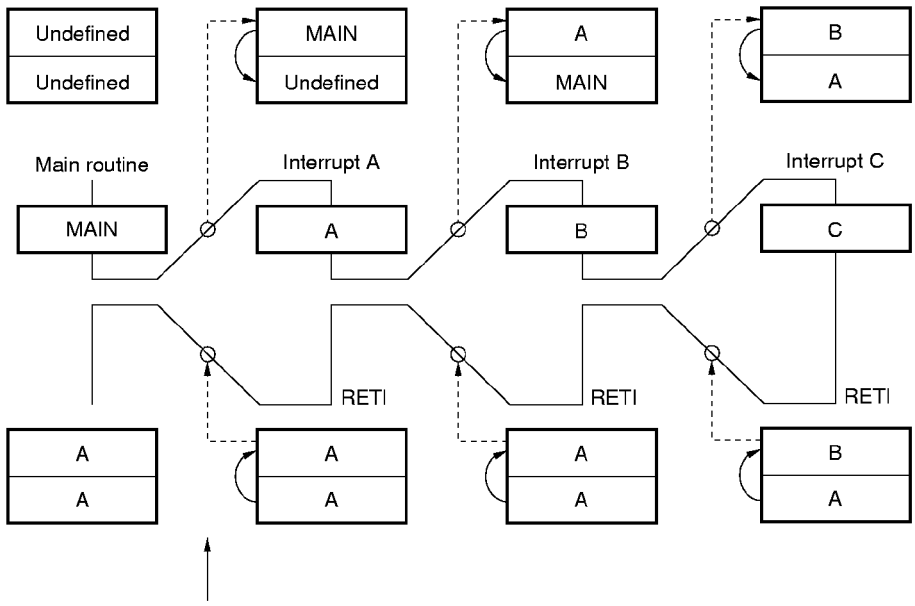
The restoring processing for the interrupt stack is different, when the RETI instruction is executed, from that, when the RET instruction is executed. Use the RET instruction as the last return instruction, when performing nesting, exceeding the maximum stack level for each model.

Figure 12-11. Interrupt Stack during Nesting

(a) Ordinary nesting

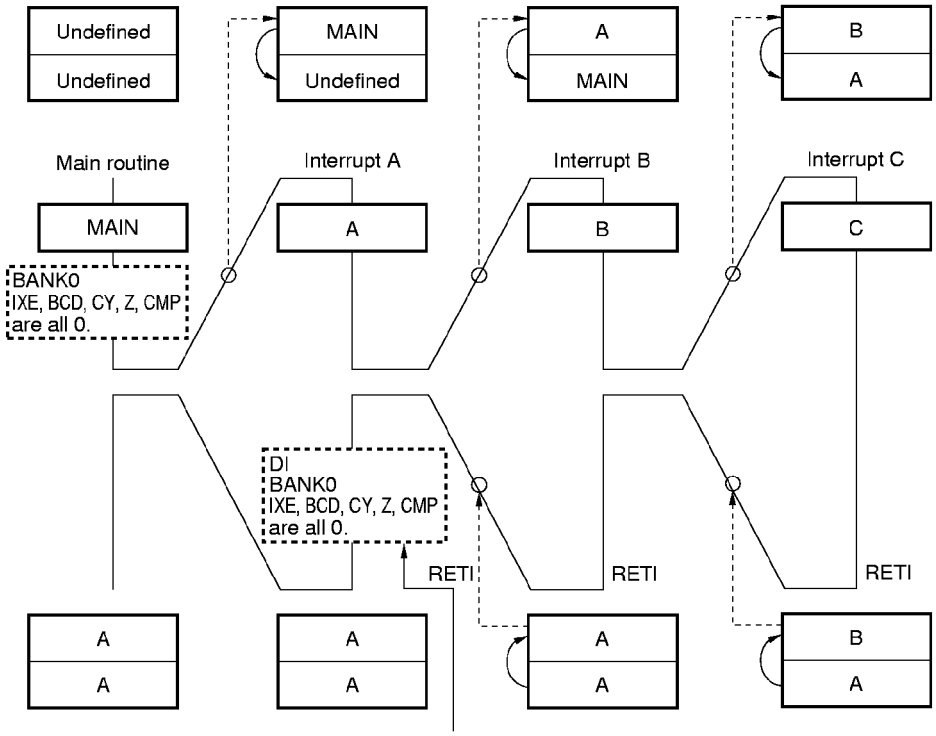


(b) Nesting exceeding maximum stack level (when interrupt stack is at level 2)



If execution is returned to main routine at this point, BANK and PSWORD of interrupt A are not correctly restored, and operation of main routine is not performed correctly.

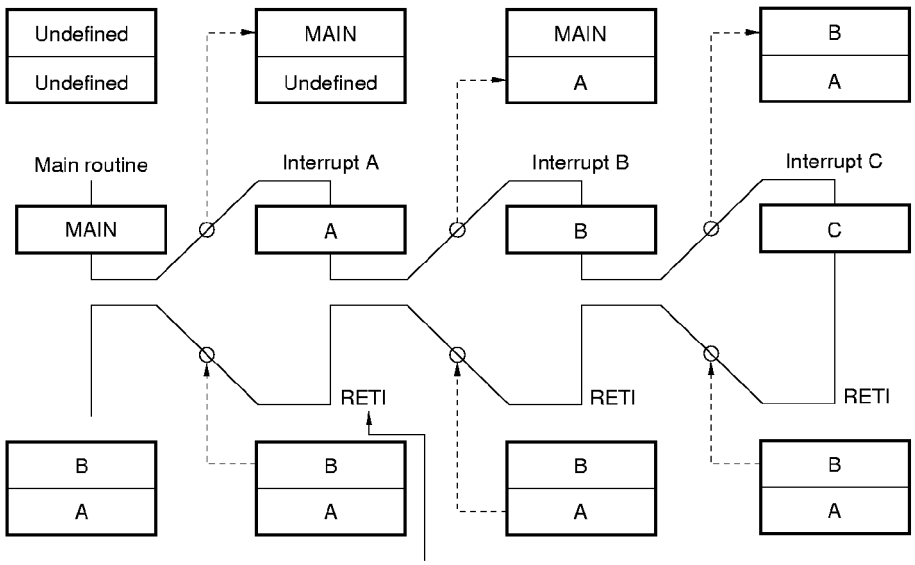
Figure 12-12. Example showing Nesting Exceeding Maximum Stack Level (when interrupt stack is at level 2)



In the example in Figure 12-12, in the main routine where the interrupt A priority is always lower than B and C priorities, and where interrupt A is enabled, nesting for 3 levels can be implemented, if the bank register and program status word are always fixed. The RET instruction is used, after the bank register and program status word in the main routine are specified when the interrupt A processing has ended.

★ If the bank register and program status word for interrupt A are identical to those in the main routine, the RETI instruction can be used. However, the 17K series Emulator (IE-17K, IE-17K-ET) cannot debug the RETI instruction, because its operations are different from those for the actual device, as shown in Figure 12-13.

Figure 12-13. Interrupt Stack Operation, When Maximum Stack Level Is Exceeded with 17K Series Emulator (IE-17K, IE-17K-ET) Used



★ When the RETI instruction is used by the Emulator (IE-17K, IE-17K-ET), the bank register (BANK) contents and index enable flag for interrupt B are restored, as shown in Figure 12-13.

CHAPTER 13 STANDBY FUNCTIONS

The standby function is to reduce the current consumption for the device, when the program processing is stopped.

The HALT mode is to reduce the current consumption, when only some device functions, for example the clock, are operating.

The clock stop function is to reduce the current consumption, for example, when only the data memory contents are retained.

The current consumption, however, is not reduced as specified, even when the standby function is used, depending on the status for the hardware peripherals and pins. Take necessary measures, in terms of software as well as hardware, to reduce the current consumption to the rated level, while referring to the Data Sheet for each model.

13.1 Configuration of Standby Block

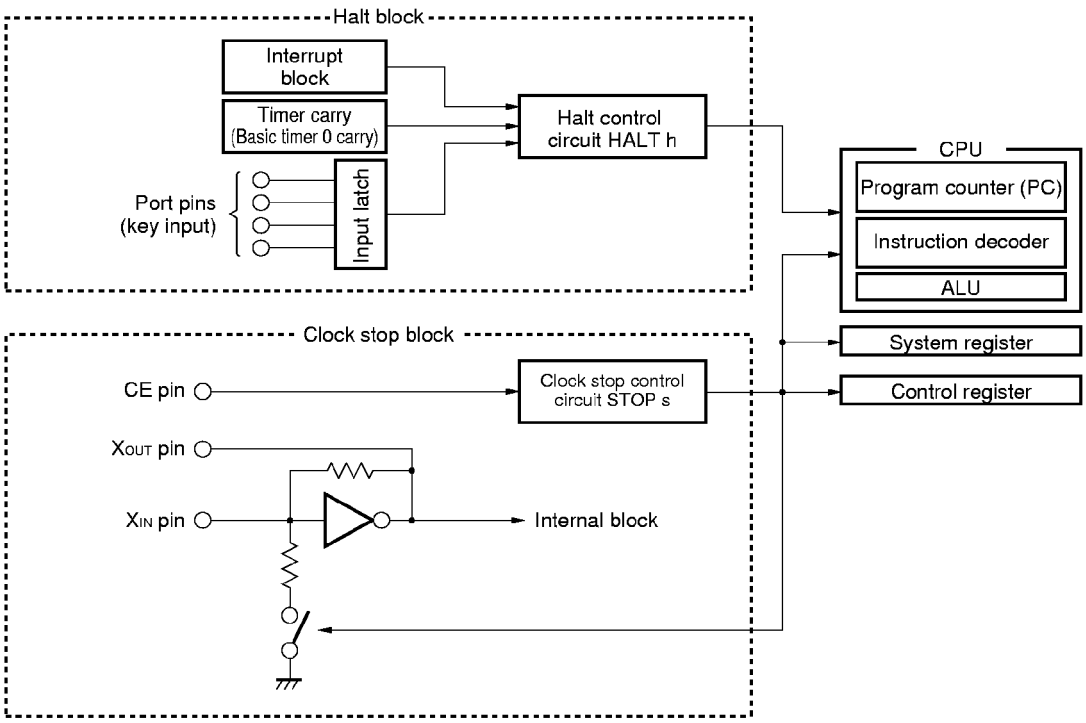
Figure 13-1 shows the configuration of the standby block.

As shown in the figure, the standby block is divided into two blocks: halt control block and clock stop control block.

The halt control block consists of a halt control circuit, interrupt control block, timer carry (basic timer 0 carry), and key input pins (port pins) and controls the operation of the CPU (program counter, instruction decoder, and ALU block).

The clock stop control block controls the operation clock oscillation circuit, CPU, system register, and control registers, by using the clock stop control circuit.

Figure 13-1. Configuration Example of Standby Block



13.2 Standby Function

The standby function reduces the current consumption of the device by stopping some or all the operations of the device.

The standby function can be used in two modes: halt and clock stop.

The halt mode is to reduce the current consumption of the device by executing a dedicated instruction "HALT h" and stopping the operation of the CPU.

The clock stop mode is to reduce the current consumption of the device by executing a dedicated instruction "STOP s" and stopping the crystal oscillation circuit.

In addition to the halt and clock stop modes, the operation mode of the device can be also set by the CE pin.

The CE pin is used to control the operation of the PLL frequency synthesizer and reset the device, and can be said to be a type of the standby function in that it controls the operation of the PLL frequency synthesizer.

13.3 explains how to set the operation mode of the device by using the CE pin.

13.4 and **13.5** respectively explain the halt and clock stop modes.

13.3 Selecting Device Operation Mode with CE Pin

The CE pin controls the following functions (1) through (3) by using the level and rising edge of an externally input signal.

(1) Controls operation of internal peripheral hardware
(2) Enables or disables clock stop instruction
(3) Resets device

13.3.1 Controlling operation of internal peripheral hardware

The hardware peripheral operates when the CE pin is at high level, and stops when it is at low level.

13.3.2 Enabling and disabling clock stop instruction

The clock stop instruction "STOP s" is enabled only when the CE pin is low.

The "STOP s" instruction is executed as a no-operation (NOP) instruction if it is executed when the CE pin is high.

13.3.3 Resetting device

The device can be reset (CE reset) by raising the CE pin.

The device can also be reset through power application (power-ON reset).

For details, refer to **CHAPTER 14 RESET FUNCTIONS**.

13.3.4 Signal input to CE pin

The CE pin does not accept a low-level or high-level signal that does not last for a certain duration, which is specified for each model, to prevent the device from malfunctioning due to noise.

Since the CE pin input level is reflected on the CE flag in the control register, it can be detected by the CE flag.

13.4 Halt Function

The halt function stops the operation clock of the CPU by executing the "HALT h" instruction.

When the "HALT h" instruction is executed, the program stops at the "HALT h" instruction, until the halt status is released later.

Therefore, the current consumption of the device can be reduced in the halt status by the operating current of the CPU.

The halt status can be released by key input, timer carry, or interrupt. The releasing condition of the key input, timer carry (basic timer 0 carry), and interrupt is specified by the operand "h" of the "HALT h" instruction.

The conditions specified by "h", under which that the halt mode is released, differ depending on the model. Refer to the Data Sheet for the model used.

The "HALT h" instruction is valid regardless of the input level of the CE pin.

13.4.1 through **13.4.6** explain the halt status, halt release condition, and each halt release condition.

13.4.1 Halt status

All the operations of the CPU are stopped in the halt status.

In other words, program execution is stopped at the "HALT h" instruction.

However, the peripheral hardware units continue the operations set before the "HALT h" instruction is executed.

For details, refer to the Data Sheet for the model used.

13.4.2 Halt release condition

Figure 13-2 shows the halt release conditions.

As shown in this figure, the halt release conditions are set by 4-bit data specified by operand "h" of the "HALT h" instruction.

The halt status is released when the condition specified as "1" by operand "h" is satisfied.

When the halt status is released, the execution starts from the instruction next to the "HALT h" instruction.

★ If two or more release conditions are specified, and if any one of the specified conditions is satisfied, the halt condition is released.

If the device is reset (power-ON reset or CE reset), the halt status is released, and each reset operation is performed.

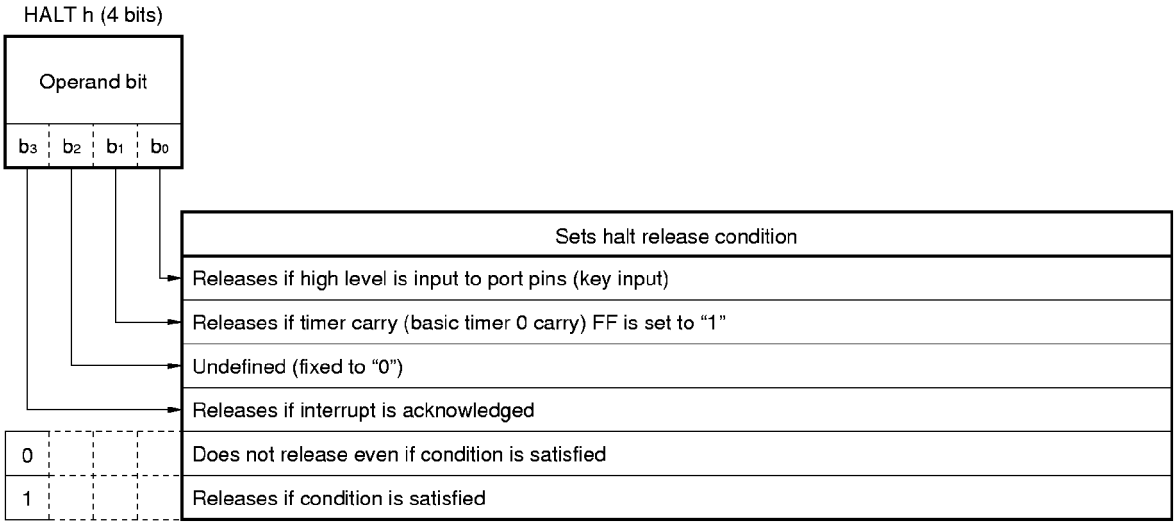
If 0000B is set as the halt release condition "h", no release condition is set.

At this time, the halt status is released if the device is reset (power-ON reset or CE reset).

13.4.3 through **13.4.5** explain halt release conditions set by key input, timer carry (basic timer 0 carry), and interrupt.

13.4.6 shows an example when two or more release conditions are specified.

Figure 13-2. Halt Release Condition



13.4.3 Releasing halt status by key input

Releasing the halt status by key input is specified by the "HALT 0001B".

If releasing the halt status by key input is specified, the halt status is released when a high level is input to any of the port 0D pins.

★ **Remark** The shared pins of port 0D pin differ depending on the model used. When using shared functions, or using a general-purpose output port as a key source signel, refer to the cautions in the Data Sheet for each model.

13.4.4 Releasing halt status by timer carry (basic timer 0 carry)

Releasing the halt status by the timer carry (basic timer 0 carry) is set by the "HALT 0010B".

When the release of the halt status is set by the timer carry (basic timer 0 carry), the halt status is released as soon as the timer carry (basic timer 0 carry) FF has been set to "1".

The timer carry (basic timer 0 carry) FF corresponds to the dedicated flag in the control register, and is set to "1" at fixed time intervals.

Therefore, the halt status can be released at fixed time intervals.

★ **Remark** The set time of the timer carry (basic timer 0 carry) differs depending on the model used. Refer to the Data Sheet of each model.

13.4.5 Releasing halt status by interrupt

Releasing the halt status by an interrupt is set by the "HALT 1000B".

If releasing the halt status by an interrupt is set, the halt status is released as soon as the interrupt has been acknowledged.

Therefore, the interrupt source to be used to release the halt status must be specified by program in advance if two or more interrupt sources exist.

So that the interrupt is acknowledged, all the interrupts must be enabled (by the EI instruction), each interrupt is enabled (by setting the corresponding interrupt permission flag), in addition that the interrupt request from each interrupt source must be issued.

Even if an interrupt request is issued, if that interrupt is not enabled, the interrupt is not acknowledged and the halt status is not released.

When the halt status has been released because the interrupt has been acknowledged, the program flow branches to the vector address of the interrupt.

If the "RETI" instruction is executed after the interrupt processing, the program flow returns to the instruction next to the "HALT" instruction.

★ **Remark** The interrupt sources differ depending on the model used. Refer to the Data Sheet of each model.

★ **Caution When executing the HALT instruction which is to be released if the interrupt request flag (IRQ×××) for which the interrupt permission flag (IP×××) is set is set, describe a NOP instruction immediately before the HALT instruction.**

If a NOP instruction is described immediately before the HALT instruction, a time of one instruction is generated in between the IRQ××× manipulation instruction and HALT instruction. In the case of the CLR1 IRQ××× instruction, for example, clearing IRQ××× is correctly reflected on the HALT instruction (refer to Example 1 below). If a NOP instruction is not described immediately before the HALT instruction, the CLR1 IRQ××× instruction is not correctly reflected on the HALT instruction, and the HALT mode is not set (refer to Example 2 below).

Examples 1. Program that correctly executes HALT instruction

```
        ⋮                   ; Sets IRQ×××
CLR1    IRQ×××
NOP                         ; Describes NOP instruction immediately before HALT instruction
                            ; (clearing IRQ××× is correctly reflected on HALT instruction)
HALT    1000B               ; Correctly executes HALT instruction (HALT mode is set)
        ⋮
```

2. Program that does not set HALT mode

```
        ⋮                   ; Sets IRQ×××
CLR1    IRQ×××              ; Clearing IRQ××× is not reflected on HALT instruction
                            ; (but on instruction next to HALT)
HALT    1000B               ; HALT instruction is ignored (HALT mode is not set)
        ⋮
```

13.4.6 If two or more release conditions are simultaneously set

If two or more release conditions are simultaneously set, and if even one of the conditions is satisfied, the halt status is released.

The method to identify the release condition that is satisfied when two or more release conditions are specified is shown below.

★ **Examples 1.**

```
        HLTINT    DAT   1000B
        HLTTMR    DAT   0010B
        HLTKEY    DAT   0001B
        INTPIN    DAT   0004H          ; INT pin interrupt vector address symbol
                                       ; definition
START:
        BR        MAIN
ORG     INTPIN
        Processing A                   ; INT pin interrupt processing
        EI
        RETI
TMRUP                                  ; Timer carry processing
        Processing B
        RET
KEYDEC:                                ; Key input processing
        Processing C
        RET
MAIN:
        SET2      TMMD1, TMMD0         ; Embedded macro
                                       ; Sets timer carry FF setting time to 1 ms
        SET1      IP                   ; Embedded macro
                                       ; Enables INT pin interrupt
        EI
LOOP:
        HALT      HLTINT OR HLTTMR OR HLTKEY
                                       ; Specifies interrupt, timer carry, and key input
                                       ; as halt release conditions
        SKF1      TMCY                 ; Embedded macro
                                       ; Detects TMCY flag
        CALL      TMRUP                ; Timer carry processing if set to "1"
        SKF4      P0D3,P0D2,P0D1,P0D0  ; Detects key input
                                       ; Detects key input latch
        CALL      KEYDEC               ; Key input processing if latched
        BR        LOOP
```

In **Example 1**, three halt release conditions are specified: INT pin interrupt, 1-ms timer carry, and key input.

To detect which condition has caused the halt status to be released, the vector address, TMCY flag, and P0D flag are detected to identify the interrupt, timer carry, and key input, respectively.

When using two or more release conditions, the following two points must be noted.

(1) All the specified release conditions must be detected when the halt status has been released.
(2) The conditions must be sequentially detected starting from the one with the highest priority.

For example, if the program below "MAIN:" in example 1 is as shown in **Example 2** below, care must be exercised. Do not develop the program as shown in **Example 2** if the timer carry has a high priority.

★ **Remark** The flag names may differ depending on the model used. Refer to the Data Sheet of each model.

Examples 2.

```
MAIN:
        SET4    P1C3, P1C2, P1C1, P1C0      ; Uses general-purpose output port as key
        SET2    TMMD1, TMMD0                ; source signal
        SET1    IP
        EI
 LOOP:
        HALT    HLTINT OR HLTTMR OR HLTKEY
        SKF4    P0D3, P0D2, P0D1, P0D0      ; Detects key input
        BR      KEYDEC
        SKF1    TMCY
        CALL    TMRUP
        BR      LOOP
KEYDEC:                                     ; Key input processing
        ┌──────────────┐
        │ Processing C │
        └──────────────┘
        BR      LOOP
```

In **Example 2**, suppose the timer carry FF is set to "1" immediately after the halt status has been released by key input.

Then the program executes the "HALT" instruction again after executing the key input processing.

Because the timer carry FF remains set at this time, the halt status is immediately released.

Usually, however, a high level is input for about 100 ms as key input. Consequently, execution further branches to the key input processing.

As a result, the timer carry FF is not correctly detected.

★ **Remark** The flag names may differ depending on the model used. Refer to the Data Sheet of each model.

13.5 Clock Stop Function

The clock stop function stops the crystal oscillation circuit by executing the "STOP s" instruction (clock stop status). Therefore, the current consumption of the device is decreased to the minimum value.

The current consumption differs depending on the model used. Refer to the Data Sheet of each model.

Specify "0000B" as operand "s" of the "STOP s" instruction.

The "STOP s" instruction is valid only while the CE pin is low.

It is executed as a no-operation (NOP) instruction even when executed while the CE pin is high.

In other words, the "STOP s" instruction must be executed while the CE pin is low.

The clock stop status is released by raising the level of the CE pin from low to high (CE reset).

13.5.1 through **13.5.3** explain the clock stop status, how to release the clock stop status, and notes on using the clock stop instruction.

13.5.1 Clock stop status

Because the crystal oscillation circuit is stopped in the clock stop status, all the device operations, such as those of the CPU and peripheral hardware, are stopped.

For the operations of the CPU and peripheral hardware, refer to the Data Sheet of each model.

The power failure detection circuit does not operate in the clock stop status even if the supply voltage V_{DD} of the device is lowered to 2.2 V. Therefore, the data memory can be backed up at a low voltage. For the details on the power failure detection circuit, refer to **CHAPTER 14 RESET FUNCTIONS**.

13.5.2 Releasing clock stop status

The clock stop status is released either by raising the level of the CE pin from low to high (CE reset), or by lowering the supply voltage V_{DD} of the device to 2.2 V or less once, and then increasing it to 4.5 V (power-ON reset).

Figures 13-3 and 13-4 respectively show how the clock stop is released on CE reset and power-ON reset.

If the clock stop status is released by power-ON reset, the power failure detection circuit operates.

For the details on power-ON reset, refer to **14.4 Power-ON Reset**.

Figure 13-3. Releasing Clock Stop Status by CE Reset

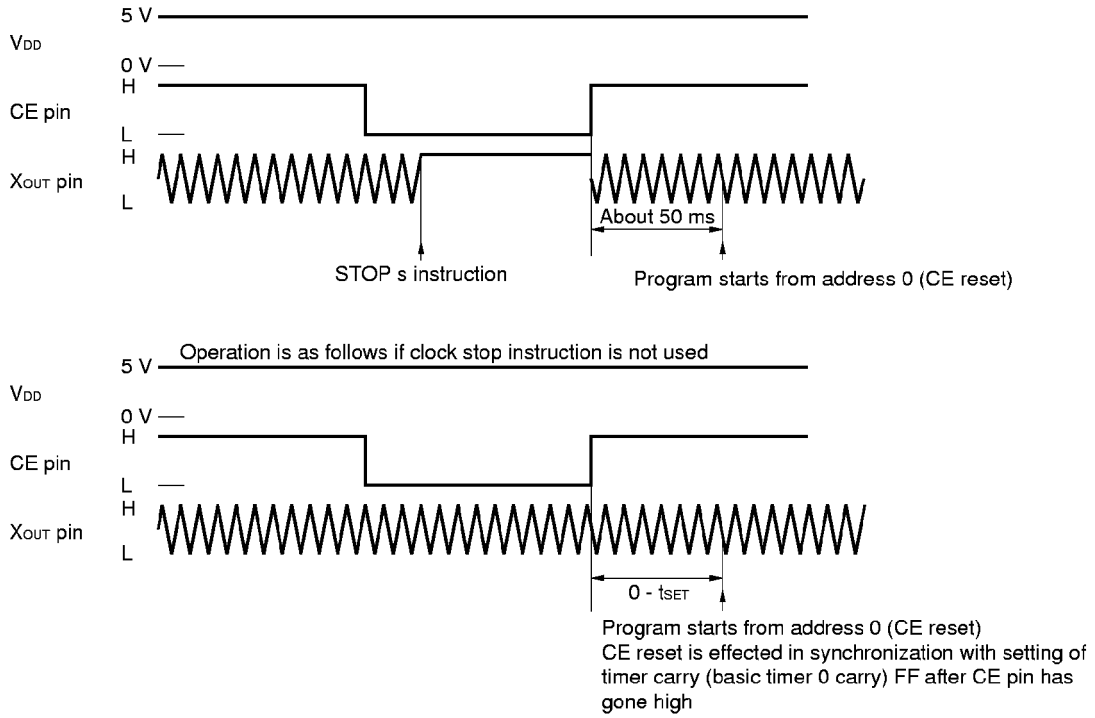
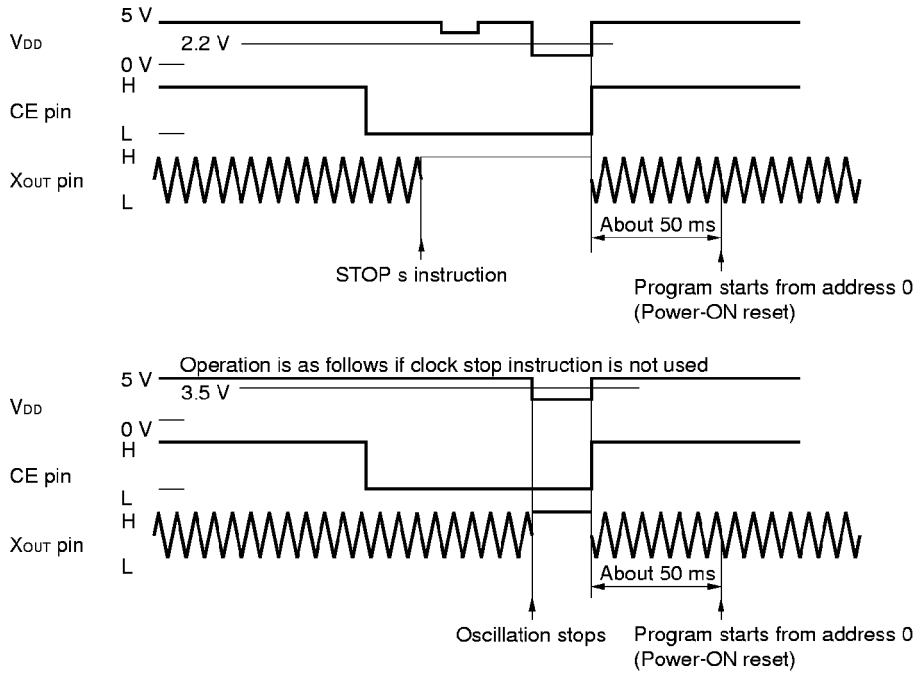


Figure 13-4. Releasing Clock Stop Status by Power-ON Reset



13.5.3 Troubles occurring as result of executing clock stop instruction, when CE pin is high, and remedies therefor

The clock stop instruction (STOP s) is valid, when the CE pin is at the low level.

Therefore, it is necessary to design processing that can be executed, when the CE pin happens to be at the high level.

In the following program example, the CE pin status is detected in <1>. If it is at a low level, the clock stop instruction "STOP XTAL" in <2> is executed, after processing A has been performed.

However, if the CE pin goes high, while the STOP XTAL instruction is being executed, as shown in Figure 13-5, the instruction is treated as a no-operation (NOP) instruction.

At this time, assuming that the branch instruction "BR $-1" in <3> is missing, the program execution enters the main processing, and malfunctioning may take place.

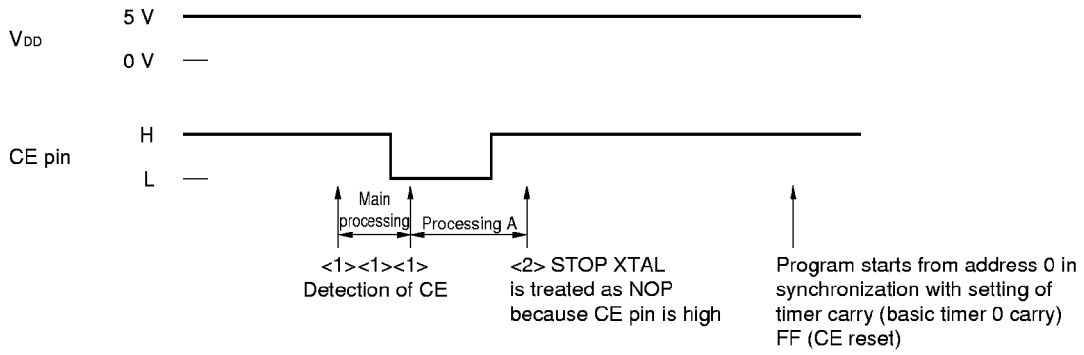
Therefore, either insert the branch instruction, as shown in <3>, or the program must be designed so that no malfunction takes place, even after the execution enters the main processing.

When a branch instruction is used as in <3>, CE reset is effected in synchronization with the next setting for the timer carry (basic timer 0 carry) FF, as shown in Figure 13-5, even when the CE pin level remains high.

Example

```
        XTAL    DAT     0000B   ; Symbol definition for clock stop condition
CEJDG:
    ; <1>
        SKF1    CE              ; Macroinstruction
                                ; Detects input level for CE pin
        BR      MAIN            ; Branches to main processing if CE = high level
        ┌──────────┐
        │Processing A│          ; Processing, when CE = low
        └──────────┘
    ; <2>
        STOP    XTAL            ; Clock stops
    ; <3>
        BR      $-1
MAIN:
        ┌──────────────┐
        │Main processing│
        └──────────────┘
        BR      CEJDG
```

Figure 13-5. Malfunctioning in Clock Stop Instruction, Due to CE Pin Input, and Remedy



CHAPTER 14 RESET FUNCTIONS

The reset functions are to initialize the device operation and are divided into power-ON reset and CE reset functions. Which of the two reset functions is to be effected is determined by the program, after the operation has been started, and an appropriate initialization program is executed.

The operation performed by the hardware and program, to determine which reset function is to be implemented, is called power failure detection.

★ Power-ON reset is performed when the supply voltage V_{DD} of the device is lowered from a specific level. At power-ON reset, all the contents of the internal registers that can be rewritten are initialized.

The CE reset function is to resume operations, after data are retained at a low voltage in the range permitted by the specifications in the clock stop status. Usually, therefore, the data memory contents are not rewritten, and the main routine execution is resumed, after the major registers have been initialized.

★ **Remark** The names of the timers and flags used for reset differ depending on the product. Refer to the Data Sheet of each model (in this chapter, timer carry, TMCY flag, and TMMD0 and 1 flags are used).

- Timer carry ↔ Basic timer 0 carry
- TMCY flag ↔ BTM0CY flag
- TMMD0, TMMD1, TMMD2, TMMD3 flags ↔ BTM0CY0, BTM0CY1, BTM1CY0, BTM1CY1 flags

14.1 Configuration of Reset Block

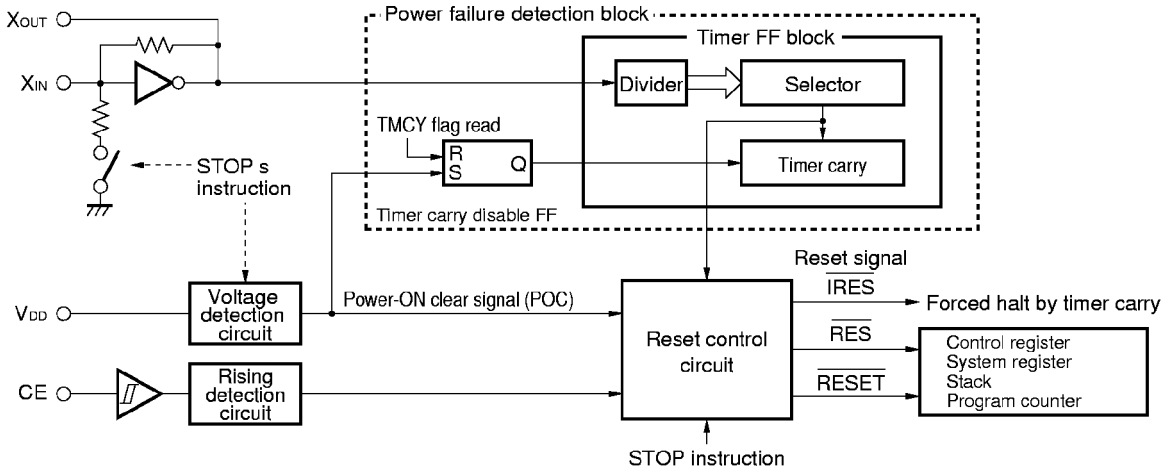
Figure 14-1 shows the configuration of the reset block.

The device is reset in two ways: by applying supply voltage V_{DD} (power-ON reset or V_{DD} reset) and by using the CE pin (CE reset).

The power-ON reset block consists of a voltage detection circuit that detects a voltage input to the V_{DD} pin, a power failure detection circuit, and a reset control circuit.

The CE reset block consists of a circuit that detects the rising of a signal input to the CE pin, and a reset control circuit.

Figure 14-1. Configuration Example of Reset Block



14.2 Reset Function

Power-ON reset is effected when supply voltage V_{DD} rises from a specific level, and CE reset is effected when the CE pin goes high.

Power-ON reset initializes the program counter, stack, system register, and control registers, and executes the program from address 0000H.

CE reset initializes the program counter, stack, system register, and some control registers, and executes the program from address 0000H.

The major differences between power-ON reset and CE reset are the control registers that are initialized and the operation of the power failure detection circuit that is explained in **14.6**.

Both power-ON reset and CE reset are controlled by the reset signals $\overline{IRES}$, $\overline{RES}$, and $\overline{RESET}$ output from the reset control circuit shown in Figure 14-1.

Table 14-1 shows the relation among the $\overline{IRES}$, $\overline{RES}$, and $\overline{RESET}$ signals, and power-ON reset, and CE reset.

The reset control circuit also operates when the clock stop instruction (STOP s) explained in **CHAPTER 13 STANDBY FUNCTIONS** is executed.

14.3 and **14.4** respectively explain CE reset and power-ON reset.

14.5 explains the relation between CE reset and power-ON reset.

Table 14-1. Relation between Internal Reset Signals and Each Reset Operation

Internal Reset Signal	Output Signal			Control Operation by Each Reset Signal
	CE Reset	Power-ON Reset	Clock Stop	
$\overline{IRES}$	×	○	○	Forcibly sets device in halt status. Halt status is released when timer carry FF is set.
$\overline{RES}$	×	○	○	Initializes some control registers.
$\overline{RESET}$	○	○	○	Initializes program counter, stack, system register, and some control registers.

14.3 CE Reset

CE reset is effected when the CE pin goes high.

When the CE pin goes high, the $\overline{\text{RESET}}$ signal is output in synchronization with the rising edge of the next timer carry FF setting pulse, and the device is reset.

When CE reset is effected, the $\overline{\text{RESET}}$ signal initializes the program counter, stack, system register, and some control registers, and the program is executed starting from address 0000H.

For the value to which each of the above registers is initialized, refer to the Data Sheet of each model.

The operation of CE reset differs depending on whether the clock stop instruction is used.

The differences in operation are explained in **14.3.1** and **14.3.2**.

14.3.3 explains the points to be noted on using CE reset.

14.3.1 CE reset when clock stop (STOP s) instruction is not used

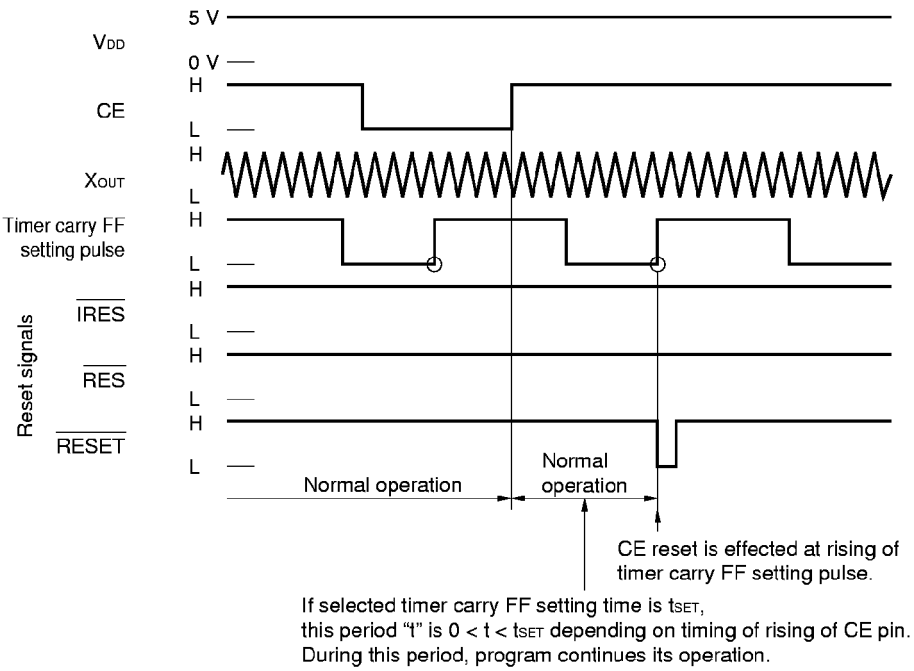
Figure 14-2 shows the operation of CE reset when the clock stop (STOP s) instruction is not used.

When the STOP s instruction is not used, the timer mode select register of the control registers is not initialized.

After the CE pin has gone high, therefore, the $\overline{\text{RESET}}$ signal is output at the rising edge of the timer carry FF setting pulse selected at that time, and the device is reset.

★ **Remark** The timer mode select register is sometimes referred to as the basic timer clock select register depending on the model. Refer to the Data Sheet of each model.

Figure 14-2. CE Reset Operation When Clock Stop Instruction Is Not Used



14.3.2 CE reset when clock stop (STOP s) instruction is used

Figure 14-3 shows the operation of CE reset when the clock stop (STOP s) instruction is used.

When the STOP s instruction is used, the $\overline{\text{IRES}}$, $\overline{\text{RES}}$, and $\overline{\text{RESET}}$ signals are output as soon as the STOP s instruction has been executed.

At this time, the timer mode select register of the control registers is initialized to 0000B by the $\overline{\text{RES}}$ signal, the timer carry FF setting signal is set to 100 ms.

Because the $\overline{\text{IRES}}$ signal is output while the CE pin is low, the halt status, which can be released by the timer carry, is forcibly set.

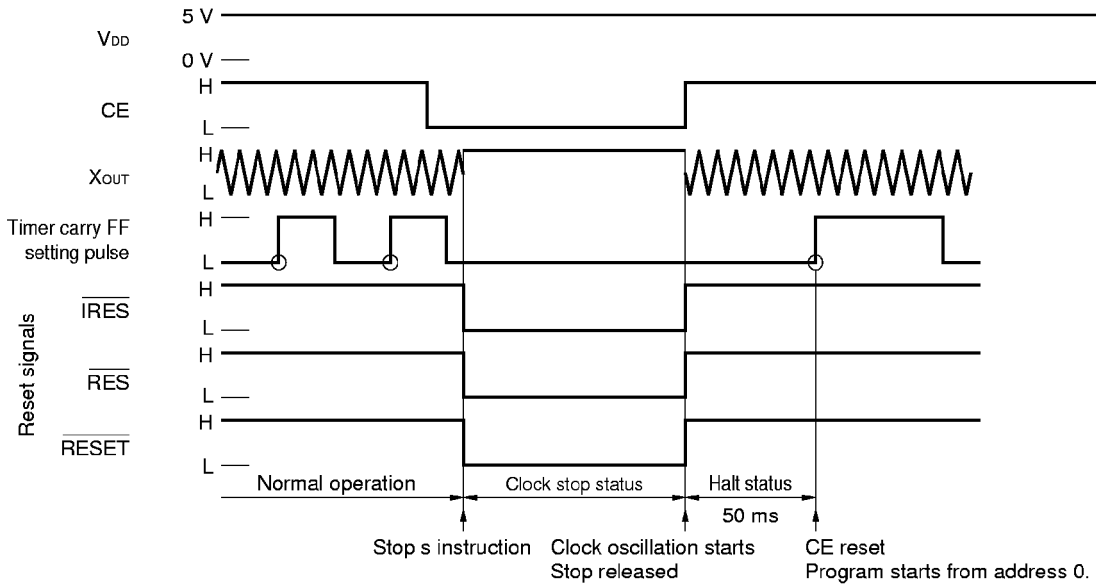
However, the device stops operation because the clock is stopped.

When the CE pin goes high, the clock stop status is released, and oscillation starts.

Because the halt status that can be released by the timer carry is set at this time by the $\overline{\text{IRES}}$ signal, the program starts from address 0 when the CE pin goes high and then the timer carry FF setting pulse rises.

Because the timer carry FF setting pulse is initialized to 100 ms, CE reset is effected 50 ms after the CE pin has gone high.

Figure 14-3. CE Reset Operation When Clock Stop Instruction Is Used



14.3.3 Notes on CE reset

Because CE reset is effected regardless of the instruction under execution, the following points <1> and <2> must be noted.

(1) Time to execute timer processing such as watch

When developing a watch program by using the timer carry or timer interrupt, the processing of that program must be completed in specific time.

For details, refer to the Data Sheet of each model.

(2) Processing of data and flag used for program

Care must be exercised in rewriting the contents of data or flag that cannot be processed with one instruction and whose contents must not change even when CE reset is effected, such as a security code.

This is explained in detail by using the following examples.

★ **Examples 1.**

```
R1      MEM     0.01H           ; First digit of key input data of security code
R2      MEM     0.02H           ; Second digit of key input data of security code
R3      MEM     0.03H           ; First digit data for changing security code
R4      MEM     0.04H           ; Second digit data for changing security code
M1      MEM     0.11H           ; First digit of current security code
M2      MEM     0.12H           ; Second digit of current security code

START:

        Key input processing
            R1 ← contents of key A  ; Security code input wait mode
            R2 ← contents of key B  ; Substitutes contents of pressed key into R1 and R2.

        SET2    CMP, Z          ;<1>; Compares security code with input data.
        SUB     R1, M1
        SUB     R2, M2
        SKT1    Z
        BR      ERROR           ; Input data is different from security code.
MAIN:

        Key input processing
            R3 ← contents of key C  ; Security code rewriting mode
            R4 ← contents of key D  ; Substitutes contents of pressed key into R3 and R4.

        ST      M1, R3          ;<2>; Rewrites security code.
        ST      M2, R4          ;<3>
        BR      MAIN
ERROR:

        Must not operate
```

Suppose the current security code is "12H" in the above program, the contents of data memory areas M1 and M2 are "1H" and "2H", respectively.

If CE reset is effected, the contents of key input are compared with security code "12H" in <1>. If they coincide, the normal processing is performed.

If the security code is changed by the main processing, the new code is written to M1 and M2 in <2> and <3>.

Suppose the security code is changed to "34H", "3H" and "4H" are written to M1 and M2, respectively, in <2> and <3>.

If CE reset is effected at the point where <2> is executed, the program is executed from address 0000H without <3> executed.

Consequently, the security code is changed to "32H", making impossible to clear the security.

In this case, use the program shown in following **Example 2**.

★ **Examples 2.**

```
R1      MEM     0.01H           ; First digit of key input data of security code
R1      MEM     0.01H           ; First digit of key input data of security code
R2      MEM     0.02H           ; Second digit of key input data of security code
R3      MEM     0.03H           ; First digit data for changing security code
R4      MEM     0.04H           ; Second digit data for changing security code
M1      MEM     0.11H           ; First digit of current security code
M2      MEM     0.12H           ; Second digit of current security code
CHANGE  FLG     0.13H.0         ; "1" while security code is changed

START:

        Key input processing
          R1 ← contents of key A  ; Security code input wait mode
          R2 ← contents of key B  ; Substitutes contents of pressed key into R1 and R2.

        SKT1    CHANGE      ;<4>; If CHANGE flag is "1"
        BR      SECURITY_CHK
        ST      M1, R3              ; rewrites M1 and M2.
        ST      M2, R4
        CLR1    CHANGE
SECURITY_CHK:
        SET2    CMP, Z      ;<1>; Compares security code with input data.
        SUB     R1, M1
        SUB     R2, M2
        SKT1    Z
        BR      ERROR               ; Input data is different from security code.
MAIN:

        Key input processing
          R3 ← contents of key C  ; Security code rewriting mode
          R4 ← contents of key D  ; Substitutes contents of pressed key into R3 and R4.

        SET1    CHANGE      ;<5>; Until security code is changed
                                ; Sets CHANGE flag to "1".
        ST      M1, R3      ;<2>; Rewrites security code
        ST      M2, R4      ;<3>
        CLR1    CHANGE              ; When security code has been changed, sets
                                    ; CHANGE flag to "0".
        BR      MAIN
ERROR:

        Must not operate
```

In the program in **Example 2**, the CHANGE flag is set to "1" in <5> before the security code is changed in <2> and <3>.

Therefore, the security code is rewritten in <4> even if CE reset is effected before <3> is executed.

14.4 Power-ON Reset

Power-ON reset is effected when the supply voltage V_{DD} of the device rises from a specific level (called power-ON clear voltage).

If the supply voltage V_{DD} is lower than the power-ON clear voltage, a power-ON clear signal (POC) is output from the voltage detection circuit shown in Figure 14-1.

When the power-ON clear voltage is output, the crystal oscillation circuit is stopped, and the device operation is stopped.

While the power-ON clear signal is output, the $\overline{IRES}$, $\overline{RES}$, and $\overline{RESET}$ signals are output.

If supply voltage V_{DD} exceeds the power-ON clear voltage, the power-ON clear signal is cleared, and crystal oscillation is started. At the same time, the $\overline{IRES}$, $\overline{RES}$, and $\overline{RESET}$ signals are also cleared.

At this time, the halt status is set to be released by the timer carry due to the $\overline{IRES}$ signal. Therefore, power-ON reset is effected at the rising edge of the next timer carry FF setting signal.

★ The timer carry FF setting signal is initialized to 100 ms by the $\overline{RESET}$ signal. For this reason, reset is effected 50 ms after supply voltage V_{DD} has exceeded the power-ON clear voltage, and the program is started from address 0.

This operation is illustrated in Figure 14-4.

The program counter, stack, system register, and control registers are initialized as soon as the power-ON clear signal has been output.

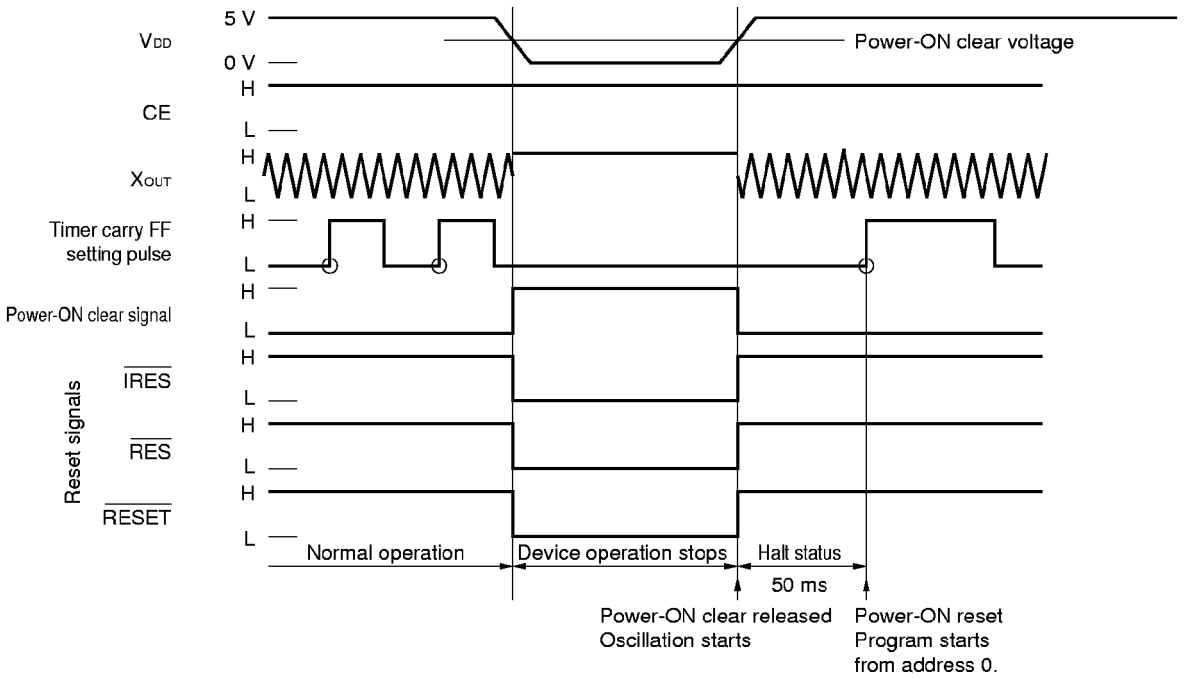
For the value to which each of the above registers is to be initialized, refer to the Data Sheet of each model.

The power-ON clear voltage is 3.5 V (rated value) during normal operation, and 2.2 V (rated value) in the clock stop status.

The operations performed when the power-ON clear voltage is at the respective levels are explained in **14.4.1** and **14.4.2**.

The operation to be performed if the supply voltage V_{DD} rises from 0 V is explained in **14.4.3**.

Figure 14-4. Operation of Power-ON Reset



14.4.1 Power-ON reset during normal operation

Figure 14-5 (a) shows the operation.

As shown in the figure, the power-ON clear signal is output and the device operation stops regardless of the input level of the CE pin, if the supply voltage V_{DD} drops below 3.5 V.

If V_{DD} rises beyond 3.5 V again, the program starts from address 0000H after 50 ms of halt status.

The "normal operation" is when the clock stop instruction is not used and includes the halt status that is set by the halt instruction.

14.4.2 Power-ON reset in clock stop status

Figure 14-5 (b) shows the operation.

As shown in the figure, the power-ON clear signal is output and the device operation stops if supply voltage V_{DD} drops below 2.2 V.

However, it seems as if the device operation were not changed because the device is in the clock stop status.

When supply voltage V_{DD} rise beyond 3.5 V next time, the program starts from address 0000H after 50 ms of halt status.

14.4.3 Power-ON reset when supply voltage V_{DD} rises from 0 V

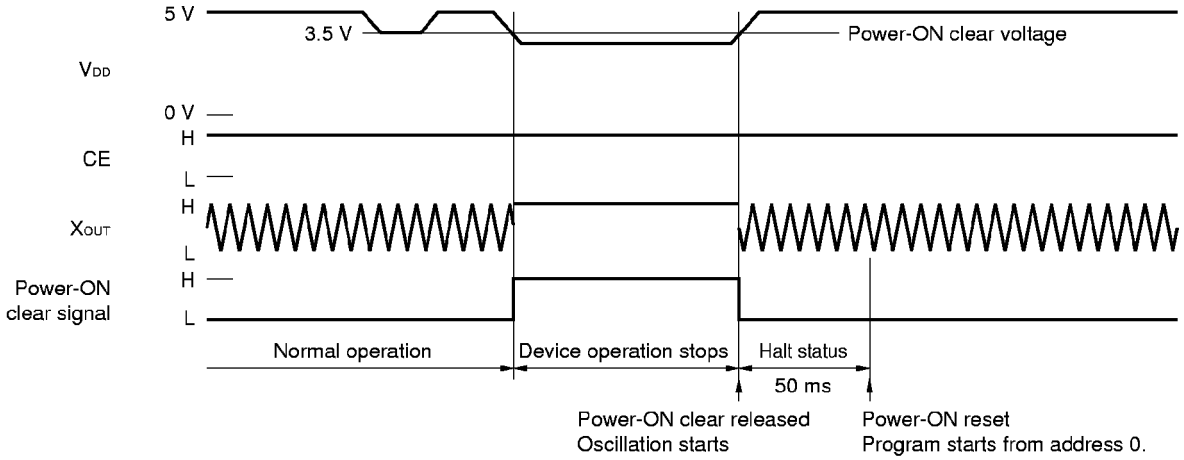
Figure 14-5 (c) shows the operation.

As shown in the figure, the power-ON clear signal is output until supply voltage V_{DD} rises from 0 V to 3.5 V.

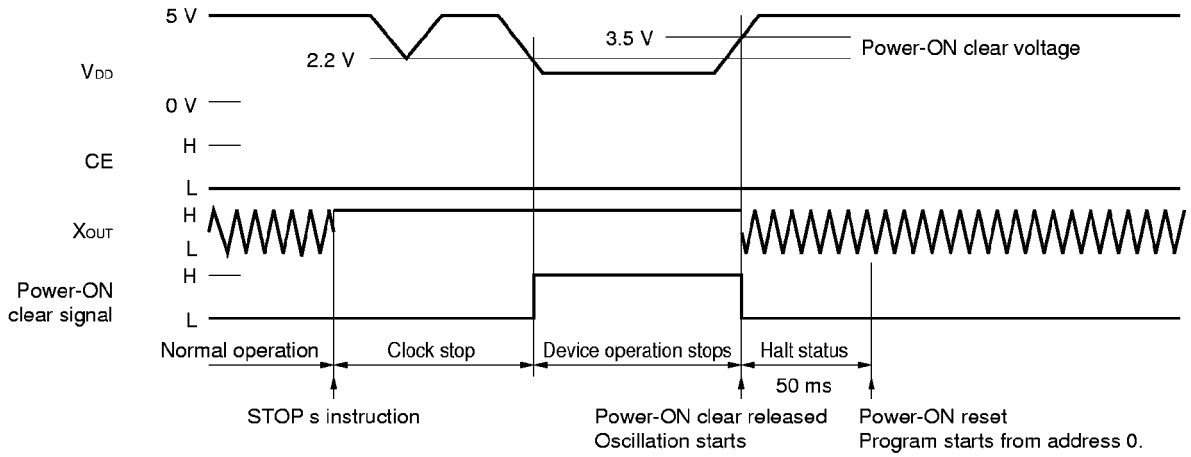
When V_{DD} rises beyond the power-ON clear voltage, the crystal oscillation circuit starts operating, and the program starts from address 0000H after 50 ms of halt status.

Figure 14-5. Power-ON Reset and Supply Voltage V_{DD}

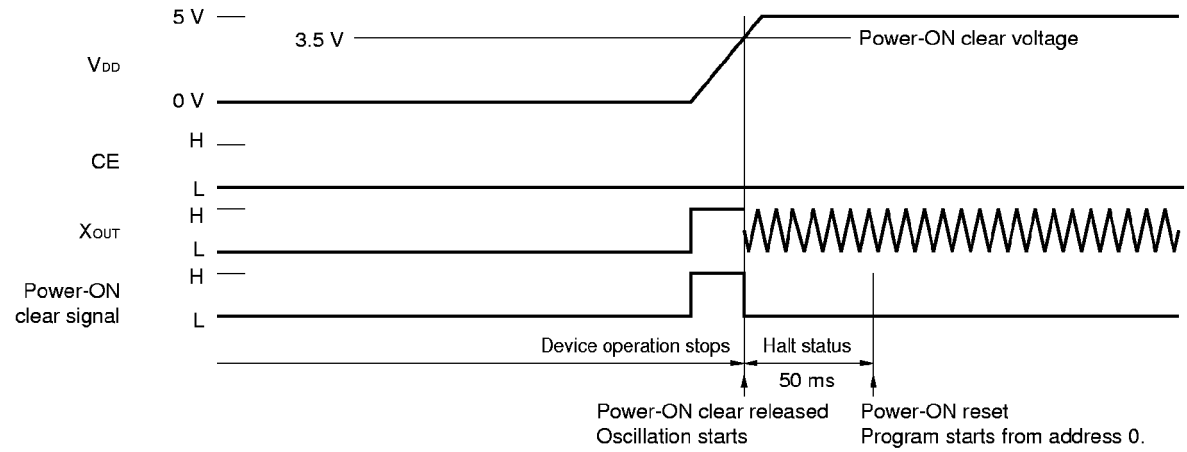
(a) During normal operation (including halt status)



(b) In clock stop status



(c) When supply voltage V_{DD} rises from 0 V



14.5 Relation between CE Reset and Power-ON Reset

There is a possibility that power-ON reset and CE reset are effected at the same time when power is first applied. The reset operations performed at this time are explained in **14.5.1** through **14.5.3**.

14.5.4 explains the points to be noted in raising supply voltage V_{DD}.

14.5.1 If V_{DD} pin and CE pin rise simultaneously

Figure 14-6 (a) shows the operation.

At this time, the program starts from address 0000H due to power-ON reset.

14.5.2 If CE pin rises in forced halt status of power-ON reset

Figure 14-6 (b) shows the operation.

At this time, the program starts from address 0000H due to power-ON reset in the same manner as in **14.5.1**.

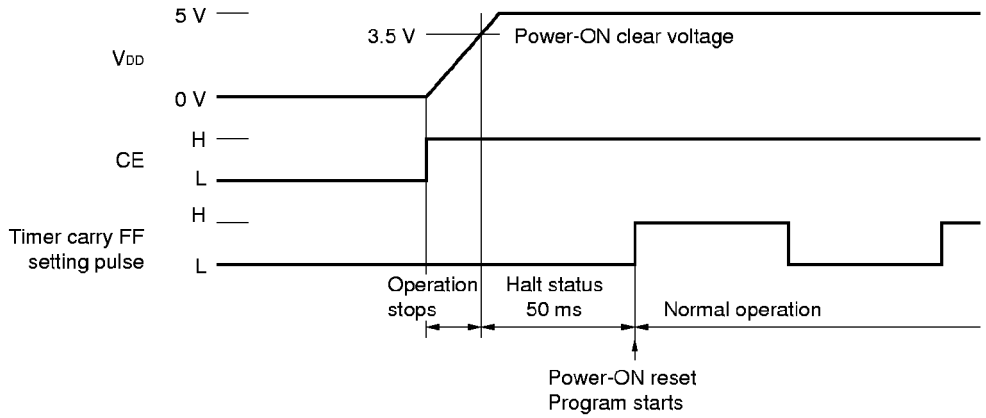
14.5.3 If CE pin rises after power-ON reset

Figure 14-6 (c) shows the operation.

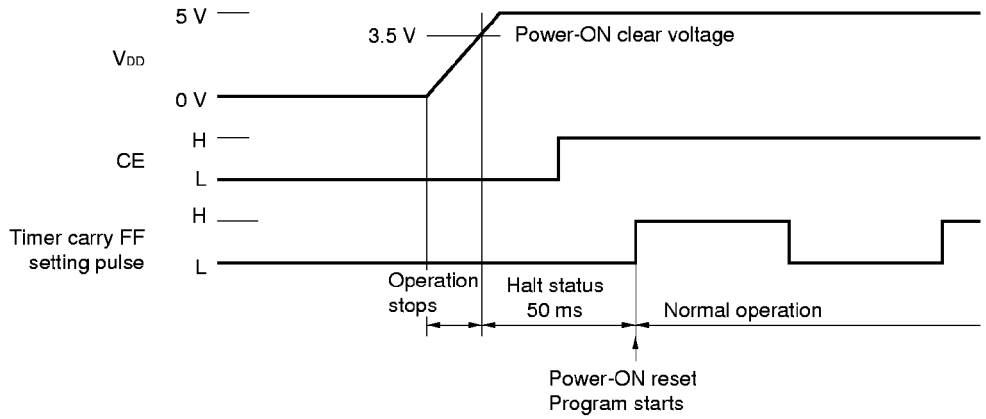
At this time, the program starts from address 0000H due to power-ON reset, and the program starts from address 0000H again at the rising of the next timer carry FF setting signal because of CE reset.

Figure 14-6. Relation between Power-ON Reset and CE Reset

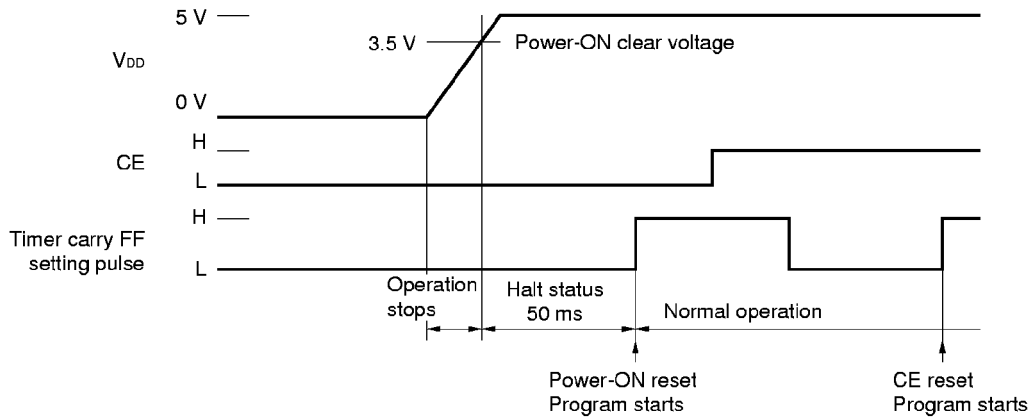
(a) If V_{DD} and CE pins rise simultaneously



(b) If CE pin rises in halt status



(c) If CE pin rises after power-ON reset



14.5.4 Notes on raising supply voltage V_{DD}

When raising supply voltage V_{DD}, keep in mind the following points (1) and (2).

(1) When raising supply voltage V_{DD} from power-ON clear voltage

It is necessary to raise supply voltage V_{DD} to higher than 3.5 V at least once.

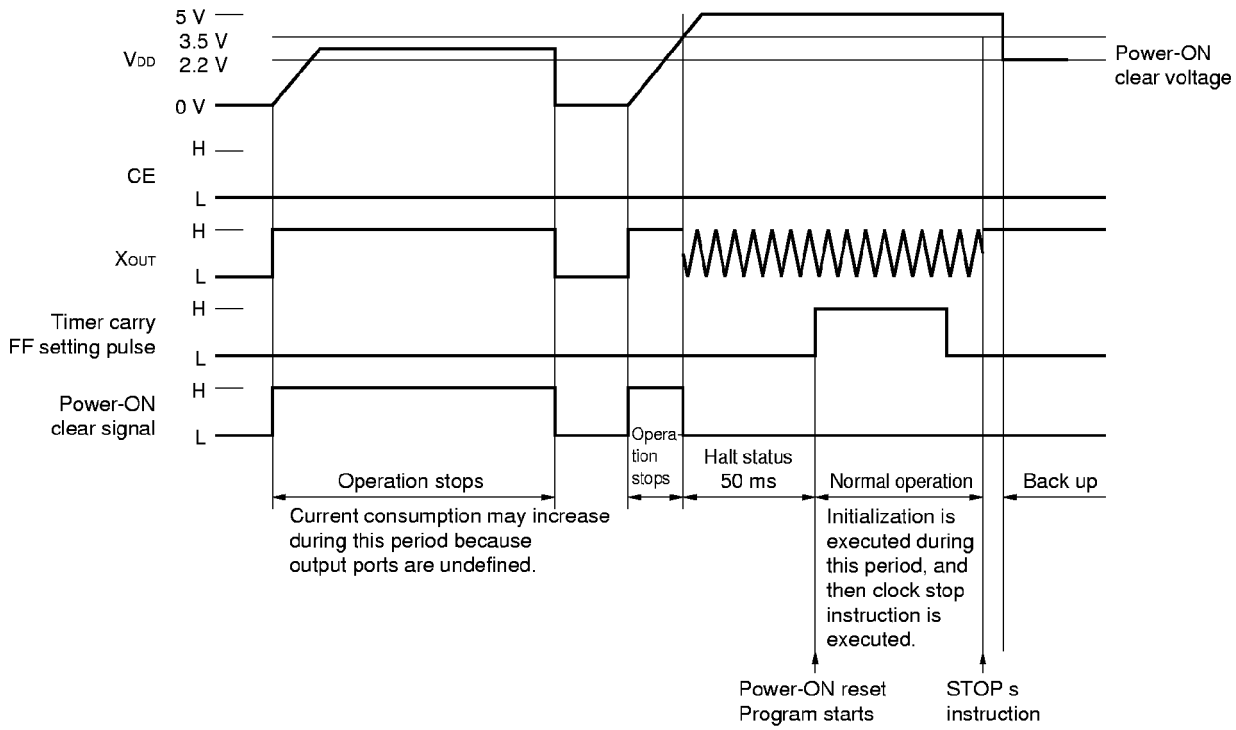
This is illustrated in Figure 14-7.

Suppose, for example, only a voltage less than 3.5 V is applied on application of V_{DD} with a program that backs up V_{DD} at 2.2 V by using the clock stop instruction, as shown in Figure 14-7, the power-ON clear signal is continuously output, and the program does not operate.

Because the output ports of the device output undefined values, the current consumption increases in some cases.

If the device is backed up by batteries, therefore, the back-up time is substantially shortened.

Figure 14-7. Notes on Raising V_{DD}



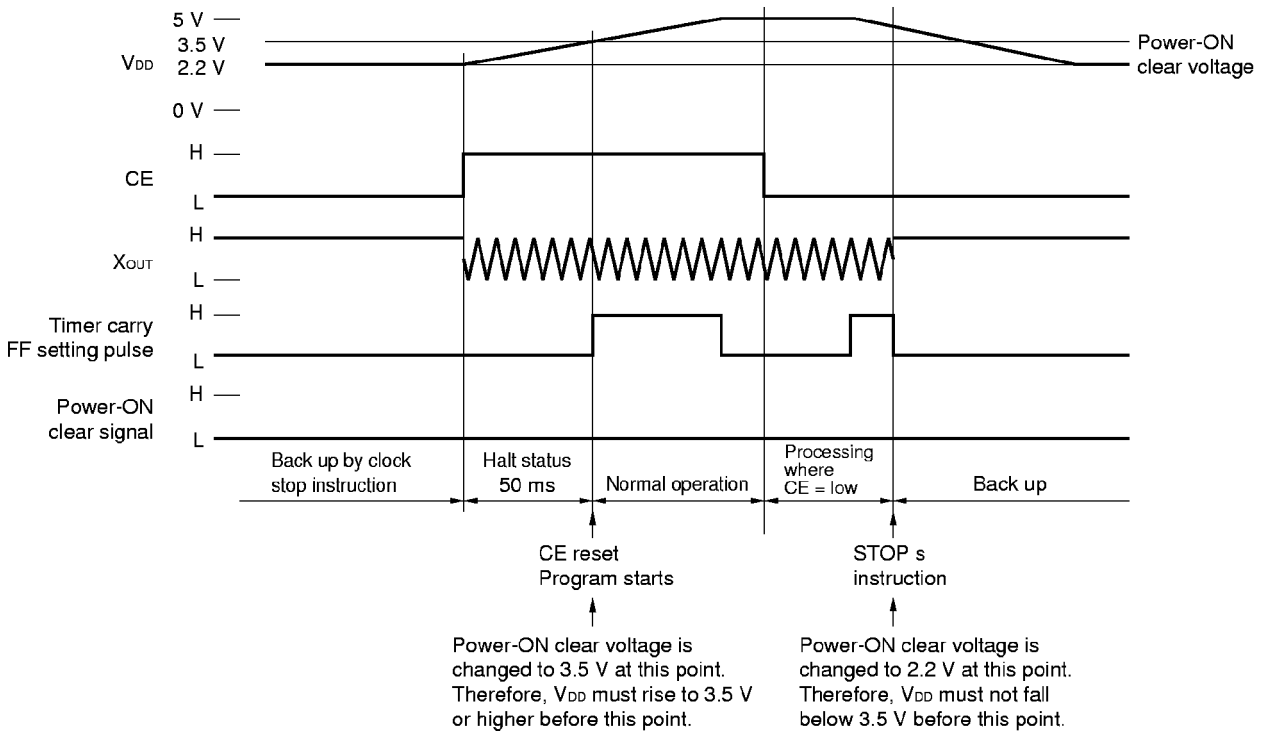
(2) Restoring from clock stop status

To restore the device from the back-up status while supply voltage V_{DD} is backed up at 2.2 V by using the clock stop instruction, V_{DD} must be raised to 3.5 V or higher within 50 ms after the CE pin has gone high.

As shown in Figure 14-8, the device is restored from the clock stop status by means of CE reset. Because the power-ON clear voltage is changed to 3.5 V 50 ms after the CE pin has gone high, power-ON reset is effected unless V_{DD} is 3.5 V or higher at this point.

The same applies when V_{DD} is lowered.

Figure 14-8. Restoring from Clock Stop Status



14.6 Power Failure Detection

Power failure detection is used to judge whether power-ON reset by application of supply voltage V_{DD}, or CE reset has been effected when the device is reset, as shown in Figure 14-9.

Because the contents of the data memory and ports are "undefined" on power application, these contents are initialized by means of power failure detection.

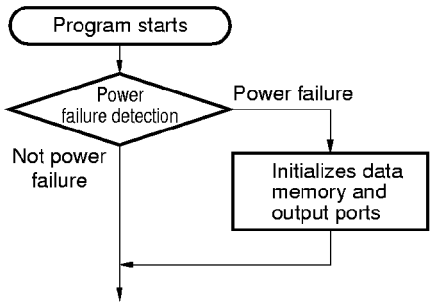
In contrast, the previous values of the data memory and output port remain unchanged at CE reset, and therefore, they do not have to be initialized.

A power failure can be detected in two ways: by using the power failure detection circuit to detect the TMCY flag, and by detecting the contents of the data memory (RAM judgement).

14.6.1 and **14.6.2** explain how a power failure is detected by using the power failure detection circuit and TMCY flag.

14.6.3 and **14.6.4** explain how a power failure is detected by RAM judgement method.

Figure 14-9. Power Failure Detection Flow Chart



14.6.1 Power failure detection circuit

The power failure detection circuit consists of a voltage detection circuit, a timer carry disable flip-flop that is set by the output (power-ON clear signal) of the voltage detection circuit, and a timer carry, as shown in Figure 14-1.

The timer carry disable FF is set to "1" by the power-ON clear signal, and is reset to "0" when an instruction that reads the TMCY flag is executed.

When the timer carry disable FF is set to "1", the TMCY flag is not set to "1".

When the power-ON clear signal is output (at power-ON reset), the program is started with the TMCY flag reset, and the TMCY flag is disabled from being set until an instruction that reads the TMCY flag is executed.

Once the instruction that reads the TMCY flag has been executed, the TMCY flag is set each time the timer carry FF setting pulses has risen. It can be judged whether power-ON reset (power failure) or CE reset (not power failure) has been effected by detecting the contents of the TMCY flag when the device is reset. Power-ON reset has been effected if the TMCY flag is reset to "0"; CE reset has been effected if it is set to "1".

The voltage at which a power failure can be detected is the same as the voltage at which power-ON reset is effected, or V_{DD} = 3.5 V during crystal oscillation, or V_{DD} = 2.2 V in the clock stop status.

Figure 14-10 shows the transition of the status of the TMCY flag.

Figures 14-11 and 14-10 show the timing chart and the operation of the TMCY flag.

Figure 14-10. Status Transition of TMCY Flag

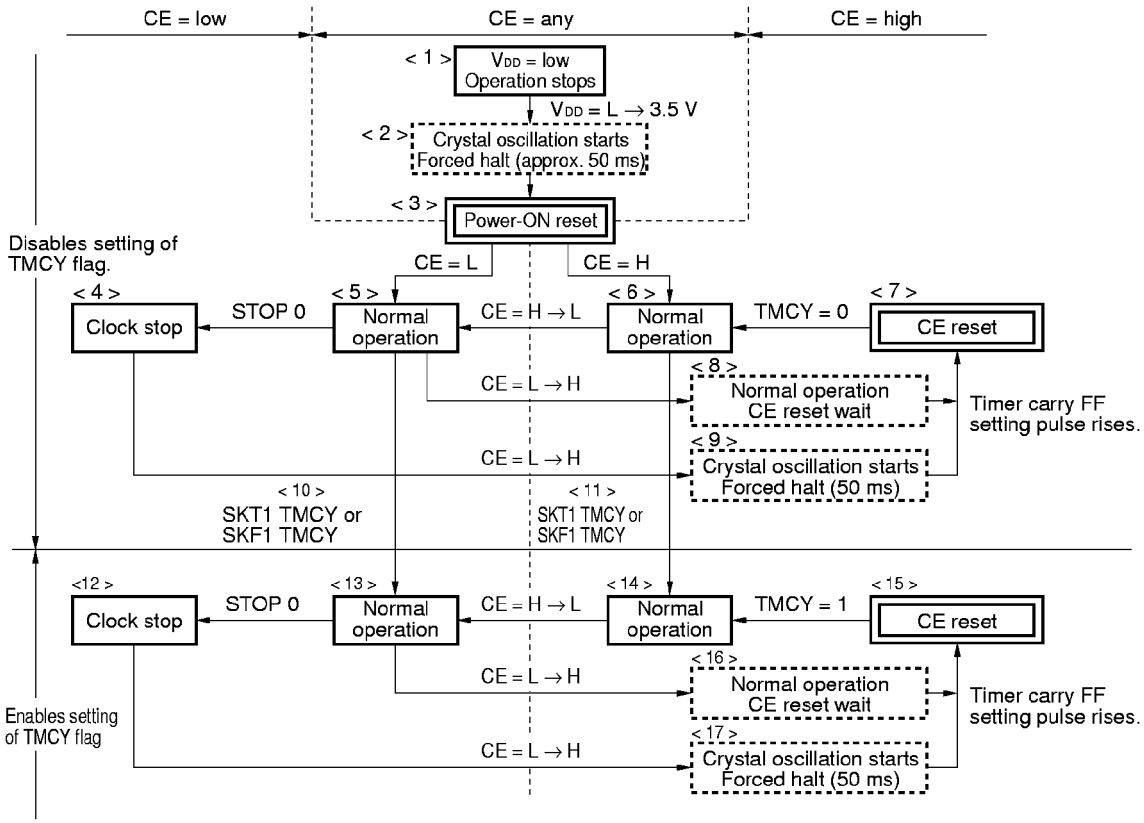
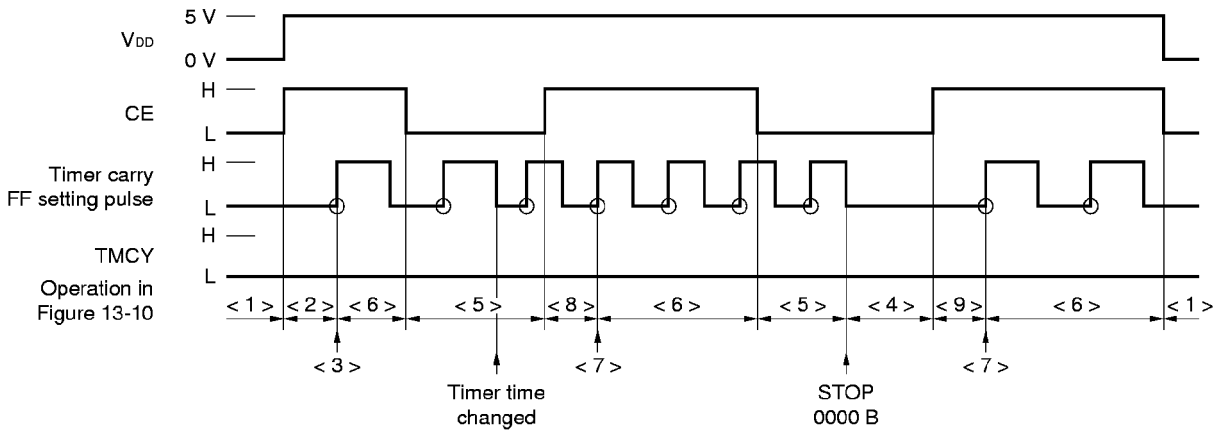
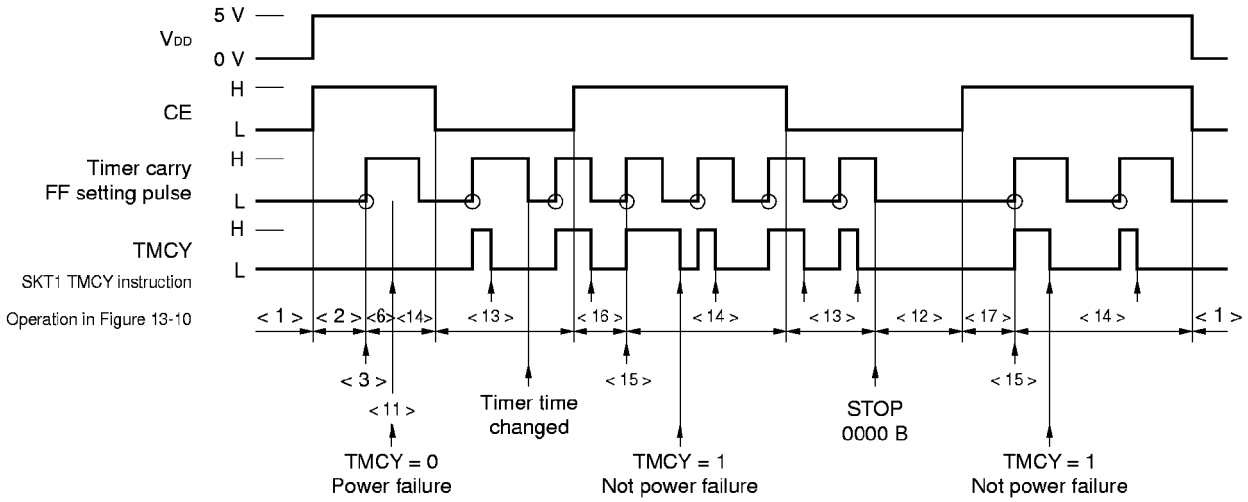


Figure 14-11. Operation of TMCY Flag

(a) When BTM0CY flag never detected (SKT1 TMCY or SKF1 TMCY is not executed)



(b) When detecting power failure by TMCY flag



14.6.2 Notes on detecting power failure by TMCY flag

The following points must be noted when using the TMCY flag for watch counting.

(1) Updating watch

When developing a watch program by using the timer carry, the watch must be updated after a power failure has been detected.

This is because counting of the watch is skipped once because the TMCY flag is reset to "0" when the TMCY flag is read on detection of a power failure.

(2) Watch updating processing time

The processing to update the watch must be completed before the next timer carry FF setting pulse rises.

This is because, if the CE pin goes high during the watch updating processing, CE reset is effected without the watch updating processing completed.

When detecting a power failure, the following points must be noted.

(3) Timing of power failure detection

To count the watch by using the TMCY flag, the TMCY flag must be read to detect a power failure within the time since the program has started from address 0000H until the next timer carry FF setting pulse rises.

For example if the timer carry FF setting time is set to 5 ms, and a power failure is detected 6 ms after the program has been started, the TMCY flag is overlooked once.

Power failure detection and initial processing must be completed within the timer carry FF setting time as shown in the following example.

This is because, if the CE pin goes high and CE reset is effected during power failure detection processing and initial processing, these processing may be stopped in midway, and thus problems may occur.

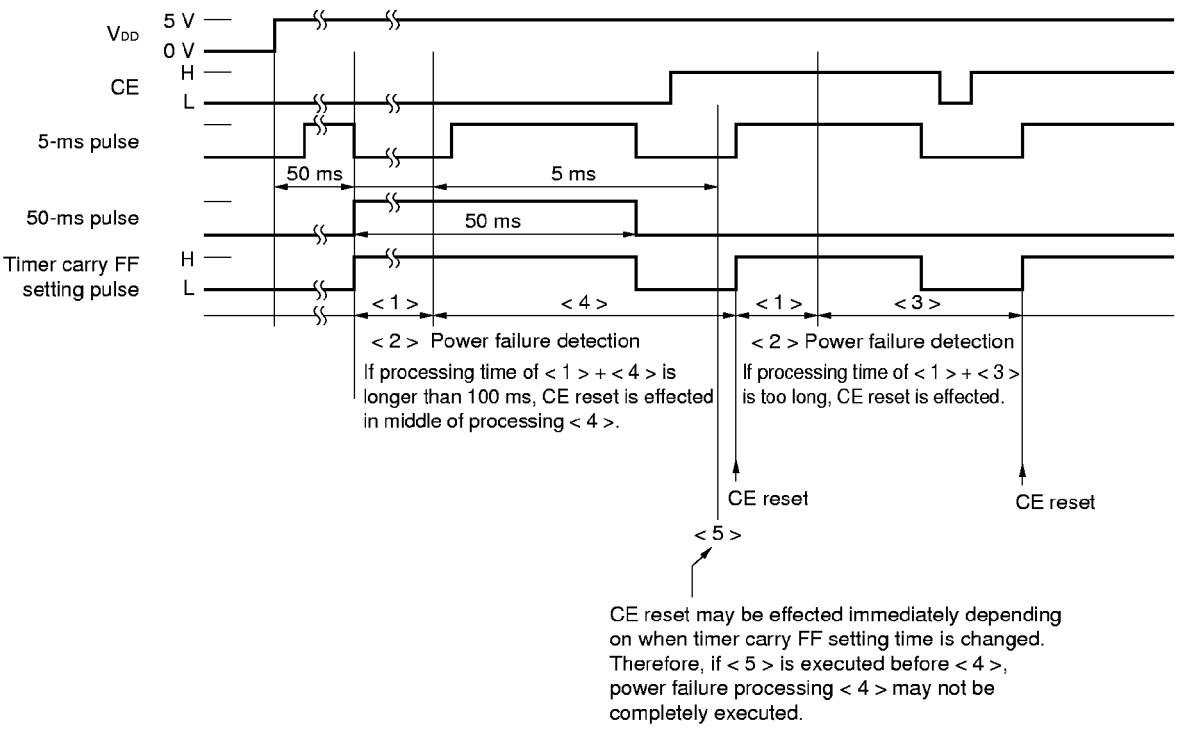
To change the timer carry FF setting time by the initial processing, the instruction that changes the time must be executed at the end of the initial processing, and the instruction must be one instruction.

This is because the initial processing may not be completely executed because of CE reset if the timer carry FF setting time is changed before the initial processing is executed, as shown in the following example.

Example

```
START:                                  ; Program address 0000H
;<1>
   Processing on reset
;<2>
   SKT1      TMCY                       ; Power failure detection
   BR        INITIAL
BACKUP:
;<3>
   Watch updating
   BR        MAIN
INITIAL:
;<4>
   Initial processing
;<5>
   INITFLG TMMD1, NOT TMMD0             ; Embedded macro
                                        ; Sets timer carry FF setting time to 5 ms
MAIN:
   Main processing
   SKT1      TMCY
   BR        MAIN
   Watch updating
   BR        MAIN
```

Example of operation



14.6.3 Power failure detection by RAM judgement method

The RAM judgement method is to detect a power failure by judging whether the contents of the data memory at a specific address are the specified value.

An example of a program that detects a power failure by the RAM judgement method is shown below.

★ The RAM judgement method detects a power failure by comparing an "undefined" value with the "specified value" because the contents of the data memory are "undefined" on application of supply voltage V_{DD}.

Therefore, there is a possibility that a wrong judgment may be made as explained in **14.6.4 Notes on detecting power failure by RAM judgement method**.

When the RAM judgement method is used, however, the device can be backed up at a voltage lower than that at which a power failure is detected, by using the power failure detection circuit, as shown in Table 14-2.

Table 14-2. Comparing Power Failure Detection by Power Failure Detection Circuit and RAM Judgement Method

	Power Failure Detection Circuit		RAM Judgement Method	
Data hold voltage	Effective value	Rated value	Effective value	Rated value
(in clock stop status)	1-2 V	2.2 V	0-1 V	2.0 V
Operating status	No malfunctioning		Malfunctioning may occur	

★ **Example Program to detect power failure by RAM judgement method**

```
M012    MEM     0.12H
M034    MEM     0.34H
M056    MEM     0.56H
M107    MEM     1.07H
M128    MEM     1.28H
M16F    MEM     1.6FH
DATA0   DAT     1010B
DATA1   DAT     0101B
DATA2   DAT     0110B
DATA3   DAT     1001B
DATA4   DAT     1100B
DATA5   DAT     0011B

START:
        SET2    CMP, Z
        SUB     M012, #DATA0        ; If M012 = DATA0 and
        SUB     M034, #DATA1        ; M034 = DATA1 and
        SUB     M056, #DATA2        ; M035 = DATA2 and
        BANK1
        SUB     M107, #DATA3        ; M107 = DATA3 and
        SUB     M128, #DATA4        ; M128 = DATA4 and
        SUB     M16F, #DATA5        ; M16F = DATA5,
        BANK0
        SKF1    Z
        BR      BACKUP              ; branches to BACKUP
;INITIAL:

        [ Initial processing ]

        MOV     M012, #DATA0
        MOV     M034, #DATA1
        MOV     M056, #DATA2
        BANK1
        MOV     M107, #DATA3
        MOV     M128, #DATA4
        MOV     M16F, #DATA5
        BR      MAIN
BACKUP:

        [ Backup processing ]

MAIN:

        [ Main processing ]
```

14.6.4 Notes on detecting power failure by RAM judgement method

The value of the data memory on application of supply voltage V_{DD} is basically "undefined", and therefore, the following points (1) and (2) must be noted.

(1) Data to be compared

Where the number of bits of the data memory to be compared by the RAM judgement method is "n bits", the probability at which the value of the data memory coincides with the value to be compared on application of V_{DD} is $(1/2)^n$.

This means that backup is judged at a probability of $(1/2)^n$ when a power failure is detected by the RAM judgement method.

To lower this probability, as many bits as possible must be compared.

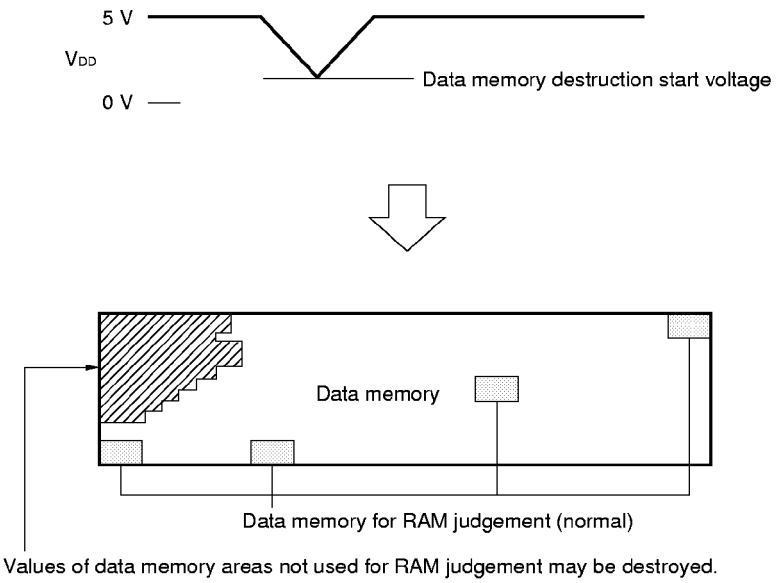
Because the contents of the data memory on application of V_{DD} are likely to be the same value such as "0000B" and "1111B", it is recommended to mix "0" and "1" as data to be compared, such a "1010B" and "0110B" to reduce the possibility of a wrong judgment.

(2) Notes on program

If V_{DD} rises from the level at which the data memory contents may be destroyed as shown in Figure 14-12, and even if the value of the data memory area to be compared is normal, the values of the other data memory areas may be destroyed.

This is judged as backup if a power failure is detected by the RAM judgement method. Therefore, consideration must be given so that the program does not hang up even if the contents of the data memory are destroyed.

Figure 14-12. V_{DD} and Destruction of Data Memory Contents



★ Some mnemonics are not supported by some devices. Refer to the Data Sheet of the target device or User's Manual of the device file.

15.1 Instruction Set Outline

$b_{14} - b_{11}$ \ b_{15} BIN	HEX	0		1	
0 0 0 0	0	ADD	r, m	ADD	m, #n4
0 0 0 1	1	SUB	r, m	SUB	m, #n4
0 0 1 0	2	ADDC	r, m	ADDC	m, #n4
0 0 1 1	3	SUBC	r, m	SUBC	m, #n4
0 1 0 0	4	AND	r, m	AND	m, #n4
0 1 0 1	5	XOR	r, m	XOR	m, #n4
0 1 1 0	6	OR	r, m	OR	m, #n4
0 1 1 1	7	INC	AR		
		INC	IX		
		MOVT	DBF, @AR		
		BR	@AR		
		CALL	@AR		
		RET			
★		SYSCAL	entry		
		RETSK			
		EI			
		DI			
		RETI			
		PUSH	AR		
		POP	AR		
		GET	DBF, p		
		PUT	p, DBF		
		PEEK	WR, rf		
		POKE	rf, WR		
		RORC	r		
		STOP	s		
		HALT	h		
		NOP			
1 0 0 0	8	LD	r, m	ST	m, r
1 0 0 1	9	SKE	m, #n4	SKGE	m, #n4
1 0 1 0	A	MOV	@r, m	MOV	m, @r
1 0 1 1	B	SKNE	m, #n4	SKLT	m, #n4
1 1 0 0	C	BR	addr (page 0)	CALL	addr
1 1 0 1	D	BR	addr (page 1)	MOV	m, #n4
1 1 1 0	E	BR	addr (page 2)	SKT	m, #n
1 1 1 1	F	BR	addr (page 3)	SKF	m, #n

15.2 Legend

AR : Address register
ASR : Address stack register indicated by stack pointer
addr : Program memory address (11 bits, with highest bit fixed to 0)
BANK : Bank register
CMP : Compare flag
CY : Carry flag
DBF : Data buffer
entry : Entry address of system segment
h : Halt release condition
INTEF : Interrupt enable flag
INTR : Register automatically saved to stack when interrupt occurs
INTSK : Interrupt stack register
IX : Index register
MP : Data memory row address pointer
  MPE : Memory pointer enable flag
m : Data memory address indicated by m_R, m_C
  m_R : Data memory row address (high)
  m_C : Data memory column address (low)
n : Bit position (4 bits)
n4 : Immediate data (4 bits)
PC : Program counter
p : Peripheral address
  p_H : Peripheral address (high-order 3 bits)
  p_L : Peripheral address (low-order 4 bits)
r : General register column address
rf : Register file address
  rf_R : Register file row address (high-order 3 bits)
  rf_C : Register file column address (low-order 4 bits)
SP : Stack pointer
s : Stop release condition
WR : Window register
(×) : Contents addressed by ×

15.3 Instruction List

Instruction	Mnemonic	Operand	Operation	Machine Code			
				Op Code	Operand		
Addition	ADD	r, m	(r) ← (r) + (m)	00000	m_R	m_C	r
		m, #n4	(m) ← (m) + n4	10000	m_R	m_C	n4
	ADDC	r, m	(r) ← (r) + (m) + CY	00010	m_R	m_C	r
		m, #n4	(m) ← (m) + n4 + CY	10010	m_R	m_C	n4
	INC	AR	AR ← AR + 1	00111	000	1001	0000
		IX	IX ← IX + 1	00111	000	1000	0000
Subtraction	SUB	r, m	(r) ← (r) – (m)	00001	m_R	m_C	r
		m, #n4	(m) ← (m) – n4	10001	m_R	m_C	n4
	SUBC	r, m	(r) ← (r) – (m) – CY	00011	m_R	m_C	r
		m, #n4	(m) ← (m) – n4 – CY	10011	m_R	m_C	n4
Logical operation	OR	r, m	(r) ← (r) ∨ (m)	00110	m_R	m_C	r
		m, #n4	(m) ← (m) ∨ n4	10110	m_R	m_C	n4
	AND	r, m	(r) ← (r) ∧ (m)	00100	m_R	m_C	r
		m, #n4	(m) ← (m) ∧ n4	10100	m_R	m_C	n4
	XOR	r, m	(r) ← (r) ∀ (m)	00101	m_R	m_C	r
		m, #n4	(m) ← (m) ∀ n4	10101	m_R	m_C	n4
Test	SKT	m, #n	CMP ←0, if (m) ∧ n = n, then skip	11110	m_R	m_C	n
	SKF	m, #n	CMP ← 0, if (m) ∧ n = 0, then skip	11111	m_R	m_C	n
Compare	SKE	m, #n4	(m) – n4, skip if zero	01001	m_R	m_C	n4
	SKNE	m, #n4	(m) – n4, skip if not zero	01011	m_R	m_C	n4
	SKGE	m, #n4	(m) – n4, skip if not borrow	11001	m_R	m_C	n4
	SKLT	m, #n4	(m) – n4, skip if borrow	11011	m_R	m_C	n4
Rotate	RORC	r	→ CY → $(r)_{b3}$ → $(r)_{b2}$ → $(r)_{b1}$ → $(r)_{b0}$ →	00111	000	0111	r
Transfer	LD	r, m	(r) ← (m)	01000	m_R	m_C	r
	ST	m, r	(m) ← (r)	11000	m_R	m_C	r
	MOV	@r, m	if MPE = 1 : (MP, (r)) ← (m) if MPE = 0 : (BANK, m_R, (r)) ← (m)	01010	m_R	m_C	r
		m, @r	if MPE = 1 : (m) ← (MP, (r)) if MPE = 0 : (m) ← (BANK, m_R, (r))	11010	m_R	m_C	r
		m, #n4	(m) ← n4	11101	m_R	m_C	n4
	MOVT[Note]	DBF, @AR	SP ← SP – 1, ASR ← PC, PC ← AR, DBF ← (PC), PC ← ASR, SP ← SP +1	00111	000	0001	0000
	PUSH	AR	SP ← SP – 1, ASR ← AR	00111	000	1101	0000
	POP	AR	AR ← ASR, SP ← SP + 1	00111	000	1100	0000
	PEEK	WR, rf	WR ← (rf)	00111	rf_R	0011	rf_C
	POKE	rf, WR	(rf) ← WR	00111	rf_R	0010	rf_C

Note As an exception, two machine cycles are necessary for executing the MOVT instruction.

Instruction	Mnemonic	Operand	Operation	Op Code			
				Op Code	Operand		
Transfer	GET	DBF, p	DBF ← (p)	00111	P_H	1011	P_L
	PUT	p, DBF	(p) ← DBF	00111	P_H	1010	P_L
Branch	BR	addr	PC_{10-0} ← addr, PAGE ← 0	01100	addr		
			PC_{10-0} ← addr, PAGE ← 1	01101			
			PC_{10-0} ← addr, PAGE ← 2	01110			
			PC_{10-0} ← addr, PAGE ← 3	01111			
		@AR	PC ← AR	00111	000	0100	0000
Subroutine	CALL	addr	SP ← SP – 1, ASR ← PC, PC_{10-0} ← addr, PAGE ← 0	11100	addr		
		@AR	SP ← SP – 1, ASR ← PC, PC ← AR	00111	000	0101	0000
★	SYSCAL	entry	SP ← SP – 1, ASR ← PC, SGR ← 1, $PC_{12, 11}$ ← 0, PC_{10-8} ← $entry_H$, PC_{7-4} ← 0, PC_{3-0} ← $entry_L$	00111	$entry_H$	0000	$entry_L$
	RET		PC ← ASR, SP ← SP + 1	00111	000	1110	0000
	RETSK		PC ← ASR, SP ← SP + 1 and skip	00111	001	1110	0000
	RETI		PC ← ASR, INTR ← INTSK, SP ← SP + 1	00111	100	1110	0000
Interrupt	EI		INTEF ← 1	00111	000	1111	0000
	DI		INTEF ← 0	00111	001	1111	0000
Other operations	STOP	s	STOP	00111	010	1111	s
	HALT	h	HALT	00111	011	1111	h
	NOP		No operation	00111	100	1111	0000

★ 15.4 Assembler (RA17K) Macro instructions

Legend

flag n : FLG type symbol

< > : Can be omitted

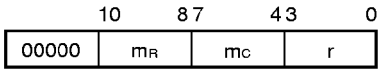
	Mnemonic	Operand	Operation	n
Embedded macro	SKTn	flag 1, ... flag n	if (flag 1) to (flag n)=all "1", then skip	1 ≤ n ≤ 4
	SKFn	flag 1, ... flag n	if (flag 1) to (flag n)=all "0", then skip	1 ≤ n ≤ 4
	SETn	flag 1, ... flag n	(flag 1) to (flag n) ← 1	1 ≤ n ≤ 4
	CLRn	flag 1, ... flag n	(flag 1) to (flag n) ← 0	1 ≤ n ≤ 4
	NOTn	flag 1, ... flag n	if (flag n)="0", then (flag n) ← 1 if (flag n)="1", then (flag n) ← 0	1 ≤ n ≤ 4
	INITFLG	<NOT> flag 1, ... <<NOT> flag n>	if description=NOT flag n, then (flag n) ← 0 if description=flag n, then (flag n) ← 1	1 ≤ n ≤ 4
	BANKn		(BANK) ← n	0 ≤ n ≤ 2
Extension	BRX	Label	Jump Label	—
	CALLX	function-name	CALL sub-routine	—
	INITFLGX	<NOT/INV> flag 1, ... <NOT/INV> flag n	if description = NOT (or INV) ← 0 flag, (flag) ← 0 if description = flag, (flag) ← 1	n ≤ 4

15.5 Instruction Functions

15.5.1 Addition Instructions

(1) ADD r,m **Add data memory to general register**

<1> OP code



<2> Function

When CMP = 0 (r) ← (r) + (m)

Adds the contents of a specified data memory address to the contents of a specified general register, and stores the result in the general register.

When CMP = 1 (r) + (m)

The result is not stored in the register, and the carry flag (CY) and Zero flag (Z) are affected according to the result.

If a carry has occurred as a result of the addition, the carry flag (CY) is set. If not, the carry flag is reset.
If the result of the addition is other than zero, the zero flag (Z) is reset, regardless of the compare flag (CMP).
If the result of the addition becomes zero, with the compare flag reset (CMP = 0), the zero flag (Z) is set.
If the result of the addition becomes zero, with the compare flag set (CMP = 1), the zero flag (Z) is not changed.
Addition can be executed in binary or BCD, which can be selected by the BCD flag (BCD) of the PSWORD.

<3> Example 1

To add the contents of address 0.2FH to those of address 0.03H and store the result in address 0.03H when row address 0 (0.00H-0.0FH) of bank 0 is specified as the general register (RPH=0, RPL=0):

(0.03H) ← (0.03H) + (0.2FH)

```
MEM003  MEM  0.03H
MEM02F  MEM  0.2FH
        MOV  BANK, #00H      ; Data memory bank 0
        MOV  RPH, #00H       ; General register bank 0
        MOV  RPL, #00H       ; General register row address 0
        ADD  MEM003, MEM02F
```

★ **Example 2**

To add the contents of address 0.2FH to those of address 0.23H and store the result in address 0.23H when row address 2 (0.20H-0.2FH) of bank 0 is specified as the general register (RPH=0, RPL=4):

```
                (0.23H) ← (0.23H) + (0.2FH)
MEM023   MEM   0.23H
MEM02F   MEM   0.2FH
         MOV   BANK, #00H        ; Data memory bank 0
         MOV   RPH, #00H         ; General register bank 0Note
         MOV   RPL, #04H         ; General register row address 2
         ADD   MEM023, MEM02F
```

Note

Register	RP							
	RPH				RPL			
Bit	b_3	b_2	b_1	b_0	b_3	b_2	b_1	b_0
Data	Bank				Row address			B / C / D
	0	0	0	0				

The assignment of RP (general register pointer) in the system register is as shown above.
Therefore, to set bank 0 and row address 2 in a general register, 00H must be stored in RPH and 04H, in RPL.
In this case, the arithmetic operations to be performed thereafter are carried out in binary and 4-bit units, because the BCD (binary coded decimal) flag is reset.

Example 3

To add the contents of address 0.6FH to those of address 0.03H and store the result in address 0.03H: If IXE = 1, IXH = 0, IXM = 4, and IXL = 0, i.e., if IX = 0.40H, data memory address 0.6FH can be specified by specifying address 2FH.

```
                (0.03H) ← (0.03H) + (0.6FH)
                                    └──Address obtained by ORing index register contents 0.40H with
                                       data memory address 0.2FH
MEM003   MEM   0.03H
MEM02F   MEM   0.2FH
         MOV   RPH, #00H          ; General register bank 0
         MOV   RPL, #00H          ; General register row address 0
         MOV   IXH, #00H          ; IX ← 00001000000B
         MOV   IXM, #04H          ;
         MOV   IXL, #00H          ;
         SET1  IXE                ; IXE flag ← 1
         ADD   MEM003, MEM02F     ; IX                    00001000000B(0.40H)
                                  ; Bank operand   OR)00000101111B(0.2FH)
                                  ; Specified address  00001101111B(0.6FH)
```

★ **Example 4**

To add the contents of address 0.3FH to those for address 0.03H and store the result in address 0.03H: If IXE = 1, IXH = 0, IXM = 1, and IXL = 0, i.e., if IX = 0.10H, data memory address 0.3FH can be specified by specifying address 2FH.

(0.03H) ← (0.03H) + (0.3FH)

└ Address obtained by ORing index register contents 0.10H with data memory address 0.2FH

```
MEM003  MEM   0.03H
MEM02F  MEM   0.2FH
        MOV   BANK, #00H
        MOV   RPH, #00H          ; General register bank 0
        MOV   RPL, #00H          ; General register row address 0
        MOV   IXH, #00H          ; IX ← 00000010000B (0.10H)Note
        MOV   IXM, #01H
        MOV   IXL, #00H
        SET1  IXE                ; IXE flag ← 1
        ADD   MEM003, MEM02F ; IX                    00000010000B(0.10H)
                             ; Bank operand  OR)00000101111B(0.2FH)
                             ; Specified address 00100111111B(0.3FH)
```

Note

Register	IX											
	IXH				IXM				IXL			
Bit	b_3	b_2	b_1	b_0	b_3	b_2	b_1	b_0	b_3	b_2	b_1	b_0
Data	MPE	0	0	0	0	Row address			Column address			
		Bank										

The IX (index pointer) assignment in the system register is as shown above.
Therefore, in order that IX = 0.10H, 00H must be stored in IXH, 01H in IXM, and 00H in IXL.
In this case, since the MPE (memory pointer enable) flag is reset, MP (memory pointer) is invalid for general register indirect transfer.

<4> Precaution

The first operand for the ADD r, m instruction is the column address of a general register. Therefore, if the instruction is described as follows, the column address of the general register is 03H:

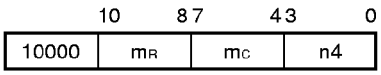
```
MEM013  MEM   0.13H
MEM02F  MEM   0.2FH
        ADD   MEM013, MEM02F
```

└ Means column address of general register. Low-order 4 bits (03H in this case) are valid.

When CMP flag = 1, the addition result is not stored.
When the BCD flag is 1, the BCD operation result is stored.

(2) ADD m, #n4 — Add immediate data to data memory

<1> OP code



<2> Function

When CMP = 0 (m) ← (m) + n4

Adds the immediate data to the contents of a specified data memory address and stores the results in the data memory.

When CMP = 1 (m) + n4

The result is not stored in the data memory, and the carry flag (CY) and zero flag (Z) are affected according to the result.

If a carry has occurred as a result of the addition, the carry flag (CY) is set. If not, the carry flag is reset.
If the result of the addition is other than zero, the zero flag (Z) is reset, regardless of the compare flag (CMP).
If the result of the addition becomes zero, with the compare flag reset (CMP = 0), the zero flag (Z) is set.
If the result of the addition becomes zero, with the compare flag set (CMP = 1), the zero flag (Z) is not changed.
Addition can be executed in binary 4-bit units or BCD, which can be selected by the BCD flag (BCD) of the PSWORD.

<3> Example 1

To add 5 to the contents of address 0.2FH and store the result in address 0.2FH:

```
                (0.2FH) ← (0.2FH) + 5
MEM02F  MEM     0.2FH
        ADD     MEM02F, #05H
```

Example 2

To add 5 to the contents of address 0.6FH and store the result in address 0.6FH: At this time, if IXE = 1, IXH = 0, IXM = 4, and IXL = 0, i.e., if IX = 0.40H, data memory address 0.6FH can be specified by specifying address 2FH.

```
                (0.6FH) ← (0.6FH) + 05H
                          └── Address obtained by ORing index register contents 0.40H with data
                              memory address 0.2FH
MEM02F  MEM     0.2FH
        MOV     BANK, #00H          ; Data memory bank 0
        MOV     IXH, #00H           ; IX ← 00001000000B(0.40H)
        MOV     IXM, #04H
        MOV     IXL, #00H
        SET1    IXE                 ; IXE flag ← 1
        ADD     MEM02F, #05H        ; IX                    00001000000B(0.40H)
                                    ; Bank operand  OR)00000101111B(0.2FH)
                                    ; Specified address 00001101111B(0.6FH)
```

★ **Example 3**

To add 5 to the contents of address 0.2FH and store the result in address 0.2FH: If IXE = 1, IXH = 0, IXM = 0, and IXL = 0, i.e., if IX = 0.00H, data memory address 0.2FH can be specified by specifying address 2FH.

(2.2FH) ← (0.2FH) + 05H
└── Address obtained by ORing index register contents 0.00H with data memory address 0.2FH

```
MEM02F  MEM   0.2FH
        MOV   BANK, #00H        ; Data memory bank 0
        MOV   IXH, #00H         ; IX ← 00000000000B
        MOV   IXM, #00H
        MOV   IXL, #00H
        SET1  IXE               ; IXE flag ← 1
        ADD   MEM02F, #05H      ; IX                    00000000000B(0.00H)
                                ; Bank operand   OR)00000101111B(0.2FH)
                                ; Specified address  00000101111B(0.2FH)
```

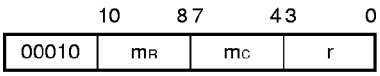
<4> Precaution

When CMP flag = 1, the result of the addition is not stored.

When BCD flag = 1, the result of a BCD operation is stored.

(3) ADDC r, m **Add data memory to general register with carry flag**

<1> OP code



<2> Function

When CMP = 0 (r) ← (r) + (m) + CY

Adds the contents of a specified data memory address and the carry flag CY value to the contents of a general register, and stores the result in the general register specified by r.

When CMP = 1 (r) + (m) + CY

The result is not stored in the register, and the carry flag (CY) and zero flag (Z) are affected by the result.

You can use this ADDC instruction to easily add, two or more words.

If a carry has occurred as a result of the addition, the carry flag (CY) is set. If not, the carry flag is reset.

If the result of the addition is other than zero, the zero flag (Z) is reset regardless of the compare flag (CMP).

If the addition results in zero, with the compare flag reset (CMP = 0), the zero flag (Z)is set.

If the result of the addition results in zero, with the compare flag set (CMP = 1), the zero flag (Z) is not affected.

You can perform addition in binary and 4-bit units or BCD, which you can select by the BCD flag of the PSWORD.

<3> Example 1

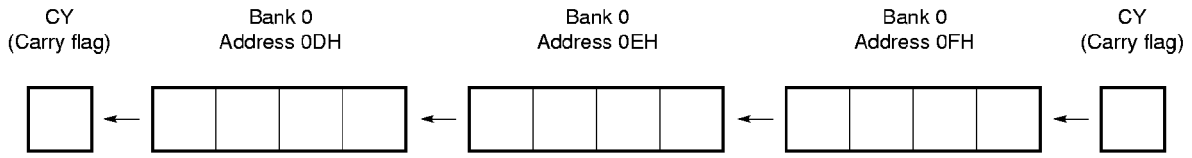
To add the contents of 12-bit addresses 0.2DH through 0.2FH to the 12-bit contents of addresses 0.0DH through 0.0FH and store the result in the 12 bits of addresses 0.0DH through 0.0FH when row address 0 in bank 0 (0.00H-0.0FH) is specified as a general register:

0.0FH ← (0.0FH) + (0.2FH)
0.0EH ← (0.0EH) + (0.2EH) + CY
0.0DH ← (0.0DH) + (0.2DH) + CY

```
MEM00D  MEM   0.0DH
MEM00E  MEM   0.0EH
MEM00F  MEM   0.0FH
MEM02D  MEM   0.2DH
MEM02E  MEM   0.2EH
MEM02F  MEM   0.2FH
        MOV   BANK, #00H        ; Data memory bank 0
        MOV   RPH, #00H         ; General register bank 0
        MOV   RPL, #00H         ; General register row address 0
        ADD   MEM00F, MEM02F
        ADDC  MEM00E, MEM02E
        ADDC  MEM00D, MEM02D
```

★ **Example 2**

To shift the 12-bit contents of addresses 0.2DH through 0.2FH 1 bit to the left with the carry flag when row address 2 (0.20H-0.2FH) of bank 0 is specified as a general register:




```
MEM00D  MEM   0.0DH
MEM00E  MEM   0.0EH
MEM00F  MEM   0.0FH
MEM02D  MEM   0.2DH
MEM02E  MEM   0.2EH
MEM02F  MEM   0.2FH
        MOV   RPH, #00H          ; General register bank 0
        MOV   RPL, #04H          ; General register row address 2
        MOV   BANK, #00H         ; Data memory bank 0
        ADDC  MEM00F, MEM02F
        ADDC  MEM00E, MEM02E
        ADDC  MEM00D, MEM02D
```

Example 3

To add the contents of addresses 0.40H through 0.4FH to the contents of address 0.0FH and store the result in address 0.0FH:

```
        (0.0FH) ← (0.0FH) + (0.40H) + (0.41H) + ····· + (0.4FH)
MEM00F  MEM   0.0FH
MEM000  MEM   0.00H
        MOV   BANK, #00H         ; Data memory bank 0
        MOV   RPH, #00H          ; General register bank 0
        MOV   RPL, #00H          ; General register row address 0
        MOV   IXH, #00H          ; IX ← 00001000000B (0.40H)
        MOV   IXM, #04H
        MOV   IXL, #00H
LOOP1:
        SET1  IXE                ; IXE flag ← 1
        ADD   MEM00F, MEM000
        CLR1  IXE                ; IXE flag ← 0
        INC   IX                 ; IX ← IX + 1
        SKE   IXL, #0
        JMP   LOOP1
```

★ **Example 4**

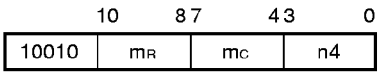
To add the 12-bit contents of addresses 0.40H through 0.42H to the 12-bit contents of addresses 0.0DH through 0.0FH and store the result in the 12 bits of addresses 0.0DH through 0.0FH:

(0.0DH) ← (0.0DH) + (0.40H)
(0.0EH) ← (0.0EH) + (0.41H) + CY
(0.0FH) ← (0.0FH) + (0.42H) + CY

```
MEM000  MEM   0.00H
MEM001  MEM   0.01H
MEM002  MEM   0.02H
MEM00D  MEM   0.0DH
MEM00E  MEM   0.0EH
MEM00F  MEM   0.0FH
        MOV   BANK, #00H          ; Data memory bank 0
        MOV   RPH, #00H           ; General register bank 0
        MOV   RPL, #00H           ; General register row address 0
        MOV   IXH, #00H           ; IX ← 00001000000 (0.40H)
        MOV   IXM, #04H
        MOV   IXL, #00H
        SET1  IXE                 ; IXE flag ← 1
        ADD   MEM00D, MEM000      ; (0.0DH) ← (0.0DH) + (0.40H)
        ADDC  MEM00E, MEM001      ; (0.0EH) ← (0.0EH) + (0.41H)
        ADDC  MEM00F, MEM002      ; (0.0FH) ← (0.0FH) + (0.42H)
```

(4) ADDC m, #n4 — Add immediate data to data memory with carry flag

<1> OP code



10	8 7	4 3	0
10010	m_R	m_C	n4

<2> Function

When CMP = 0 (m) ← (m) + n4 + CY

Adds the immediate data to the contents of a specified data memory address, including the carry flag (CY), and stores the results in the data memory address.

When CMP = 1 (m) + n4 + CY

The result is not stored in the data memory, and the carry flag (CY) and zero flag (Z) are affected by the result.

If a carry has occurred as a result of the addition, the carry flag (CY) is set. If not, the carry flag is reset.
If the result of the addition is other than zero, the zero flag (Z) is reset, regardless of the compare flag (CMP).
If the result of the addition becomes zero, with the compare flag reset (CMP = 0), the zero flag is set.
If the result of the addition becomes zero, with the compare flag set (CMP = 1), the zero flag is not affected.
You can perform addition in binary or BCD, which you can select by the BCD flag of the PSWORD.

<3> Example 1

To add 5 to the 12-bit contents of addresses 0.0DH through 0.0FH and store the result in addresses 0.0DH through 0.0FH:

```
                (0.0FH) ← (0.0FH) + 05H
                (0.0EH) ← (0.0EH) + CY
                (0.0DH) ← (0.0DH) + CY
MEM00D  MEM  0.0DH
MEM00E  MEM  0.0EH
MEM00F  MEM  0.0FH
        MOV  BANK, #00H        ; Data memory bank 0
        ADD  MEM00F, #05H
        ADDC MEM00E, #00H
        ADDC MEM00D, #00H
```

Example 2

To add 5 to the 12-bit contents of addresses 0.4DH through 0.4FH and store the result in addresses 0.4DH through 0.4FH:

```
                (0.4FH) ← (0.4FH) + 05H
                (0.4EH) ← (0.4EH) + CY
                (0.4DH) ← (0.4DH) + CY
MEM00D  MEM   0.0DH
MEM00E  MEM   0.0EH
MEM00F  MEM   0.0FH
        MOV   BANK, #00H        ; Data memory bank 0
        MOV   IXH, #00H         ; IX ← 00001000000B(0.40H)
        MOV   IXM, #04H
        MOV   IXL, #00H
        SET1  IXE               ; IXE flag ← 1
        ADD   MEM00F, #5        ; (0.4FH) ← (0.4FH) + 5H
        ADDC  MEM00E, #0        ; (0.4EH) ← (0.4EH) + CY
        ADDC  MEM00D, #0        ; (0.4DH) ← (0.4DH) + CY
```

(5) INC AR — Increment address register

<1> OP code

10	8 7	4 3	0
00111	000	1001	0000

<2> Function

AR ← AR + 1

Increments the contents of the address register (AR).

<3> Example 1

To add 1 to the 16-bit contents of AR3 through AR0 (address registers) in the system register and store the result in AR3 through AR0:

```
; AR0 ← AR0 + 1
; AR1 ← AR1 + CY
; AR2 ← AR2 + CY
; AR3 ← AR3 + CY
INC AR
```

This instruction effect can also be implemented by an addition instruction, as follows:

```
ADD   AR0, #01H
ADDC  AR1, #00H
ADDC  AR2, #00H
ADDC  AR3, #00H
```

Example 2

To transfer table data in 16-bit units (1 address) to DBF (data buffer) by using the table reference instruction (for details, refer to **10.4 Data Buffer and Table Reference**):

```
; Address           Table data
010H      DW        0F3FFH
011H      DW        0A123H
012H      DW        0FFF1H
013H      DW        0FFF5H
014H      DW        0FF11H
          :
          :
          MOV       AR3, #0H      ; Table data address
          MOV       AR2, #0H      ; Sets 0010H in address register
          MOV       AR1, #1H
          MOV       AR0, #0H
LOOP:
          MOVT       @AR          ; Reads table data to DBF
          :
          :
          :                       ; Processing referencing table data
          INC       AR            ; register by 1
          BR        LOOP
```

<4> Precaution

The number of bits of the address registers (AR0 through AR3) that can be used differs according to the model. For details, refer to the Data Sheet of each model.

(6) INC IX — Increment index register

<1> OP code

10	87	43	0
00111	000	1000	0000

<2> Function

IX ← IX + 1

Increments the contents of the index register (IX).

<3> Example 1

To add 1 to the 12-bit contents of IXH, IXM, and IXL (index registers) in the system register and store the result in IXH, IXM, and IXL:

```
; IXL ← IXL + 1
; IXM ← IXM + CY
; IXH ← IXH + CY
; INC IX
```

You can also execute this instruction by an addition instruction, as follows:

```
ADD  IXL, #01H
ADDC IXM, #00H
ADDC IXH, #00H
```

Example 2

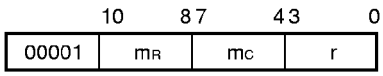
To clear all the contents of data memory addresses 0.00H through 0.73H to 0 by using the index register:

```
        MOV     IXH,    #00H    ; Sets index register contents to 00H in bank 0
        MOV     IXM,    #00H    ;
        MOV     IXL,    #00H
RAM clear:
MEM000  MEM     0.00H
        SET1    IXE             ; IXE flag ← 1
        MOV     MEM000, #00H    ; Writes 0 to data memory indicated by index register
        CLR1    IXE             ; IXE flag ← 0
        INC     IX
        SET2    CMP, Z          ; CMP flag ← 1, Z flag ← 1
        SUB     IXL, #03H       ; Checks if index register contents are 73H for bank 0
        SUBC    IXM, #07H       ;
        SUBC    IXH, #00H       ;
        SKT1    Z               ; Loops until index register contents become 73H for bank 0
        BR      RAM clear       ;
```

15.5.2 Subtraction instructions

(1) SUB r, m **Subtract data memory from general register**

<1> OP code



<2> Function

When CMP = 0 (r) ← (r) – (m)

Subtracts the contents of a specified data memory address from the contents of a specified general register, and stores the result in the general register.

When CMP = 1 (r) – (m)

The result is not stored in the register, and the carry flag (CY) and zero flag (Z) are affected by the result.

If a borrow has occurred as a result of the subtraction, the carry flag (CY) is set. If not, the carry flag is reset.

If the result of the subtraction is other than zero, the zero flag (Z) is reset, regardless of the compare flag (CMP).

If the subtraction results in zero, with the compare flag reset (CMP = 0), the zero flag (Z) is set.

If the subtraction results in zero, with the compare flag set (CMP = 1), the zero flag (Z) is not affected.

You can perform subtraction in binary and 4-bit units or BCD, which you can select by the BCD flag of the PSWORD.

<3> Example 1

To subtract the contents of address 0.2FH from those of address 0.03H and store the result in address 0.03H when the row address 0 (0.00H-0.0FH) of bank 0 is specified as a general register (RPH=0, RPL=0):

(0.03H) ← (0.03H) + (0.2FH)

```
MEM003  MEM  0.03H
MEM02F  MEM  0.2FH
        SUB  MEM003, MEM02F
```

★ **Example 2**

To subtract the contents of address 0.2FH from those of address 0.23H and store the result in address 0.23H when row address 2 (0.20H-0.2FH) of bank 0 is specified as a general register (RPH=0, RPL=4):

(0.23H) ← (0.23H) – (0.2FH)

```
MEM023  MEM  0.23H
MEM02F  MEM  0.2FH
        MOV  BANK, #00H      ; Data memory bank 0
        MOV  RPH, #00H       ; General register bank 0
        MOV  RPL, #04H       ; General register row address 2
        SUB  MEM023, MEM02F
```

Example 3

To subtract the contents of address 0.6FH from those of address 0.03H, and store the result in address 0.03H: If IXE = 1, IXH = 0, IXM = 4, and IXL = 0, i.e., if IX = 0.40H, data memory address 0.6FH can be specified by specifying address 2FH.

```
                (0.03H) ← (0.03H) – (0.6FH)
MEM003   MEM   0.03H
MEM02F   MEM   0.2FH
         MOV   BANK,#00H          ; Data memory bank 0
         MOV   RPH, #00H          ; General register bank 0
         MOV   RPL, #00H          ; General register row address 0
         MOV   IXH, #00H          ; IX ← 00001000000B (0.40H)
         MOV   IXM, #04H          ;
         MOV   IXL, #00H          ;
         SET1  IXE                ; IXE ← flag  1
         SUB   MEM003, MEM02F     ; IX                  00001000000B(0.40H)
                                  ; Bank operand   OR)00000101111B(0.2FH)
                                  ; Specified address  00001101111B(0.6FH)
```

★ **Example 4**

To subtract the contents of address 0.3FH from those of address 0.03H and store the result in address 0.03H: If IXE = 1, IXH = 0, IXM = 1, and IXL = 0, i.e., if IX = 0.10H, data memory address 0.3FH can be specified by specifying address 2FH.

```
                (0.03H) ← (0.03H) + (0.3FH)
MEM003   MEM   0.03H
MEM02F   MEM   0.2FH
         MOV   BANK,#00H          ; Data memory bank 0
         MOV   RPH, #00H          ; General register bank 0
         MOV   RPL, #00H          ; General register row address 0
         MOV   IXH, #00H          ; IX ← 00000010000B (0.10H)
         MOV   IXM, #01H          ;
         MOV   IXL, #00H          ;
         SET1  IXE                ; IXE flag ← 1
         SUB   MEM003, MEM02F     ; IX                  00000010000B(0.10H)
                                  ; Bank operand   OR)00000101111B(0.2FH)
                                  ; Specified address  00000111111B(0.3FH)
```

<4> Precaution

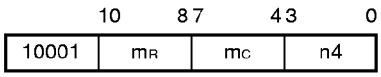
The first operand of the SUB r, m instruction must be a general register address. Therefore, if you make the following description, address 03H is specified as a register.

```
MEM013   MEM   0.13H
MEM02F   MEM   0.2FH
         SUB   MEM013, MEM02F
                 └── General register address must be within the 00H-0FH range (set register
                     pointer to other than row address 1).
```

When CMP flag = 1, the result of the subtraction is not stored.
When the BCD flag = 1, the result of the BCD operation is stored.

(2) SUB m, #n4 — Subtract immediate data from data memory

<1> OP code



<2> Function

When CMP = 0 (m) ← (m) – n4

Subtracts specified immediate data from the contents of a specified data memory address, and stores the result in the data memory address.

When CMP = 1 (m) – n4

The result is not stored in the data memory, and the carry flag (CY) and zero flag (Z) are affected by the result.

If a borrow has occurred as a result of the subtraction, the carry flag (CY) is set. If not, the carry flag is reset.

If the result of the subtraction is other than zero, the zero flag (Z) is reset regardless of the compare flag (CMP).

If the subtraction results in zero when the compare flag is reset (CMP = 0), the zero flag is set.

If the subtraction results in zero when the compare flag is set (CMP = 1), the zero flag is not affected.

You can perform subtraction in binary and 4-bit units and BCD, which you can select by the BCD flag for the PSWORD.

<3> Example 1

To subtract 5 from the address 0.2FH contents and store the result in address 0.2FH:

```
                (0.2FH) ← (0.2FH) – 5
MEM02F  MEM     0.2FH
        SUB     MEM02F, #05H
```

Example 2

To subtract 5 from the contents of address 0.6FH and store the result in address 0.6FH: At this time, if IXE = 1, IXH = 0, IXM = 4, and IXL = 0, i.e., if IX = 0.40H, data memory address 0.6FH can be specified by specifying address 2FH.

```
                0.6FH ← (0.6FH) – 5
                        └─ Address obtained by ORing index register contents 0.40H with data
                           memory address 0.2FH
MEM02F  MEM     0.2FH
        MOV     BANK, #00H      ; Data memory bank 0
        MOV     IXH, #00H       ; IX ← 00001000000B (0.40H)
        MOV     IXM, #04H       ;
        MOV     IXL, #00H       ;
        SET1    IXE             ; IXE flag ← 1
        SUB     MEM02F, #05H    ; IX                    00001000000B(0.40H)
                                ; Bank operand  OR)00000101111B(0.2FH)
                                ; Specified address   00001101111B(0.6FH)
```

★ **Example 3**

To subtract 5 from the contents of address 0.2FH and store the result in address 0.2FH: If IXE = 1, IXH = 0, IXM = 0, and IXL = 0, i.e., if IX = 0.00H, data memory address 0.2FH can be specified by specifying address 2FH.

(0.2FH) ← (0.2FH) – 5
└Address obtained by ORing index register contents 0.00H with data memory address 0.2FH

```
MEM02F  MEM   0.2FH
        MOV   BANK0, #00H      ; Data memory bank 0
        MOV   IXH, #00H        ; IX ← 00000000000B (0.00H)
        MOV   IXM, #00H        ;
        MOV   IXL, #00H        ;
        SET1  IXE              ; IXE flag ← 1
        SUB   MEM02F, #05H     ; IX                    00000000000B(0.00H)
                               ; Bank operand   OR)00000101111B(0.2FH)
                               ; Specified address  00000101111B(0.2FH)
```

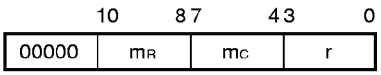
<4> Precaution

When CMP flag = 1, the result of the subtraction is not stored.

When BCD flag = 1, the result of the BCD format operation is stored.

(3) SUBC r, m — Subtract data memory from general register with carry flag

<1> OP code



<2> Function

When CMP = 0 (r) ← (r) – (m) – CY

Subtracts the contents of a specified data memory, including the carry flag (CY), from the contents of a specified general register, and stores the result in the general register. By using this SUBC instruction, subtraction of two or more words can be easily carried out.

When CMP = 1 (r) – (m) – CY

The result is not stored in the register, and the carry flag (CY) and zero flag (Z) are affected by the result.

If a borrow has occurred, as a result of the subtraction, the carry flag (CY) is set. If not, the carry flag is reset.

If the result of the subtraction is other than zero, the zero flag (Z) is reset, regardless of the compare flag (CMP).

If the subtraction results in zero when the compare flag is reset (CMP = 0), the zero flag is reset.

If the subtraction results in zero when the compare flag set (CMP = 1), the zero flag is not changed.

You can perform subtraction in binary and 4-bit units or BCD, which you can select by the BCD flag of the PSWORD.

<3> Example 1

To subtract the 12-bit contents of addresses 0.2DH through 0.2FH from the 12-bit contents of addresses 0.0DH through 0.0FH and store the result in the 12 bits of addresses 0.0DH through 0.0FH when row address 0 of bank 0 (0.00H-0.0FH) is specified as a general register:

(0.0FH) ← (0.0FH) – (0.2FH)
(0.0EH) ← (0.0EH) – (0.2EH) – CY
(0.0DH) ← (0.0DH) + (0.2DH) – CY

```
MEM00D  MEM   0.0DH
MEM00E  MEM   0.0EH
MEM00F  MEM   0.0FH
MEM02D  MEM   0.2DH
MEM02E  MEM   0.2EH
MEM02F  MEM   0.2FH
        SUB   MEM00F, MEM02F
        SUBC  MEM00E, MEM02E
        SUBC  MEM00D, MEM02D
```

Example 2

To subtract the 12-bit contents of addresses 0.40H through 0.42H from the 12-bit contents of addresses 0.0DH through 0.0FH and store the result in the 12 bits of addresses 0.0DH through 0.0FH:

```
                (0.0DH) ← (0.0DH) – (0.40H)
                (0.0EH) ← (0.0EH) – (0.41H) – CY
                (0.0FH) ← (0.0FH) + (0.42H) – CY
MEM000    MEM   0.00H
MEM001    MEM   0.01H
MEM002    MEM   0.02H
MEM00D    MEM   0.0DH
MEM00E    MEM   0.0EH
MEM00F    MEM   0.0FH
          MOV   BANK, #00H          ; Data memory bank 0
          MOV   RPH, #00H           ; General register bank 0
          MOV   RPL, #00H           ; General register row address 0
          MOV   IXH, #00H           ; IX ← 00001000000B (0.40H)
          MOV   IXM, #04H           ;
          MOV   IXL, #00H           ;
          SET1  IXE                 ; IXE flag ← 1
          SUB   MEM00D, MEM000 ; (0.0DH) ← (0.0DH) – (0.40H)
          SUBC  MEM00E, MEM001 ; (0.0EH) ← (0.0EH) – (0.41H)
          SUBC  MEM00F, MEM002 ; (0.0FH) ← (0.0FH) – (0.42H)
```

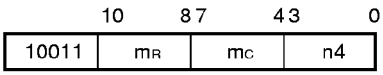
Example 3

To compare the 12-bit contents of addresses 0.00H through 0.03H with the 12-bit contents of addresses 0.0CH through 0.0FH and jump to LAB1, if both the 12-bit contents are the same. If not, jump to LAB2.

```
MEM000    MEM   0.00H
MEM001    MEM   0.01H
MEM002    MEM   0.02H
MEM003    MEM   0.03H
MEM00C    MEM   0.0CH
MEM00D    MEM   0.0DH
MEM00E    MEM   0.0EH
MEM00F    MEM   0.0FH
          SET2  CMP, Z              ; CMP flag ← 1, Z flag ← 1
          SUB   MEM000, MEM00C ; 0.00H-0.03H, because CMP flag is set
          SUBC  MEM001, MEM00D ; Address contents are not affected
          SUBC  MEM002, MEM00E ;
          SUBC  MEM003, MEM00F ;
          SKF1  Z                   ; Z flag = 1, if result is the same.
          BR    LAB1                ; Z flag = 0, if result is not the same.
          BR    LAB2
                :
LAB1 :    :
                :
LAB2 :    :
                :
```

(4) SUBC m, #n4 — Subtract immediate data from data memory with carry flag

<1> OP code



<2> Function

When CMP = 0 (m) ← (m) – n4 – CY

Subtracts specified immediate data from the contents of a specified data memory, including the carry flag, and stores the result in the data memory address.

When CMP = 1 (m) – n4 – CY

The result is not stored in the data memory, and the carry flag (CY) and zero flag (Z) are affected by the result.

If a borrow has occurred as a result of the subtraction, the carry flag (CY) is set. If not, the carry flag is reset.

If the result of the subtraction is other than zero, the zero flag (Z) is reset, regardless of the compare flag (CMP).

If the subtraction results in zero when the compare flag is reset (CMP = 0), the zero flag is set.

If the subtraction results in zero when the compare flag is set (CMP = 1), the zero flag is not changed.

You can perform subtraction in binary and 4-bit units or BCD, which can be selected by the BCD flag of the PSWORD.

<3> Example 1

To subtract 5 from the 12-bit contents of addresses 0.0DH through 0.0FH and store the result in addresses 0.0DH through 0.0FH:

```
                (0.0FH) ← (0.0FH) – 05H
                (0.0EH) ← (0.0EH) – CY
                (0.0DH) ← (0.0DH) – CY
MEM00D  MEM     0.0DH
MEM00E  MEM     0.0EH
MEM00F  MEM     0.0FH
        SUB     MEM00F, #05H
        SUBC    MEM00E, #00H
        SUBC    MEM00D, #00H
```

Example 2

To subtract 5 from the 12-bit contents of addresses 0.4DH through 0.4FH and store the result in addresses 0.4DH through 0.4FH:

```
                (0.4FH) ← (0.4FH) – 05H
                (0.4EH) ← (0.4EH) – CY
                (0.4DH) ← (0.4DH) – CY
MEM00D  MEM     0.0DH
MEM00E  MEM     0.0EH
MEM00F  MEM     0.0FH
        MOV     BANK, #00H      ; Data memory bank 0
        MOV     IXH, #00H       ; IX ← 00001000000B (0.40H)
        MOV     IXM, #04H       ;
        MOV     IXL, #00H       ;
        SET1    IXE             ; IXE flag ← 1
        SUB     MEM00F, #5      ; (0.4FH) ← (0.4FH) – 5
        SUBC    MEM00E, #0      ; (0.4EH) ← (0.4EH) – CY
        SUBC    MEM00D, #0      ; (0.4DH) ← (0.4DH) – CY
```

Example 3

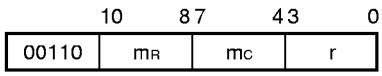
To compare the 12-bit contents of addresses 0.00H through 0.03H with immediate data 0A3FH and jump to LAB1 when both the 12-bit contents are the same. If not, jump to LAB2.

```
MEM000  MEM     0.00H
MEM001  MEM     0.01H
MEM002  MEM     0.02H
MEM003  MEM     0.03H
        SET2    CMP, Z          ; CMP flag ← 1, Z flag ← 1
        SUB     MEM000, #0H     ; 0.00H-0.03H, because CMP flag is set
        SUBC    MEM001, #0AH    ; Address contents are not affected
        SUBC    MEM002, #3H     ;
        SUBC    MEM003, #0FH    ;
        SKF1    Z               ; Z flag = 1, if result is the same.
        BR      LAB1            ; Z flag = 0, if result is not the same.
        BR      LAB2
                :
LAB1:           :
                :
LAB2:           :
                :
                :
```

15.5.3 Logical operation instructions

(1) OR r, m **OR between general register and data memory**

<1> OP code



<2> Function

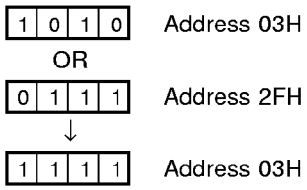
(r) ← (r) ∨ (m)

ORs the contents of a specified data memory address with the contents of a specified general register, and stores the result in the general register.

<3> Example

To OR the contents of address 0.03H (1010B) with the contents of address 0.2FH (0111B) and store the result (1111B) in address 0.03H.

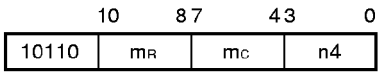
(0.03H) ← (0.03H) ∨ (0.2FH)




```
MEM003   MEM   0.03H
MEM02F   MEM   0.2FH
         MOV   MEM003, #1010B
         MOV   MEM02F, #0111B
         OR    MEM003, MEM02F
```

(2) OR m, #n4 — OR between data memory and immediate data

<1> OP code



<2> Function

(m) ← (m) ∨ n4

ORs the contents of a specified data memory address with specified immediate data and stores the result in the data memory address.

<3> Example 1

To set bit 3 (MSB) of address 0.03H.

(0.03H) ← (0.03H) ∨ 1000B

Address 0.03H

1	×	×	×

× : don't care

```
MEM003   MEM   0.03H
         OR    MEM003, #1000B
```

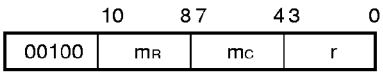
Example 2

To set all the bits of address 0.03H.

```
MEM003   MEM   0.03H
         OR    MEM003, #1111B
   or
MEM003   MEM   0.03H
         MOV   MEM003, #0FH
```

(3) AND r, m — AND between general register and data memory

<1> OP code



<2> Function

(r) ← (r) ∧ (m)

ANDs the contents of a specified data memory address with the contents of a specified general register, and stores the result in the general register.

<3> Example

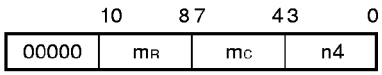
To AND the contents of address 0.03H (1010B) with the contents of address 0.2FH (0110B) and store the result (0010B) in address 0.03H.

(0.03H) ← (0.03H) ∧ (0.2FH)


```
MEM003   MEM 0.03H
MEM02F   MEM 0.2FH
         MOV   MEM003, #1010B
         MOV   MEM02F, #0110B
         AND   MEM003, MEM02F
```

(4) AND m, #n4 **AND between data memory and immediate data**

<1> OP code



<2> Function

(m) ← (m) ∧ n4

ANDs the contents of a specified data memory address with specified immediate data, and stores the result in the data memory address.

<3> Example 1

To reset bit 3 (MSB) of address 0.03H.

(0.03H) ← (0.03H) ∧ 0111B

Address 0.03H

0	×	×	×

×: don't care

```
MEM003   MEM   0.03H
         AND   MEM003, #0111B
```

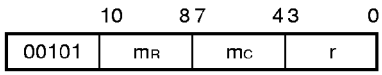
Example 2

To reset all the bits of address 0.03H.

```
MEM003   MEM   0.03H
         AND   MEM003, #0000B
   or,
MEM003   MEM   0.03H
         MOV   MEM003, #00H
```

(5) XOR r, m — Exclusive OR between general register and data memory

<1> OP code



<2> Function

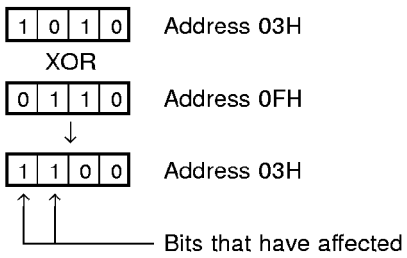
(r) ← (r) ⊽ (m)

Exclusive-ORs the contents of a specified data memory address with the contents of a specified general register, and stores the result in the general register.

<3> Example 1

To compare the contents of address 0.03H with those of address 0.0FH, set and store in address 0.03H bits not in agreement. If all the bits of address 0.03H are reset (i.e., if the address 0.03H contents are the same as those of address 0.0FH), jump to LBL1; otherwise, to jump to LBL2.

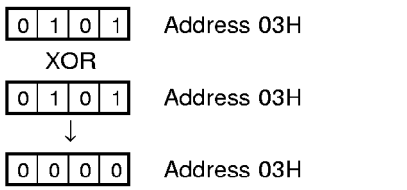
This example compares the status of an alternate switch (address 0.03H contents) with the internal status (address 0.0FH contents) and to branch to the processing of the switch that has affected.




```
MEM003   MEM   0.03H
MEM00F   MEM   0.0FH
         XOR   MEM003, MEM00F
         SKNE  MEM003, #00H
         BR    LBL1
         BR    LBL2
```

Example 2

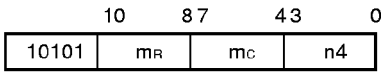
To clear the address 0.03H contents




```
MEM003   MEM   0.03H
         XOR   MEM003, MEM003
```

(6) XOR m, #n4 **Exclusive OR between data memory and immediate data**

<1> OP code



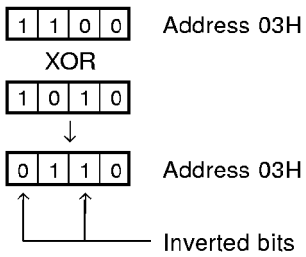
<2> Function

(m) ← (m) ∀ n4

Exclusive-ORs the contents of a specified data memory address with specified immediate data, and stores the result in the data memory address.

<3> Example

To invert bits 1 and 3 of address 0.03H and store the results in address 03H:

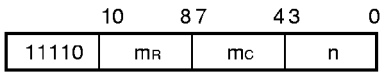



```
MEM003   MEM   0.03H
         XOR   MEM003, #1010B
```

15.5.4 Test instructions

(1) SKT m, #n **Skip next instruction if data memory bits are true**

<1> OP code



<2> Function

CMP ← 0, if (m) ∧ n = n, then skip

★ Skips the next one instruction if the result of ANDing the specified data memory contents with immediate data n is equal to n (Excecutes as NOP instruction).

<3> Example 1

To jump to AAA if bit 0 of address 03H is '1'; if it is '0', to jump to BBB.

```
SKT     03H, #0001B
BR      BBB
BR      AAA
```

Example 2

To skip the next instruction if both bits 0 and 1 of address 03H are '1':

```
SKT     03H, #0011B
```

		b_3	b_2	b_1	b_0	
Skip condition	03H	×	×	1	1	×: don't care

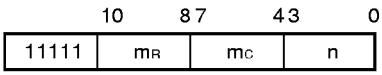
Example 3

The results of executing of the following two instructions are the same:

```
SKT     13H, #1111B
SKE     13H, #0FH
```

(2) SKF m, #n — Skip next instruction if data memory bits are false

<1> OP code



<2> Function

CMP ← 0, if (m) ∧ n = 0, then skip

Skips the next one instruction if the result of ANDing the specified data memory contents with immediate data n is 0 (Executes as NOP instruction).

<3> Example 1

To store immediate data 00H in data memory address 0FH if bit 2 of address 13H is 0; if it is 1, to jump to ABC.

```
MEM013   MEM   0.13H
MEM00F   MEM   0.0FH
         SKF   MEM013, #0100B
         BR    ABC
         MOV   MEM00F, #00H
```

Example 2

To skip the next instruction if both bits 3 and 0 of address 29H are '0'.

```
SKF      29H, #1001B
```

		b_3	b_2	b_1	b_0	
Skip condition	29H	0	×	×	0	×: don't care

Example 3

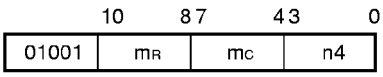
The results of executing the following two instructions are the same:

```
SKF      34H, #1111B
SKE      34H, #00H
```

15.5.5 Compare instructions

(1) SKE m, #n4 **Skip if data memory equal to immediate data**

<1> OP code



★ **<2> Function**

(m) – n4, skip if zero

Skips the next one instruction if the contents of a specified data memory address are equal to the value of the immediate data (Executes as NOP instruction).

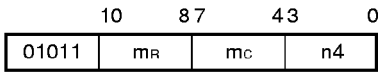
<3> Example

To transfer 0FH to address 24H, if the address 24H contents are 0. If not, jump to OPE1.

```
MEM024  MEM  0.24H
        SKE  MEM024, #00H
        BR   OPE1
        MOV  MEM024, #0FH
OPE1    :
```

(2) SKNE m, #n4 — Skip if data memory not equal to immediate data

<1> OP code



★ #### <2> Function

(m) – n4, skip if not zero

Skips the next one instruction if the contents of a specified data memory address are not equal to the value of the immediate data (Executes as NOP instruction).

<3> Example

To jump to XYZ if the contents of address 1FH are 1 and if the address 1EH contents are 3; otherwise, jump to ABC.

To compare 8-bit data, this instruction is used in the following combination.

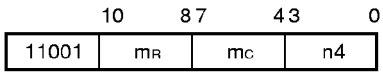

```
MEM01E  MEM   0.1EH
MEM01F  MEM   0.1FH
        SKNE  MEM01F, #01H
        SKE   MEM01E, #03H
        BR    ABC
        BR    XYZ
```

The same operation can be performed by using the compare and zero flags as follows:

```
MEM01E  MEM   0.1EH
MEM01F  MEM   0.1FH
        SET2  CMP, Z            ; CMP flag ← 1, Z flag ← 1
        SUB   MEM01F, #01H
        SUBC  MEM01E, #03H
        SKT1  Z
        BR    ABC
        BR    XYZ
```

(3) SKGE m, #n4 — Skip if data memory greater than or equal to immediate data

<1> OP code



★ #### <2> Function

(m) – n4, skip if not borrow

Skips the next one instruction if the contents of a specified data memory address are greater than the value of the immediate data (Executes as NOP instruction).

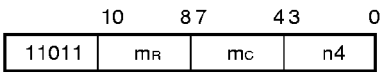
<3> Example

To execute RET if the 8-bit data, stored in addresses 1FH (higher) and 2FH (lower) is greater than immediate data 17H; otherwise, execute RETSK.

```
MEM01F  MEM  0.1FH
MEM02F  MEM  0.2FH
        SKGE MEM01F, #1
        RETSK
        SKNE MEM01F, #1
        SKLT MEM02F, #8      ; 7+1
        RET
        RETSK
```

(4) SKLT m, #n4 — Skip if data memory less than immediate data

<1> OP code



★ #### <2> Function

(m) – n4, skip if borrow

Skips the next one instruction if the contents of a specified data memory address are less than the value of the immediate data (Executes as NOP instruction).

<3> Example

To store 01H in address 0FH if the address 10H contents is greater than immediate data '6'; otherwise, to store 02H in address 0FH.

```
MEM00F  MEM  0.0FH
MEM010  MEM  0.10H
        MOV  MEM00F, #02H
        SKLT MEM010, #06H
        MOV  MEM00F, #01H
```

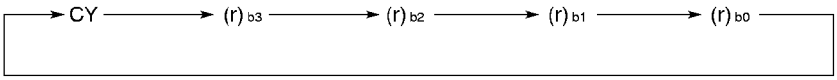
15.5.6 Rotation instruction

(1) RORC r **Rotate right general register with carry flag**

<1> OP code

	3	0
00111	000	0111	r

<2> Function



Rotates the contents of a general register specified by r 1 bit to the right with the carry flag.

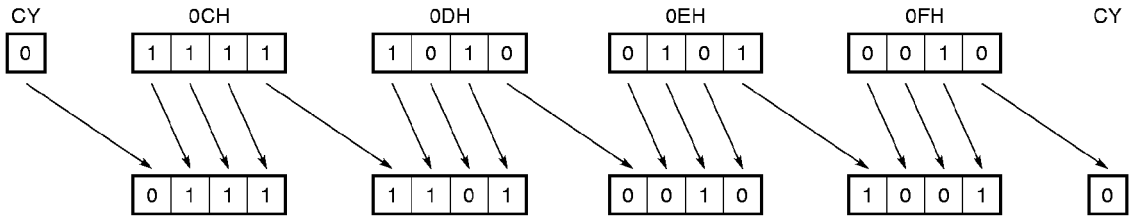
<3> Example 1

To rotate the value of address 0.00H (1000B) 1 bit to the right when row address 0 (0.00H-0.0FH) of bank 0 is specified as a general register (RPH = 0, RPL = 0). As a result, the value of the address becomes 0100B.

```
                (0.00H) ← (0.00H) ÷ 2
MEM000  MEM     0.00H
        MOV     RPH, #00H       ; General register bank 0
        MOV     RPL, #00H       ; General register row address 0
        CLR1    CY              ; Carry flag ← 0
        RORC    MEM000
```

Example 2

To rotate the value 0FA52H of the data buffer (DBF) 1 bit to the right when row address 0 (0.00H-0.0FH) of bank 0 is specified as a general register (RPH = 0, RPL = 0). As a result, the value of the data buffer becomes 7D29H.




```
MEM00C  MEM     0.0CH
MEM00D  MEM     0.0DH
MEM00E  MEM     0.0EH
MEM00F  MEM     0.0FH
        MOV     RPH, #00H       ; General register bank 0
        MOV     RPL, #00H       ; General register row address 0
        CLR1    CY              ; Carry flag ← 0
        RORC    MEM00C
        RORC    MEM00D
        RORC    MEM00E
        RORC    MEM00F
```

15.5.7 Transfer instructions

(1) LD r, m **Load data memory to general register**

<1> OP code

10	87	43	0
01000	mR	mC	r

<2> Function

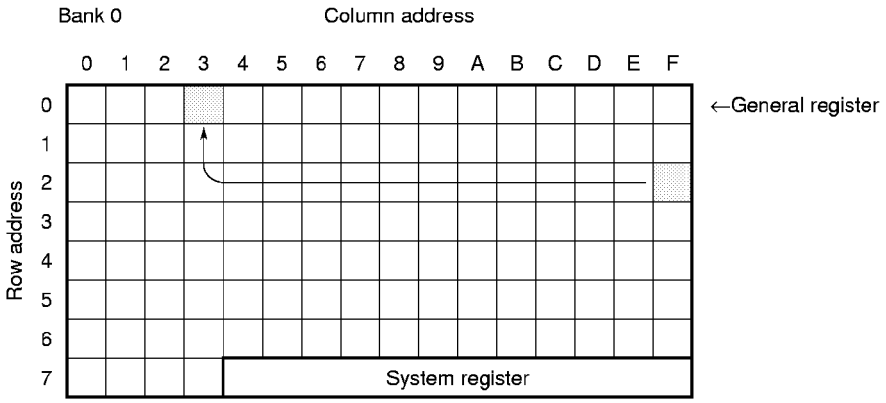
(r) ← (m)

Loads the contents of a specified data memory address to a specified general register.

★ **<3> Example 1**

To load the address 0.2FH contents to address 0.03H.

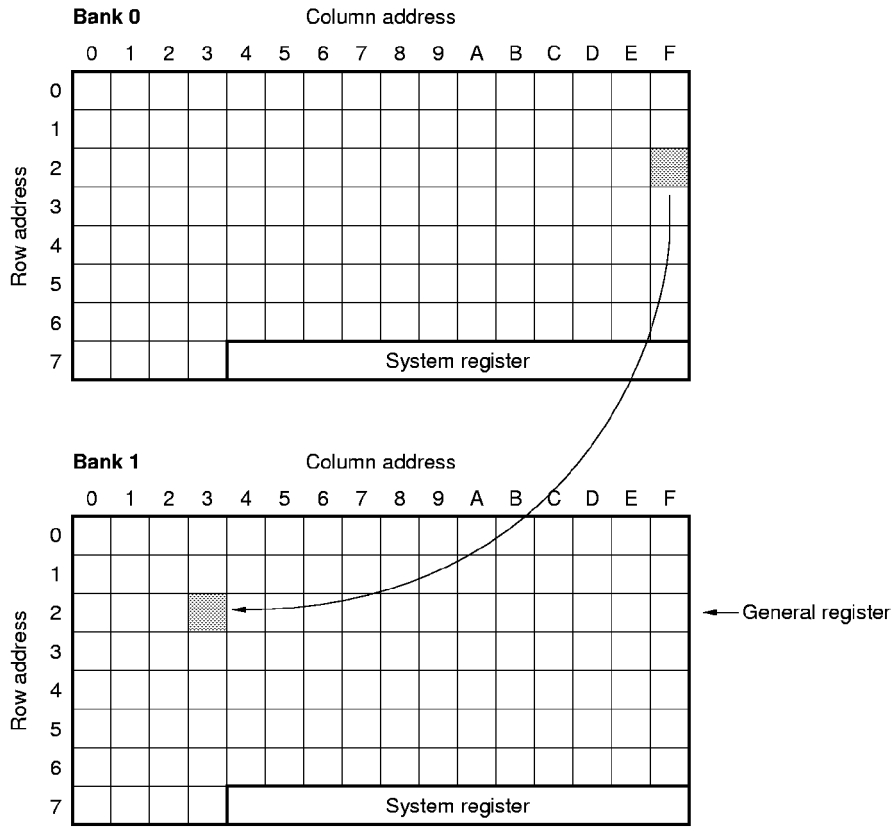
```
                (0.03H) ← (0.2FH)
MEM003  MEM     0.03H
MEM02F  MEM     0.2FH
        MOV     RPH, #00H       ; General register bank 0
        MOV     RPL, #00H       ; General register row address 0
        LD      MEM003, MEM02F
```



Example 2

To store the address 0.2FH contents to address 1.23H, when row address 2 (1.20H-1.2FH) in bank 1 is specified as a general register (RPH = 1, RPL = 4):

```
                (1.23H) ← (0.2FH)
MEM123  MEM     1.23H
MEM02F  MEM     0.2FH
        MOV     RPH, #01H       ; General register bank 1
        MOV     RPL, #04H       ; General register row address 0
        LD      MEM123, MEM02F
```



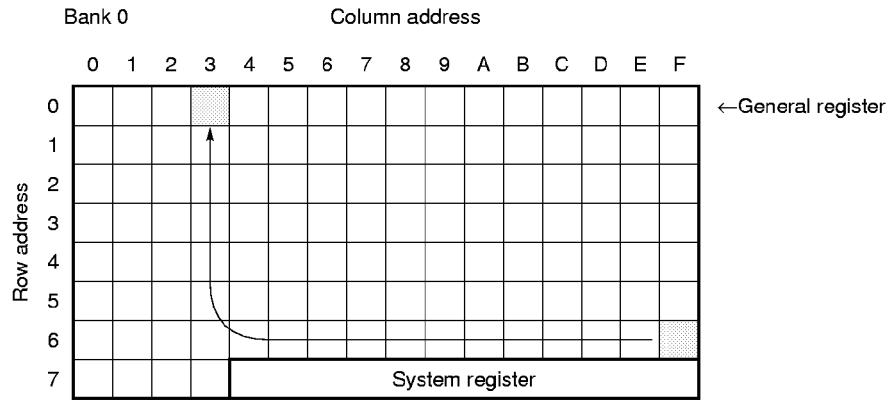
Example 3

To load the address 0.6FH contents to address 0.03H. At this time, if IXE = 1, IXH = 0, IXM = 4, and IXL = 0, i.e., if IX = 0.40H, data memory address 0.6FH can be specified by specifying address 2FH.

IXH ← 00H
IXM ← 04H
IXL ← 00H
IXE flag ← 1
(0.03H) ← (0.6FH)

Address obtained by ORing index register contents 040H with data memory contents 0.2FH

```
MEM003   MEM   0.03H
MEM02F   MEM   0.2FH
         MOV   IXH, #00H          ; IX ← 00001000000B (0.40H)
         MOV   IXM, #04H
         MOV   IXL, #00H
         SET1  IXE                ; IXE flag ← 1
         LD    MEM003, MEM02F
```



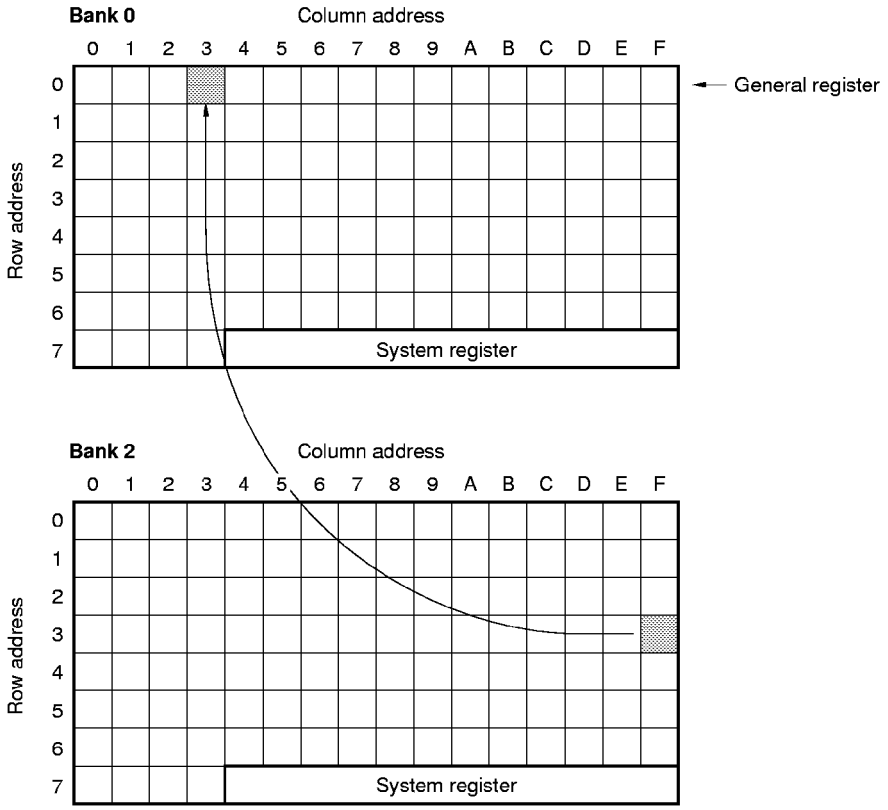
Example 4

To store the address 2.3FH contents to address 0.03H. At this time, data memory address 2.3FH can be specified by selecting data memory address 2FH, if IXE = 1, IXH = 1, IXM = 1, and IXL = 0, i.e., IX = 2.10H.

(0.03H) ← (2.3FH)

└ Address obtained as result of ORing index register contents, 2.10H, and data memory address 0.2FH

```
MEM003   MEM   0.03H
MEM02F   MEM   0.2FH
         MOV   IXH, #01H          ; IX ← 00100010000B (2.10H)
         MOV   IXM, #01H
         MOV   IXL, #00H
         SET1  IXE                ; IXE flag ← 1
         LD    MEM003, MEM02F
```



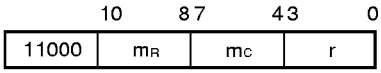
<4> Precaution

The first operand for the LD r, m instruction is a column address. Therefore, if the instruction is described as follows, the column address for the general register is 03H:

```
MEM013   MEM   0.13H
MEM02F   MEM   0.2FH
         LD    MEM013, MEM02F
                  └── Indicates general register column address. Low-order 4 bits (in this case,
                      03H) are valid. In this case, address 03H is specified, if row address 0 in
                      bank 0 is selected as general register.
```

(2) ST m, r — Store general register to data memory

<1> OP code



<2> Function

(m) ← (r)

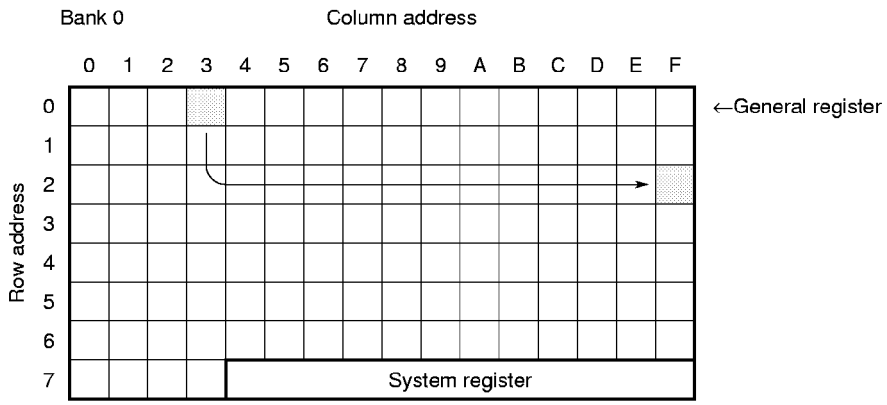
Stores the contents of a specified general register to a specified data memory address.

★ **<3> Example 1**

To store the contents of address 0.03H in address 0.2FH.

(0.2FH) ← (0.03H)

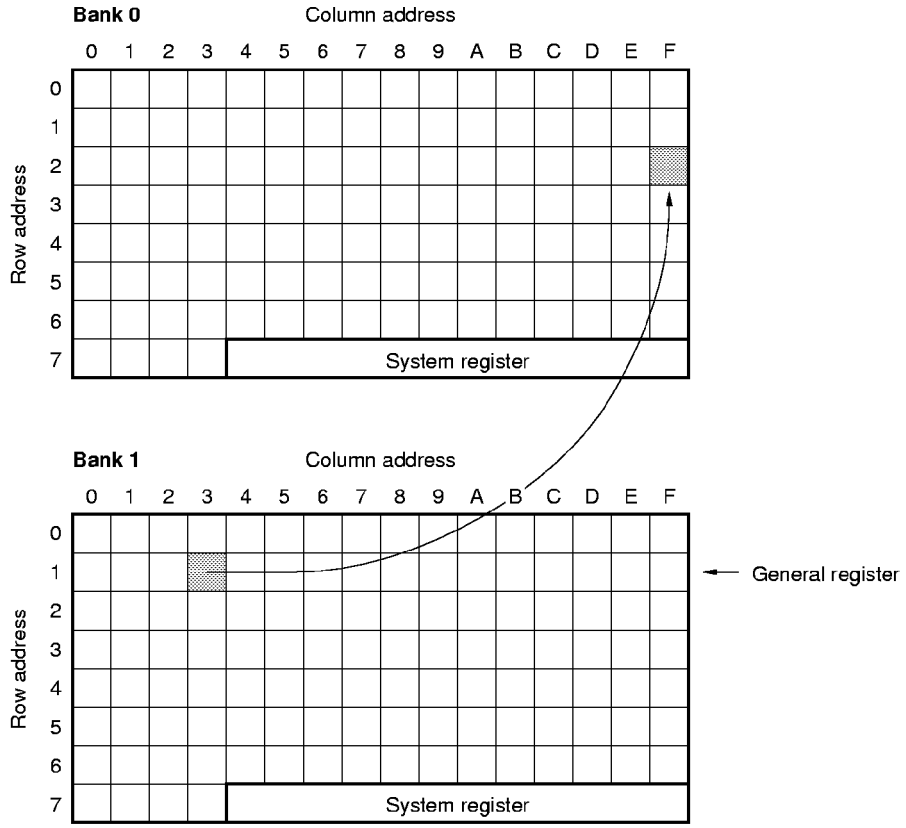
```
MOV   RPH, #00H      ; General register bank 0
MOV   RPL, #00H      ; General register row address 0
ST    2FH, 03H       ; Transfers general register contents to data memory
```



Example 2

To store the address 1.13H contents to address 0.2FH. Row address 1 in bank 1 (1.10H-1.1FH) is specified as the general register by the register pointer.

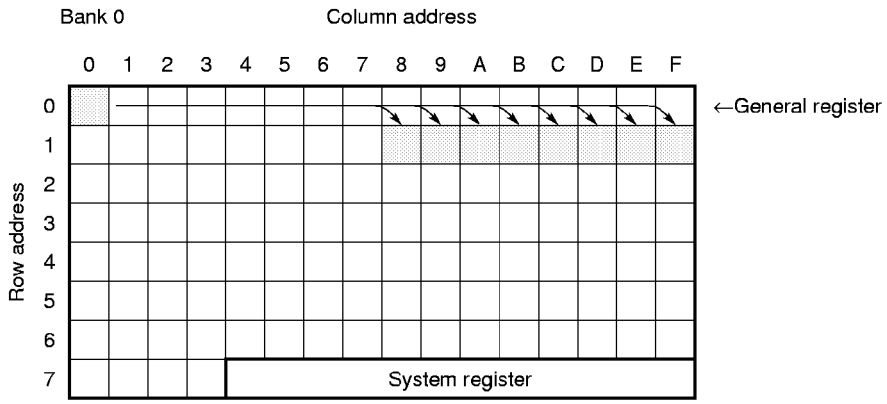
```
    (0.2FH) ← (1.13H)
MEM02F  MEM  0.2FH
MEM113  MEM  1.13H
        MOV  RPH, #01H          ; General register bank 1
        MOV  RPL, #02H          ; General register row address 1
        ST   MEM02F, MEM113 ; Transfers general register contents to data memory
```



Example 3

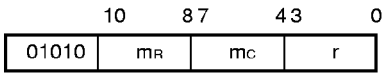
To store the contents of 0.00H in addresses 0.18H through 0.1FH. Data memory (18H to 1FH) is addressed by the index register.

```
(0.18H) ← (0.00H)
(0.19H) ← (0.00H)
          :
          :
          :
          :
(0.1FH) ← (0.00H)
          MOV   IXH, #00H          ; IX ← 00000000000B (0.00H)
          MOV   IXM, #00H
          MOV   IXL, #00H          ; Specifies address 0.00H in data memory.
MEM018    MEM   0.18H
MEM000    MEM   0.00H
  LOOP1:
          SET1  IXE                ; IXE flag ← 1
          ST    MEM018, MEM000     ; (0.1 x H) ← (0.00H)
          CLR1  IXE                ; IXE flag ← 0
          INC   IX                 ; IX ← IX + 1
          SKGE  IXL, #08H
          BR    LOOP1
```



(3) MOV @r, m — Move data memory to destination indirect

<1> OP code



<2> unction

When MPE = 1

(MP,(r)) ← (m)

When MPE = 0

(BANK, mR, (r)) ← (m)

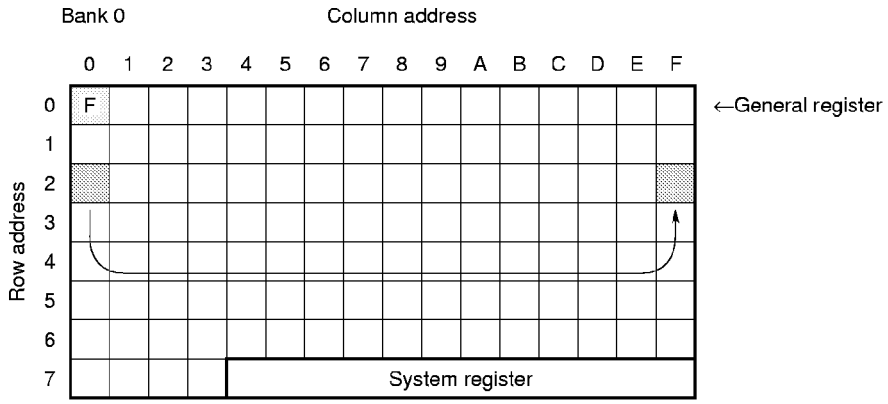
Stores the contents of a specified data memory address to the data memory addressed by the contents of a specified general register.

When MPE = 0, transfer is executed in the same row address of the same bank.

<3> Example 1

To store the contents of address 0.20H in address 0.2FH with the MPE flag cleared to 0. The destination data memory source address is the same row address as that of the transfer source, and the contents of the general register at address 0.00H are the column address.

```
                (0.2FH) ← (0.20H)
MEM000  MEM  0.00H
MEM020  MEM  0.20H
        CLR1 MPE                ; MPE flag  0
        MOV  MEM000,#0FH        ; Sets column address in general register
        MOV  @MEM000, MEM020 ; Stores
```

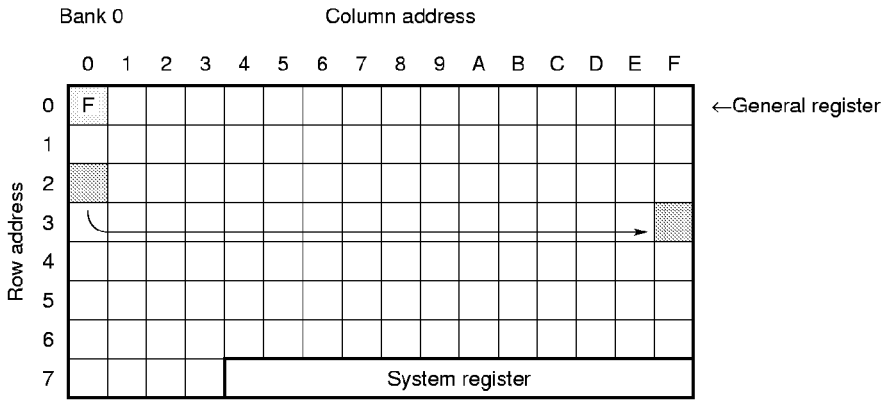


Example 2

To store the contents of address 0.20H in address 0.3FH with the MPE flag set to 1. The row address of the data memory at the transfer destination is specified is the contents of memory pointer MP, and the column address is the contents of the general register at address 0.00H.

```
                (0.3FH) ← (0.20H)
MEM000  MEM  0.00H
MEM020  MEM  0.20H
```

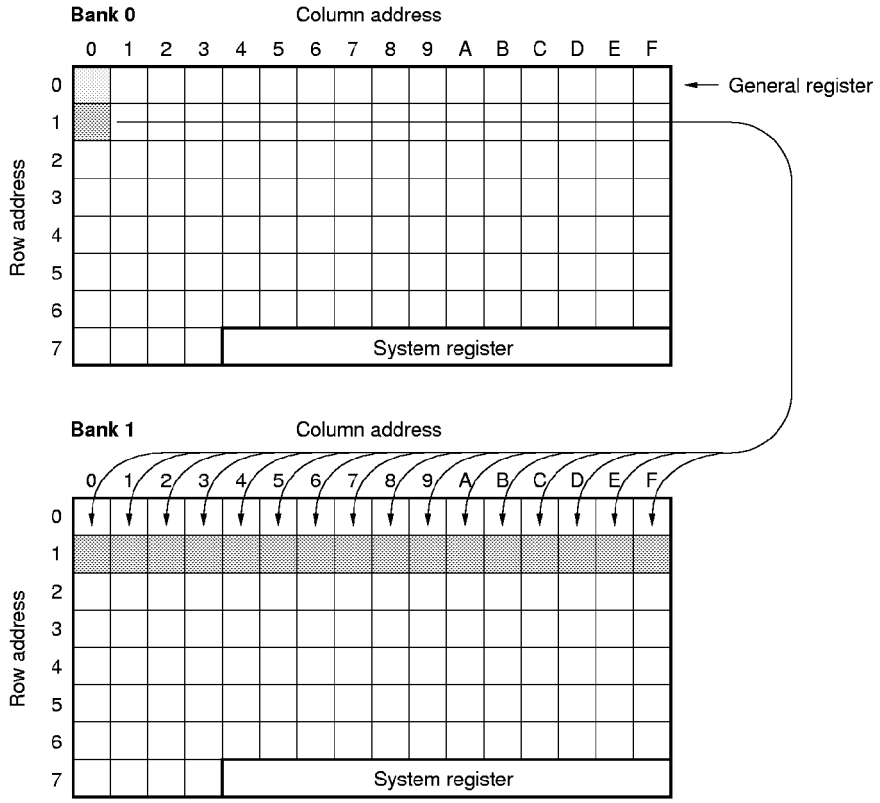
```
MOV   RPH, #00H          ; General register bank 0
MOV   RPL, #00H          ; General register row address 0
MOV   00H, #0FH          ; Sets column address in general register
MOV   MPH, #00H          ; Sets row address in memory pointer
MOV   MPL, #03H          ;
SETI  MPE                ; MPE flag ← 1
MOV   @MEM000, MEM020 ; Stores
```



Example 3

To store the address 0.1FH contents to addresses 1.10H through 1.1FH:

```
          (1.10H) ← (0.10H)
          (1.11H) ← (0.10H)
                  :
                  :
          (1.1FH) ← (0.10H)
MEM000    MEM   0.00H
MEM010    MEM   0.10H
          MOV   RPH, #00H          ; General register bank 0
          MOV   RPL, #00H          ; General register row address 0
          MOV   MEM000, #00H       ; Sets column address in general register
          MOV   MPH, #00H          ; Sets bank 1 and row address 1
          MOV   MPL, #09H          ; in memory pointer
          SET1  MPE                ; MPE flag ← 1
LOOP1:
          MOV   @MEM000, MEM010    ; ((MP), (00H)) ← (10H)
          ADD   MEM000, #01H       ; Column address + 1
          SKT1  CY                 ; Finished up to address 1FH in bank 1
          BR    LOOP1
```



(4) MOV m, @r — Move data memory to destination indirect

<1> OP code

10	8 7	4 3	0
11010	m_R	m_C	r

<2> Function

When MPE = 1

(m) ← (MP,(r))

When MPE = 0

(m) ← (BANK,m_R, (r))

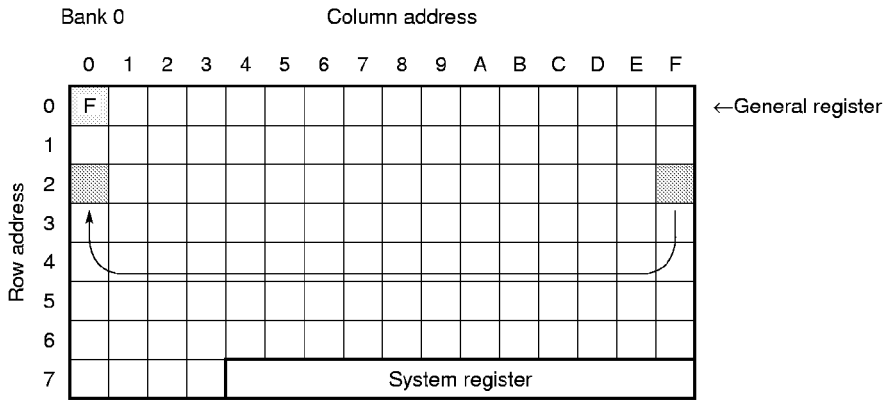
Stores the contents of the data memory addressed by the contents of a specified general register to another data memory address.

When MPE = 0, transfer is executed in the same row address of the same bank.

<3> Example 1

To store the contents of address 0.2FH in address 0.20H with the MPE flag cleared to 0. The data memory at the transfer source is at the same row address as the destination, and the column address is the contents of the general register at address 0.00H.

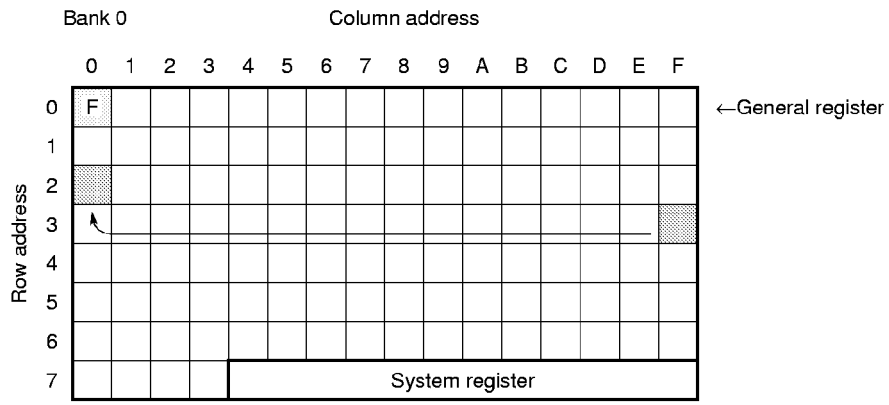
```
                (0.20H) ← (0.2FH)
MEM000  MEM  0.00H
MEM020  MEM  0.20H
        CLR1 MPE                ; MPE flag ← 0
        MOV  MEM000, #0FH       ; Sets column address in general register
        MOV  MEM020, @MEM000 ; Stores
```



Example 2

To store the contents of address 0.3FH in address 0.20H with the MPE flag set to 1. The row address of the data memory at the transfer source is the contents of the memory pointer, and the column address is the contents of the general register at address 0.00.

```
                (0.20H) ← (0.3FH)
MEM000  MEM     0.00H
MEM020  MEM     0.20H
        MOV     MEM000, #0FH        ; Sets column address in general register
        MOV     MPH, #00H           ; Sets row address in memory pointer
        MOV     MPL, #03H           ;
        SETI    MPE                 ; MPE flag ← 1
        MOV     MEM020, @MEM000 ; Stores
```

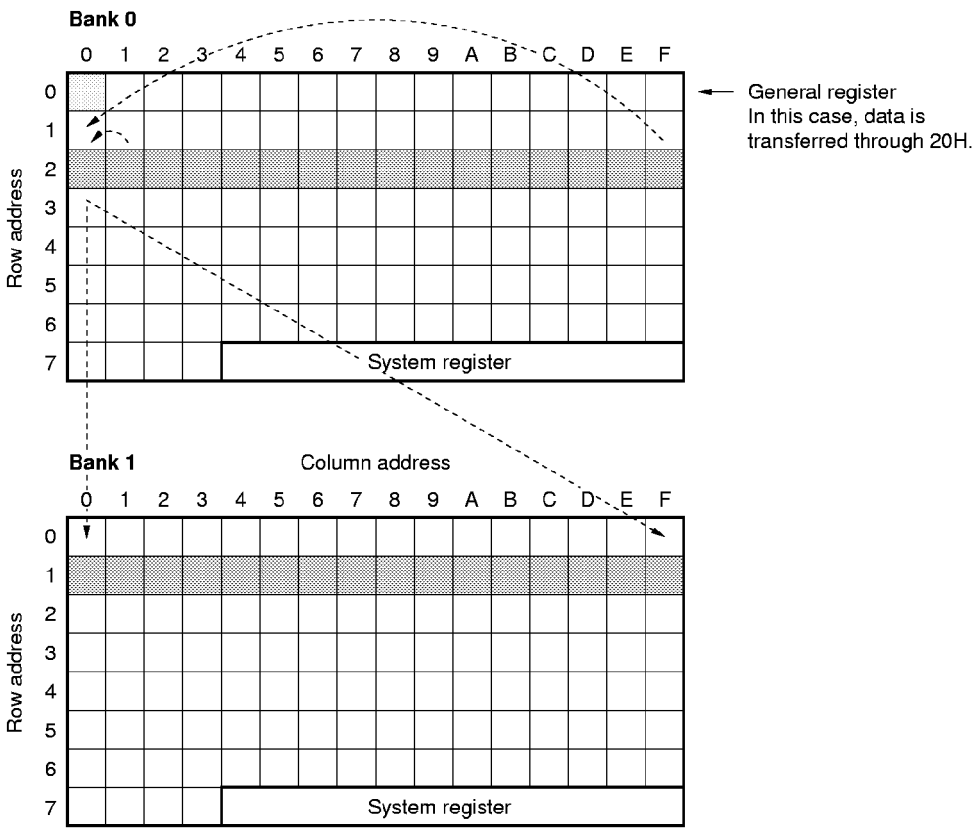


Example 3

To store the addresses 0.20H contents through 0.2FH to addresses 1.10H through 1.1FH. The row addresses for the transfer source data memory addresses are the same as those for the data memory addresses used for relaying. The column addresses are specified by the general register at address 0.00H.

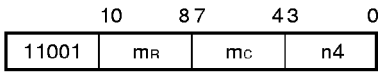
The row addresses for the destination data memory addresses are specified by the memory pointer M contents. The column addresses are specified by the address 0.00H contents.

```
                (1.10H) ← (0.20H)
                (1.11H) ← (0.21H)
                (1.12H) ← (0.22H)
                        :
                        :
                (1.1FH) ← (0.2FH)
MEM000  MEM  0.00H
MEM020  MEM  0.20H
        CLR1 MPE                ; MPE flag ← 0
        MOV  MEM000, #00H       ; Sets column address in general register
        MOV  MPH, #00H          ; Sets memory pointer
        MOV  MPL, #09H          ; Bank 1, row address 1
LOOP1:
        MOV  MEM020, @MEM000    ; (20H) ← (2, (00H))
        SET1 MPE                ; MPE flag ← 1
        MOV  @MEM000, MEM020    ; ((MP), (00H)) ← (20H)
        CLR1 MPE                ; MPE flag ← 0
        ADD  MEM000, #01H       ; Column address + 1
        SKT1 CY                 ; Finished up to 1FH in bank 1
        BR   LOOP1
```



(5) MOV m, #n4 **Move immediate data to data memory**

<1> OP code



<2> Function

(m) ← n4

Stores immediate data in a specified data memory address.

<3> Example 1

To store immediate data 0AH in data memory address 0.50H.

```
            (0.50H) ← 0AH
MEM050      MEM  0.50H
            MOV  MEM050, #0AH
```

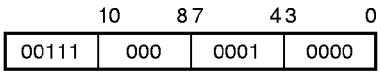
Example 2

To store immediate data 07H in address 0.32H when data memory address 0.00H is specified and if IXH = 0, IXM = 3, IXL = 2, and IXE flag = 1.

```
            (0.32H) ← 07H
MEM000      MEM  0.00H
            MOV  IXH, #00H          ; IX ← 00000110010B (0.32H)
            MOV  IXM, #03H
            MOV  IXL, #02H
            SET1 IXE                ; IXE flag ← 1
            MOV  MEM000, #07H
```

(6) MOVT DBF, @AR **Move program memory data specified by AR to DBF**

<1> OP code



<2> Function

SP ← SP – 1, ASR ← PC, PC ← AR,

DBF ← (PC), PC ← ASR, SP ← SP + 1

Stores the program memory contents, addressed by address register AR, in data buffer DBF.

Because this instruction temporarily uses one level of stack, pay attention to the nesting of subroutines and interrupts.

<3> Example 1

To transfer 16 bits of table data to data buffers (DBF3, DBF2, DBF1, and DBF0) according to the values of the address registers (AR3, AR2, AR1, and AR0) in the system register.

```
                ; *
                ; **    Table data
                ; *
Address         ORG     0010H
0010H           DW      0000000000000000B ; (0000H)
0011H           DW      1010101111001101B ; (0ABCDH)
                                :
                                :
                ; *
                ; **    Table reference program
                ; *
                MOV     AR3, #00H       ; AR3 ← 00H     Sets 0011H in address register
                MOV     AR2, #00H       ; AR2 ← 00H
                MOV     AR1, #01H       ; AR1 ← 01H
                MOV     AR0, #01H       ; AR0 ← 01H
                MOVT    DBF, @AR        ; Transfers data of address 0011H to DBF
```

In this case, the data is stored in DBF as follows:

DBF3 = 0AH
DBF2 = 0BH
DBF1 = 0CH
DBF0 = 0DH

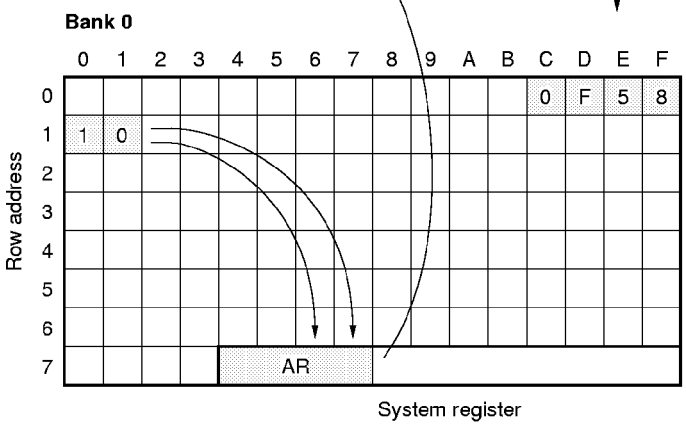
Example 2

To set channel numbers in data memory addresses 0.10H and 011H, obtain the PLL (value N) frequency division, and transfers it to the PLL register:

```
                ; *
                ; **    Table data for value N
                ; *
Address         ORG     0010H
0010H           DW      0F58H           ; 87.5 MHz (minimum frequency:  channel 00)
0011H           DW      0F5CH           ; 87.6 MHz
0012H           DW      0F60H           ; 87.7 MHz
0013H           DW      0F64H           ; 87.8 MHz
0014H           DW      0F68H           ; 87.9 MHz
0015H           DW      0F6CH           ; 88.0 MHz
0016H           DW      0F70H           ; 88.1 MHz
0017H           DW      0F74H           ; 88.2 MHz
                                :
                                :
                ; *
                ; **    Value N setting program
                ; *
```

```
MEM010  MEM   0.10H
MEM011  MEM   0.11H
        MOV   RPH, #00H        ; RPH ← 00H Set row address 7
        MOV   RPL, #0EH        ; RPL ← 0EH (0.70H-0.7FH) as general register
        MOV   AR3, #00H        ; AR3 ← 0
        MOV   AR2, #00H        ; AR2 ← 0
        LD    AR1, MEM010      ; AR1 ← 10H Channel data, high
        LD    AR0, MEM011      ; AR0 ← 11H Channel data, low
        ADD   AR1, #01H        ; 0010H as table data start address
        ADDC  AR2, #00H        ; Adds 0010H to address register as transfer is made
        ADDC  AR3, #00H        ; from address
        MOVT  DBF, @AR         ; Stores table data in DBF
        PUT   PLLR, DBF        ; Transfers value N to PLL register (PLLR)
              :
              :
```

```
;*
;**      Table data
;*
0010H   DW    0F58H    ; 87.5 MHz
0011H   DW    0F5CH    ; 87.6 MHz
0012H   DW    0F60H    ; 87.7 MHz
0013H   DW    0F64H    ; 87.8 MHz
0014H   DW    0F68H    ; 87.9 MHz
```



<4> Precaution 1

The number of bits, in address registers AR3, AR2, AR1, and AR0, differ depending on the microcontroller model to be used. Refer to the Data Sheet for your microcontroller.

Precaution 2

When using the MOVT instruction, one stack level is used. Therefore, pay attention to the stack level, when the instruction is used in subroutines or interrupt processing.

(7) PUSH AR — Push address register

<1> OP code

10	87	43	0
00111	000	1101	0000

<2> Function

SP ← SP–1,

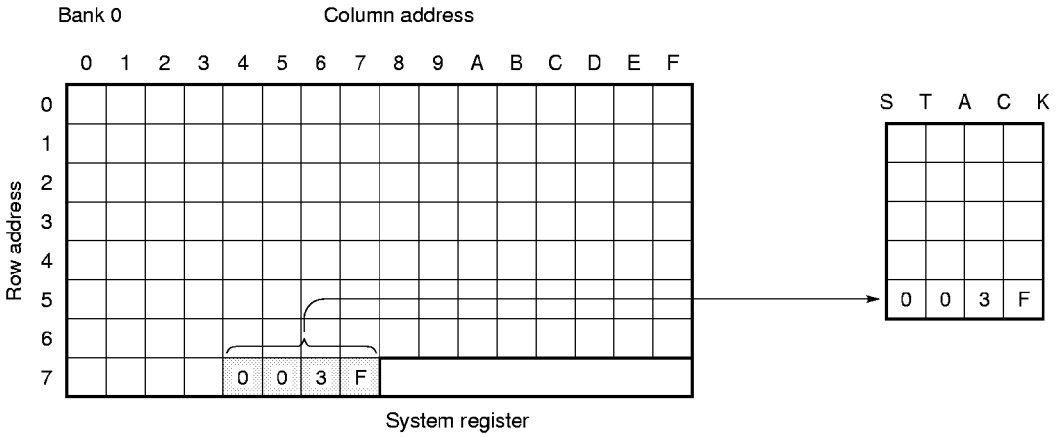
ASR ← AR

Decrements the stack pointer SP and stores the value of the address register AR in the address stack register specified by the stack pointer.

<3> Example 1

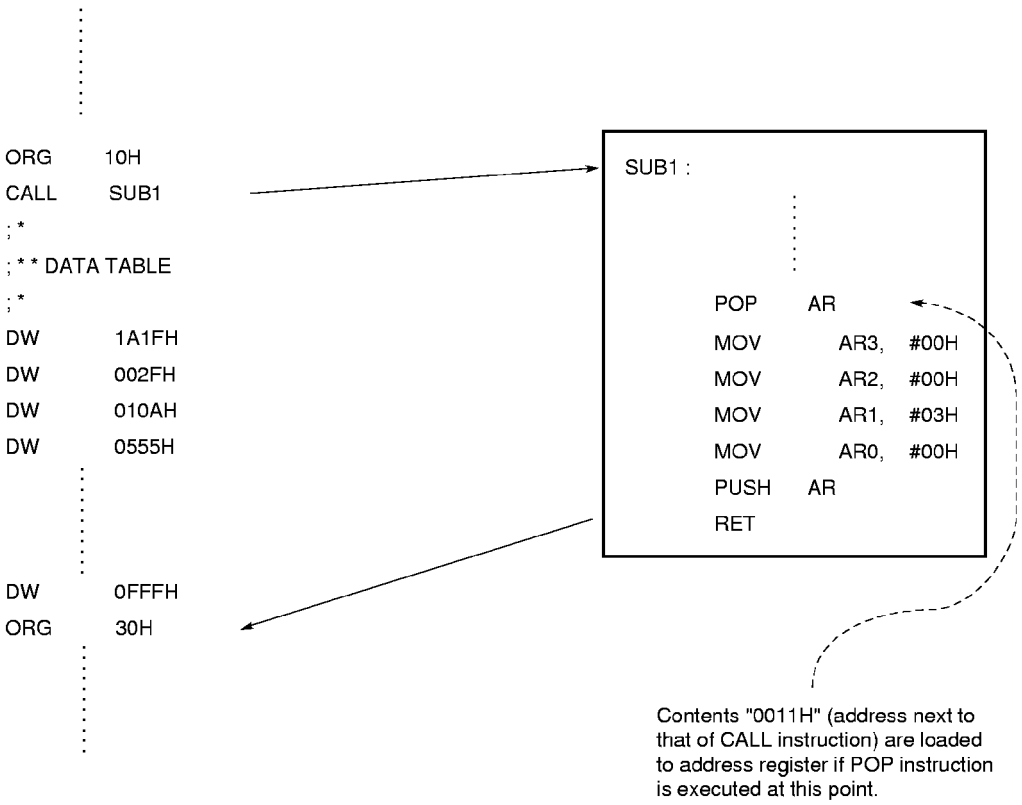
To set 003FH in the address register and store it in stack.

```
MOV     AR3, #00H
MOV     AR2, #00H
MOV     AR1, #03H
MOV     AR0, #0FH
PUSH    AR
```



Example 2

To set the return address of a subroutine in the address register and return if there is a data table following the CALL instruction.



(8) POP AR — Pop address register

<1> OP code

00111	000	1100	0000

<2> Function

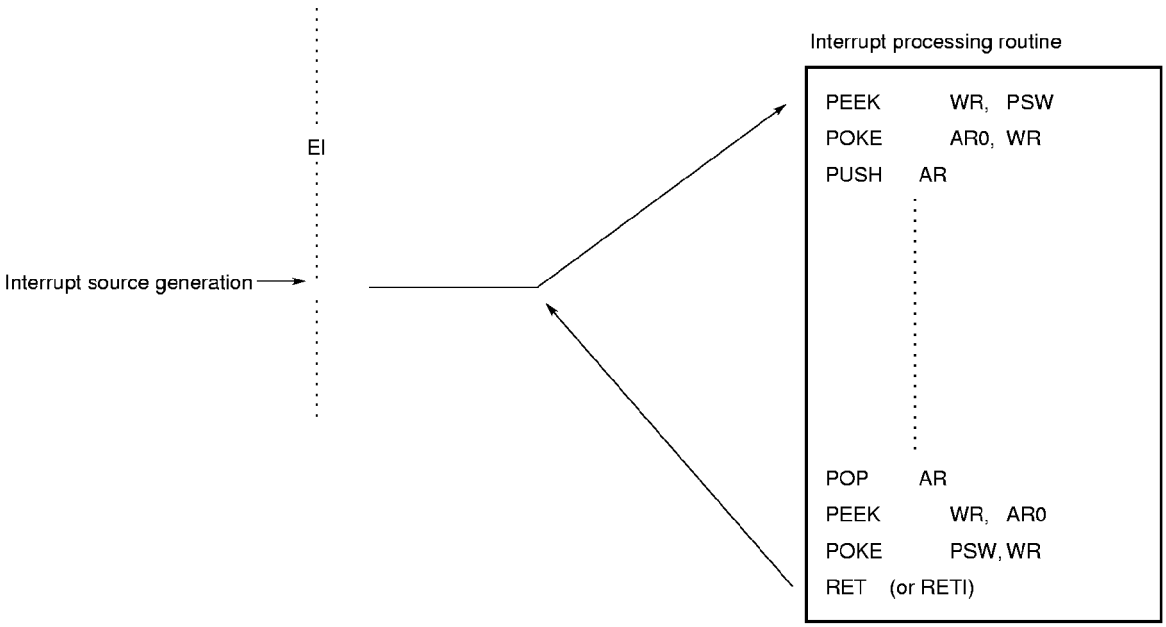
AR ← ASR

SP ← SP+1

Loads the contents of the address stack register specified by the stack pointer to address register AR, and then increments the value of stack pointer SP.

<3> Example

If the PSW contents have been changed in an interrupt processing routine when interrupt processing is performed and you want to transfer the PSW contents to the address register through WR, you should save them at the beginning of the interrupt processing, to the address stack register by the PUSH instruction. Then, restore them to the address register by the POP instruction before return, and transfer them to PSW through WR.



(9) PEEK WR, rf

Peek register file to window register

<1> OP code

00111	rfB	0011	rfC

<2> Function

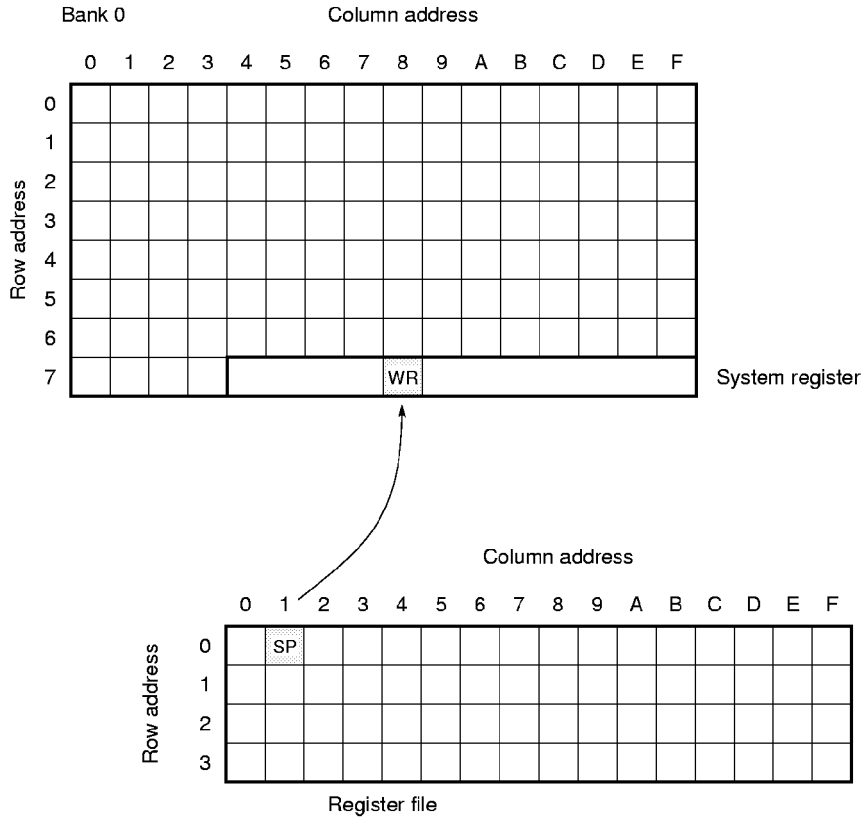
WR ← (rf)

Stores the register file contents to window register WR.

<3> Example

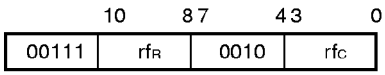
To store the stack pointer contents (SP) at address 01H in the register file to the window register.

```
PEEK    WR, SP
```



(10) POKE rf, WR — Poke window register to register file

<1> OP code



<2> Function

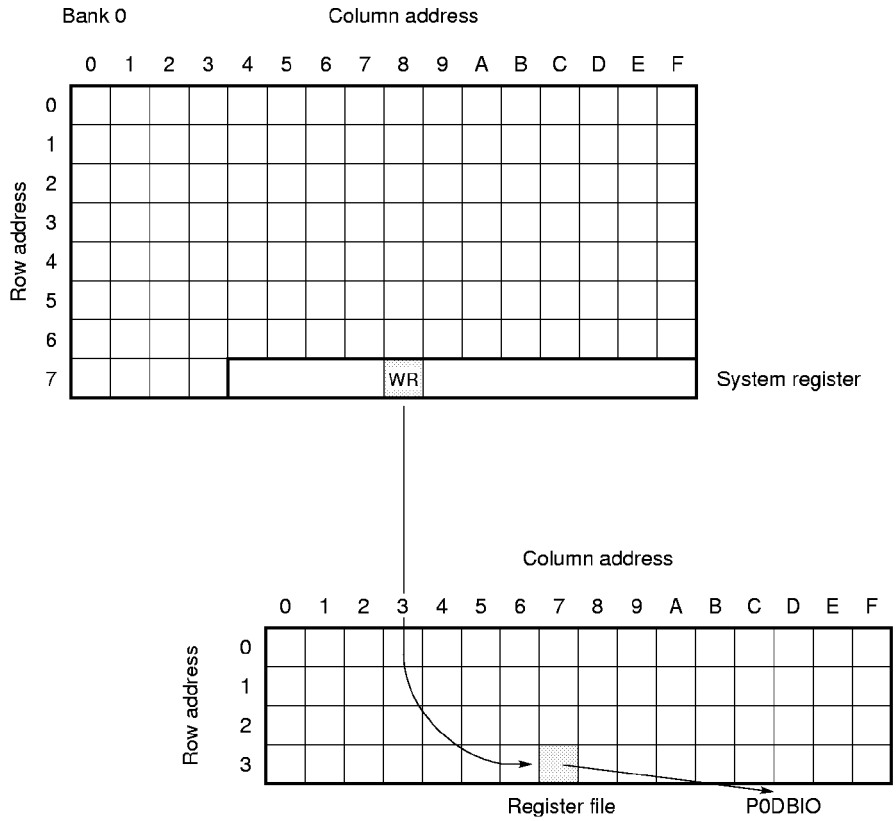
(rf) ← WR

Stores the window register WR contents to the register file.

<3> Example

To store immediate data 0FH in P0DBIO of the register file through the window register.

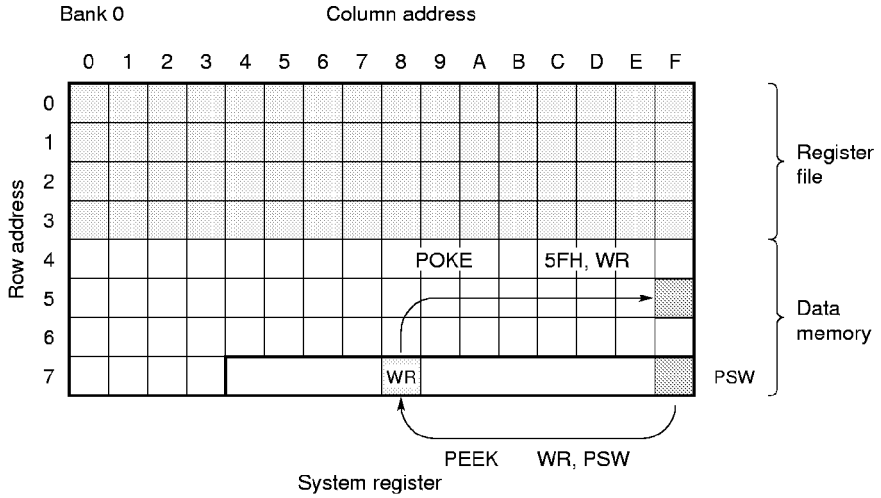
```
MOV     WR, #0FH
POKE    P0DBIO, WR          ; Sets all P0D0, P0D1, P0D2, and P0D3 in output mode
```



<4> Precaution

Addresses 40H through 7FH in the register file appear in data memory in a program. Consequently, the PEEK and POKE instructions can access addresses 40H through 7FH of each bank of the data memory in addition to the register file. For example, these instructions can also be used as follows:

```
MEM05F   MEM   0.5FH
         PEEK  WR, PSW          ; Stores PSW (7FH) contents in system register to WR
         POKE  MEM05F, WR       ; Stores WR contents to data memory at address 5FH
```



(11) GET DBF, p

Get peripheral data to data buffer

<1> OP code

10	8 7	4 3	0
00000	P_H	1011	P_L

<2> Function

DBF ← (p)

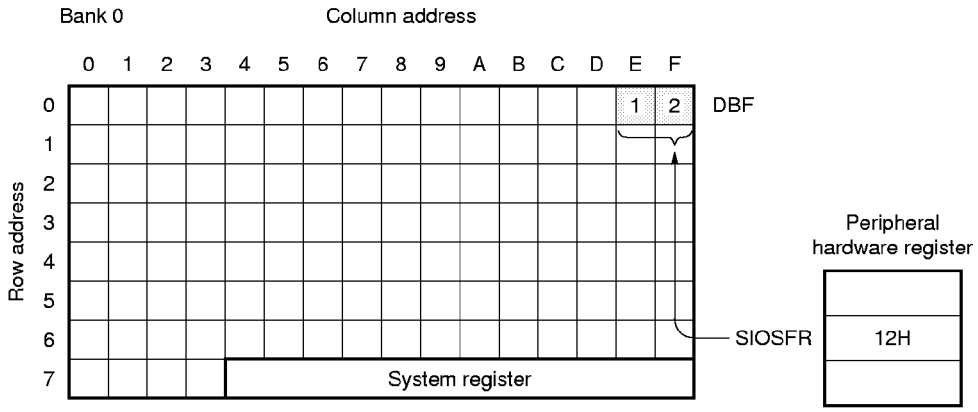
Stores the peripheral circuit contents to data buffer DBF.

DBF is a 16-bit area of addresses 0CH through 0FH of BANK0 of the data memory regardless of the value of the bank register.

<3> Example 1

To store the 8-bit contents of the shift register SIOSFR of the serial interface in data buffers DBF0 and DBF1.

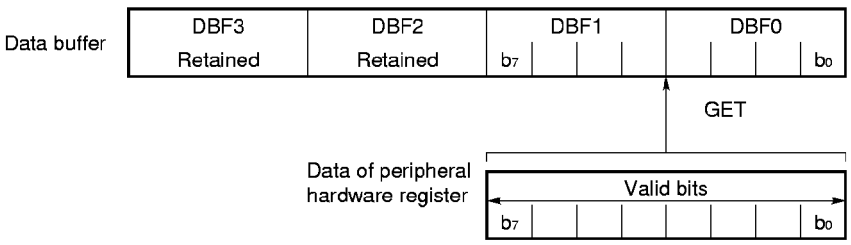
```
GET   DBF,   SIOSFR
```



<4> Precaution

The data buffer is 16 bits wide. The number of bits differs depending on the peripheral hardware to be accessed. For example, when the GET instruction is executed to a peripheral hardware register whose valid bit length is 8 bits, data is stored in the low-order 8 bits of data buffer DBF (DBF1, DBF0).

★



(12) PUT p, DBF — Put data buffer to peripheral

<1> OP code

10	87	43	0
00111	P_M	1010	P_L

<2> Function

(p) ← DBF

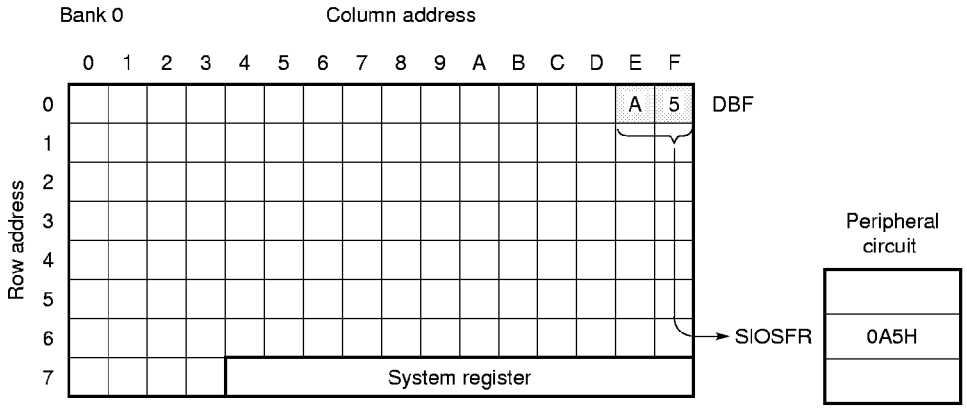
Stores the data buffer DBF contents in the peripheral register.

★ DBF is a 16-bit area of addresses 0CH through 0FH of BANK0 of the data memory regardless of the value of the bank register.

<3> Example 1

To set 0AH and 05H in data buffers DBF1 and DBF0, respectively, and transfer them to the shift register (SIOSFR) for serial interface.

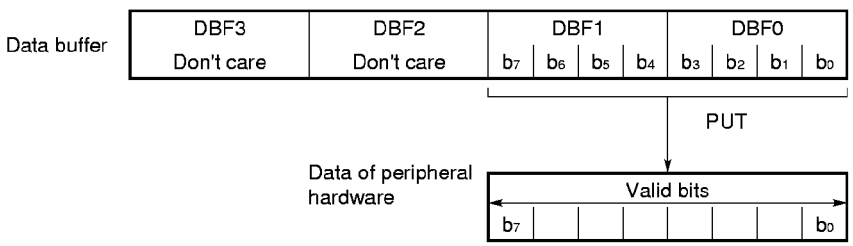
```
MOV     BANK, #00H          ; Data memory bank 0
MOV     DBF0, #05H
MOV     DBF1, #0AH
PUT     SIOSFR, DBF
```



<4> Precaution

The data buffer is 16 bits wide. The number of bits differs depending on the peripheral hardware to be accessed. For example, when the PUT instruction is executed to the peripheral hardware register whose valid bit length is 8-bit, the low-order 8 bits data of data buffer DBF (DBF1, DBF0) is transferred to a peripheral hardware register (DBF3 and DBF2 data is not transferred).

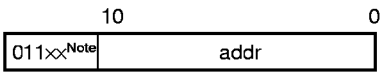
★



15.5.8 Branch instructions

(1) BR addr **Branch to the address**

<1> OP code



Note Refer to **<4> Precaution**

<2> Function

$PC_{10-0} \leftarrow$ addr

Branches to an address specified by addr.

The address range to which execution can be directly branched by this instruction is 8K steps from address 0000H to 1FFFH.

To branch address 2000H or those that follow, use the BR @AR instruction.

<3> Example

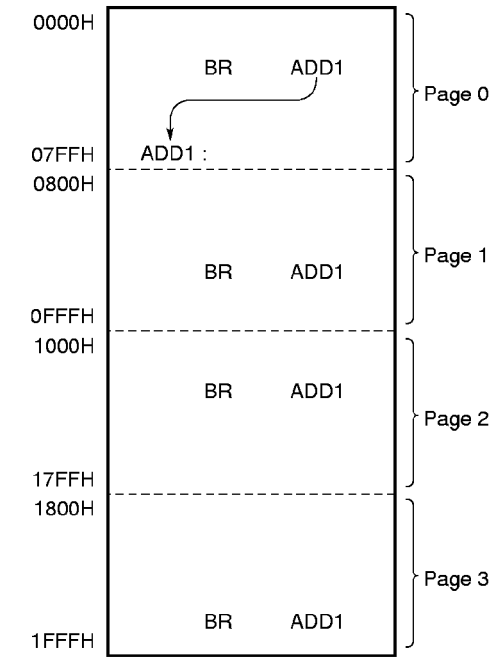
```
FLY     LAB   0FH      ; Defines FLY = 0FH
          :
          :
        BR    FLY      ; Jumps to address 0F
          :
          :
        BR    LOOP1    ; Jumps to LOOP1
          :
          :
        BR    $ + 2    ; Jumps to address two addresses  lower than the current address
          :
        BR    $ – 3    ; Jumps to address three addresses higher than the current address
          :
          :
LOOP1:
```

<4> Precation

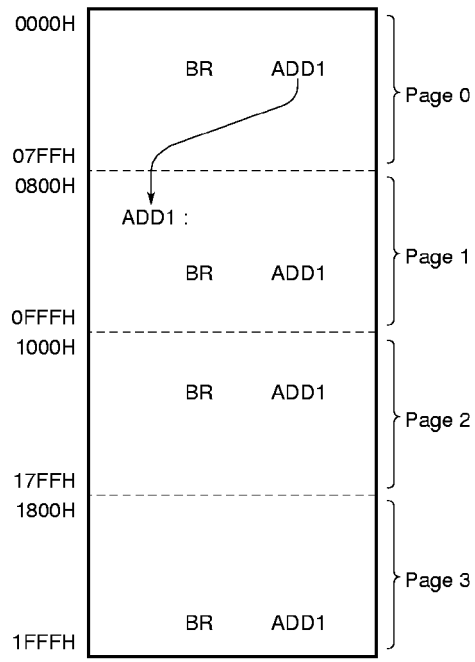
The BR instruction does not use the division of “page”, and the instruction can be written in the ROM addresses 0000H-1FFFH. However, the BR instruction branching within page 0 (addresses 0000H-07FFH) and the BR instruction branching in page 1 (07FFH-0FFFH), and the BR instruction branching in page 2 (1000H-17FFH) and the BR instruction branching in page 3 (17FFH-1FFFH) differ in OP code. The OP codes are 0C in page 0, 0D in page 1, 0E in page 2, and 0F in page 3.

If these instructions are assembled with the 17K-series assembler, the jump destination is automatically referenced.

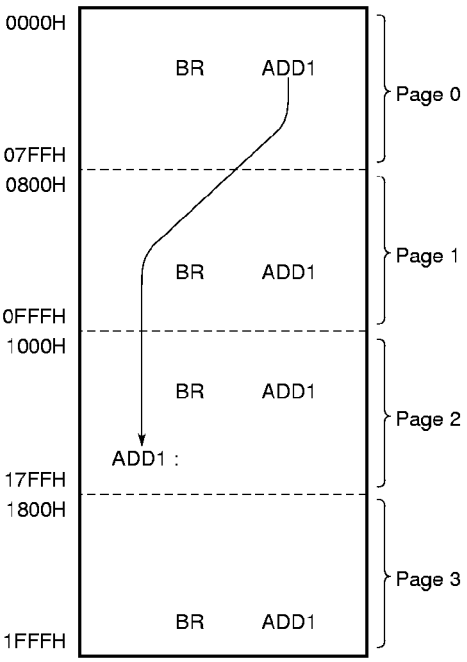
If OP code is 0C
(if jump destination address is in page 0)



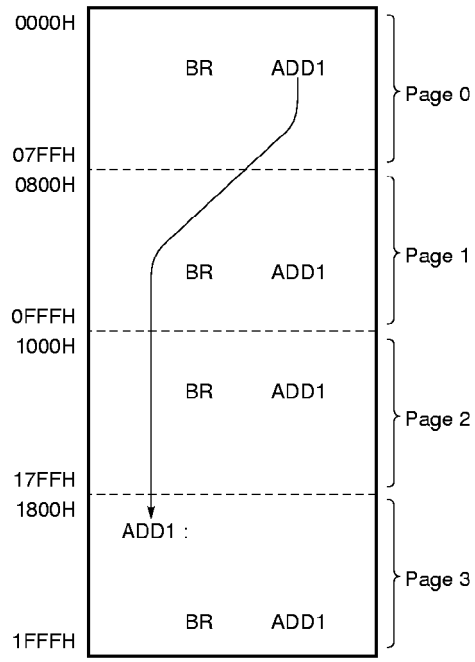
If OP code is 0D
(if jump destination address is in page 1)



If OP code is 0E
(if jump destination address is in page 2)

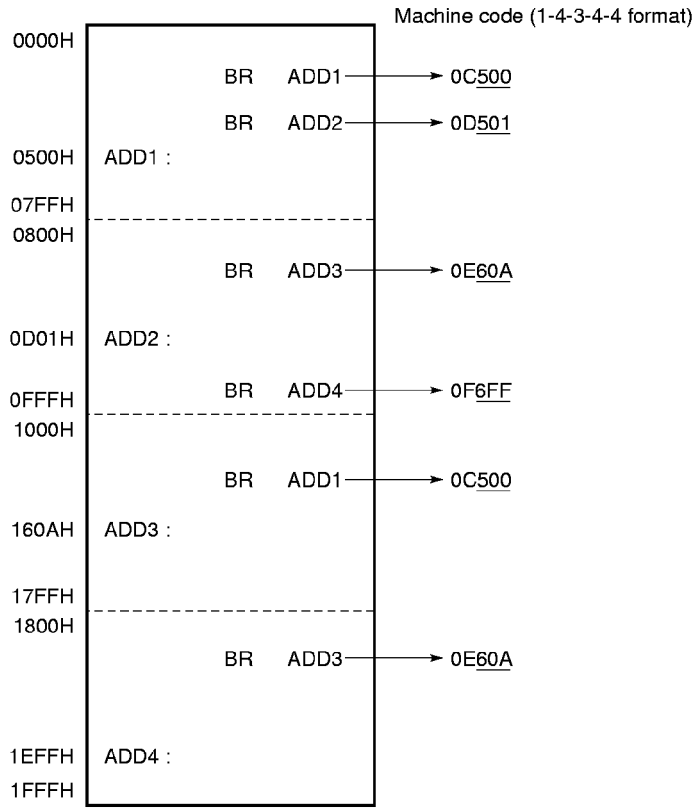


If OP code is 0F
(if jump destination address is in page 3)



To perform patch correction during debugging, it is necessary for the programmer to convert 0C, 0D, 0E, and 0F.

Address conversion is also necessary when the jump destination of the BR instructions are in addresses 0000H-07FFH, 0800H-0FFFH, 1000H-17FFH, and 1800H-1FFFH. In other words, addresses 0000H, 0800H, 1000H, and 1800H are treated as address 000H, starting from which the subsequent addresses are incremented by 1.



Caution The number of pages of each model in the μPD170×× series differs. For details, refer to the Data Sheet of each model.

(2) BR @AR — Branch to the address specified by address register

<1> OP code

00111	000	0100	0000

<2> Function

PC ← AR

Branches to a program address specified by address register AR.

<3> Example 1

To set 003FH in the address registers AR (AR0-AR3) and jump to address 003FH by the BR @AR instruction.

```
MOV     AR3, #00H       ; AR3 ← 00H
MOV     AR2, #00H       ; AR2 ← 00H
MOV     AR1, #03H       ; AR1 ← 03H
MOV     AR0, #0FH       ; AR0 ← 0FH
BR      @AR             ; Jumps to address 003FH
```

Example 2

To change the branch destination as follows according to the contents of data memory address 0.10H.

```
0.10H contents    Branch destination label
  00H       →     AAA
  01H       →     BBB
  02H       →     CCC
  03H       →     DDD
  04H       →     EEE
  05H       →     FFF
  06H       →     GGG
  07H       →     HHH
08H-0FH     →     ZZZ
;*
;** Jump table
;*
ORG     10H
BR      AAA
BR      BBB
BR      CCC
BR      DDD
BR      EEE
BR      FFF
BR      GGG
BR      HHH
BR      ZZZ
```

```
                    :
                    :
                    :
MEM010  MEM     0.10H
        MOV     AR3,  #00H      ; AR3 ← 00H  001 x H in AR
        MOV     AR2,  #00H      ; AR2 ← 00H
        MOV     AR1,  #01H      ; AR1 ← 01H
        MOV     RPH,  #00H      ; General register bank 0
        MOV     RPL,  #02H      ; General register row address 1
        ST      AR0,  MEM010    ; AR0 ← 0.10H
        SKLT    AR0,  #08H
        MOV     AR0,  #08H      ; Sets 08H in AR0 if AR0 contents are greater than 08H
        BR      @AR
```

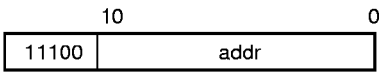
<4> Precaution

The number of bits of the address registers (AR0 through AR3) differs depending on the model. Refer to the Data Sheet of each model.

15.5.9 Subroutine instructions

(1) CALL addr **Call subroutine**

<1> OP code



<2> Function

SP ← SP – 1, ASR ← PC,

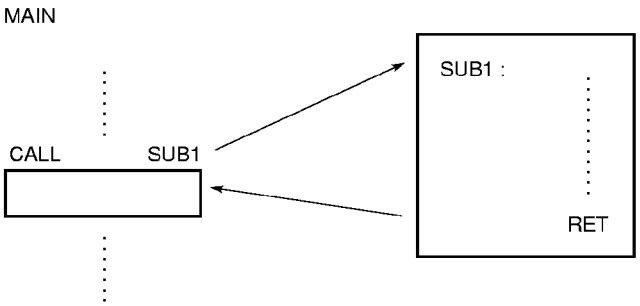
PC_{10-0} ← addr, PAGE ← 0

Increments the value of the program counter (PC), saves it in the stack, and branches to the subroutine whose address is specified by addr.

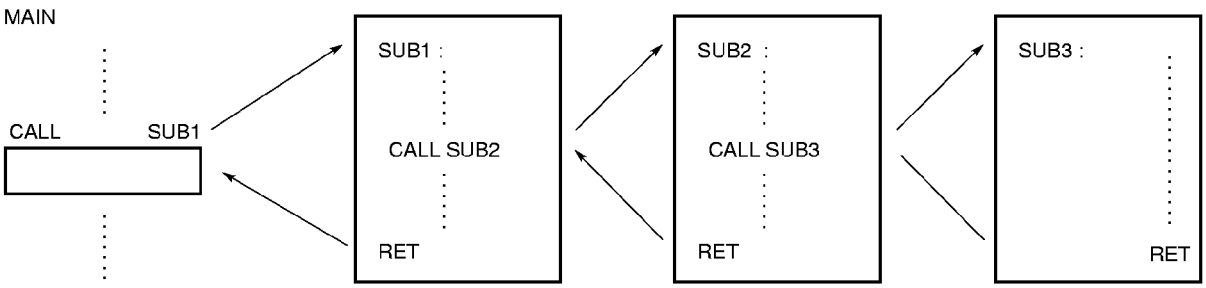
The subroutine that can be called by this instruction is within 2K steps from address 0000H to 07FFH. Therefore, it is recommended that a subroutine that is frequently used be located in the range of address 0000H to 07FFH.

To call a subroutine located at address 0800H or those that follow, use the CALL @AR instruction.

<3> Example 1



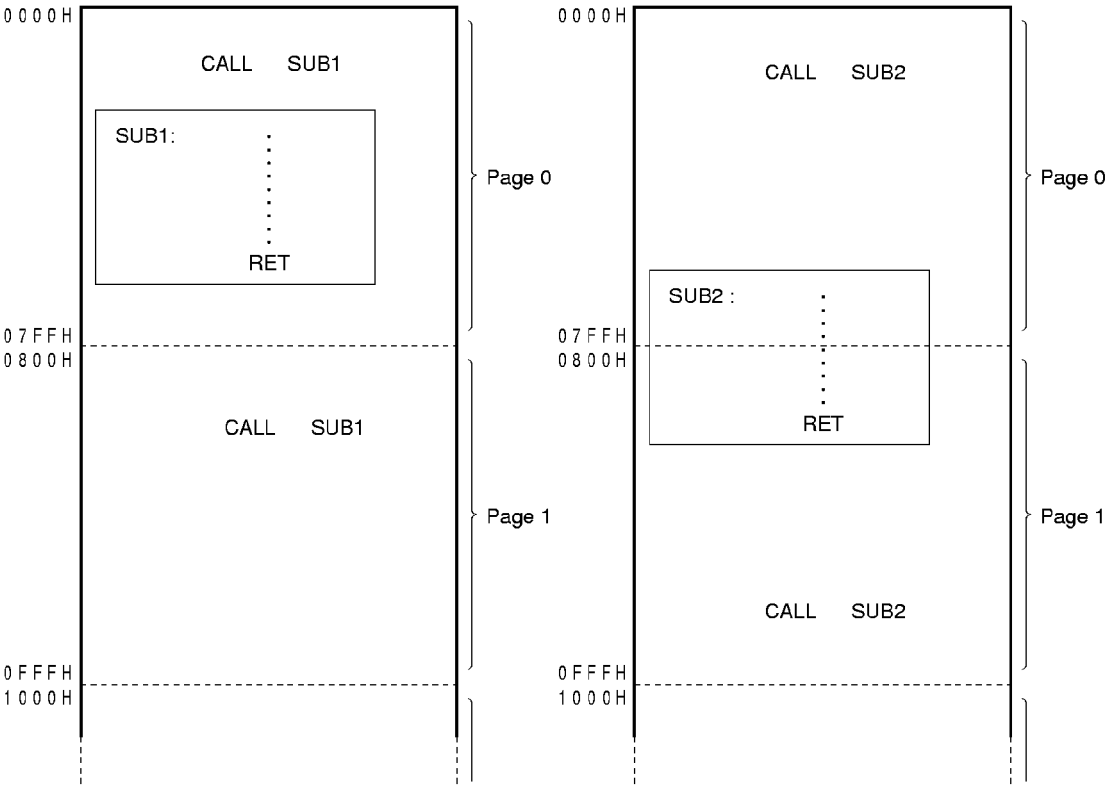
Example 2



<4> Precaution

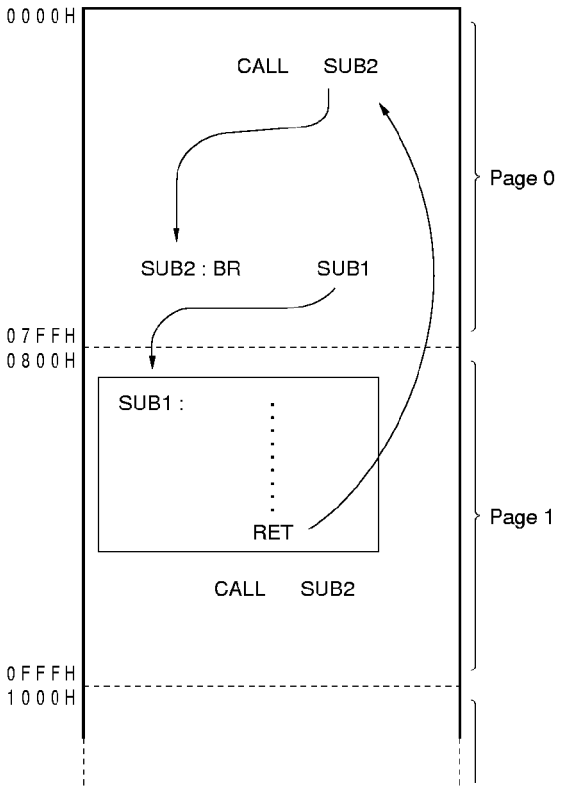
When using the CALL instruction, the address to be called, i.e., the first address in a subroutine, must be placed in page 0 (0000H-07FFH). To call a subroutine, whose first address is outside page 0, use the CALL @AR instruction.

When first address for subroutine is in page 0



If the first address for a subroutine is in page 0 as shown in the figure above, the end address for the subroutine (RET or RETSK instruction) may be outside page 0.

As long as the first address for the subroutine is in page 0, the CALL instruction can be used without applying the page concept. However, if the first page in the subroutine cannot be placed in page 0, use the BR instruction in page 0, and branch the execution to the subroutine through this BR instruction, as follows:


0000H
CALL SUB2
Page 0
SUB2 : BR SUB1
07FFH
0800H
SUB1 :
RET
Page 1
CALL SUB2
0FFFH
1000H

(2) CALL @AR — Call subroutine specified by address register

<1> OP code

00111	000	0101	0000

<2> Function

SP ← SP – 1,
ASR ← PC,
PC ← AR

Increments the value of the program counter (PC), saves it in the stack, and branches to the subroutine that starts from the address specified by the address register (AR).

<3> Example 1

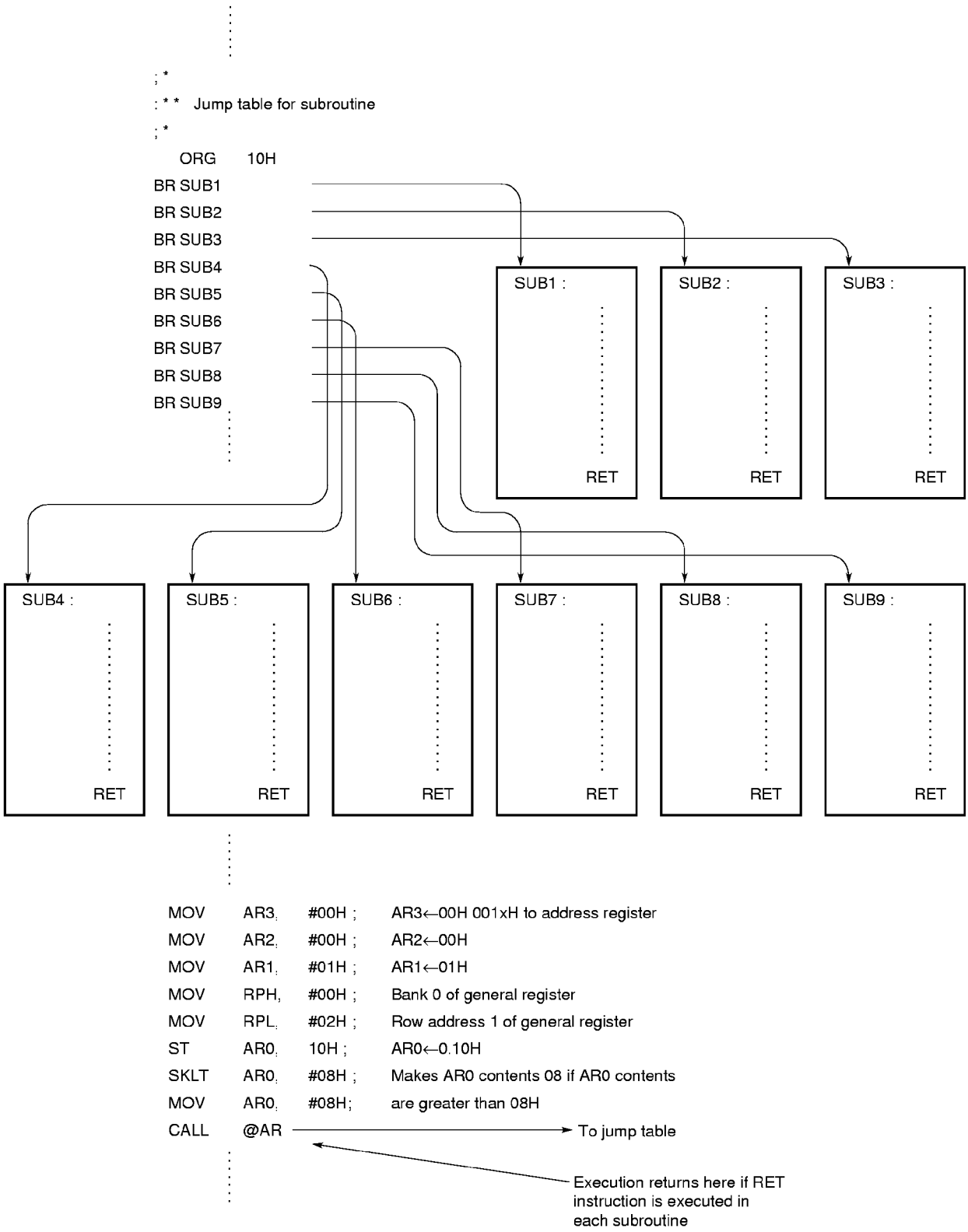
To set 0020H in address register AR (AR0-AR3) and call the subroutine at address 0020H by the CALL @AR instruction.

```
MOV     AR3,  #00H      ; AR3 ← 00H
MOV     AR2,  #00H      ; AR2 ← 00H
MOV     AR1,  #02H      ; AR1 ← 02H
MOV     AR0,  #00H      ; AR0 ← 00H
CALL    @AR             ; Calls subroutine at address 0020H
```

Example 2

To call the following subroutine by the data memory address 0.10H contents:

0.10H contents		Subroutine name
00H	→	SUB1
01H	→	SUB2
02H	→	SUB3
03H	→	SUB4
04H	→	SUB5
05H	→	SUB6
06H	→	SUB7
07H	→	SUB8
08H-0FH	→	SUB9



<4> Precaution

The number of bits of the address registers (AR0 through AR3) that can be used differs depending on the model of the device. For details, refer to the Data Sheet of each model.

★ **(3) SYSCAL entry** **Call system segment entry address**

<1> OP code

00111	$entry_H$	0000	$entry_L$

<2> Function

SP ← (SP) – 1, ASR ← PC + 1,
SGR ← SYSSEG, PAGE ← 0, PC (10-8) ← $entry_H$,
PC (7-4) ← 0, PC (3-0) ← $entry_L$

Increments the value of the program counter (PC), saves it in the stack, and branches to the subroutine specified by entry on page 0 of the system segment.
The subroutines that can be called by this instruction are 256 steps of the system segment entry address on page 0 of the system segment.

<3> Example 1

```
MAIN:
          :
          :
          SYSCAL 34H
          :
          :
CSEG      n
ORG       304H
          :
          :
          RET
```

(n: system segment)

Example 2

```
MAIN:
          :
          :
          SYSCAL.D.L. ((ENTRY SHR 4 AND 0070H) OR (ENTRY AND 000FH))
          :
          :
ENTRY:
          :
          :
          RET
```

<4> Precaution

As an operand of SYSCAL instruction, specify the symbol with data type, not label type. When the value operand exceeds 7 bits, the assembler (RA17K) generates an error.

In **Example 2** above, however, an error does not occur even though the "ENTRY" address does not exist in the system segment entry address. In this case, the address branched to by the SYSCAL instruction differs from the address expected by the user. Therefore, care must be taken when debugging.

(4) RET — Return to the main program from subroutine

<1> OP code

10	8 7	4 3	0
00111	000	1110	0000

<2> Function

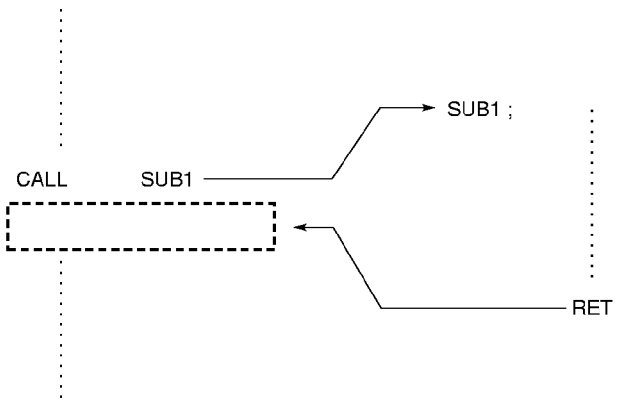
PC ← ASR,

SP ← SP + 1

Returns execution from a subroutine to the main program.

Restores the return address, saved by the CALL instruction to the stack, to the program counter.

<3> Example



(5) RETSK — Return to the main program then skip next instruction

<1> OP code

00111	001	1110	0000

<2> Function

PC ← ASR, SP ← SP + 1 and skip

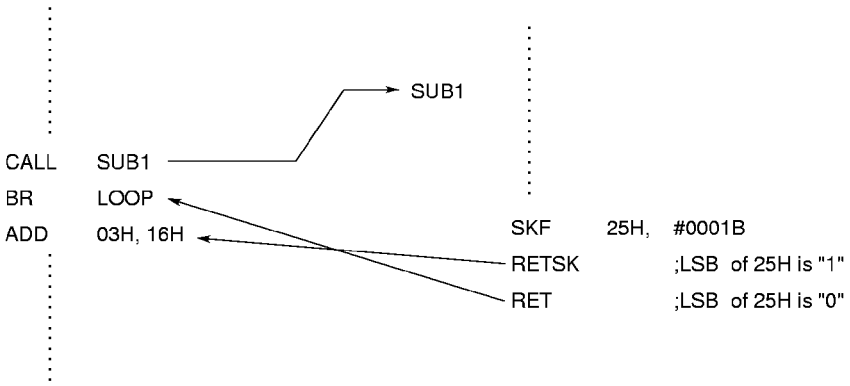
Returns execution from a subroutine to the main program.

★ Skips the instruction next to the CALL instruction (Executes as NOP instruction).

Restores the return address, saved by the CALL instruction to the stack, to the program counter PC, and then increments the program counter contents.

<3> Example

To execute the RET instruction and return the execution to the instruction next to the CALL instruction if the LSB (least significant bit) at address 25H of the data memory (RAM) is 0; if the LSB is 1, to execute the RETSK instruction to return the execution to the instruction after the next to the CALL instruction (ADD 03H, 16H in this example).



(6) RETI **Return to the main program from interrupt service routine**

<1> OP code

00111	100	1110	0000

<2> Function

PC ← ASR, INTR ← INTSK, SP ← SP + 1

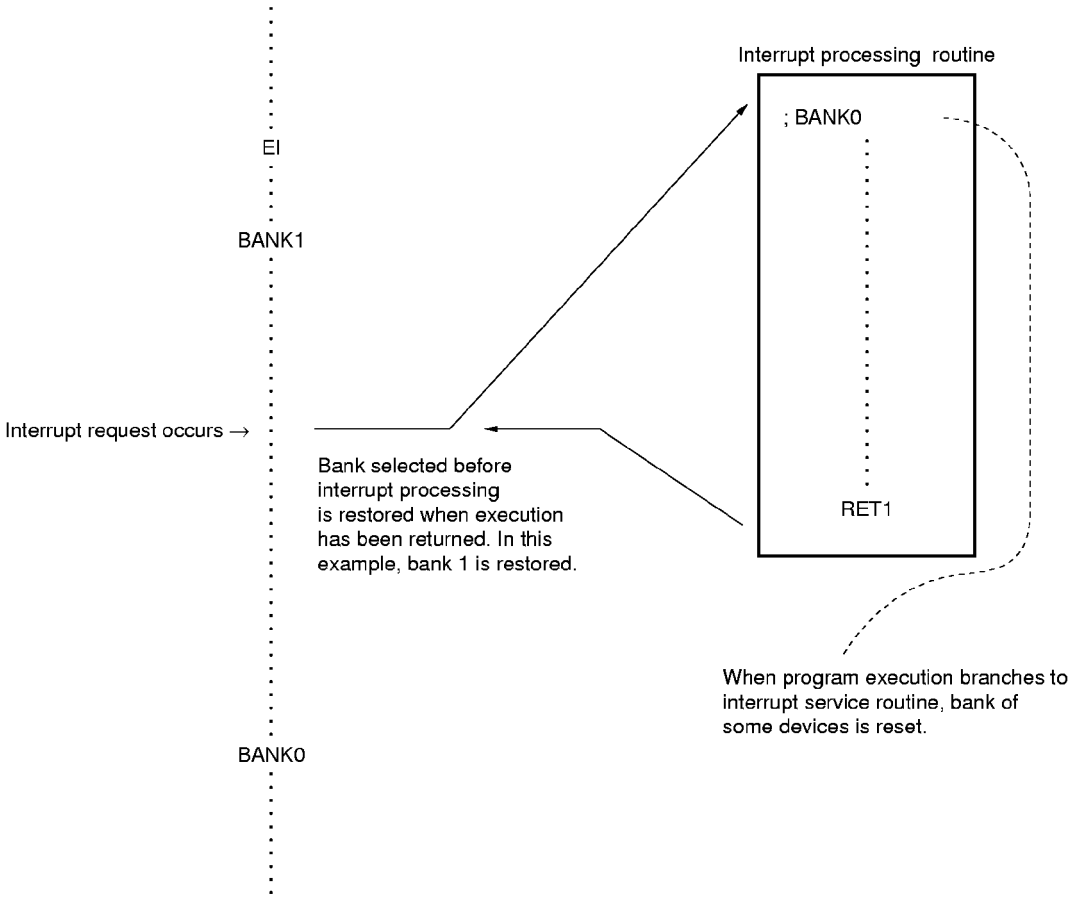
Returns execution from an interrupt processing program to the main program.

Restores to the program counter the return address which was saved in the stack by a vectored interrupt.

A part of the system registers is also restored to the states before the occurrence of the vectored interrupt.

<3> Example

If it is necessary to save the current bank address, because a vectored interrupt occurs, when the data memory is in bank 1 and data memory bank 0 is used for the interrupt processing:



<4> Precaution 1

The contents of the system register are automatically saved by an interrupt (which can be restored by the RETI instruction) are the PSWORD.

Precaution 2

If the RETI instruction is used in the place of the RET instruction to return from an ordinary subroutine, the bank contents (which were saved when the interrupt has occurred) may be replaced with the contents of the interrupt stack. Consequently, probably the bank contents are unknown to the user. To avoid this, be sure to use the RET (or RETSK) instruction to return from a subroutine.

15.5.10 Interrupt instructions

(1) EI **Enable Interrupt**

<1> OP code

00111	000	1111	0000

<2> Function

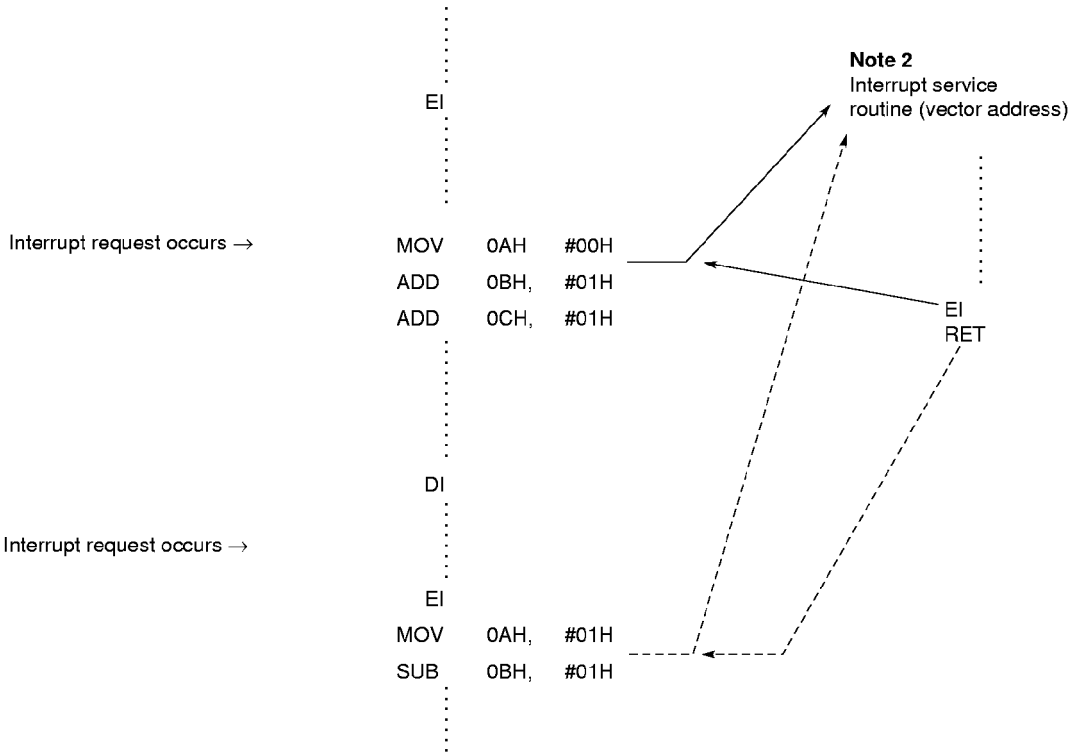
INTEF ← 1

Enables the vectored interrupt.

The interrupt is enabled after the instruction next to the EI instruction has been executed.

<3> Example 1

As shown in the following example, the interrupt request is accepted after the next instruction (except the instruction that manipulates the program counter) has been executed, and then the execution flow shifts to a vector address[Note1].

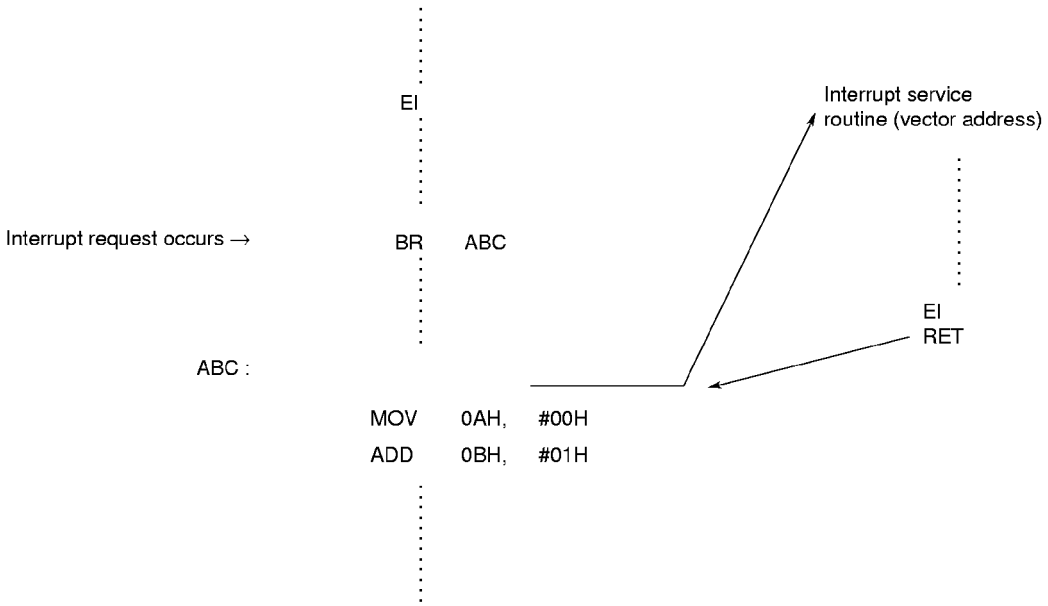


Notes 1. The vector address differs depending on the interrupt accepted. For details, refer to the Data Sheet of the model used.

2. The interrupt accepted (an interrupt request occurs after the execution of the EI instruction and the execution flow shifts to an interrupt service routine) is the interrupt whose interrupt enable flag (IPxxx) is set. The flow of the program is not changed (i.e., the interrupt is not accepted) even if an interrupt request occurs after the EI instruction has been executed with the interrupt enable flag of each interrupt not set. However, the interrupt request flag (IRQxxx) is set. The interrupt is therefore accepted at the point where the interrupt enable flag is set. For details, refer to the Data Sheet of the model used.

Example 2

An example of an interrupt that is caused by an interrupt request that has been accepted while an instruction that manipulates the program counter is executed as follows.



(2) DI — Disable interrupt

<1> OP code

00111	001	1111	0000

<2> Function

★ INTEF ← 0

Disables the vectored interrupt.

<3> Example

Refer to **Example 1** in (1) EI.

15.5.11 Other instructions

(1) STOP s **Stop CPU and release by condition s**

<1> OP code

		3	0
00111	010	1111	s

<2> Function

Stops the system clock and sets the device in the STOP mode.
By setting the device in the STOP mode, the current consumption of the device can be minimized.
The condition, under which the STOP mode is released, is specified by the operand (s).
The STOP release condition (s) differs depending on each model. Refer to the Data Sheet of the model used.

(2) HALT h **Halt CPU and release by condition h**

<1> OP code

		3	0
00111	011	1111	h

<2> Function

Sets the device in the HALT mode.
By setting the device in the HALT mode, the current consumption of the device can be reduced.
The condition, under which the HALT mode is released, is specified by the operand (h).
The HALT release condition (h) differs depending on each model. Refer to the Data Sheet of the model used.

(3) NOP **No operation**

<1> OP code

00111	100	1111	0000

<2> Function

Executes nothing but consumes one machine cycle.

APPENDIX A DEVELOPMENT TOOLS

The following tools are available for program development for the μPD170×× series.

★ A.1 Hardware

Name	Description
In-circuit emulator (IE-17K, IE-17K-ET[Note 1], EMU-17K[Note 2])	IE-17K, IE-17K-ET, and EMU-17K are in-circuit emulators common to the 17K series. IE-17K and IE-17K-ET are used by connecting to the host machine (PC-9800 series or IBM PC/AT™) via RS232-C. By using in combination with a system evaluation board (SE board) specific to each model, they can operate as an emulator matched to the model used. Use of *SIMPLEHOST*, a man-machine interface software, realizes a more powerful debugging environment. Moreover, the EMU-17K is provided with a function to check the contents of data memory in real time.
SE board	Used for system evaluation by itself, and used for debugging in combination with in-circuit emulator.
Emulation probe	Used in combination with a conversion socket to connect the SE board and the target system. It varies depending on the model used.
Conversion socket	Used to connect the emulation probe and the target system. It varies depending on the device package.
PROM programmer (AF-9703[Notes 3, 4] AF-9704[Notes 3, 4] AF-9705[Note 4] AF-9706[Note 4])	By connecting with the program adapter, this can be used to program PROM products.
Program adapter	This is used in combination with a PROM programmer. It varies depending on the model used.

Notes 1. Standard price version: External power supply type

2. This is a product of Naito Densei Machida Mfg. Co., Ltd. For details, contact Naito Densei Machida Mfg. Co., Ltd. (TEL: 044-822-3813)

3. Production stopped.

4. This is a product of Ando Electric Co., Ltd. For details, contact Ando Electric Co., Ltd. (TEL: 03-3733-1163)

★ A.2 Software

Name	Description	Host Machine	OS	Supply Media	Part Number
17K assembler (RA17K)	RA17K is an assembler common to 17K series products. Used in combination with device files for program development.	PC-9800 series	Japanese Windows™	3.5" 2HD	μSAA13RA17K
		IBM PC/AT and compatible	Japanese Windows	3.5" 2HC	μSAB13RA17K
			English Windows		μSBB13RA17K
Device file (AS170××)	Used in combination with RA17K. Differs depending on the model used.	PC-9800 series	Japanese Windows	3.5" 2HD	μSAA13AS170××
		IBM PC/AT and compatible	Japanese Windows	3.5" 2HC	μSAB13AS170××
			English Windows		μSBB13AS170××
Support software (*SIMPLEHOST*)	Software to provide man-machine interface in Windows when developing programs using in-circuit emulator and personal computer.	PC-9800 series	Japanese Windows	3.5" 2HD	μSAA13ID17K
		IBM PC/AT and compatible	Japanese Windows	3.5" 2HC	μSAB13ID17K
			English Windows		μSBB13ID17K

Remark ××: Differs depending on the model used.

APPENDIX B HOW TO ORDER THE MASK ROM

After you have developed your program, place your order for a mask ROM as follows:

(1) Reservation for ordering mask ROM

Inform NEC in advance when you need the mask ROM; otherwise, the mask ROM may not be delivered in time to meet your needs.

(2) Creating ordering medium

The medium in which the mask ROM is ordered is a UV-EPROM.

First, create a hex file (with extension characters .PRO) for ordering the mask ROM by adding assemble option
★ /PROM of the assembler (RA17K).

Next, write the hex file for ordering the mask ROM in the UV-EPROM.

When ordering with a UV-EPROM, create three UV-EPROM, all having identical contents.

Caution You cannot order a mask ROM by creating a hex file with .ICE.

(3) Creating necessary documents

Fill out the following forms, when ordering for the mask ROM:

- Mask ROM ordering sheet
- Mask ROM ordering check sheet

(4) Ordering

Submit the medium created in (2) and documents created in (3) to NEC by the deadline date for ordering.

★ **Remark** For details, refer to **ROM Code Ordering Procedure (IEM-1366)**.

[MEMO]

APPENDIX C INSTRUCTION INDEX

★ C.1 Instruction Index (by function)

C.2 Instruction Index (by alphabetic order)

APPENDIX D REVISION HISTORY

A history of the revisions up to this edition is shown below. "Applied to:" indicates the chapters to which the revision was applied.

★

Edition	Contents	Applied to:
2nd	Assembler changed (AS17K → RA17K)	Throughout
	In-circuit emulator IE-17K-ET added	Throughout
	CHAPTER 14 ONE-TIME PROM MODEL in previous edition deleted	CHAPTER 14 ONE-TIME PROM MODEL in previous edition
	Related Documents in **INTRODUCTION** changed	INTRODUCTION
	μPD170×× Product Development and **List of Functions** in **CHAPTER 1 GENERAL** deleted	CHAPTER 1 GENERAL
	6.7.3 Notes on using general register pointer added	CHAPTER 6 SYSTEM REGISTER (SYSREG)
	Diagram of the relationship between program status word (PSWORD) and status flip-flop in **Figure 8-1. Configuration of ALU Block** added	CHAPTER 8 ARITHEMETIC LOGICAL UNIT (ALU)
	Operation and description of instructions added to **Table 8-1. ALU Processing Instructions**	
	The following descriptions added to **8.2.3 Status flip-flop functions**: (1) Z flag (2) CY flag (3) CMP flag (4) BCD flag	
	Table 8-2. Results for Binary 4-bit and BCD Operations changed	
	Table 8-3. Arithmetic Operation Instructions added	
	Table 8-4. Logical Operation Instructions added	
	Table 8-6. Bit Testing Instructions added	
	Table 8-7. Compare Instructions added	
	Example 3 added to **9.4.2 Symbol definition of register file and reserved words**	CHAPTER 9 REGISTER FILE (RF)
	Description added to **12.2.6 Interrupt enable flip-flop (INTE)**	CHAPTER 12 INTERRUPT FUNCTIONS
	Remarks added to: **13.4.3 Releasing halt status by key input** **13.4.4 Releasing halt status by timer carry (basic timer 0 carry)**	CHAPTER 13 STANDBY FUNCTIONS
	Remark and Caution added to **13.4.5 Releasing halt status by interrupt**	
	12.6 Current Dissipation in Halt and Clock Stop Modes in previous edition deleted	
	Description added to **14.4 Power-ON Reset**	CHAPTER 14 RESET FUNCTION
	15.5.9 (3) SYSCAL entry added	CHAPTER 15 INSTRUCTION SET
	A.1 Hardware and **A.2 Software** changed	APPENDIX A DEVELOPMENT TOOLS
	C.1 Instruction Index (by function) added	APPENDIX C INSTRUCTION INDEX
	APPENDIX D REVISION HISTORY added	APPENDIX D REVISION HISTORY

[MEMO]

NEC

Facsimile Message

From:

Name

Company

Tel. FAX

Address

Although NEC has taken all possible steps to ensure that the documentation supplied to our customers is complete, bug free and up-to-date, we readily accept that errors may occur. Despite all the care and precautions we've taken, you may encounter problems in the documentation. Please complete this form whenever you'd like to report errors or suggest improvements to us.

Thank you for your kind support.

North America
NEC Electronics Inc.
Corporate Communications Dept.
Fax: 1-800-729-9288
1-408-588-6130

Europe
NEC Electronics (Europe) GmbH
Technical Documentation Dept.
Fax: +49-211-6503-274

South America
NEC do Brasil S.A.
Fax: +55-11-6465-6829

Hong Kong, Philippines, Oceania
NEC Electronics Hong Kong Ltd.
Fax: +852-2886-9022/9044

Korea
NEC Electronics Hong Kong Ltd.
Seoul Branch
Fax: 02-528-4411

Taiwan
NEC Electronics Taiwan Ltd.
Fax: 02-719-5951

Asian Nations except Philippines
NEC Electronics Singapore Pte. Ltd.
Fax: +65-250-3583

Japan
NEC Semiconductor Technical Hotline
Fax: 044-548-7900

I would like to report the following error/make the following suggestion:

Document title: _____

Document number: _____ Page number: _____

If possible, please fax the referenced page or drawing.

Document Rating	Excellent	Good	Acceptable	Poor
Clarity	❑	❑	❑	❑
Technical Accuracy	❑	❑	❑	❑
Organization	❑	❑	❑	❑