

To our customers,

---

## Old Company Name in Catalogs and Other Documents

---

On April 1<sup>st</sup>, 2010, NEC Electronics Corporation merged with Renesas Technology Corporation, and Renesas Electronics Corporation took over all the business of both companies. Therefore, although the old company name remains in this document, it is a valid Renesas Electronics document. We appreciate your understanding.

Renesas Electronics website: <http://www.renesas.com>

April 1<sup>st</sup>, 2010  
Renesas Electronics Corporation

Issued by: Renesas Electronics Corporation (<http://www.renesas.com>)

Send any inquiries to <http://www.renesas.com/inquiry>.

## Notice

1. All information included in this document is current as of the date this document is issued. Such information, however, is subject to change without any prior notice. Before purchasing or using any Renesas Electronics products listed herein, please confirm the latest product information with a Renesas Electronics sales office. Also, please pay regular and careful attention to additional and different information to be disclosed by Renesas Electronics such as that disclosed through our website.
2. Renesas Electronics does not assume any liability for infringement of patents, copyrights, or other intellectual property rights of third parties by or arising from the use of Renesas Electronics products or technical information described in this document. No license, express, implied or otherwise, is granted hereby under any patents, copyrights or other intellectual property rights of Renesas Electronics or others.
3. You should not alter, modify, copy, or otherwise misappropriate any Renesas Electronics product, whether in whole or in part.
4. Descriptions of circuits, software and other related information in this document are provided only to illustrate the operation of semiconductor products and application examples. You are fully responsible for the incorporation of these circuits, software, and information in the design of your equipment. Renesas Electronics assumes no responsibility for any losses incurred by you or third parties arising from the use of these circuits, software, or information.
5. When exporting the products or technology described in this document, you should comply with the applicable export control laws and regulations and follow the procedures required by such laws and regulations. You should not use Renesas Electronics products or the technology described in this document for any purpose relating to military applications or use by the military, including but not limited to the development of weapons of mass destruction. Renesas Electronics products and technology may not be used for or incorporated into any products or systems whose manufacture, use, or sale is prohibited under any applicable domestic or foreign laws or regulations.
6. Renesas Electronics has used reasonable care in preparing the information included in this document, but Renesas Electronics does not warrant that such information is error free. Renesas Electronics assumes no liability whatsoever for any damages incurred by you resulting from errors in or omissions from the information included herein.
7. Renesas Electronics products are classified according to the following three quality grades: "Standard", "High Quality", and "Specific". The recommended applications for each Renesas Electronics product depends on the product's quality grade, as indicated below. You must check the quality grade of each Renesas Electronics product before using it in a particular application. You may not use any Renesas Electronics product for any application categorized as "Specific" without the prior written consent of Renesas Electronics. Further, you may not use any Renesas Electronics product for any application for which it is not intended without the prior written consent of Renesas Electronics. Renesas Electronics shall not be in any way liable for any damages or losses incurred by you or third parties arising from the use of any Renesas Electronics product for an application categorized as "Specific" or for which the product is not intended where you have failed to obtain the prior written consent of Renesas Electronics. The quality grade of each Renesas Electronics product is "Standard" unless otherwise expressly specified in a Renesas Electronics data sheets or data books, etc.

"Standard": Computers; office equipment; communications equipment; test and measurement equipment; audio and visual equipment; home electronic appliances; machine tools; personal electronic equipment; and industrial robots.

"High Quality": Transportation equipment (automobiles, trains, ships, etc.); traffic control systems; anti-disaster systems; anti-crime systems; safety equipment; and medical equipment not specifically designed for life support.

"Specific": Aircraft; aerospace equipment; submersible repeaters; nuclear reactor control systems; medical equipment or systems for life support (e.g. artificial life support devices or systems), surgical implantations, or healthcare intervention (e.g. excision, etc.), and any other applications or purposes that pose a direct threat to human life.

8. You should use the Renesas Electronics products described in this document within the range specified by Renesas Electronics, especially with respect to the maximum rating, operating supply voltage range, movement power voltage range, heat radiation characteristics, installation and other product characteristics. Renesas Electronics shall have no liability for malfunctions or damages arising out of the use of Renesas Electronics products beyond such specified ranges.
9. Although Renesas Electronics endeavors to improve the quality and reliability of its products, semiconductor products have specific characteristics such as the occurrence of failure at a certain rate and malfunctions under certain use conditions. Further, Renesas Electronics products are not subject to radiation resistance design. Please be sure to implement safety measures to guard them against the possibility of physical injury, and injury or damage caused by fire in the event of the failure of a Renesas Electronics product, such as safety design for hardware and software including but not limited to redundancy, fire control and malfunction prevention, appropriate treatment for aging degradation or any other appropriate measures. Because the evaluation of microcomputer software alone is very difficult, please evaluate the safety of the final products or system manufactured by you.
10. Please contact a Renesas Electronics sales office for details as to environmental matters such as the environmental compatibility of each Renesas Electronics product. Please use Renesas Electronics products in compliance with all applicable laws and regulations that regulate the inclusion or use of controlled substances, including without limitation, the EU RoHS Directive. Renesas Electronics assumes no liability for damages or losses occurring as a result of your noncompliance with applicable laws and regulations.
11. This document may not be reproduced or duplicated, in any form, in whole or in part, without prior written consent of Renesas Electronics.
12. Please contact a Renesas Electronics sales office if you have any questions regarding the information contained in this document or Renesas Electronics products, or if you have any other inquiries.

(Note 1) "Renesas Electronics" as used in this document means Renesas Electronics Corporation and also includes its majority-owned subsidiaries.

(Note 2) "Renesas Electronics product(s)" means any product developed or manufactured by or for Renesas Electronics.

To all our customers

---

## **Regarding the change of names mentioned in the document, such as Hitachi Electric and Hitachi XX, to Renesas Technology Corp.**

---

The semiconductor operations of Mitsubishi Electric and Hitachi were transferred to Renesas Technology Corporation on April 1st 2003. These operations include microcomputer, logic, analog and discrete devices, and memory chips other than DRAMs (flash memory, SRAMs etc.) Accordingly, although Hitachi, Hitachi, Ltd., Hitachi Semiconductors, and other Hitachi brand names are mentioned in the document, these names have in fact all been changed to Renesas Technology Corp. Thank you for your understanding. Except for our corporate trademark, logo and corporate statement, no changes whatsoever have been made to the contents of the document, and these changes do not constitute any alteration to the contents of the document itself.

Renesas Technology Home Page: <http://www.renesas.com>

Renesas Technology Corp.  
Customer Support Dept.  
April 1, 2003

## Cautions

Keep safety first in your circuit designs!

1. Renesas Technology Corporation puts the maximum effort into making semiconductor products better and more reliable, but there is always the possibility that trouble may occur with them. Trouble with semiconductors may lead to personal injury, fire or property damage.  
Remember to give due consideration to safety when making your circuit designs, with appropriate measures such as (i) placement of substitutive, auxiliary circuits, (ii) use of nonflammable material or (iii) prevention against any malfunction or mishap.

Notes regarding these materials

1. These materials are intended as a reference to assist our customers in the selection of the Renesas Technology Corporation product best suited to the customer's application; they do not convey any license under any intellectual property rights, or any other rights, belonging to Renesas Technology Corporation or a third party.
2. Renesas Technology Corporation assumes no responsibility for any damage, or infringement of any third-party's rights, originating in the use of any product data, diagrams, charts, programs, algorithms, or circuit application examples contained in these materials.
3. All information contained in these materials, including product data, diagrams, charts, programs and algorithms represents information on products at the time of publication of these materials, and are subject to change by Renesas Technology Corporation without notice due to product improvements or other reasons. It is therefore recommended that customers contact Renesas Technology Corporation or an authorized Renesas Technology Corporation product distributor for the latest product information before purchasing a product listed herein.  
The information described here may contain technical inaccuracies or typographical errors.  
Renesas Technology Corporation assumes no responsibility for any damage, liability, or other loss rising from these inaccuracies or errors.  
Please also pay attention to information published by Renesas Technology Corporation by various means, including the Renesas Technology Corporation Semiconductor home page (<http://www.renesas.com>).
4. When using any or all of the information contained in these materials, including product data, diagrams, charts, programs, and algorithms, please be sure to evaluate all information as a total system before making a final decision on the applicability of the information and products. Renesas Technology Corporation assumes no responsibility for any damage, liability or other loss resulting from the information contained herein.
5. Renesas Technology Corporation semiconductors are not designed or manufactured for use in a device or system that is used under circumstances in which human life is potentially at stake. Please contact Renesas Technology Corporation or an authorized Renesas Technology Corporation product distributor when considering the use of a product contained herein for any specific purposes, such as apparatus or systems for transportation, vehicular, medical, aerospace, nuclear, or undersea repeater use.
6. The prior written approval of Renesas Technology Corporation is necessary to reprint or reproduce in whole or in part these materials.
7. If these products or technologies are subject to the Japanese export control restrictions, they must be exported under a license from the Japanese government and cannot be imported into a country other than the approved destination.  
Any diversion or reexport contrary to the export control laws and regulations of Japan and/or the country of destination is prohibited.
8. Please contact Renesas Technology Corporation for further details on these materials or the products contained therein.

# Renesas Microcomputer H8/300H Series

Application Notes for CPU

## Notice

When using this document, keep the following in mind:

1. This document may, wholly or partially, be subject to change without notice.
2. All rights are reserved: No one is permitted to reproduce or duplicate, in any form, the whole or part of this document without Hitachi's permission.
3. Hitachi will not be held responsible for any damage to the user that may result from accidents or any other reasons during operation of the user's unit according to this document.
4. Circuitry and other examples described herein are meant merely to indicate the characteristics and performance of Hitachi's semiconductor products. Hitachi assumes no responsibility for any intellectual property claims or other problems that may result from applications based on the examples described herein.
5. No license is granted by implication or otherwise under any patents or other rights of any third party or Hitachi, Ltd.
6. **MEDICAL APPLICATIONS:** Hitachi's products are not authorized for use in **MEDICAL APPLICATIONS** without the written consent of the appropriate officer of Hitachi's sales company. Such use includes, but is not limited to, use in life support systems. Buyers of Hitachi's products are requested to notify the relevant Hitachi sales offices when planning to use the products in **MEDICAL APPLICATIONS**.

# Contents

Section 1	CPU Architecture.....	1
1.1	Introduction .....	1
1.1.1	Features .....	1
1.1.2	Register Configuration.....	2
1.1.3	Data Configuration.....	4
1.1.4	Address Space .....	6
1.1.5	Addressing Mode .....	7
1.1.6	Instructions.....	16
Section 2	Instructions .....	17
2.1	Data Transfer Instructions.....	17
2.1.1	MOV.....	17
2.1.2	PUSH, POP .....	19
2.2	Arithmetic Operation Instructions.....	21
2.2.1	ADD, SUB .....	21
2.2.2	ADDX, SUBX .....	22
2.2.3	INC, DEC.....	23
2.2.4	ADDS, SUBS.....	24
2.2.5	DAA, DAS .....	25
2.2.6	MULXU, DIVXU, MULXS, DIVXS .....	25
2.2.7	CMP .....	27
2.2.8	NEG .....	28
2.2.9	EXTS, EXTU .....	28
2.3	Logic Operation Instructions.....	30
2.3.1	AND, OR, XOR, NOT .....	30
2.4	Shift Instructions .....	32
2.4.1	SHAL, SHAR, SHLL, SHLR, ROTL, ROTR, ROTXL, ROTXR.....	32
2.5	Bit Manipulation Instructions.....	34
2.5.1	BSET, BCLR, BNOT, BTST, BLD, BILD, BST, BIST, BAND, BIAN, BOR, BIOR, BXOR, BIXOR .....	34
2.6	Branch Instructions .....	36
2.6.1	Bcc .....	36
2.6.2	JMP.....	38
2.6.3	BSR, JSR.....	38
2.6.4	RTS.....	40
2.7	System Control Instructions .....	41
2.7.1	RTE .....	41
2.7.2	SLEEP .....	41
2.7.3	LDC, STC.....	42
2.7.4	ANDC, ORC, XORC.....	43

2.7.5	NOP .....	44
2.7.6	TRAPA .....	44
2.8	Block Transfer Instructions .....	45
2.8.1	EEPMOV .....	45
Section 3 Load Module Conversion Procedures .....		47
Section 4 Examples of Software Applications .....		49
4.1	Software Applications Examples .....	49
4.2	Using Software Examples .....	50
4.2.1	Program Listing Page Format (Format 4) .....	51
4.3	Block Transfer .....	52
4.3.1	Description of Functions .....	54
4.3.2	Cautions for Use .....	56
4.3.3	Description of Data Memory .....	56
4.3.4	Examples of Use .....	57
4.3.5	Principles of Operation .....	57
4.3.6	Program Listing .....	59
4.4	Block Transfer Using Block Transfer Instruction .....	60
4.4.1	Description of Functions .....	63
4.4.2	Cautions for Use .....	64
4.4.3	Description of Data Memory .....	64
4.4.4	Examples of Use .....	65
4.4.5	Principles of Operation .....	65
4.4.6	Program Listing .....	66
4.5	Branching Using a Table .....	67
4.5.1	Description of Functions .....	69
4.5.2	Cautions for Use .....	70
4.5.3	Description of Data Memory .....	70
4.5.4	Examples of Use .....	71
4.5.5	Principles of Operation .....	72
4.5.6	Program Listing .....	74
4.6	Counting the Number of Logical 1s in 8-Bit Data .....	75
4.6.1	Description of Functions .....	76
4.6.2	Cautions for Use .....	77
4.6.3	Description of Data Memory .....	77
4.6.4	Examples of Use .....	77
4.6.5	Principles of Operation .....	78
4.6.6	Program Listing .....	80
4.7	Find the First 1 in 32-Bit Data .....	81
4.7.1	Description of Functions .....	83
4.7.2	Cautions for Use .....	83
4.7.3	Description of Data Memory .....	83

4.7.4	Examples of Use.....	84
4.7.5	Principles of Operation.....	84
4.7.6	Program Listing.....	86
4.8	64-Bit Binary Addition.....	87
4.8.1	Description of Functions.....	90
4.8.2	Cautions for Use.....	90
4.8.3	Description of Data Memory.....	90
4.8.4	Examples of Use.....	91
4.8.5	Principles of Operation.....	92
4.8.6	Program Listing.....	93
4.9	64-Bit Binary Subtraction.....	94
4.9.1	Description of Functions.....	97
4.9.2	Cautions for Use.....	97
4.9.3	Description of Data Memory.....	97
4.9.4	Examples of Use.....	98
4.9.5	Principles of operation.....	99
4.9.6	Program Listing.....	100
4.10	Unsigned 32-Bit Binary Multiplication.....	101
4.10.1	Description of functions.....	104
4.10.2	Cautions for Use.....	104
4.10.3	Description of Data Memory.....	104
4.10.4	Examples of Use.....	105
4.10.5	Principles of Operation.....	106
4.10.6	Program Listing.....	109
4.11	Unsigned 32-Bit Binary Division.....	110
4.11.1	Description of Functions.....	113
4.11.2	Cautions for Use.....	113
4.11.3	Description of Data Memory.....	113
4.11.4	Examples of Use.....	114
4.11.5	Principles of Operation.....	115
4.11.6	Program Listing.....	117
4.12	Signed 16-Bit Binary Multiplication.....	118
4.12.1	Description of Functions.....	120
4.12.2	Cautions for Use.....	120
4.12.3	Description of Data Memory.....	120
4.12.4	Examples of Use.....	121
4.12.5	Principles of Operation.....	121
4.12.6	Program Listing.....	122
4.13	Signed 32-Bit Binary Multiplication.....	123
4.13.1	Description of Functions.....	126
4.13.2	Cautions for Use.....	126
4.13.3	Description of Data Memory.....	126
4.13.4	Examples of Use.....	127

4.13.5	Principles of Operation .....	128
4.13.6	Program Listing.....	132
4.14	Signed 32-Bit Binary Division (16-Bit Divisor) .....	133
4.14.1	Description of Functions .....	136
4.14.2	Cautions for Use.....	136
4.14.3	Description of Data Memory .....	136
4.14.4	Examples of Use.....	137
4.14.5	Principles of Operation.....	137
4.14.6	Program Listing.....	140
4.15	Signed 32-Bit Binary Division (32-Bit Divisor) .....	141
4.15.1	Description of Functions .....	144
4.15.2	Cautions for Use.....	144
4.15.3	Description of Data Memory .....	144
4.15.4	Examples of Use.....	145
4.15.5	Principles of Operation.....	146
4.15.6	Program Listing.....	147
4.16	8-Digit Decimal Addition.....	148
4.16.1	Description of Functions .....	151
4.16.2	Cautions for Use.....	151
4.16.3	Description of Data Memory .....	151
4.16.4	Examples of Use.....	152
4.16.5	Principles of Operation.....	152
4.16.6	Program Listing.....	154
4.17	8-Digit Decimal Subtraction .....	155
4.17.1	Description of Functions .....	158
4.17.2	Cautions for Use.....	158
4.17.3	Description of Data Memory .....	158
4.17.4	Examples of Use.....	159
4.17.5	Principles of Operation.....	159
4.17.6	Program Listing.....	161
4.18	Sum of Products .....	162
4.18.1	Description of Functions .....	165
4.18.2	Cautions for Use.....	165
4.18.3	Description of Data Memory .....	166
4.18.4	Examples of Use.....	166
4.18.5	Principles of Operation.....	166
4.18.6	Program Listing.....	168
4.19	Sorting.....	169
4.19.1	Description of Functions .....	171
4.19.2	Description of Data Memory .....	171
4.19.3	Examples of Use.....	172
4.19.4	Principles of Operation.....	173
4.19.5	Processing Method in Program .....	173

4.19.6 Program Listing.....	175
Appendix A Instruction Set.....	177
A.1 Number of Execution States.....	178
Appendix B Assembler.....	190
B.1 .CPU.....	190
B.2 .SECTION.....	191
B.3 .EQU.....	193
B.4 .ORG.....	194
B.5 .DATA.....	195
B.6 .RES.....	196
B.7 .END.....	197



# Section 1 CPU Architecture

## 1.1 Introduction

The H8/300H is a high-speed CPU with an internal 32-bit configuration and architecture that is upward-compatible with the H8/300. The H8/300H CPU has sixteen 16-bit general registers, can handle 16 Mbyte of linear address space, and is ideal for realtime control.

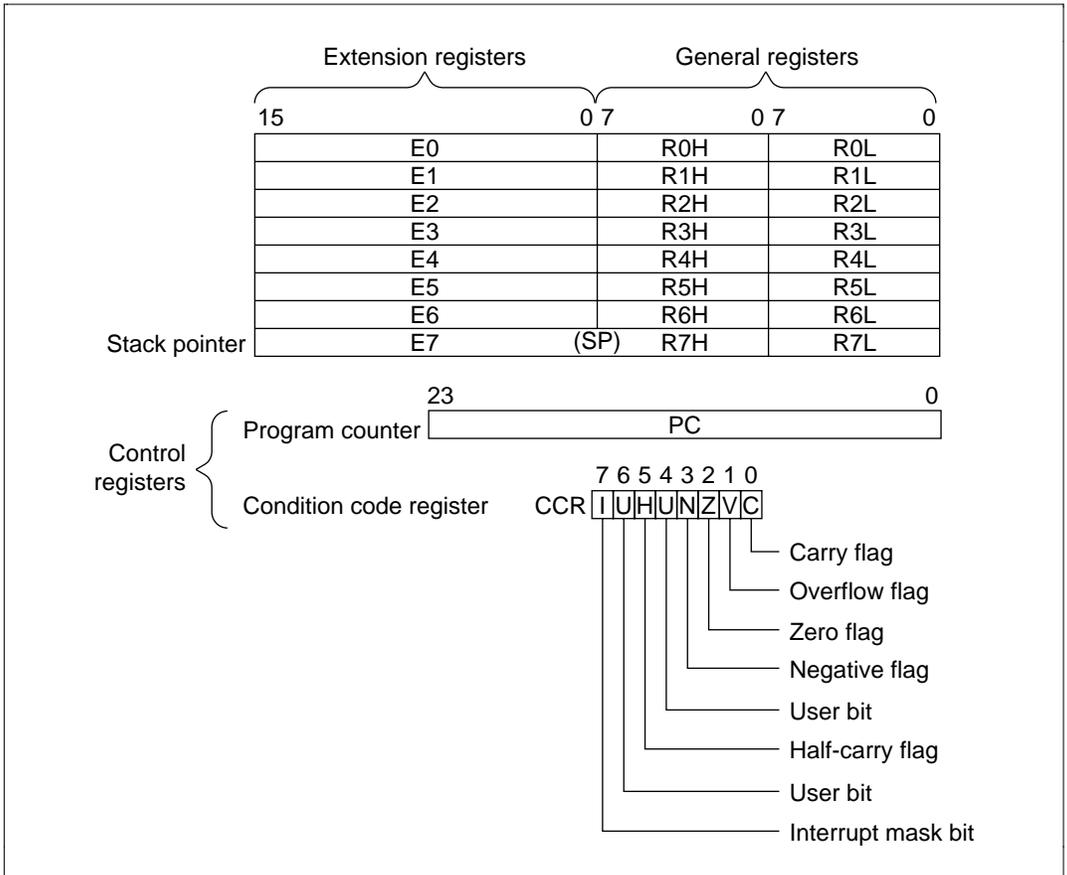
### 1.1.1 Features

The H8/300H has the following features:

- Upward compatibility with the H8/300: H8/300 object programs can be run without any changes
- Sixteen 16-bit general registers (can also be used as a sixteen 8-bit registers or eight 32-bit registers)
- Sixty two basic instructions: 8/16/32 bit operation instructions, multiplication/division instructions, powerful bit-manipulation instructions
- Eight types of addressing modes:
  - Register direct (Rn)
  - Register indirect (@ERn)
  - Register indirect with displacement (@(d:16, ERn)/@(d:24, ERn))
  - Post-increment/pre-decrement register indirect (@ERn+/@-ERn),
  - Absolute addressing (@aa:8/@aa:16/@aa:24)
  - Immediate (#xx:8/#xx:16/#xx:32)
  - Program counter relative (d:8, d:16)
  - Memory indirect (@@aa:8)
- 16 Mbyte address space
- High-speed operation:
  - Almost all common instructions executed in 2, 4, or 6 states
  - Maximum operating frequency: 16 MHz
  - Addition/subtraction between 8/16/32-bit registers: 0.17  $\mu$ s
  - Multiplication of two 8-bit registers: 1.2  $\mu$ s
  - Division of a 16-bit by an 8-bit register: 1.2  $\mu$ s
  - Multiplication of two 16-bit registers: 1.8  $\mu$ s
  - Division of a 32-bit by a 16-bit register: 1.8  $\mu$ s
- Two CPU operating modes: Normal mode/advanced mode
- Power-down mode: SLEEP instruction activates power-down mode

## 1.1.2 Register Configuration

Figure 1.1 shows the register configuration for the H8/300H. The H8/300H CPU is composed of sixteen 8-bit general register (R0H/R0L–R7H/R7L), eight 16-bit extended registers (E0–E7), one 24-bit program counter (PC) and one 8-bit condition code register (CCR), which are used as control registers.



**Figure 1.1 Composition of Registers**

**Extended Registers:** There are two ways of using extended registers:

- When working with 32-bit data and addresses (24 bits), 16-bit general registers (R0–R7) are combined as shown in table 1.1 and used as the upper 16 bits of 32-bit registers (ERn).
- They can also be used as independent 16-bit registers (En).

**Note:** The function of E7 as the upper 16 bits of the stack pointer (SP) is already allocated and is used implicitly in exception processing and subroutine calls.

## General Registers:

- General registers can be used as independent 8-bit registers (R0H/R0L–R7H/R7L).
- 8-bit registers can be combined with each other as shown in figure 1.2 for use as 16-bit registers (Rn).
- When working with 32-bit data and addresses (24 bits) and combining extended registers (E0–E7) as shown in figure 1.3, general registers can be used as the lower 16 bits of 32-bits registers (ERn).

Note: The function of R7 as the lower 16 bits of the stack pointer (SP) is already allocated and is used implicitly in exception processing and subroutine calls.

R0	R0H	R0L	E0
R1	R1H	R1L	E1
R2	R2H	R2L	E2
R3	R3H	R3L	E3
R4	R4H	R4L	E4
R5	R5H	R5L	E5
R6	R6H	R6L	E6
R7	R7H	R7L	E7

**Figure 1.2 16-Bit Registers (Rn)**

ER0	E0	R0
ER1	E1	R1
ER2	E2	R2
ER3	E3	R3
ER4	E4	R4
ER5	E5	R5
ER6	E6	R6
ER7	E7	R7

**Figure 1.3 32-Bit Registers (ERn)**

**Program counter (PC):** The PC is a 24-bit counter that indicates the address of the next instruction to be executed by the CPU.

**Condition Code Register (CCR):** The CCR is an 8-bit register that indicates the internal status of the CPU (table 1.1).

**Table 1.1 Condition Code Register**

Bit	Function	Description
7	Interrupt mask bit (I)	When this bit is 1, interrupts are masked. Note that a nonmaskable interrupt is received regardless of the status of the I bit. When exception processing begins, this bit is set to 1.
6	User bit (UI)	Can read/write using software (LDC, STC, ANDC, ORC, XORC instructions). Can also be used as an interrupt mask bit. For more information, see the hardware manual for the product in question.
5	Half carry flag (H)	When executing the ADD.B, ADDX.B, SUB.B, SUBX.B, CMP.B, or NEG.B instructions results in a borrow or carry at bit 3, or when executing an ADD.W, SUB.W, CMP.W, or NEG.W instruction results in a borrow or carry at bit 11, or when executing an ADD.L, SUB.L, CMP.L, or NEG.L instruction results in a borrow or carry at bit 27, the bit is set to 1; otherwise, it is set to 0.
4	User bit (U)	Can read/write using software (LDC, STC, ANDC, ORC, XORC instructions).
3	Negative flag (N)	The MSB of the data is considered a sign bit and its value is saved.
2	Zero flag (Z)	When the data is zero this bit is set to 1; when the data is nonzero, the bit is cleared to 0.
1	Overflow flag (V)	When execution of an arithmetic operation instruction creates an overflow, this bit is set to 1. In all other cases, it is set to 0.
0	Carry flag (C)	When execution of an operation creates a carry, this bit is set to 1; otherwise, it is set to 0. There are three types of carries: <ol style="list-style-type: none"><li>1. Carries caused by addition</li><li>2. Borrows caused by subtraction</li><li>3. Carries caused by shift/rotates</li></ol> The carry flag has a bit accumulator function that can be used by bit manipulation instructions.

### 1.1.3 Data Configuration

The H8/300H can work with 1-bit, 4-bit BCD, 8-bit (byte), 16-bit (word), and 32-bit (longword) data. 1-bit data is handled with bit manipulation instructions and accessed as the nth bit ( $n = 0, 1,$

2, ..., 7) of the operand data (byte). In the DAA and DAS decimal adjust instructions, byte data is two columns of 4-bit BCD data.

**Data Configuration of Registers:** Table 1.2 shows the configuration of data in the registers.

**Table 1.2 Register Data Configuration**

Data Type	Register No.	Data Image
1 bit	RnH	
	RnL	
4-bit BCD	RnH	
	RnL	
Byte	RnH	
	RnL	
Word	Rn	
	En	
Long word	ERn	

**Legend**

ERn: General register (long word size)

RnH: Top of general register

RnL: Bottom of general register

MSB: Most significant bit

LSB: Least significant bit

**Data Configuration in Memory:** Table 1.3 shows the configuration of data in memory. The H8/300H CPU can access word and longword data in memory. The MOV.W and MOV.L instructions are limited to data that starts from even addresses. When accessing word or long word

data that starts from odd addressees, the LSB of the address is considered 0 and data is accessed starting from the address one before. In such cases, no address errors are produced. The same applies to instruction code.

**Table 1.3 Memory Data Configuration**

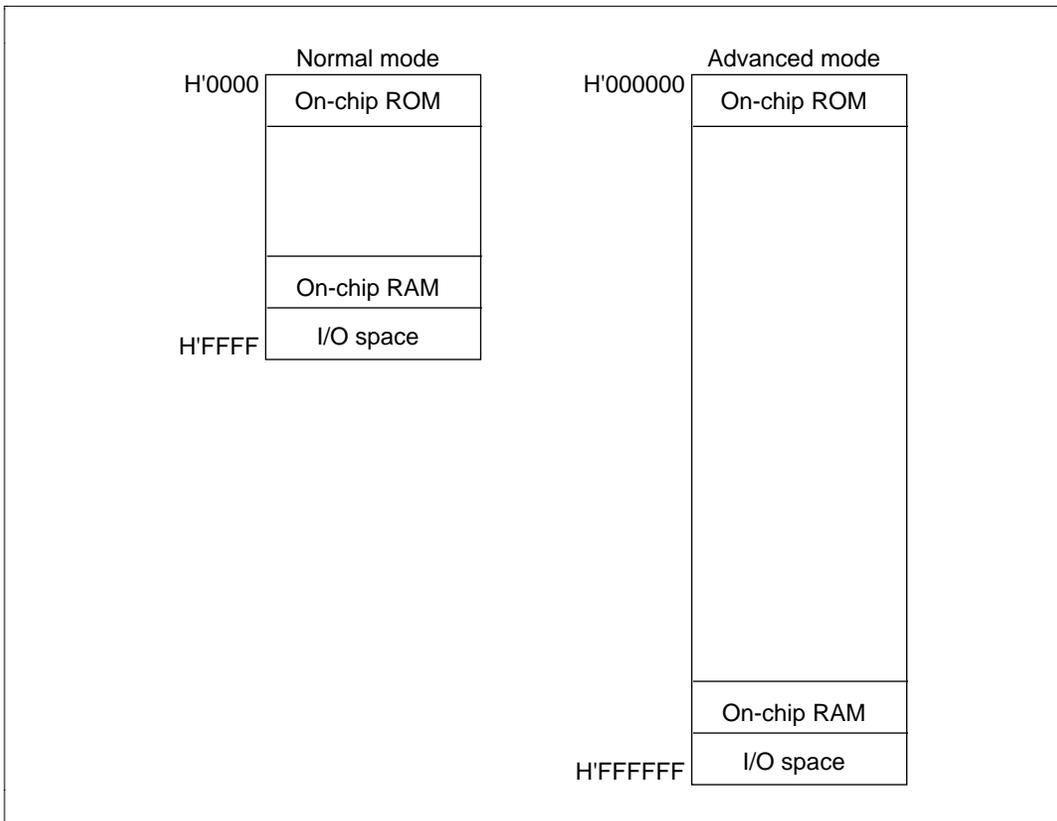
<b>Data Type</b>	<b>Memory Image</b>																		
1 bit	<table border="1"> <tr> <td style="text-align: right;">7</td> <td></td> <td></td> <td></td> <td></td> <td></td> <td></td> <td></td> <td style="text-align: left;">0</td> </tr> <tr> <td style="text-align: right;">nth address</td> <td>7</td> <td>6</td> <td>5</td> <td>4</td> <td>3</td> <td>2</td> <td>1</td> <td>0</td> </tr> </table>	7								0	nth address	7	6	5	4	3	2	1	0
7								0											
nth address	7	6	5	4	3	2	1	0											
Byte	<table border="1"> <tr> <td style="text-align: right;">nth address</td> <td>MSB</td> <td style="text-align: right;">LSB</td> </tr> </table>	nth address	MSB	LSB															
nth address	MSB	LSB																	
Word	<table border="1"> <tr> <td style="text-align: right;">Even address</td> <td>MSB</td> <td></td> </tr> <tr> <td style="text-align: right;">Odd address</td> <td></td> <td>LSB</td> </tr> </table>	Even address	MSB		Odd address		LSB												
Even address	MSB																		
Odd address		LSB																	
Long word	<table border="1"> <tr> <td style="text-align: right;">Even address</td> <td>MSB</td> <td></td> </tr> <tr> <td style="text-align: right;">Odd address</td> <td></td> <td></td> </tr> <tr> <td style="text-align: right;">Even address</td> <td></td> <td></td> </tr> <tr> <td style="text-align: right;">Odd address</td> <td></td> <td>LSB</td> </tr> </table>	Even address	MSB		Odd address			Even address			Odd address		LSB						
Even address	MSB																		
Odd address																			
Even address																			
Odd address		LSB																	

### 1.1.4 Address Space

There are two H8/300H operating modes: normal mode and advanced mode. Table 1.4 describes the operating modes and figure 1.4 shows the memory maps for these two modes. The mode pin of the LSI is used to select the mode. See the hardware manual of the product in question for more information.

**Table 1.4 Address Space for Normal and Advanced Operating Modes**

<b>CPU Operating Mode</b>	<b>Description</b>
Normal	Supports up to a maximum of 64 kbytes of address space. In this mode, the top 8 bits of the address are ignored and memory is accessed on 16-bit addresses.
Advanced	Supports up to a maximum of 16 Mbytes of address space. Can access continuous space by using the 24-bit PC and extended registers in combination.



**Figure 1.4 Memory Map**

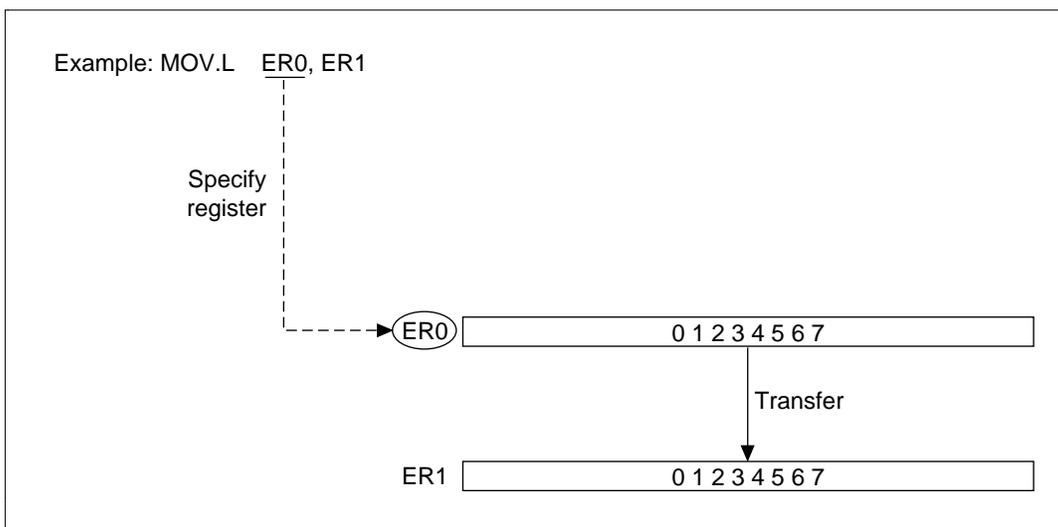
### 1.1.5 Addressing Mode

The H8/300H supports the eight addressing modes shown in table 1.5. The usable addressing modes vary for each instruction. Addressing modes are explained below using the various MOV commands as the primary example.

**Table 1.5 Addressing Modes**

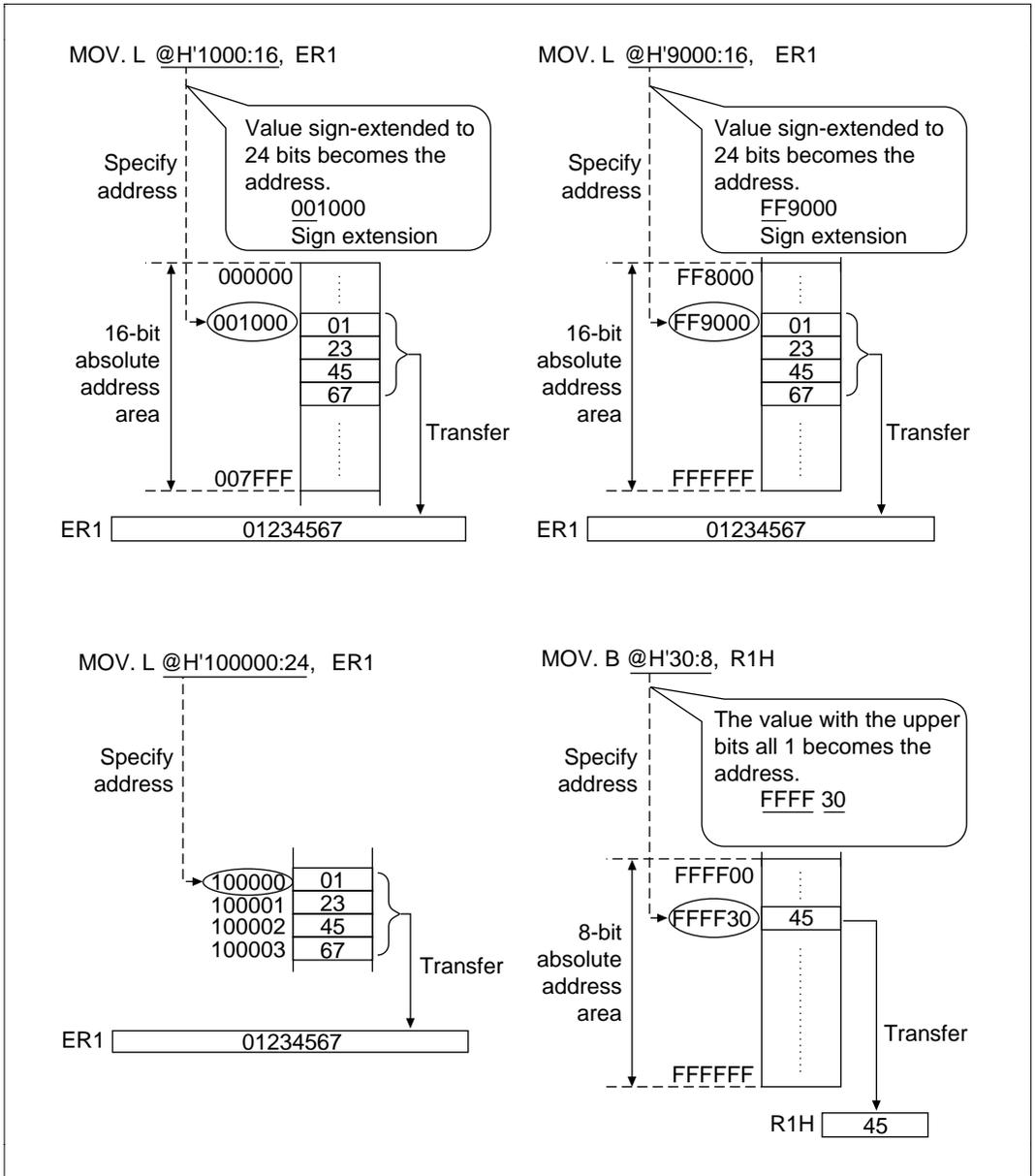
Addressing Mode	Use
Register direct	Specify registers
Absolute addressing	Specify address
Register indirect	
Post-increment register indirect	
Pre-decrement register indirect	
Register indirect with displacement	
Memory indirect	
Program counter relative	
Immediate	Specify constants

**Register Direct:** The register name (ER0–ER7, R0–R7, E0–E7, R0H/R0L–R7H/R7L) is written in the operand and the contents of that register become the subject of the instruction (figure 1.5).



**Figure 1.5 Register Direct**

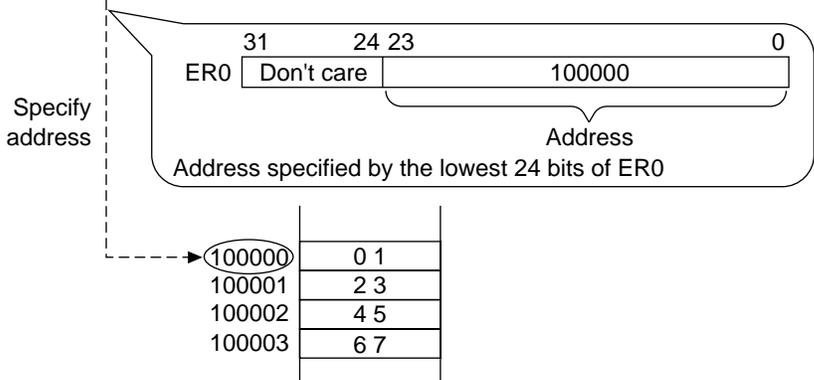
**Absolute Addressing:** Specifies the address directly. Addresses are usually specified as 24 bits in advanced mode and 16 bits in normal mode, but can be accessed by specifying only the lowest 16 bits or 8 bits when the absolute address area is 16 bits (H'000000–H'007FFF, H'FF8000–H'FFFFFF) or 8 bits (H'FFFF00–H'FFFFFF) (figure 1.6).



**Figure 1.6 Absolute Addressing**

**Register Indirect:** The address is specified by the lowest 24 bits of the 32 bit register (figure 1.7).

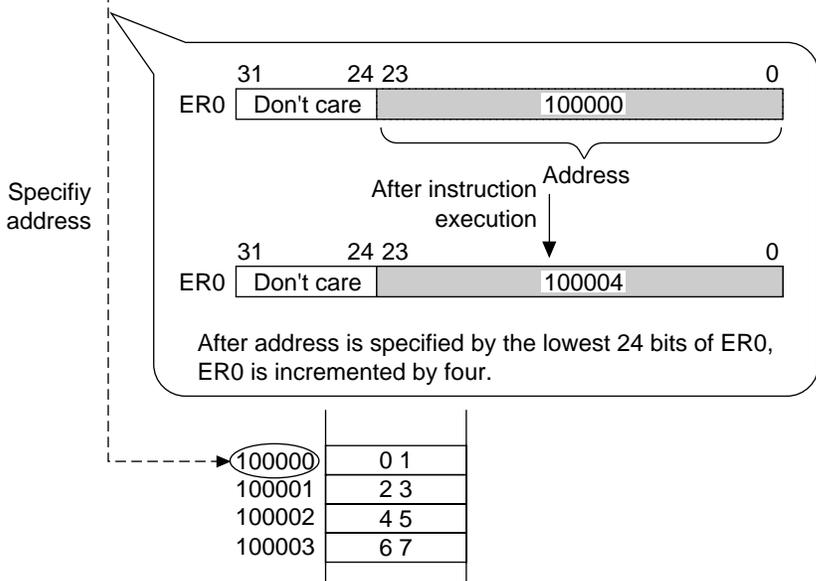
Example: MOV. L @ER0, ER1



**Figure 1.7 Register Indirect**

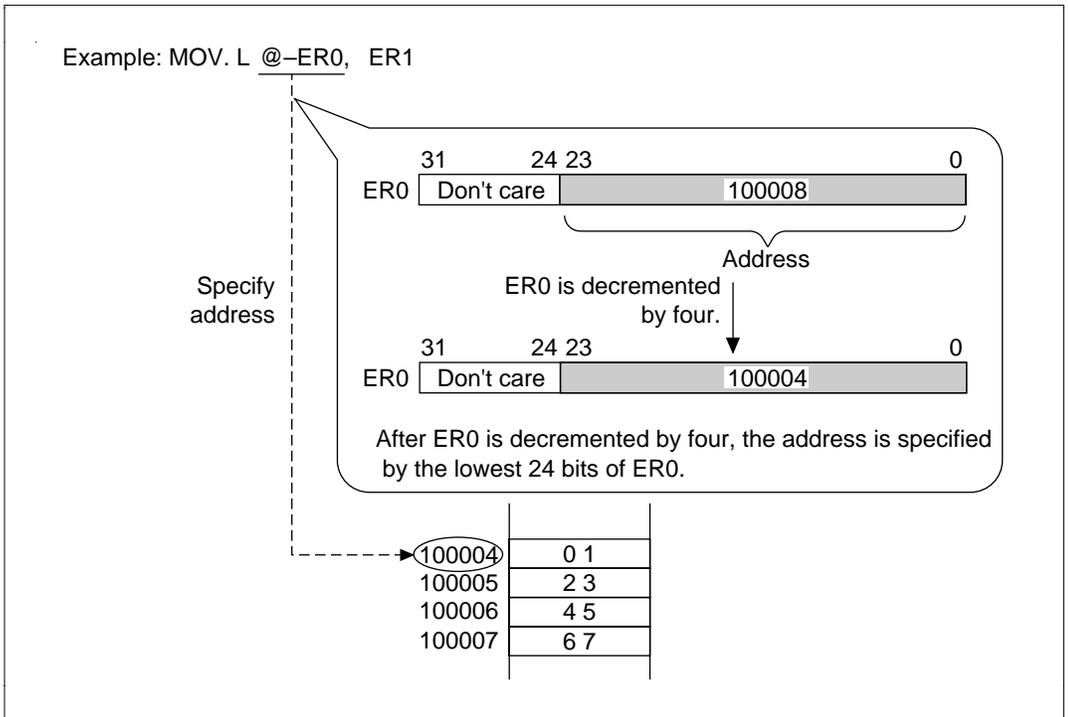
**Post-Increment Register Indirect:** The address is specified by the lowest 24 bits of the 32 bit register ERn. After instruction execution, the operand size value (B: 1, W: 2, L: 4) is added to the contents of the 32-bit register ERn (figure 1.8).

Example: MOV. L @ER0+, ER1



**Figure 1.8 Post-Increment Register Indirect**

**Pre-Decrement Register Indirect:** The address is specified by the lowest 24 bits of the 32 bit register ERn. Before instruction execution, the operand size value (B: 1, W: 2, L: 4) is subtracted from the contents of the 32-bit register ERn (figure 1.9).

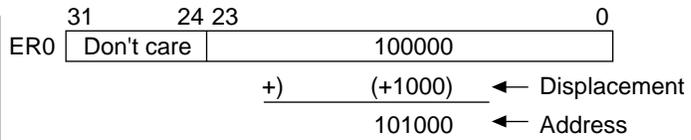


**Figure 1.9 Pre-Decrement Register Indirect**

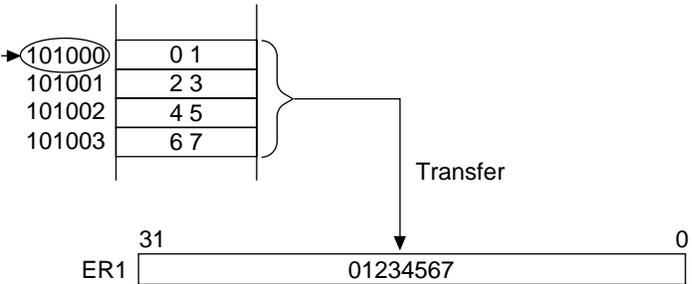
**Register Indirect with Displacement:** The address is specified by the lowest 24 bits of the 32 bit register ERn plus a signed displacement of 16 bits or 24 bits. The results of this addition are not saved in the 32-bit register ERn (figure 1.10).

Example: MOV. L @(H'1000:16. ER0), ER1

Specify address



The address is the lowest 24 bits of ER0 plus the signed 16-bit displacement (+1000).



Mnemonic:

@(displacement:16, ERn): signed displacement is 16 bits

@(displacement:24, ERn): signed displacement is 24 bits

**Figure 1.10 Register Indirect with Displacement**

Example: MOV. L @(H'F00000:24, ER0), ER1

Specify  
address

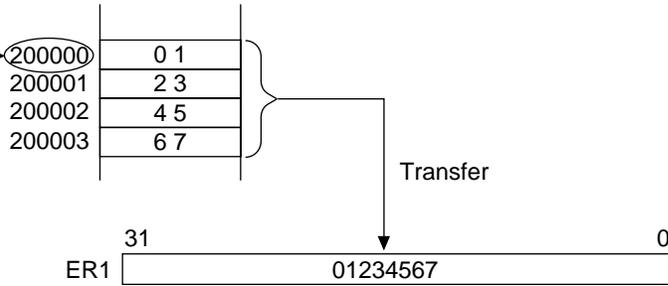
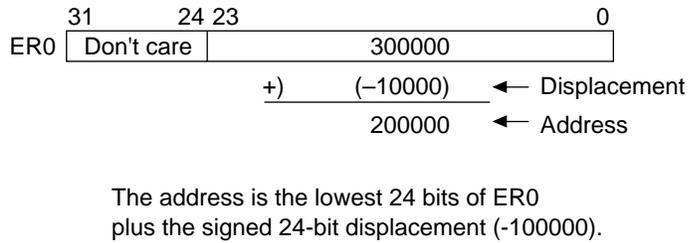
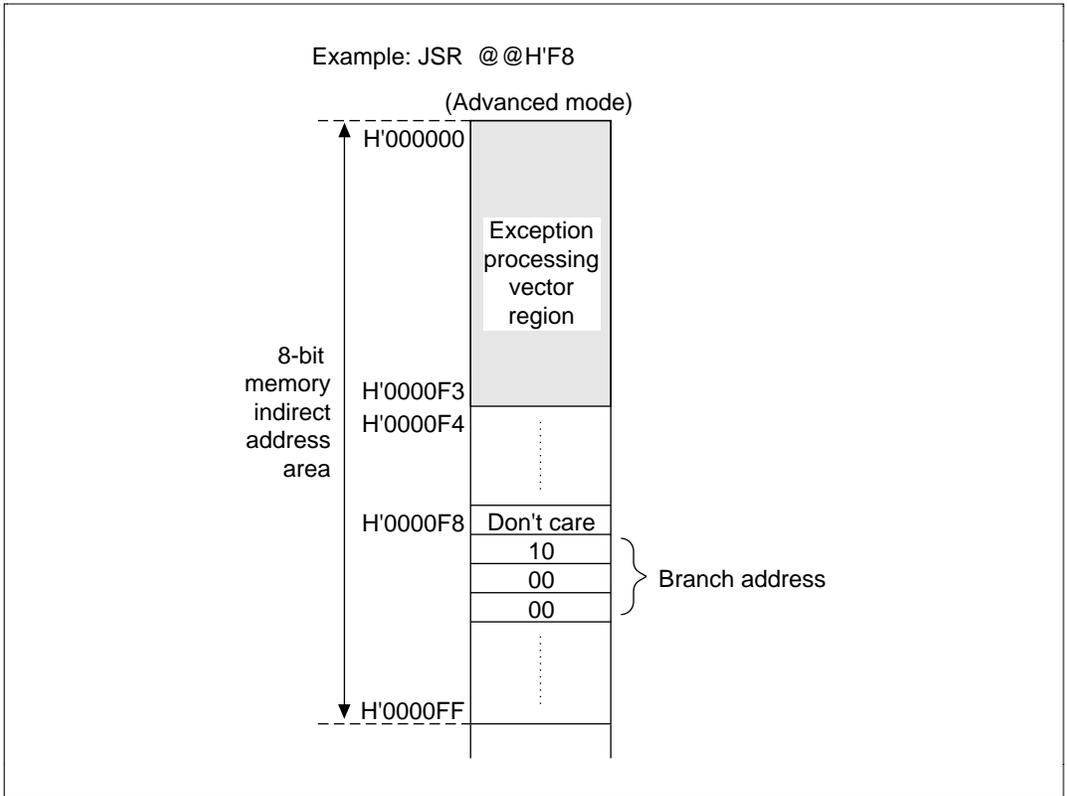


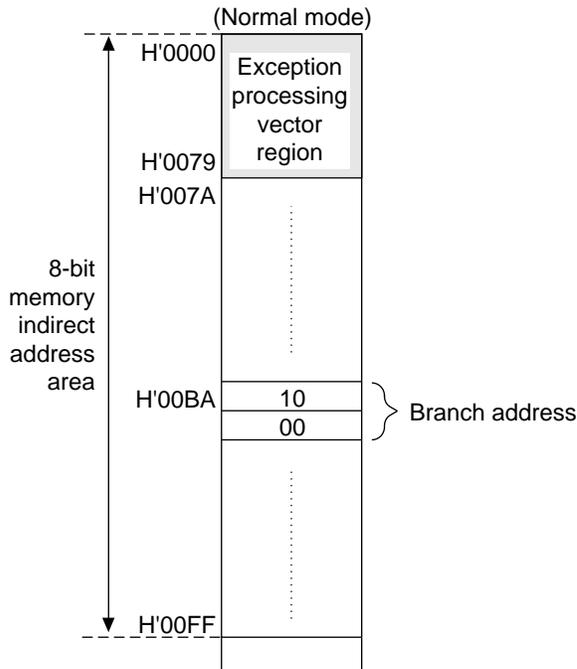
Figure 1.10 Register Indirect with Displacement (cont)

**Memory Indirect:** Uses branch address specification with the JSR and JMP instructions. The branch address is on the 8-bit memory indirect address area (advanced mode: H'000000–H'0000FF, normal mode: H'0000–H'00FF). To specify the branch address, specify the lower 8 bits of the address that stores the branch address. The address is stored in 2-byte units in normal mode and in 4-byte units for advanced mode (the first byte is ignored). Note that the top region of the 8-bit memory indirect address area is shared with the exception processing vector area. For more information, see the hardware manual for the LSI in question (figure 1.11).



**Figure 1.11 Memory Indirect**

Example: JSR @@H'BA (subroutine branch to address 1000)



**Figure 1.11 Memory Indirect (cont)**

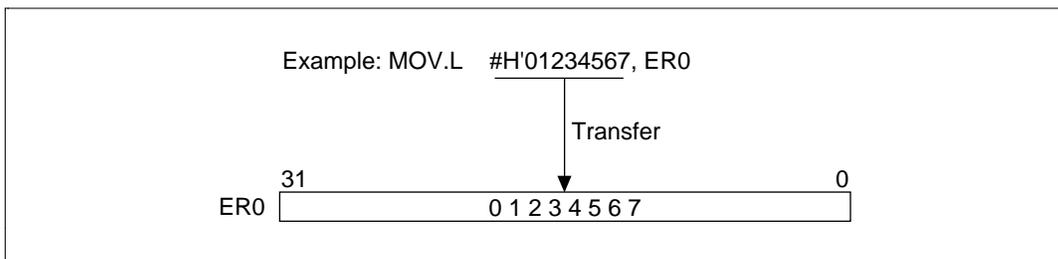
**Program Counter Relative:** Used to specify branch addresses using the Bcc or BSR instructions. It specifies the displacement of the branch address (signed 8-bit or signed 16-bit). Displacement is added to the contents of the PC and the address at the branch destination is generated. The PC contents become the start address of the next instruction, so the branchable area for the Bcc and BSR instructions are  $-126$  to  $+128$  bytes or  $-32766$  to  $+32678$  bytes. Normally, the branch destination symbol is specified rather than the displacement (figure 1.12).

```

          BSR      ABC
          ⋮
ABC: ADD. W      R0, E1
          ⋮
    
```

**Figure 1.12 Program Counter Relative**

**Immediate:** Directly specifies the data on the instruction (figure 1.13).



**Figure 1.13 Immediate Addressing**

### 1.1.6 Instructions

H8/300H CPU instructions have the following features:

- Instructions use a general register architecture
- A simplified and optimized 62-instruction basic set
- The common instruction length is 2 or 4 bytes
- High-speed executable multiplication and division instructions and powerful bit manipulation instructions
- 8 types of addressing modes

**Instruction Types:** There are a total of 62 H8/300H CPU instructions that are categorized according to function (table 1.6).

**Table 1.6 Instruction Categories**

Function	Instruction	Type
Data transfer instructions	MOV, PUSH, POP	3
Arithmetic operations instructions	ADD, SUB, ADDX, SUBX, INC, DEC, ADDS, SUBS, DAA, DAS, MULXU, DIVXU, MULXS, DIVXS, CMP, NEG, EXTS, EXTU	18
Logic operations instructions	AND, OR, XOR, NOT	4
Shift instructions	SHAL, SHAR, SHLL, SHLR, ROTL, ROTR, ROTXL, ROTXR	8
Bit manipulation instructions	BSET, BCLR, BNOT, BTST, BAND, BIAND, BOR, BIOR, BXOR, BIXOR, BLD, BILD, BST, BIST	14
Branching instructions	Bcc, JMP, BSR, JSR, RTS	5
System control instructions	RTE, SLEEP, LDC, STC, ANDC, ORC, XORC, NOP, TRAPA	9
Block transfer instructions	EEPMOV	1

# Section 2 Instructions

## 2.1 Data Transfer Instructions

### 2.1.1 MOV

MOV (Move): Transfers 8-bit, 16-bit or 32-bit data (figure 2.1).

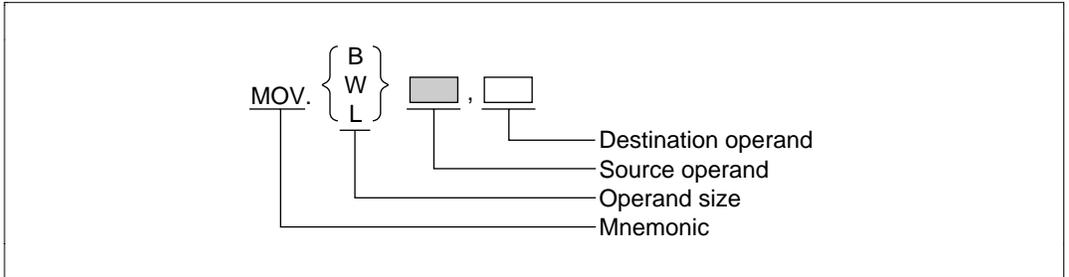
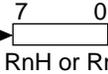
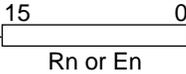
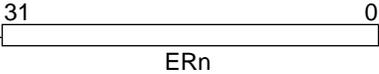


Figure 2.1 MOV

**Table 2.1 MOV**

Mnemonic	Op. Sz.	Source Operand	Dest. Op.	Description
MOV	B	RnH or RnL	RnH or RnL	
	W	Rn or En	Rn or En	
	L	ERn	ERn	
	B	@ERn @(d:16,ERn) @(d:24,ERn) @-ERn	RnH or RnL	
	W	@aa:8 @aa:16 @aa:24	Rn or En	
	L	ERn	ERn	
	B	RnH or RnL	@ERn @(d:16,ERn) @(d:24,ERn)	
	W	Rn or En	@ERn+ @aa:8 @aa:16 @aa:24	
	L	ERn	@ERn+ @aa:8 @aa:16 @aa:24	

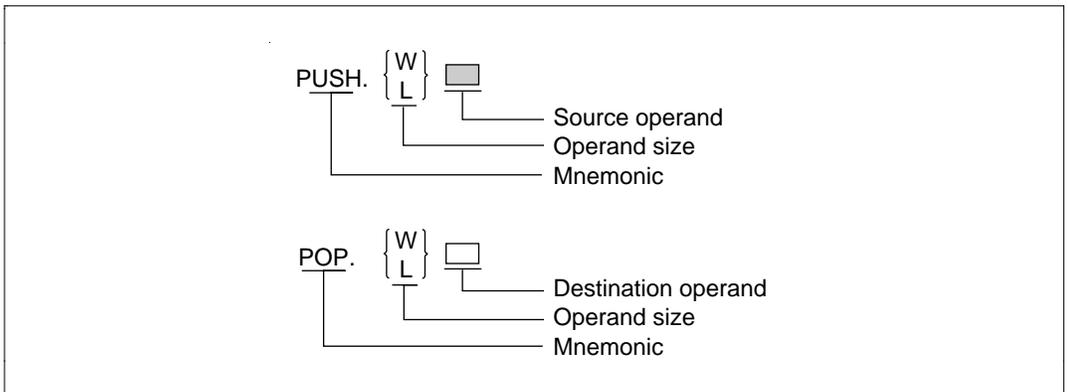
**Table 2.1 MOV (cont)**

Mnemonic	Op. Size	Source Operand	Dest. Op.	Description
MOV (cont)	B	#xx:8	RnH or RnL	#xx:8 →  RnH or RnL
	W	#xx:16	Rn or En	#xx:16 →  Rn or En
	L	#xx:32	ERn	#xx:32 →  ERn

### 2.1.2 PUSH, POP

PUSH (Push Data): Saves the contents of register to stack (figure 2.2).

POP (Pop Data): Recovers the contents of register from stack (figure 2.2).



**Figure 2.2 PUSH, POP**

**Table 2.2 PUSH, POP**

Mnemonic	Source Operand	Destination Operand (Source Operand)	Description
PUSH	W	(Rn, En)	<p>After 2 is subtracted from the stack pointer, the contents of 16-bit registers Rn and En are saved to the stack.</p> <p>The instruction is the same as MOV.W Rn, @-SP or MOV.W En, @-SP.</p>
L	(ERn)		<p>After 4 is subtracted from the stack pointer, the contents of 32-bit register ERn are saved to the stack.</p> <p>The instruction is the same as MOV.L ERn, @SP-.</p>
POP	W	Rn, En	<p>The contents of 16-bit registers Rn and En saved to the stack are recovered. After recovery 2 is added to the stack pointer.</p> <p>The instruction is the same as MOV.W @SP+, Rn or MOV.W @SO+, En.</p>
L	ERn		<p>The contents of 32-bit register ERn saved to the stack are recovered. After recovery 4 is added to the stack pointer.</p> <p>The instruction is the same as MOV.&gt; @SP+, ERn.</p>

## 2.2 Arithmetic Operation Instructions

### 2.2.1 ADD, SUB

ADD (ADD binary): Summand (8 bit) + addend (8 bit) = sum (8 bit), or  
Summand (16 bit) + addend (16 bit) = sum (16 bit), or  
Summand (32 bit) + addend (32 bit) = sum (32 bit)

SUB (Subtract binary): Subtrahend (8 bit) – minuend (8 bit) = difference (8 bit), or  
Subtrahend (16 bit) – minuend (16 bit) = difference (16 bit), or  
Subtrahend (32 bit) – minuend (32 bit) = difference (32 bit)

Figure 2.3 shows examples of ADD and SUB.

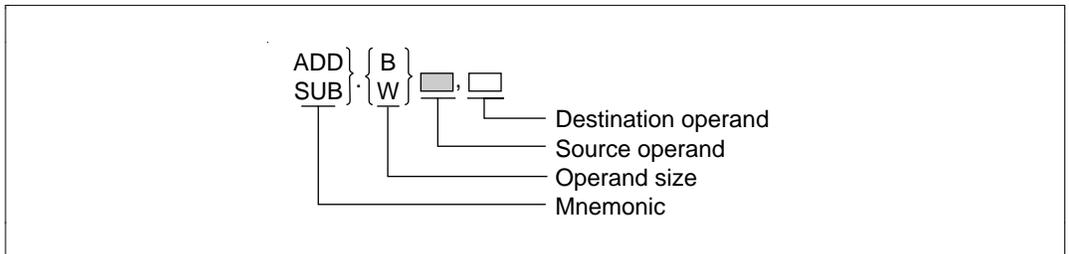


Figure 2.3 ADD, SUB

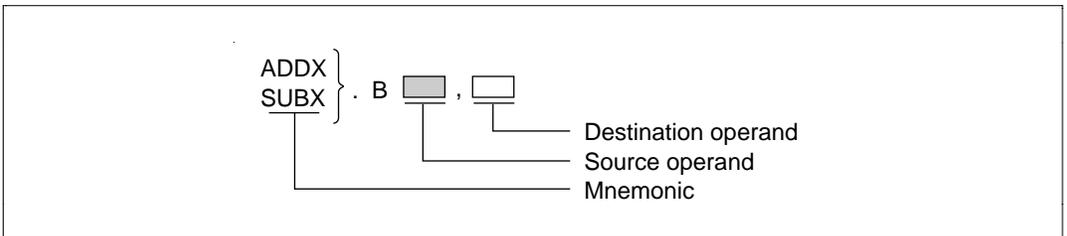
**Table 2.3 ADD, SUB**

Mnemonic	Operand Size	Destination Operand	Source Operand	Description
ADD SUB	B	RmH or RmL	#xx:8 or RnH or RnL	
W	Rm or Em	#xx:16 or Rn or En		
L	ERm	#xx:32 or ERn		

**2.2.2 ADDX, SUBX**

ADDX (ADD with Extend Carry): Adds with C flag (carry from bottom) included (figure 2.4).

SUBX (Subtract with Extend Carry): Subtracts with C flag (borrow from bottom) included (figure 2.4).



**Figure 2.4 ADDX, SUBX**

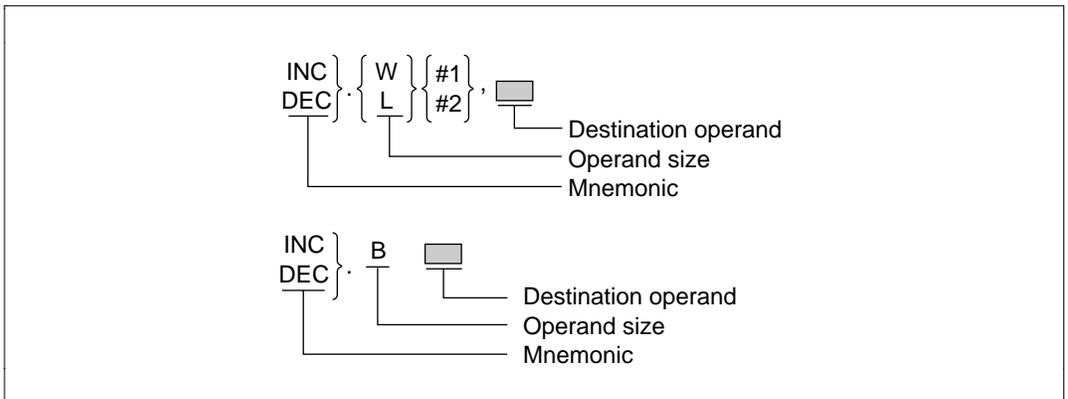
**Table 2.4 ADDX, SUBX**

Mnemonic	Operand Size	Source Operand	Destination Operand	Description
ADDX SUBX	B	#xx:8 or RnH or RnL	RmH or RmL	

### 2.2.3 INC, DEC

INC (Increment): Adds 1 to contents of 8-bit, registers RnH or RnL (figure 2.5). Adds 2 to the contents of 16-bit registers Rn or En and 32-bit register ERn.

DEC (DECrement): Subtracts 1 from contents of 8-bit, registers RnH or RnL (figure 2.5). Subtracts 2 from the contents of 16-bit registers Rn or En and 32-bit register ERn.



**Figure 2.5 INC, DEC**

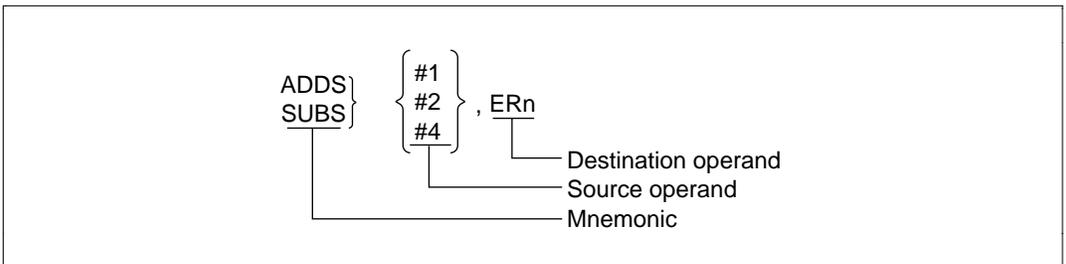
**Table 2.5 INC, DEC**

Mnemonic	Operand Size	Destination Operand	Description
INC DEC	B	RnH or RnL	 RnH or RnL
	W	Rn or En	 Rn or En
	L	ERn	 ERn

### 2.2.4 ADDS, SUBS

**ADDS (Add with Sign Extension):** Adds 1, 2 or, 4 to the contents of the 32-bit register ERn (figure 2.6).

**SUBS (Subtract with Sign Extension):** Subtracts 1, 2 or 4, from the contents of the 32-bit register ERn (figure 2.6).



**Figure 2.6 ADDS, SUBS**

**Table 2.6 ADDS, SUBS**

Mnemonic	Operand Size	Source Operand	Destination Operand	Description
ADDS SUBS	—	#1 or #2 or #4	ERn	 ERn

## 2.2.5 DAA, DAS

DAA (Decimal Adjust Add): Adjusts the sum from binary addition of 2 columns of 4-bit BCD data to 4-bit BCD data (figure 2.7).

DAS (Decimal Adjust Subtract): Adjusts the difference from binary subtraction of 2 columns of 4-bit BCD data to 4-bit BCD data (figure 2.7).

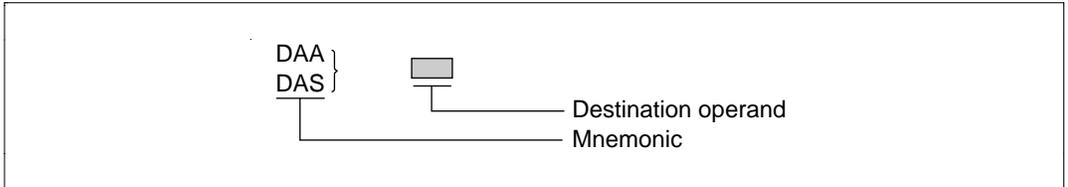


Figure 2.7 DAA, DAS

Table 2.7 DAA, DAS

Mnemonic	Destination Operand	Description
DAA	RnH or RnL	The results of binary addition or subtraction of 2 columns of 4-bit BCD data is adjusted to 2 columns of 4-bit BCD data.
DAS		

## 2.2.6 MULXU, DIVXU, MULXS, DIVXS

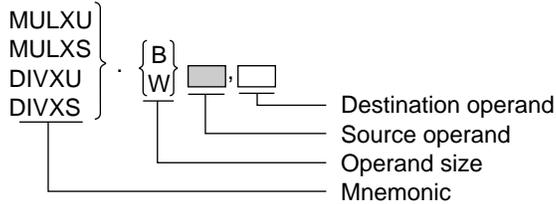
MULXU (Multiply Extended Unsigned): Multiplicand (8 bit) + multiplier (8 bit) = Product (16 bit), or Multiplicand (16 bit) + multiplier (16 bit) = Product (32 bit)

DIVXU (Divide Extended Unsigned): Dividend (16 bit) + divisor (8 bit) = Quotient (8 bit), Remainder (8 bit), or Dividend (32 bit) + divisor (16 bit) = Quotient (16 bit), Remainder (16 bit)

MULXS (Multiply Extended Signed): Multiplicand (8 bit) + multiplier (8 bit) = Product (16 bit), or Multiplicand (16 bit) + multiplier (16 bit) = Product (32 bit)

DIVXS (Divide Extended Signed): Dividend (16 bit) + divisor (8 bit) = Quotient (8 bit), Remainder (8 bit), or Dividend (32 bit) + divisor (16 bit) = Quotient (16 bit), Remainder (16 bit)

Figure 2.8 shows examples of MULXU, DIVXU, MULXS, and DIVXS.



**Figure 2.8** MULXU, DIVXU, MULXS, DIVXS

**Table 2.8** MULXU, DIVXU, MULXS, DIVXS

Mnemonic	Op. Size	Source Operand	Destination Operand	Description
MULXU MULXS	B	RnH or RnL	Rm or Em	<p>Product H'xxxx</p> $\begin{array}{ c c c } \hline 15 & 8 & 7 \\ \hline \hline & H'xx & \\ \hline \end{array} \times \begin{array}{ c c } \hline 7 & 0 \\ \hline \hline & H'xx \\ \hline \end{array} =$ <p>Rm or Em      RnH or RnL</p>
	W	Rn or En	ERm	<p>Product H'xxxxxxxx</p> $\begin{array}{ c c c c } \hline 31 & 16 & 15 & 0 \\ \hline \hline & & H'xxxx & \\ \hline \end{array} \times \begin{array}{ c c } \hline 15 & 0 \\ \hline \hline & H'xxxx \\ \hline \end{array} =$ <p>ERm      Rn or En</p>
DIVXU DIVXS	B	RnH or RnL	Rm or Em	<p>Remainder      Quotient H'xx              H'xx</p> $\begin{array}{ c c c } \hline 15 & 8 & 7 \\ \hline \hline & H'xxxx & \\ \hline \end{array} \div \begin{array}{ c c } \hline 7 & 0 \\ \hline \hline & H'xx \\ \hline \end{array} =$ <p>Rm or Em      RnH or RnL</p>
	W	Rn or En	ERm	<p>Quotient      Remainder H'xxxx              H'xxxx</p> $\begin{array}{ c c c c } \hline 31 & 16 & 15 & 0 \\ \hline \hline & & H'xxxxxxxx & \\ \hline \end{array} \div \begin{array}{ c c } \hline 15 & 0 \\ \hline \hline & H'xxxx \\ \hline \end{array} =$ <p>ERm      Rn or En</p>

## 2.2.7 CMP

CMP (Compare): Compares pairs of 8-bit, 16-bit, or 32-bit data (figure 2.9).

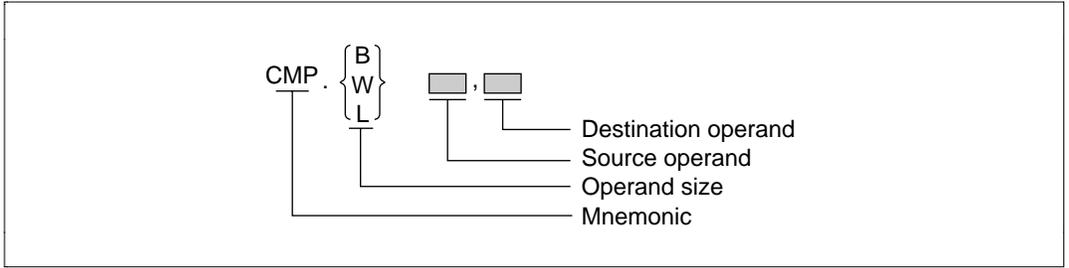


Figure 2.9 CMP

Table 2.9 CMP

Mnemonic	Op. Size	Source Op.	Dest. Op.	Description
CMP	B	#xx:8 RnH or RnL	RnH or RnL	$\left. \begin{array}{l} \text{H}'\text{xx} \\ \#xx:8 \\ \text{RnH or RnL} \end{array} \right\}$
	W	#xx:16 or Rn or En	Rn or En	$\left. \begin{array}{l} \text{H}'\text{xxxx} \\ \#xx:16 \\ \text{Rn or En} \end{array} \right\}$
	L	#xx:32 or ERn	ERn	$\left. \begin{array}{l} \text{H}'\text{xxxxxxxx} \\ \#xx:32 \\ \text{ERn} \end{array} \right\}$

## 2.2.8 NEG

NEG (Negate): Takes the two complement of 8-bit registers RnH and RnL, 16-bit registers Rn and En, and 32-bit register ERn. (figure 2.10)

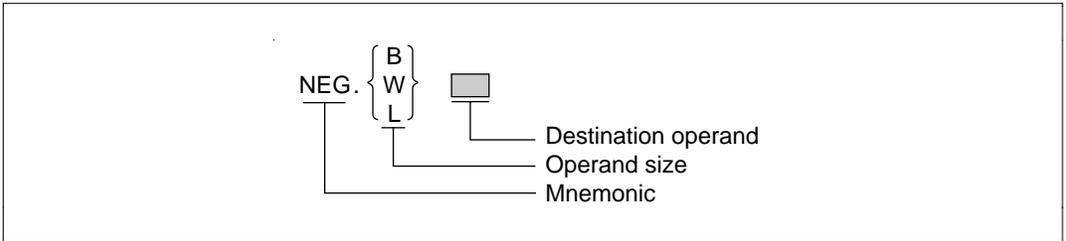


Figure 2.10 NEG

Table 2.10 NEG

Mnemonic	Op. Size	Destination Operand	Description
NEG	B	RnH or RnL	$0 - \left[ \begin{array}{c} 7 \quad 0 \\ \text{H'xx} \end{array} \right] = \text{RnH or RnL}$
	W	Rn or En	$0 - \left[ \begin{array}{c} 15 \quad 0 \\ \text{H'xxxx} \end{array} \right] = \text{Rn or En}$
	L	ERn	$0 - \left[ \begin{array}{c} 31 \quad 0 \\ \text{H'xxxxxxxx} \end{array} \right] = \text{ERn}$

## 2.2.9 EXTS, EXTU

EXTS (Extend as Signed): Sign-extends from 8 bit to 16 bit or from 16 bit to 32 bit (figure 2.11).

EXTU (Extend as Unsigned): Zero-extends from 8 bit to 16 bit or from 16 bit to 32 bit (figure 2.11).

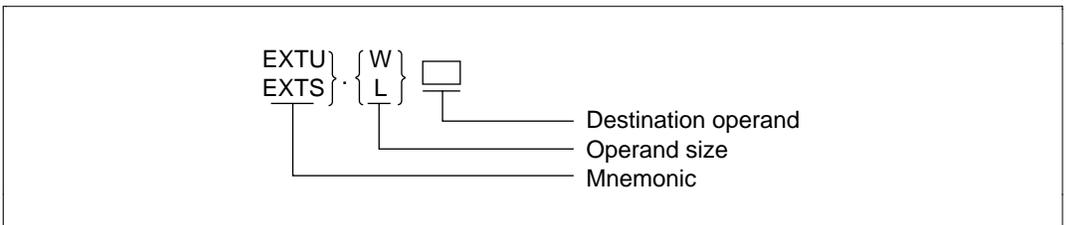


Figure 2.11 EXTS, EXTU

**Table 2.11 EXTS, EXTU**

Mnemonic	Op. Size	Destination Operand	Description
EXTU	W	Rn or En	Zero extended $\begin{array}{c} \overbrace{15 \quad 8 \quad 7} \quad 0 \\ \boxed{H' 00} \quad \boxed{H' \times \times} \\ \text{Rn or En} \end{array}$
	L	ERn	Zero extended $\begin{array}{c} \overbrace{31 \quad 16 \quad 15} \quad 0 \\ \boxed{H' 0000} \quad \boxed{H' \times \times \times \times} \\ \text{ERn} \end{array}$
EXTS	W	Rn or En	Sign extended When positive $\begin{array}{c} \overbrace{15 \quad 8 \quad 7} \quad 0 \\ \boxed{H' 00} \quad \boxed{H' \times \times} \\ \text{Rn or En} \end{array}$
			Sign extension When negative $\begin{array}{c} \overbrace{15 \quad 8 \quad 7} \quad 0 \\ \boxed{H' FF} \quad \boxed{H' \times \times} \\ \text{Rn or En} \end{array}$
	L	ERn	Sign extended When positive $\begin{array}{c} \overbrace{31 \quad 16 \quad 15} \quad 0 \\ \boxed{H' 0000} \quad \boxed{H' \times \times \times \times} \\ \text{ERn} \end{array}$
			Sign extension When negative $\begin{array}{c} \overbrace{31 \quad 16 \quad 15} \quad 0 \\ \boxed{H' FFFF} \quad \boxed{H' \times \times \times \times} \\ \text{ERn} \end{array}$

## 2.3 Logic Operation Instructions

### 2.3.1 AND, OR, XOR, NOT

AND (And logical): Takes the logical product of pairs of 8-bit, 16-bit, or 32-bit data (figure 2.12).

OR (Inclusive Or Logical): Takes the logical sum pairs of 8-bit, 16-bit, or 32-bit data (figure 2.12).

XOR (Exclusive Or Logical): Takes the exclusive logical sum of pairs of 8-bit, 16-bit, or 32-bit data (figure 2.12).

NOT (NOT = Logical Complement): Logically inverts pairs of 8-bit, 16-bit, or 32-bit data(figure 2.12).

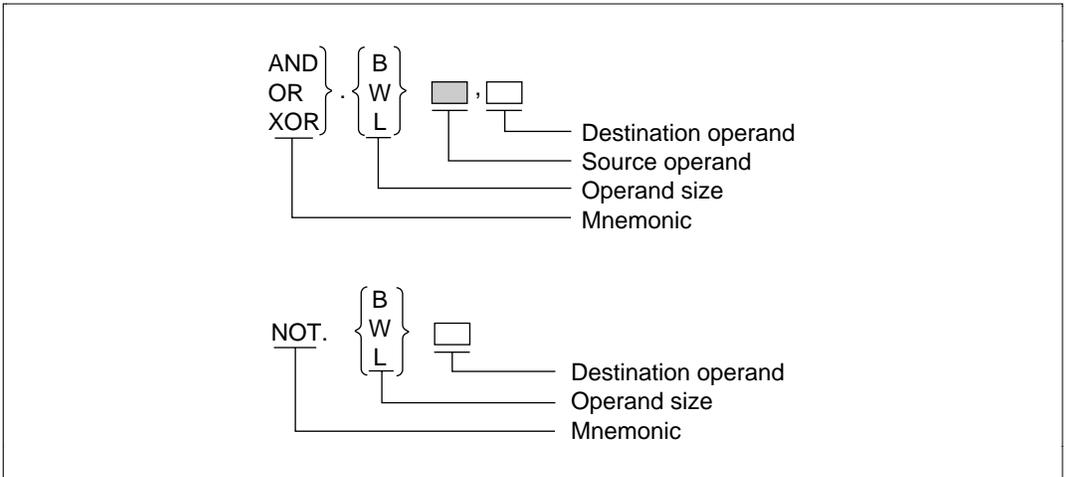


Figure 2.12 AND, OR, XOR, NOT

**Table 2.12 AND, OR, XOR, NOT**

Mnemonic	Op. Size	Dest. Op.	Source Op.	Description
AND OR XOR	B	RmH or RmL	#xx:8 or RnH or RnL	
W	Rm or Em	#xx:16 or Rn or En		
L	ERm	#xx:32 or ERn		
NOT	B	RmH or RmL	—	
W	Rm or Em	—		
L	ERm	—		

## 2.4 Shift Instructions

### 2.4.1 SHAL, SHAR, SHLL, SHLR, ROTL, ROTR, ROTXL, ROTXR

The contents of 8-bit, 16-bit, and 32-bit registers can be shifted in the eight ways shown below (figure 2.13).

SHAL (Shift Arithmetic Left): Does an arithmetic shift 1 bit left.

SHAR (Shift Arithmetic Right): Does an arithmetic shift 1 bit right.

SHLL (Shift Logical Left): Does a logical shift 1 bit left.

SHLR (Shift Logical Right): Does a logical shift 1 bit right.

ROTL (Rotate Left): Rotates 1 bit left.

ROTR (Rotate Right): Rotates 1 bit right.

ROTXL (Rotate with eXtend carry Left): Rotates 1 bit left including the C flag.

ROTXR (Rotate with eXtend carry Right): Rotates 1 bit right including the C flag.

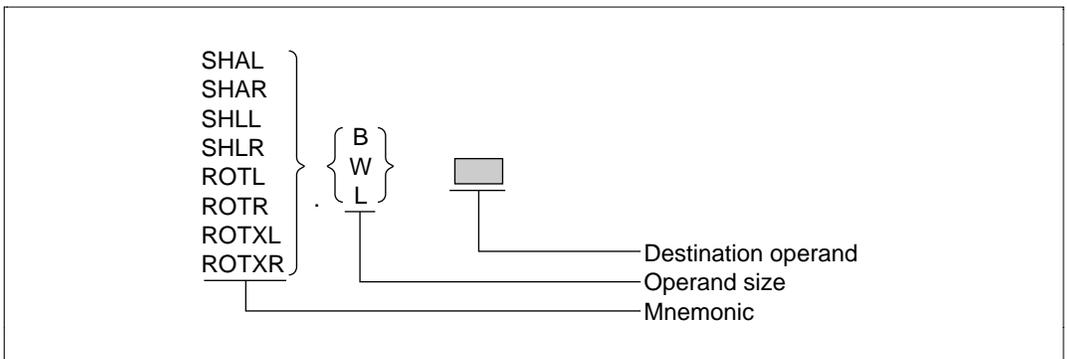


Figure 2.13 SHAL, SHAR, SHLL, SHLR, ROTL, ROTR, ROTXL, ROTXR

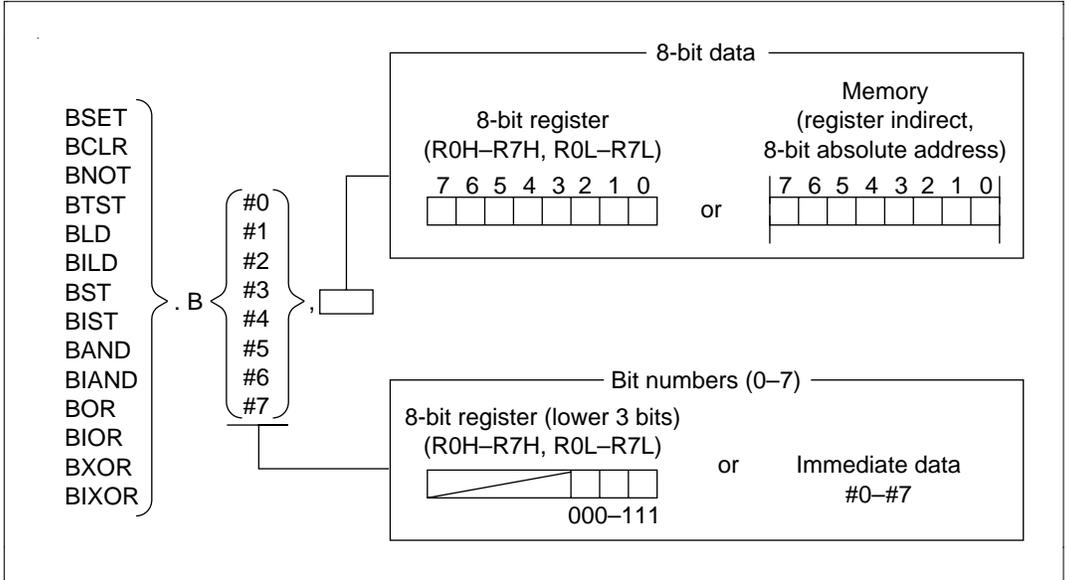
**Table 2.13 SHAL, SHAR, SHLL, SHLR, ROTL, ROTR, ROTXL, ROTXR**

Mnemonic	Destination Operand	Description
SHAL	RnH or RnL, Rn or En, ERn	<p>C flag MSB LSB RnH or RnL, Rn or En, ERn</p>
SHAR	RnH or RnL, Rn or En, ERn	<p>MSB LSB C flag RnH or RnL, Rn or En, ERn</p>
SHLL	RnH or RnL, Rn or En, ERn	<p>C flag MSB LSB RnH or RnL, Rn or En, ERn</p>
SHLR	RnH or RnL, Rn or En, ERn	<p>MSB LSB C flag RnH or RnL, Rn or En, ERn</p>
ROTL	RnH or RnL, Rn or En, ERn	<p>C flag MSB LSB RnH or RnL, Rn or En, ERn</p>
ROTR	RnH or RnL, Rn or En, ERn	<p>MSB LSB C flag RnH or RnL, Rn or En, ERn</p>
ROTXL	RnH or RnL, Rn or En, ERn	<p>C flag MSB LSB RnH or RnL, Rn or En, ERn</p>
ROTXR	RnH or RnL, Rn or En, ERn	<p>MSB LSB C flag RnH or RnL, Rn or En, ERn</p>

## 2.5 Bit Manipulation Instructions

### 2.5.1 BSET, BCLR, BNOT, BTST, BLD, BILD, BST, BIST, BAND, BIAND, BOR, BIOR, BXOR, BIXOR

Bit data can be accessed in the format of the  $n$ th bit ( $n = 0, 1, 2, \dots, 7$ ) of the 8-bit data in the 8-bit registers (R0H–R7H, R0L–R7L) or on memory. The bit numbers for such accesses are specified as 3-bit immediate data or 8-bit register contents (lower 3 bits) (figure 2.14).



**Figure 2.14 BSET, BCLR, BNOT, BTST, BLD, BILD, BST, BIST, BAND, BIAND, BOR, BIOR, BXOR, BIXOR**

**Table 2.14 BSET, BCLR, BNOT, BTST, BLD, BILD, BST, BIST, BAND, BIAND, BOR, BIOR, BXOR, BIXOR**

Category	Mnemonic (Full Name)	Description	
Bit set	BSET (Bit set)	Sets the specified bit to 1.	Specified bit 1 → 
Bit clear	BCLR (Bit clear)	Clears the specified bit to 0.	Specified bit 0 → 
Bit inversion	BNOT (Bit not)	Inverts the specified bit.	Specified bit  → 
Bit test	BTST (Bit test)	Transfers the specified bit to the zero flag.	Specified bit Z bit  → 
Bit transfer	BLD (Bit load)	Transfers the specified bit to the carry flag.	Specified bit C bit  → 
	BILD (Bit invert load)	Transfers the inversion of the specified bit to the carry flag.	Specified bit C bit  → 
	BST (Bit store)	Transfers the carry flag to the specified bit.	C bit Specified bit  → 
	BIST (Bit invert store)	Transfers the inversion of the carry flag to the specified bit.	C bit Specified bit  → 
Bit AND	BAND (Bit and)	Takes the AND of the specified bit and the carry flag and transfers the result to the carry flag.	Specified bit C bit C bit  $\wedge$  → 
	BIAND (Bit invert and)	Takes the AND of the inversion of the specified bit and the carry flag and transfers the result to the carry flag.	Specified bit C bit C bit  $\wedge$  → 

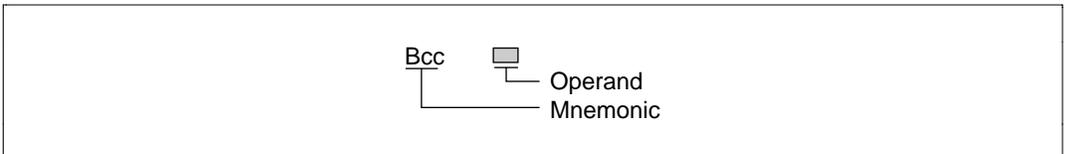
**Table 2.14 BSET, BCLR, BNOT, BTST, BLD, BILD, BST, BIST, BAND, BIAN, BOR, BIOR, BXOR, BIXOR (cont)**

Category	Mnemonic (Full Name)	Description	
Bit OR	BOR (Bit inclusive or)	Takes the OR of the specified bit and the carry flag and transfers the result to the carry flag.	
	BIOR (Bit invert inclusive or)	Takes the OR of the inversion of the specified bit and the carry flag and transfers the result to the carry flag.	
Bit exclusive Or	BXOR (Bit exclusive or)	Takes the exclusive OR of the specified bit and the carry flag and transfers the result to the carry flag.	
	BIXOR (Bit invert exclusive or)	Takes the exclusive OR of the inversion of the specified bit and the carry flag and transfers the result to the carry flag.	

## 2.6 Branch Instructions

### 2.6.1 Bcc

Bcc (Branch Conditionally): This instruction branches when a condition is met (figure 2.15).



**Figure 2.15 Bcc**

**Table 2.15 Bcc**

**Mnem-  
onic**

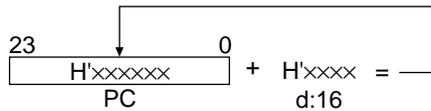
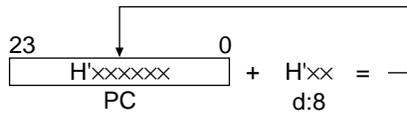
**Operand**

**Description**

Bcc

d:8 or d:16

When the condition is met, the displacement (signed 8 bit or 16 bit) to the branch



<b>Mnemonic</b>	<b>Description</b>	<b>Branch Condition</b>
BRA (BT)	Always (True)	Always
BRN (BF)	Never (False)	never
BHI	High	CVZ = 0
BLS	Low or same	CVZ = 1
BCC(BHS)	Carry clear (high or same)	C = 0
BCS(BLO)	Carry set (Low)	C = 1
BNE	Not equal	Z = 0
BEQ	Equal	Z = 1
BVC	Overflow clear	V = 0
BVS	Overflow set	V = 1
BPL	Plus	N = 0
BMI	Minus	N = 1
BGE	Greater or equal	$N \oplus V = 0$
BLT	Less than	$N \oplus V = 0$
BGT	Greater than	$ZV(N \oplus V) = 0$
BLE	Less or equal	$ZV(N \oplus V) = 0$

## 2.6.2 JMP

JMP (Jump): Jumps unconditionally to branch destination (figure 2.16).

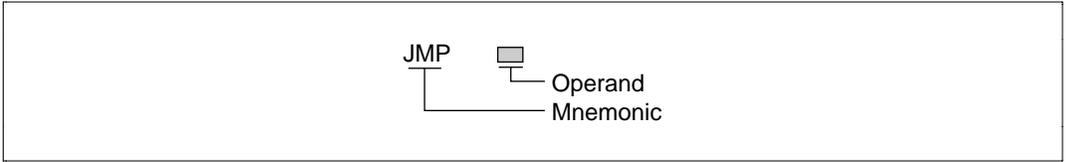


Figure 2.16 JMP

Table 2.16 JMP

Mnemonic	Operand	Description
JMP	@ERn or @aa:24 or @@aa:8	Branch destination address transferred to PC
	@ERn	
	@aa:24	H'xxxxxx
	@@aa:8	

## 2.6.3 BSR, JSR

JSR (Jump to Subroutine, BSR (Branch to Subroutine): Instructions that jump to subroutines (figure 2.17).

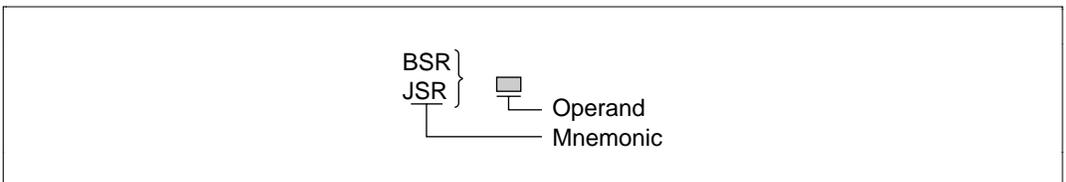
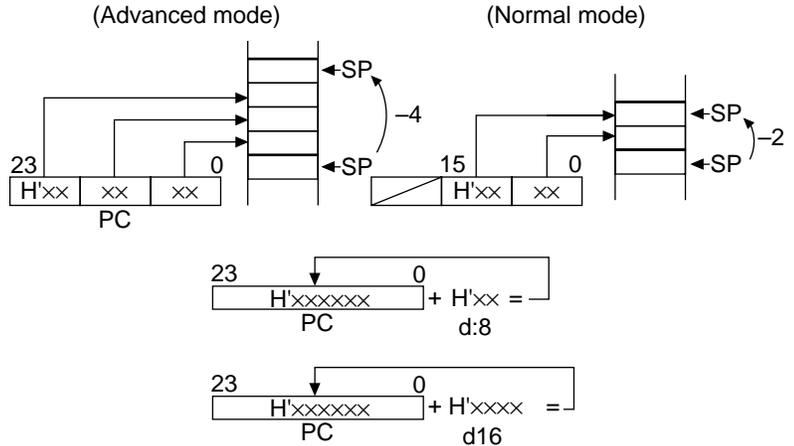


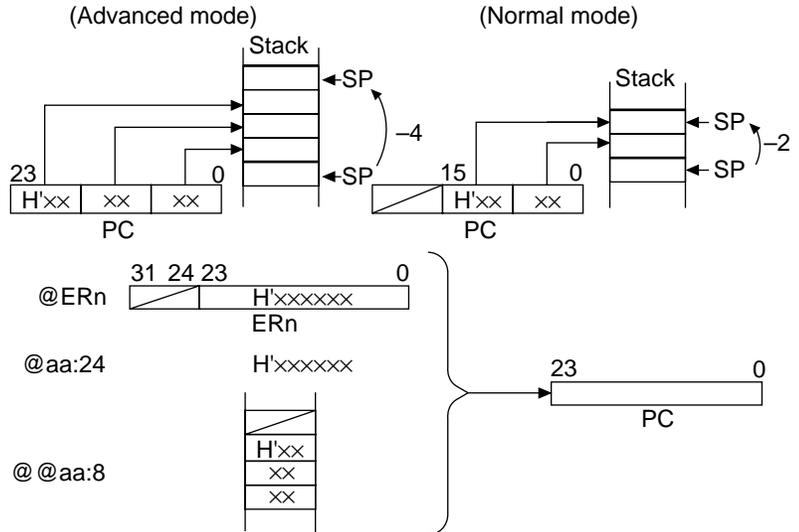
Figure 2.17 BSR, JSR

**Table 2.17 BSR, JSR**

Mnemonic	Operand	Description
BSR	d:8 or d:16	The contents of the PC are saved to the stack and the displacement (signed 8 bit or signed 16 bit) to the subroutine start destination is added to the PC contents



JSR	@ERn or @aa:24 or @@aa:8	The contents of the PC are saved to the stack
-----	--------------------------------	---



## 2.6.4 RTS

RTS (Return from Subroutine): Returns from a subroutine (figure 2.18).

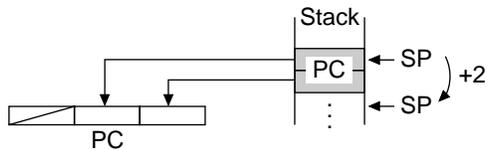
RTS  
 └── Mnemonic

**Figure 2.18 RTS**

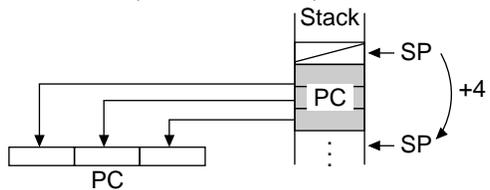
**Table 2.18 RTS**

Mnemonic	Op. Size	Source Operand	Destination Operand	Description
RTS	—	—	—	When jumping to a subroutine using BSR or JSR, the contents of the PC saved in the stack are transferred back to the PC. After the transfer, the stack pointer is incremented (+2 for normal mode and +4 for advanced mode)

(Normal mode)



(Advanced mode)



## 2.7 System Control Instructions

### 2.7.1 RTE

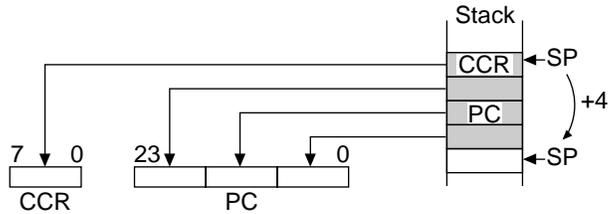
RTE (Return from Exception): Returns for exception processing program. (figure 2.19)

RTE  
 └── Mnemonic

**Figure 2.19 RTE**

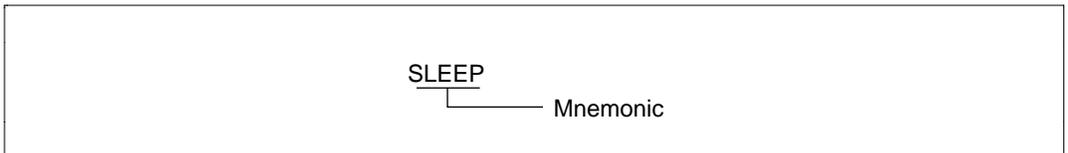
**Table 2.19 RTE**

Mnemonic	Op. Size	Source Op.	Dest. Op.	Description
RTE	—	—	—	When a hardware interrupt or software interrupt (TRAPA instruction) occurs, the CCR and PC automatically saved to the stack by the hardware are returned from the stack



### 2.7.2 SLEEP

**SLEEP (Sleep):** The SLEEP instruction places the CPU in power-down status (figure 2.20). In power-down status, the internal state of the CPU is maintained and instruction execution halted to wait for a request for exception processing to occur. When a request for exception processing does occur, the power-down state is cleared and the CPU begins exception processing. Any interrupt requests other than NMI will be masked on the CPU side at this time so the power-down status will not be cleared.



**Figure 2.20 SLEEP**

### 2.7.3 LDC, STC

**LDC (LodD to Control Register):** Transfers 8-bit data to the CCR (figure 2.21).

**STC (Store from Control Register):** Transfers the contents of the CCR to register or memory (figure 2.21).

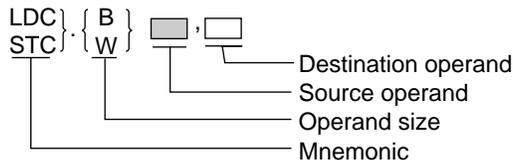


Figure 2.21 LDC, STC

Table 2.20 LDC, STC

Mnemonic	Op. Size	Destination Operand	Source Operand	Description
LDC	B	#xx:8 or RnH or RnL	CCR	The 8-bit immediate data or the contents of the RnH or RnL 8-bit registers are transferred to the CCR
	W	@ERn @(d:16,ERn) @(d:24,ERn) @-ERn @aa:8 @aa:16 @aa:24		The contents of the even address are transferred to the CCR
STC	B	CCR	RnH or RnL	The 8-bit immediate data or the contents of the RnH or RnL 8-bit registers are transferred to the CCR
	W	@ERn @(d:16,ERn) @(d:24,ERn) @ERn+ @aa:8 @aa:16 @aa:24		The contents of the even address are transferred to the CCR

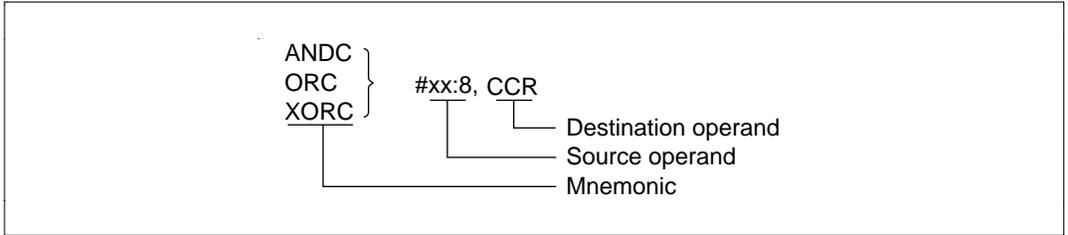
## 2.7.4 ANDC, ORC, XORC

These instructions do logical operations with the contents of the CCR (figure 2.22).

ANDC (AND Control Register): Takes the logical product.

ORC (Inclusive OR Control Register): Takes the logical sum.

XORC (Exclusive OR Control Register): Takes the exclusive logical sum.



**Figure 2.22 ANDC, ORC, XORC**

**Table 2.21 ANDC, ORC, XORC**

Mnemonic	Op. Size	Destination Operand	Source Operand	Description
ANDC ORC XORC	B	CCR	#xx:8	

### 2.7.5 NOP

NOP (No Operation): Only increments the PC by 2. No effect on the internal status of the CPU (figure 2.23).



**Figure 2.23 NOP**

### 2.7.6 TRAPA

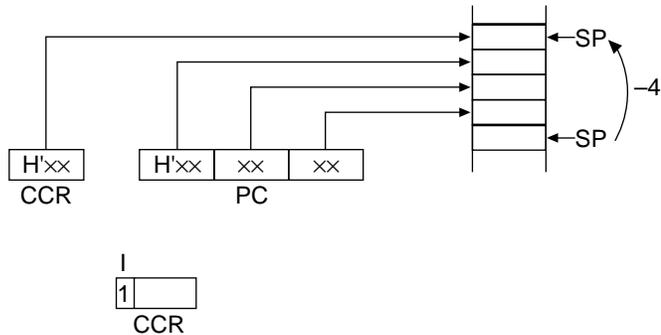
TRAPA (Trap Always): Generates a software interrupt (figure 2.24).

TRAPA { #0  
#1  
#2  
#3 }

Figure 2.24 TRAPA

Table 2.22 TRAPA

Mnemonic	Operand	Description
ANDC	#0 or #1 or	CCR and PC saved to stack
ORC	#2 or #3	
XORC		



#xx	Vector Address	
	Normal Mode	Advanced Mode
0	H'0008–H'0009	H'000010–H000013
1	H'000A–H'000B	H'000014–H000017
2	H'000C– H'000D	H'000018–H00001B
3	H'000E–H'000F	H'00001C–H00001F

## 2.8 Block Transfer Instructions

### 2.8.1 EEPMOV

EEPMOV (Move data to EEPROM): Transfer block data to any address. No interrupts will be detected during the data transfer (figure 2.25).

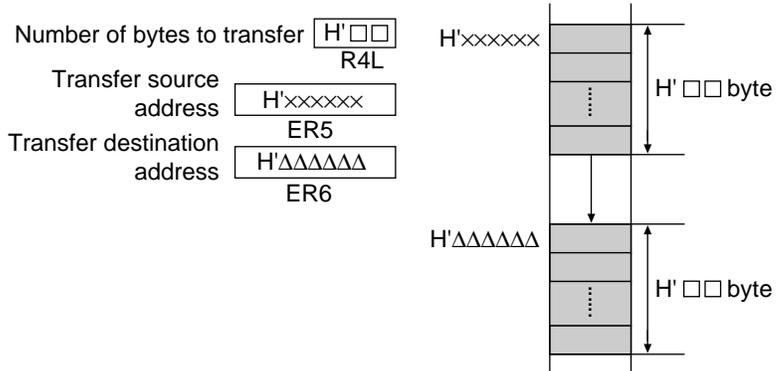
EEPMOV ·  $\left. \begin{matrix} \{B\} \\ \{W\} \end{matrix} \right\} \text{Operand size}$

**Figure 2.25 EEPMOV**

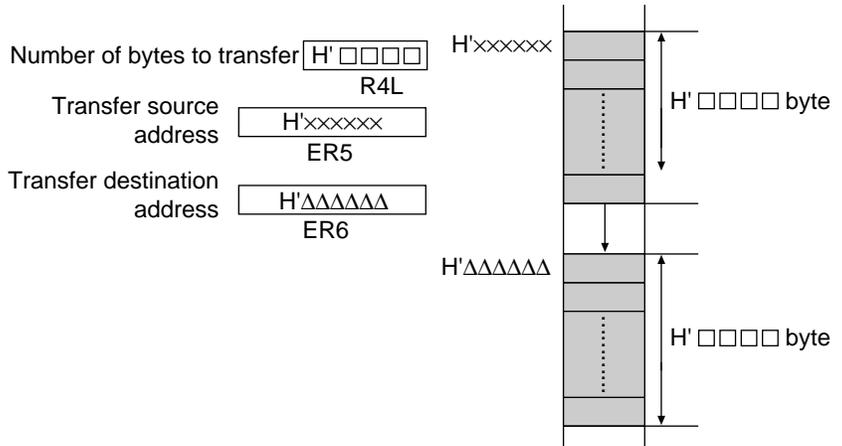
**Table 2.23 EEPMOV**

Mnemonic	Op. Size	Description
----------	----------	-------------

EEP-MOV	B	Transfers the block data that starts at the address in ER5 to the address in ER6. The maximum block data length is 255 bytes.
---------	---	---

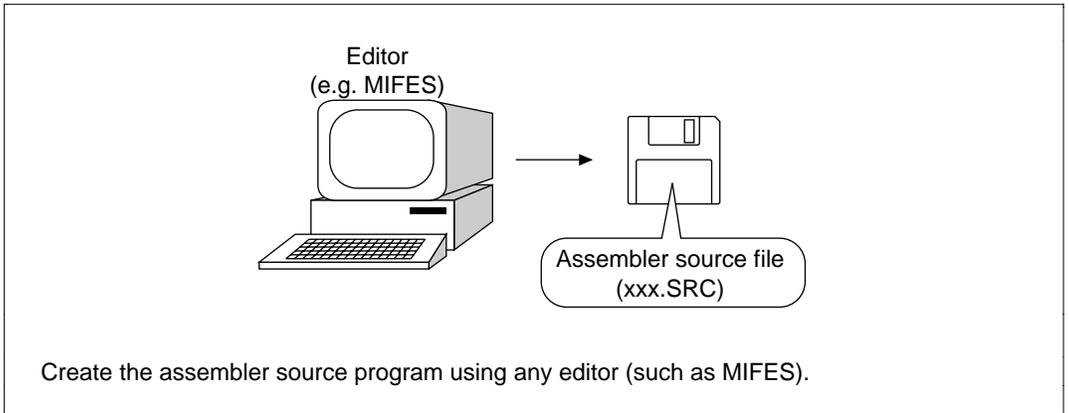


W	W	Transfers the block data that starts at the address in ER5 to the address in ER6. The maximum block data length is 65535 bytes.
---	---	---

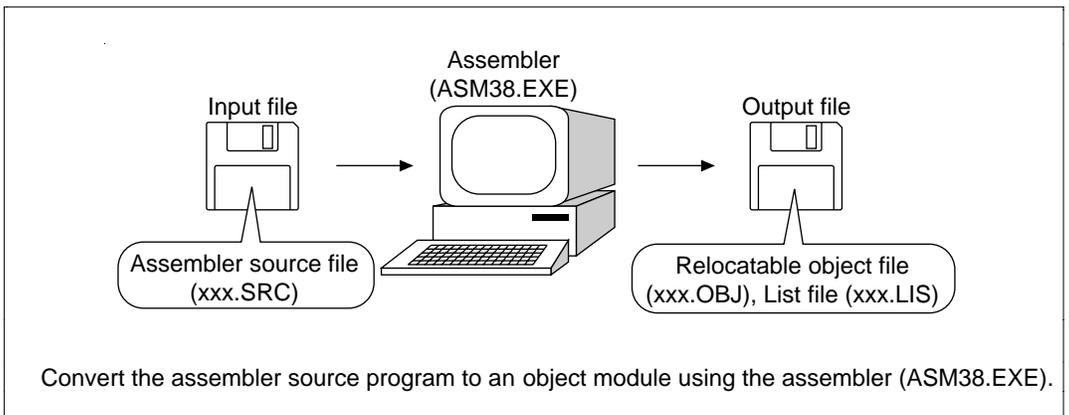


## Section 3 Load Module Conversion Procedures

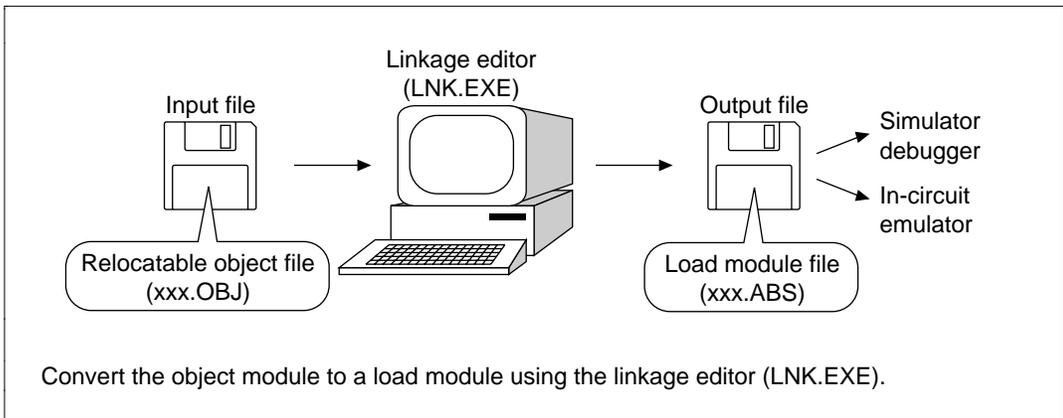
Figures 3.1 through 3.4 show the load module conversion procedures for the H8/300H.



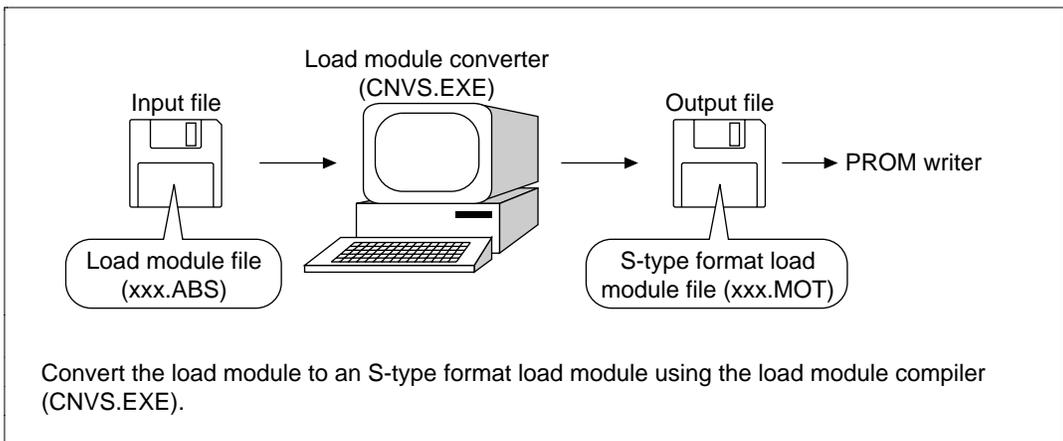
**Figure 3.1 Load Module Conversion Procedures (Step 1)**



**Figure 3.2 Load Module Conversion Procedures (Step 2)**



**Figure 3.3 Load Module Conversion Procedures (Step 3)**



**Figure 3.4 Load Module Conversion Procedures (Step 4)**

# Section 4 Examples of Software Applications

## 4.1 Software Applications Examples

Table 4.1 lists software application examples.

**Table 4.1 List of Software Application Examples**

<b>Software title</b>	<b>Label</b>	<b>Use</b>	<b>Section</b>
Block transfer	MOVE	MOV.L instruction, post-increment register indirect	4.3
Block transfer using block transfer instruction	EEMOV	EEMOV.W instruction	4.4
Branching using a table	CCASE	Register indirect with displacement	4.5
Count of number of logical 1 bits in 8-bit data	HCNT	ROTL.B instruction, ADDX.B instruction	4.6
Find first 1 in 32-bit data	FIND1	SHLL.L instruction	4.7
64-bit binary addition	ADD	ADD.L instruction	4.8
64-bit binary subtraction	SUB	SUB.L instruction	4.9
Unsigned 32-bit binary multiplication	MUL	MULXU.W instruction	4.10
Unsigned 32-bit binary division	DIV	SHLL.L instruction, ROTL.L instruction	4.11
Signed 16-bit binary multiplication	MULXS	MULXS.W instruction	4.12
Signed 32-bit binary multiplication	MULS	MULXU.W instruction	4.13
Signed 32-bit binary division (16-bit divisor)	DIVXS	DIVXS.W instruction	4.14
Signed 32-bit binary division (32-bit divisor)	DIVS	SHLL.L instruction, ROTL.L instruction, NEG.L instruction	4.15
8-digit decimal addition	ADDD	DAA.B instruction	4.16
8-digit decimal subtraction	SUBD	DAS.B instruction	4.17
Product/sum operations	SEKIWA	MULXU.W instruction	4.18
Sorting	SORT	Post-increment register indirect, pre-decrement register indirect	4.19

## 4.2 Using Software Examples

Sections 4.3 through 4.19 provide detailed information about the software applications listed in table 4.1. The following information is consistent throughout sections 4.3 through 4.19.

- Internal registers:
  - ER0–ER7: 32-bit general registers that link En and Rn n = 0, 1, 2, ... 7.
  - E0–E7: 16-bit extended registers
  - R0–R7: 16-bit general registers that link RnH and RnL n = 0, 1, 2, ... 7.
  - R0H–R7H and R0L–R7L: 8-bit general registers
- Condition code register (shown in figures labeled “Changes in Internal Registers and Flag Changes ...”):
  - C: Carry flag
  - V: Overflow flag
  - Z: Zero flag
  - N: Negative flag
  - U: User bit
  - H: Half carry bit
  - U: User bit
  - I: Interrupt mask bit
- Programming Specifications: Describes the specifications of the software.
  - Program memory bytes.: Indicates the amount of ROM used by the software.
  - Data memory bytes.: Indicates the amount of RAM used by the software.
  - Stack bytes.: Indicates the amount of stack used by the software. This does not include the stack used by subroutine calls in the user program. When executing software, the amount of stack in bytes indicated for the stack area is required, so ensure that the stack requirements are available in the data memory before execution.
  - Number of states: Indicates the number of states in which the software is executed. The execution time of the software is calculated as follows:

$$\text{Execution time (s)} = \text{No. of states} \times \text{Cycle time (s)},$$

where

$$\text{Cycle time (s)} = 1/\text{system clock frequency } \phi \text{ (Hz)},$$

and

System clock frequency  $\phi$  (Hz) = External pulse generator frequency 2 divider circuit version/2,

or

External pulse frequency 1:1 oscillation versions.

- Re-entrant: Indicates whether the structure can be used simultaneously from multiple programs.
- Relocation: Indicates whether the software will run normally no matter where in the memory space it is placed.
- Interrupts during execution: Indicates whether the software will run normally even after an interrupt routine is executed when the software is running. If it won't, inhibit interrupts prior to calling the software.

#### 4.2.1 Program Listing Page Format (Format 4)

The following list explains the format of the programming list software.

1. List line numbers
2. Location counter values
3. Object code
4. Source line numbers
5. Source statements
6. Comments
- 7–10 Assembler control instructions

Table 4.2 lists the assembler control instructions used by this software. These instructions are described further in Appendix B, Assembler Control Instruction Functions. For control instructions not listed in table 4.2, see the H8/300H Series Cross-Assembler Users Manual.

**Table 4.2 Assembler Control Instructions List**

<b>Control Instruction</b>	<b>Function</b>
.CPU	Specifies CPU
.SECTION	Specifies section
.EQU	Sets symbol value
.ORG	Sets location counter values
.DATA	Reserves integer data
.RES	Reserves integer data space
.END	End of source program

### 4.3 Block Transfer

**MCU:** H8/300H Series

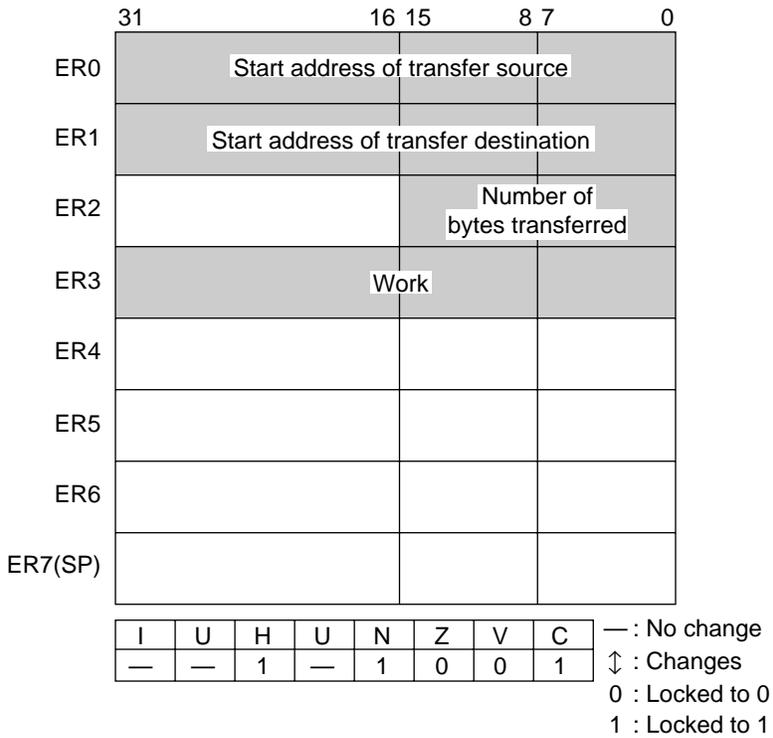
**Label Name:** MOVE

**Functions Used:** MOV.L Instruction, Post-Increment Register Indirect

**Function:** Transfers block data (up to 65535 bytes) to any even address.

**Table 4.3 MOVE Arguments**

	<b>Contents</b>	<b>Storage Location</b>	<b>Data Length (Bytes)</b>
Input	Start address of transfer source	ER0	4
	Start address of transfer destination	ER1	4
	Number of bytes transferred	ER2	2
Output	—	—	—



**Figure 4.1 Changes in Internal Registers and Flag Changes for MOVE**

Program memory (bytes)
38
Data memory (bytes)
0
Stack (bytes)
0
Number of states
491580
Re-entrant
Yes
Relocation
Yes
Interrupts during execution
Yes

Caution: The number of states given in the programming specifications is the value when H'FFFF bytes are being transferred.

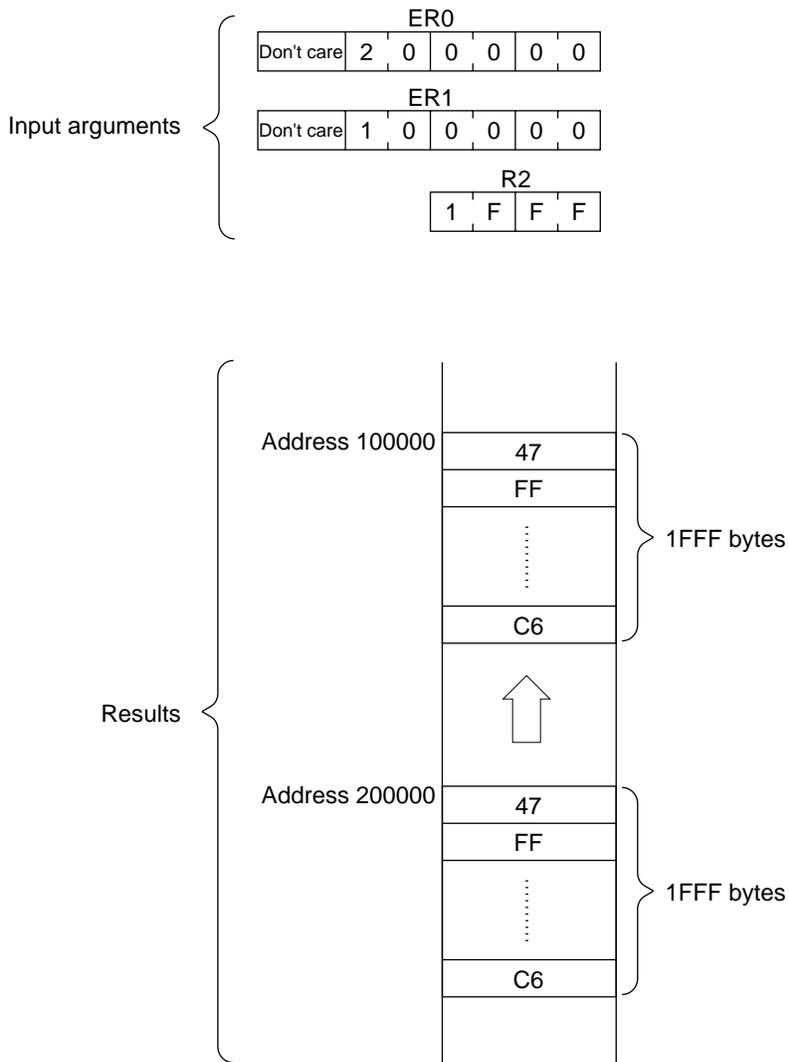
**Figure 4.2 Programming Specifications**

### 4.3.1 Description of Functions

Arguments are as follows:

- ER0: Sets the start address of the transfer source as the input argument
- ER1: Sets the start address of the transfer destination as the input argument
- R2: Sets the number of bytes to be transferred as the input argument

Figure 4.3 is an example of execution of the software MOVE. When the input arguments are set as shown, the data at the transfer source is transferred as a block to the transfer destination (results).



**Figure 4.3 Executing MOVE**

### 4.3.2 Cautions for Use

- Since R2 is 2 bytes, set data in the region  $H'0001 \leq R2 \leq H'FFFF$ .
- Set the input arguments so that the block data of the transfer source (area (A) of figure 4.4) and the block data of the transfer destination (area (B) of the figure) do not overlap.
- When the transfer source and transfer destination overlap as shown in figure 4.4, the data of the transfer source that overlaps ( area (C) in the figure) is destroyed.

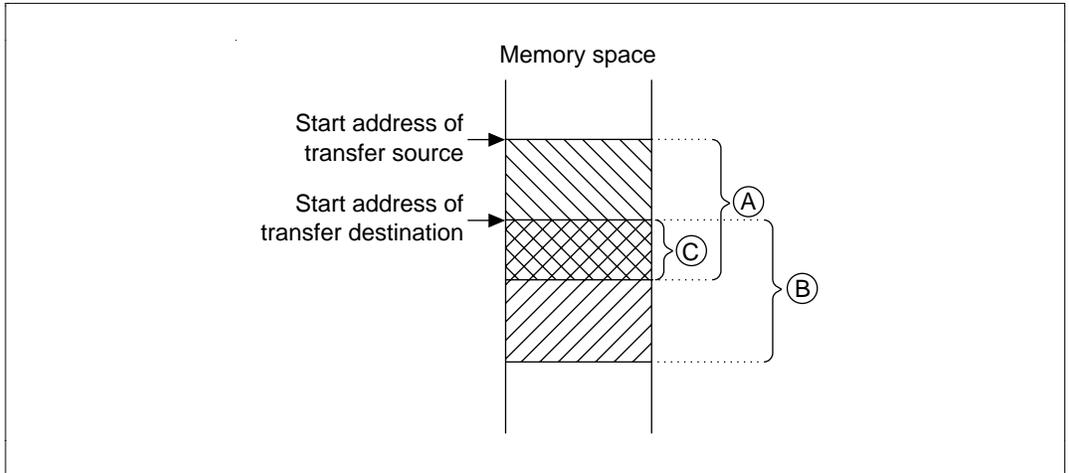


Figure 4.4 Block Transfer with Overlapping Data

### 4.3.3 Description of Data Memory

No data memory is used by MOVE.

#### 4.3.4 Examples of Use

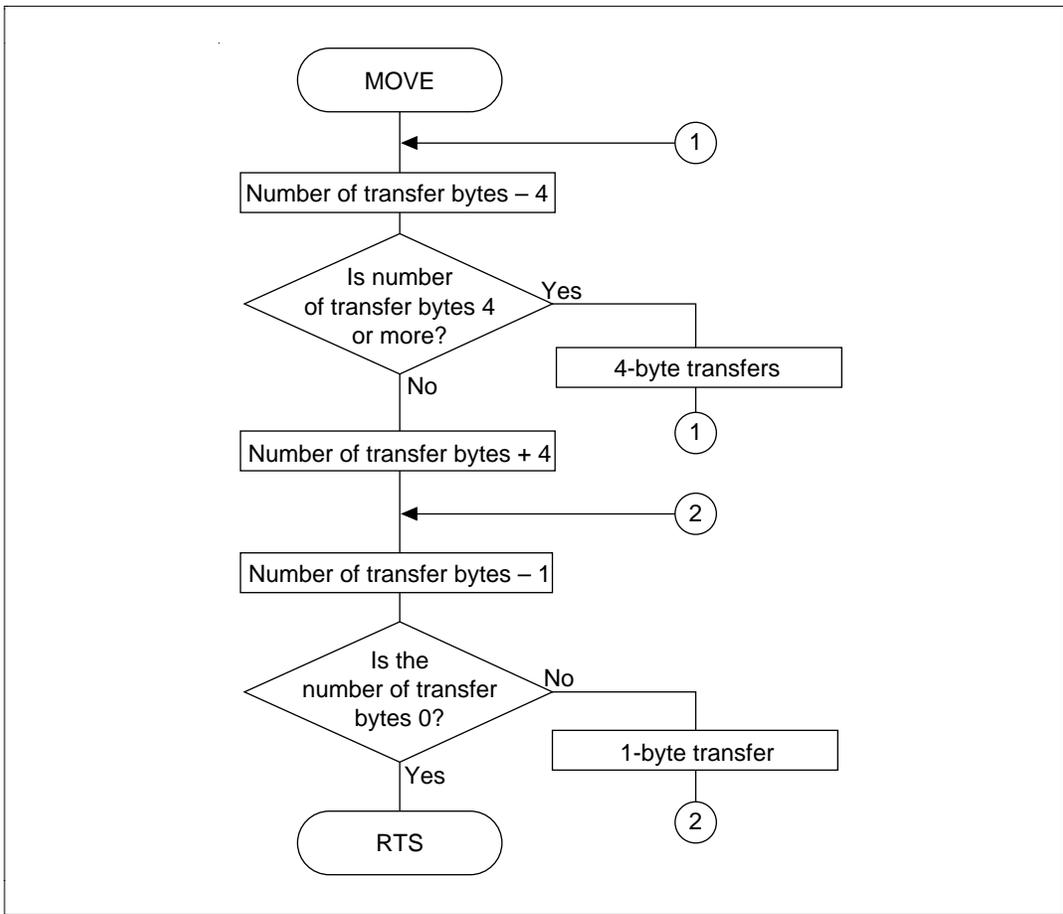
After setting the start address of the transfer source, the start address of the transfer destination and the number of bytes to be transferred, do a subroutine call to MOVE.

**Table 4.4 Block Transfer Example (MOVE)**

Label	Instruction	Action
WORK 1	.RES. L 1	Reserves the data memory area that sets the start address of the transfer source in the user program.
WORK 2	.RES. L 1	Reserves the data memory area that sets the start address of the transfer destination in the user program.
WORK 3	.RES. W 1	Reserves the data memory area that sets the number of bytes to be transferred in the user program.
	MOV. L @WORK1,ER0	Sets the start address of the transfer source as set in the user program as an input argument.
	MOV. L @WORK2,ER1	Sets the start address of the transfer destination as set in the user program as an input argument.
	MOV. L @WORK3, R2	Sets the number of bytes to be transferred as set in the user program as an input argument.
	⋮	Subroutine call to MOVE.
	JSR @MOVE	

#### 4.3.5 Principles of Operation

- When the data to be transferred is 4 bytes or more, the MOV.L instruction is used to do repeated transfers in 4-byte units.
- When the data to be transferred is less than 4 bytes, the software switches to the MOV.B instruction to do transfers in byte units.



**Figure 4.5 MOVE Flowchart**

## 4.3.6 Program Listing

## 4.4 Block Transfer Using Block Transfer Instruction

MCU: H8/300H Series

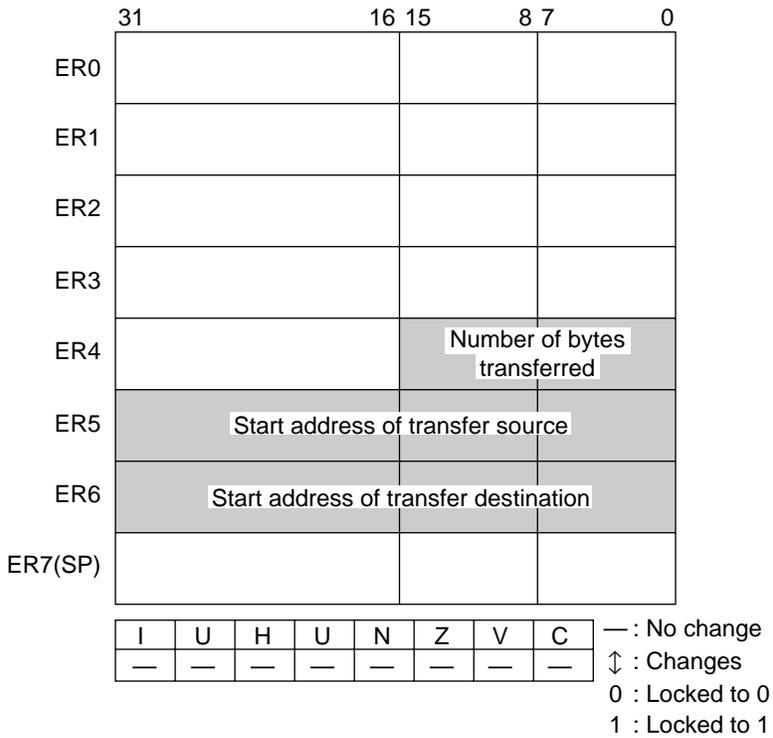
**Label Name:** EEPMOV

**Functions Used:** EEPMOV.W Instruction

**Function:** Transfers block data (up to 65535 bytes) to any even address using the block transfer instruction (EEPMOV.W).

**Table 4.5** EEPMOV Arguments

	<b>Contents</b>	<b>Storage Location</b>	<b>Data Length (Bytes)</b>
Input	Start address of transfer source	ER5	4
	Start address of transfer destination	ER6	4
	Number of bytes transferred	R4	2
Output	—	—	—



**Figure 4.6 Changes in Internal Registers and Flag Changes for EEPMOV**

Program memory (bytes)
4
Data memory (bytes)
0
Stack (bytes)
0
Number of states
262148
Re-entrant
Yes
Relocation
Yes
Interrupts during execution
Yes

Caution: The number of states given in the programming specifications is the value when H'FFFF bytes are being transferred.

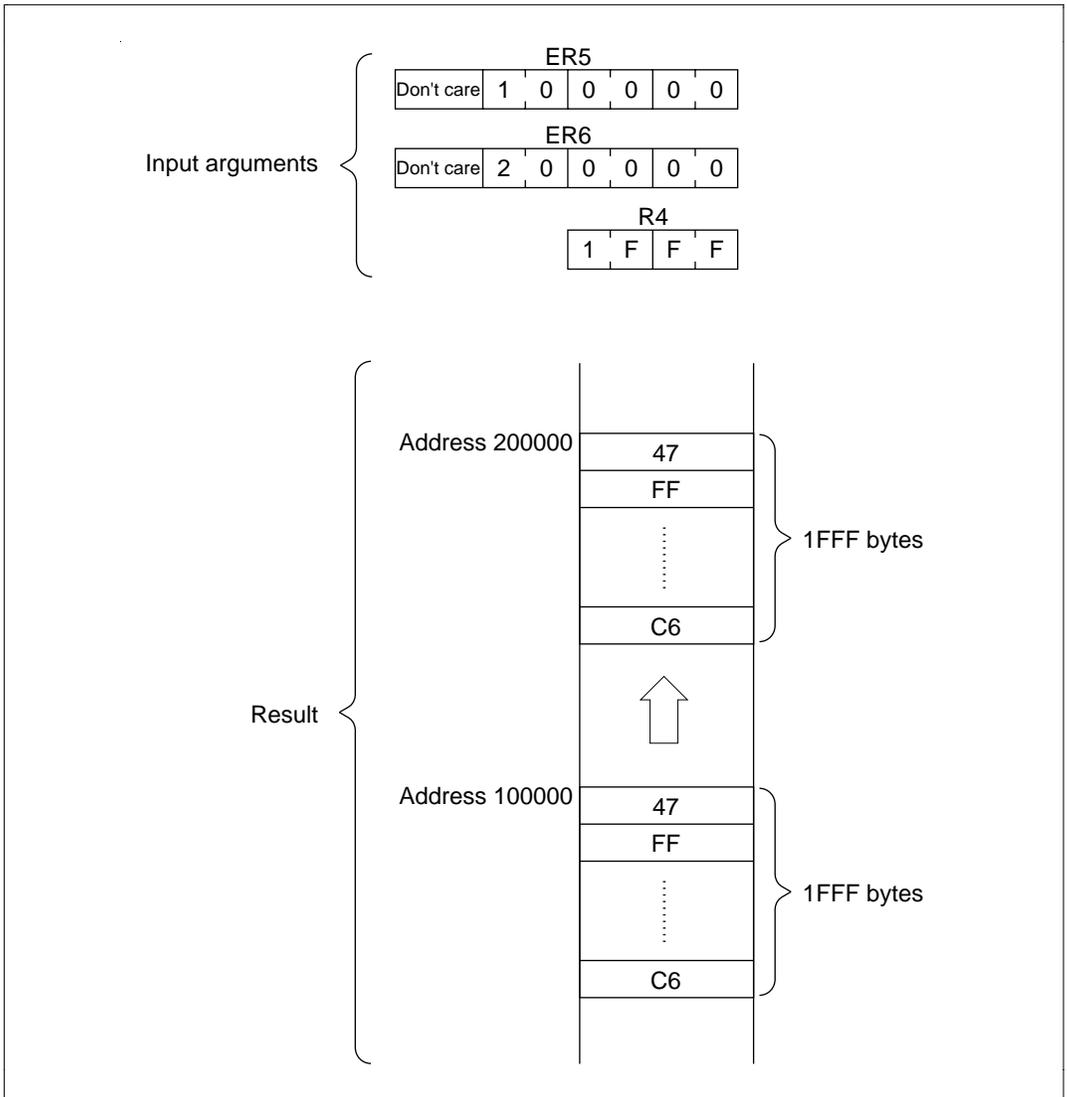
**Figure 4.7 Programming Specifications**

#### 4.4.1 Description of Functions

Arguments are as follows:

- ER5: Sets the start address (even address) of the transfer source.
- ER6: Sets the start address (even address) of the transfer destination.
- R4: Sets the number of bytes to be transferred.

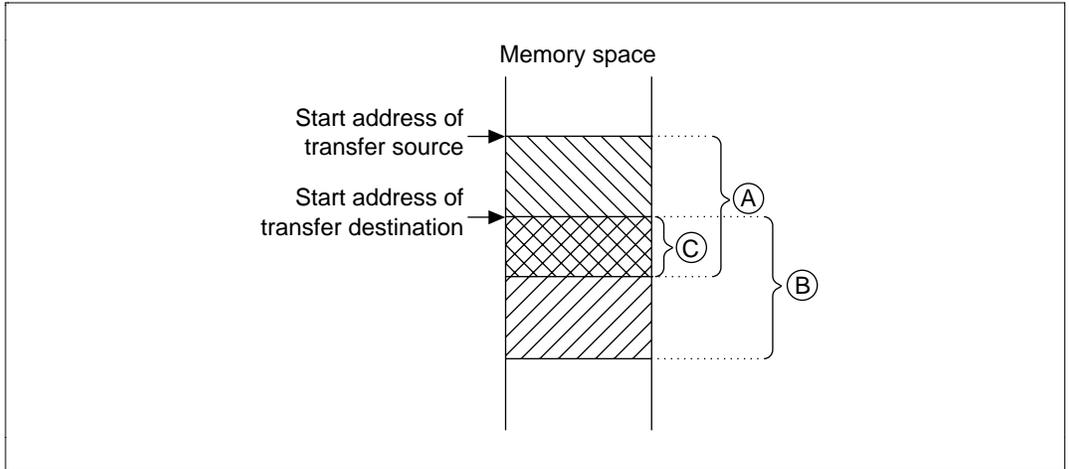
Figure 4.8 is an example of execution of the software EEPMOVE. When input arguments are set as shown, the data at the transfer source is transferred as a block to the transfer destination (result).



**Figure 4.8 Executing EEPMOVE**

#### 4.4.2 Cautions for Use

- Since R2 is 2 bytes, set data in the region  $H'0001 \leq R2 \leq H'FFFF$ .
- Interrupts cannot be detected while EEPMOVE is executing.
- Set the input arguments so that the block data of the transfer source (area (A) of figure 4.9) and the block data of the transfer destination (area (B) of the figure) do not overlap. When the transfer source and transfer destination overlap as shown in figure 4.9, the data of the transfer source that overlaps ( area (C) in the figure) is destroyed.



**Figure 4.9 Block Transfer with Overlapping Data**

#### 4.4.3 Description of Data Memory

No data memory is used by EEPMOVE.

#### 4.4.4 Examples of Use

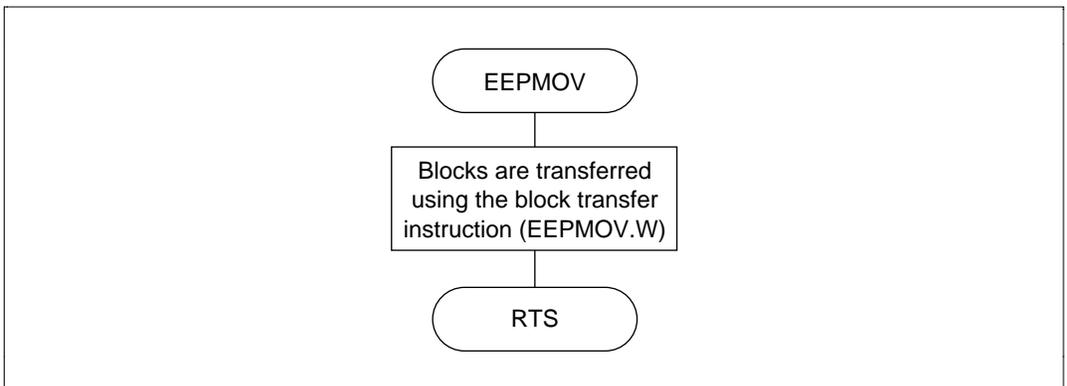
After setting the start address of the transfer source, the start address of the transfer destination and the number of bytes to be transferred, do a subroutine call to EEPMOVE.

**Table 4.6 Block Transfer Example (EEPMOVE).**

Label	Instruction	Action
WORK 1	RES. L 1	Reserves the data memory area that sets the start address of the transfer source in the user program.
WORK 2	RES. L 1	Reserves the data memory area that sets the start address of the transfer destination in the user program.
WORK 3	RES. W 1	Reserves the data memory area that sets the number of bytes to be transferred in the user program.
	MOV. L @WORK1,ER5	Sets the start address of the transfer source as set in the user program as an input argument.
	MOV. L @WORK2,ER6	Sets the start address of the transfer destination as set in the user program as an input argument.
	MOV. L @WORK3, R4	Sets the number of bytes to be transferred as set in the user program as an input argument.
	⋮	Subroutine call to EEPMOVE.
	JSR @EEPMOV	

#### 4.4.5 Principles of Operation

Use the block transfer instruction (EEPMOV.W).



**Figure 4.10 EEPMOV Flowchart**

## 4.4.6 Program Listing

## 4.5 Branching Using a Table

MCU: H8/300H Series

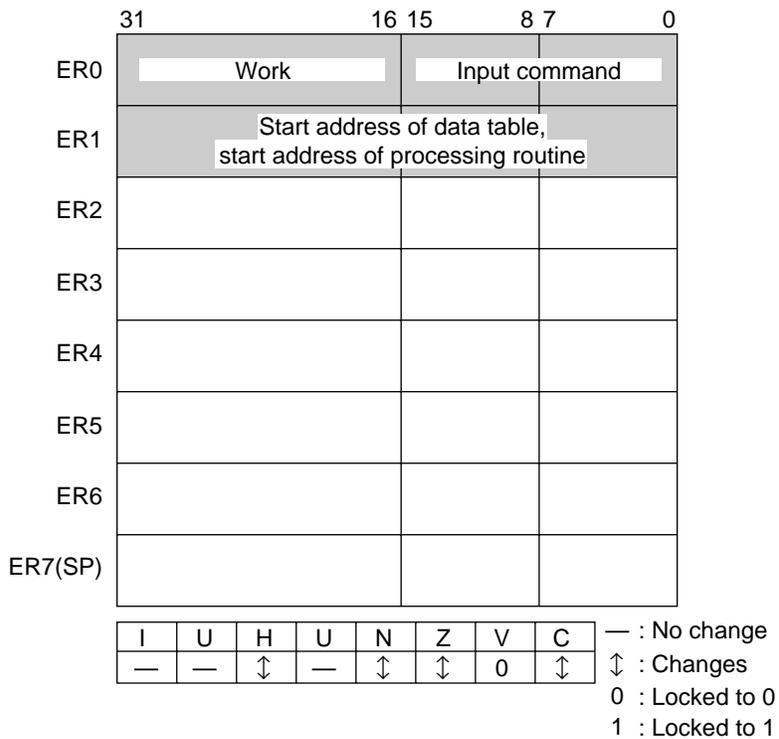
**Label Name:** CCASE

**Functions Used:** Register Indirect with Displacement

**Description:** Searches for the start address of the processing routine for the input command. This function is useful and convenient for decoding commands input from the keyboard and for processing the input command.

**Table 4.7 CCASE Arguments**

	<b>Contents</b>	<b>Storage Location</b>	<b>Data Length (Bytes)</b>
Input	Input command	R0	2
	Start address of data table	ER1	4
Output	Start address of processing routine	ER1	4
	Existence of a processing routine for the input command (yes = 0, no = 1)	Z flag (CCR)	1



**Figure 4.11 Changes in Internal Registers and Flag Changes for CCASE**

Program memory (bytes)
26
Data memory (bytes)
0
Stack (bytes)
0
Number of states
156
Re-entrant
Yes
Relocation
Yes
Interrupts during execution
Yes

Caution: The number of states given in the programming specifications is the value when the last of 6 groups of data is detected.

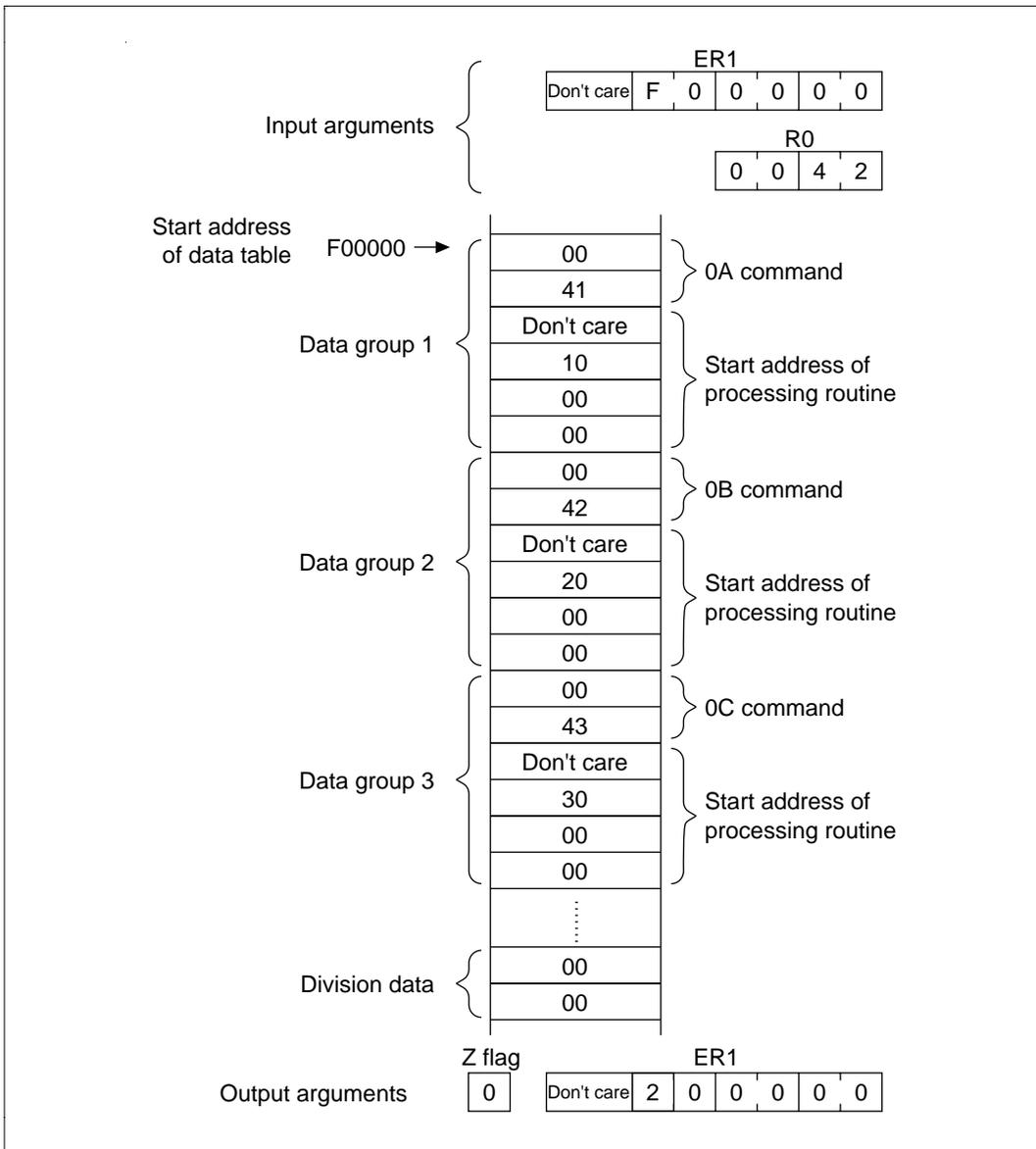
**Figure 4.12 Programming Specifications**

#### 4.5.1 Description of Functions

Arguments are as follows:

- R0: Sets the 16-bit command as an input argument.
- ER1: Sets the start address of the data table as an input argument. Also set the start address of the processing routine for the command as the output argument.
- Z flag (CCR): Indicates whether there are any errors after execution of CCASE.
  - When Z flag = 0: Indicates that there is a command on the data table that corresponds to the one set in R0.
  - When Z flag = 1: Indicates that there is no command on the data table that corresponds to the one set in R0.

Figure 4.13 is an example of execution of the software CCASE. When the input arguments are set as shown, the data table is checked and the start address of the processing routine is set in ER1.



**Figure 4.13 Executing CCASE**

### 4.5.2 Cautions for Use

Since H'0000 is used as the division data, do not use H'0000 as a command in the data table.

### 4.5.3 Description of Data Memory

No data memory is used by CCASE.



**Table 4.8 Block Transfer Example (CCASE) (cont)**

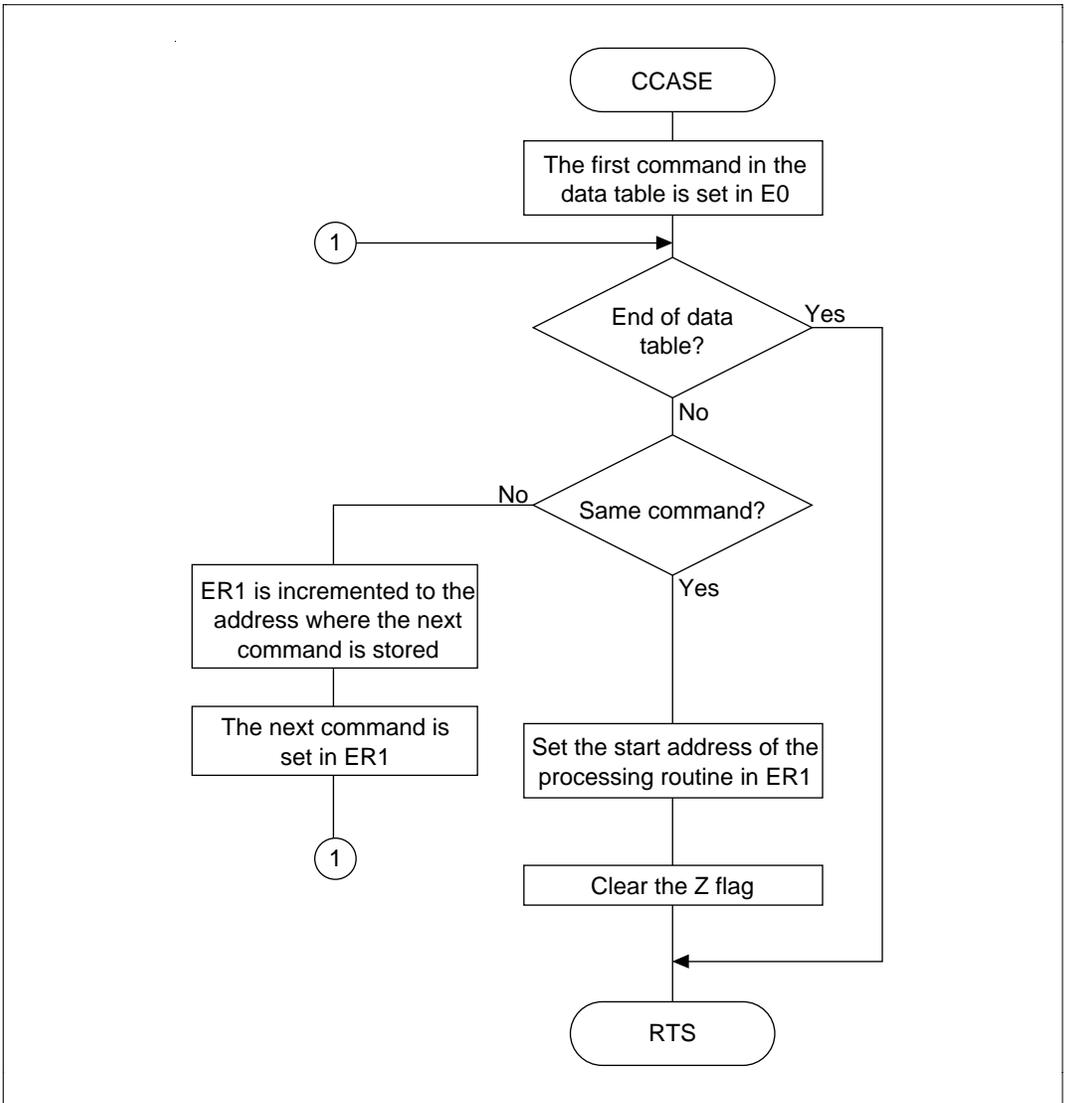
Label	Instruction	Action				
DTABLE	.ORG H'F000	Start address of data table				
	.DATA. H'0041 W	0A command				
	.DATA. H'F100 W	Start address of processing routine for 0A command				
	.DATA. H'0042 W	0B command				
	.DATA. H'F200 W	Start address of processing routine for 0B command				
	.DATA. H'0000 W	Division data				
		Subroutine call of CCASE				
	<table border="1" style="margin-left: auto; margin-right: auto;"><tr><td> </td><td>JSR</td><td>@CCASE</td><td> </td></tr></table>		JSR	@CCASE		
	JSR	@CCASE				
	BEQ ERROR	Branches to ERROR when the Z flag is set				
↑ Branches to processing routine ↓	JMP @ER1	Jumps to processing routine				
ERROR	<table border="1" style="margin-left: auto; margin-right: auto;"><tr><td> </td><td>Error program</td><td> </td></tr></table>		Error program			
	Error program					

Note: Example of program that branches to a processing routine: CCASE only sets the start address of the processing routine in ER. When actually branching to a processing routine, create a program like that shown below.

### 4.5.5 Principles of Operation

- ER1 is used as a pointer to the address storing the command on the data table.
- The command at the address indicated in ER1 of the data table is set in E0 and compared to the input command.
- When the input command and the data table command match, the start address of the processing routine located after the command is set, the Z flag is cleared and CCASE ends.

- When H'0000 is detected (indicating the end of the data table), the Z flag is set and CCASE ends.



**Figure 4.14 CCASE Flowchart**

## 4.5.6 Program Listing

## 4.6 Counting the Number of Logical 1s in 8-Bit Data

MCU: H8/300H Series

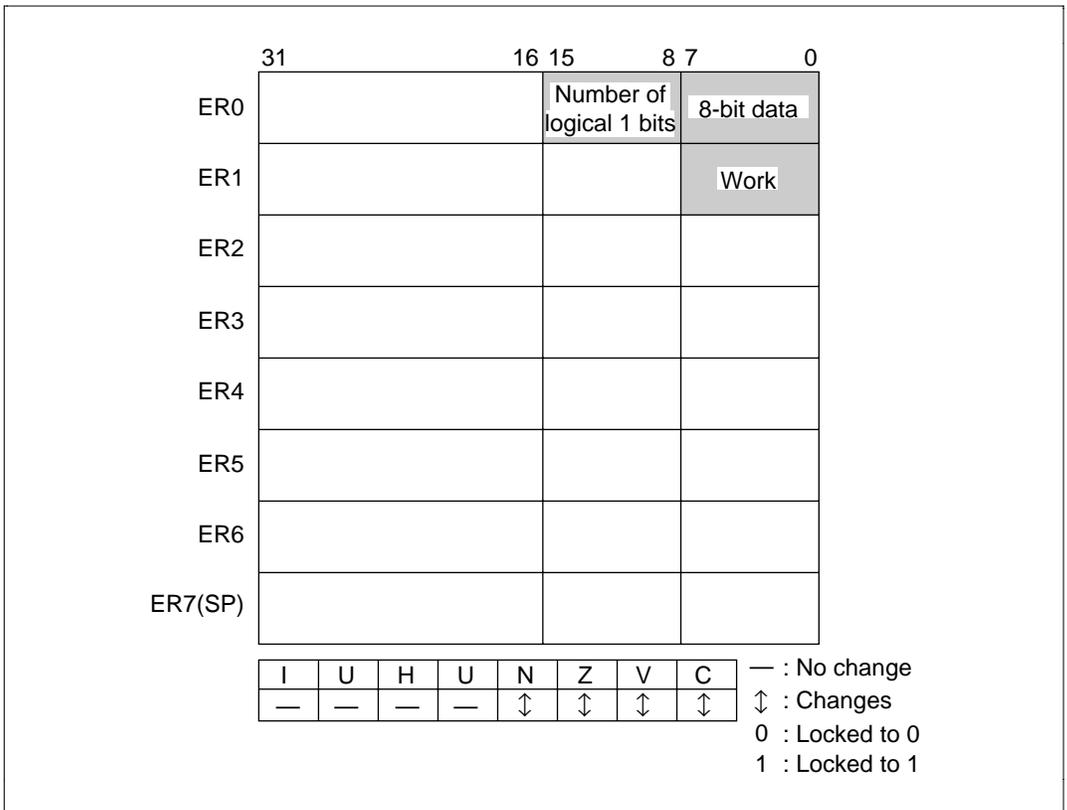
**Label Name:** HCNT

**Functions Used:** ROTL.B Instruction, ADDX.B Instruction

**Function:** Counts the number of logical 1s in 8-bit data.

**Table 4.9 HCNT Arguments**

	Contents	Storage Location	Data Length (Bytes)
Input	8-bit data	R0L	1
Output	Number of logical 1 bits	R0H	1



**Figure 4.15 Changes in Internal Registers and Flag Changes for HCNT**

Program memory (bytes)
16
Data memory (bytes)
0
Stack (bytes)
0
Number of states
126
Re-entrant
Yes
Relocation
Yes
Interrupts during execution
Yes

Caution: The number of states given in the programming specifications is the value when the 8-bit data is H'FF.

**Figure 4.16 Programming Specifications**

#### 4.6.1 Description of Functions

Arguments are as follows:

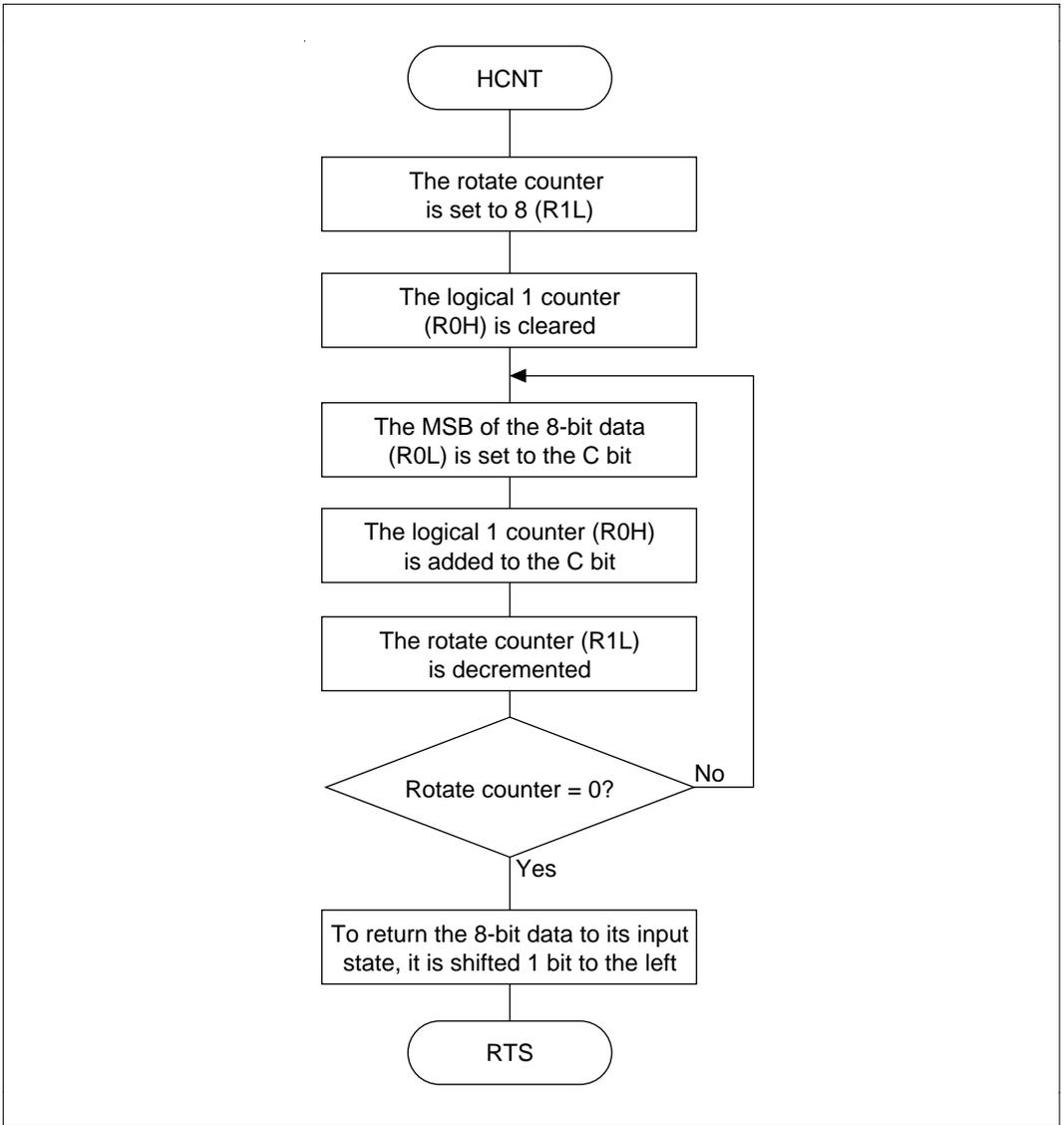
- R0L: Sets the 8-bit data.
- R0H: Sets the number of bits of logical 1s in the 8-bit data.

Figure 4.17 is an example of execution of the software HNCT. When the input arguments are set as shown, the number of bits of logical 1s are set in R0H.



#### 4.6.5 Principles of Operation

- The rotate instruction (ROTL.B) is used and the 8-bit data (ROL) is set 1 bit at a time in the C bit.
- When the logical 1 counter (R0H) is added to 0 using the add instruction with carry (ADDX.B), 1 is added to the logical 1 counter if the C bit is 1 and 0 is added to the logical 1 counter if the C bit is 0.
- The two steps above are repeated until the rotate counter (R1L) becomes 0, which reveals the number of logical 1s in the 8-bit data.



**Figure 4.18 HCNT Flowchart**

**4.6.6 Program Listing**

## 4.7 Find the First 1 in 32-Bit Data

MCU: H8/300H Series

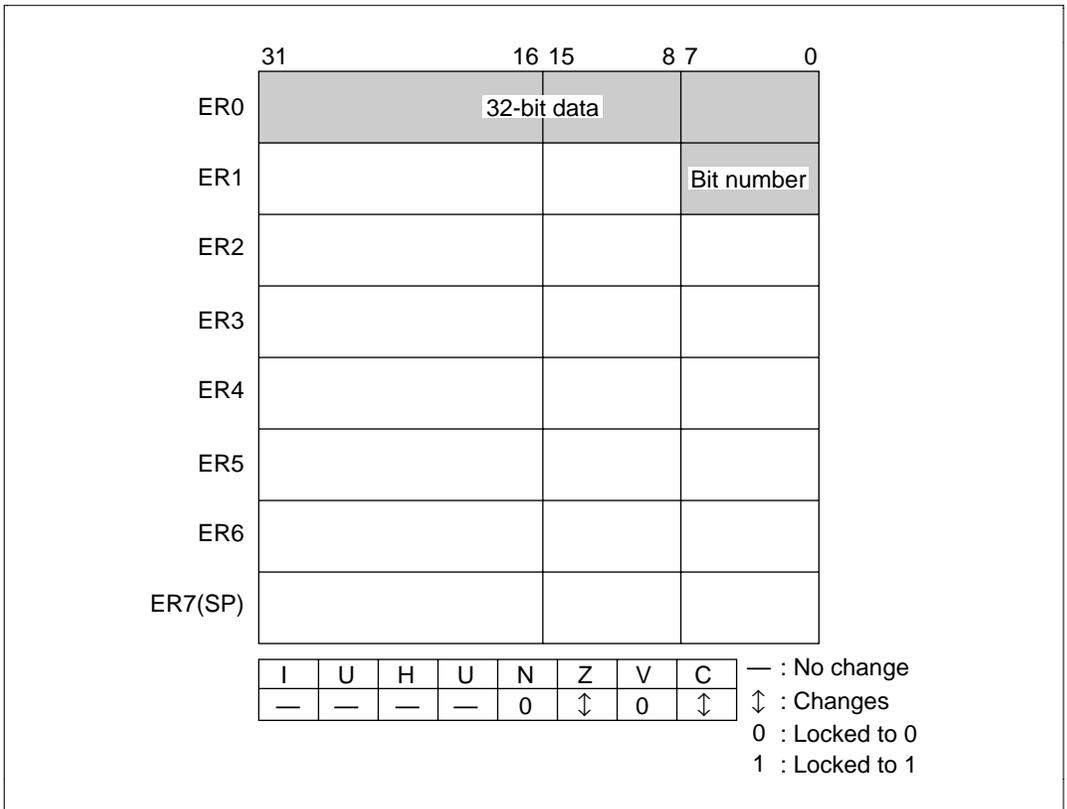
**Label Name:** FIND1

**Functions Used:** SHLL.L Instruction

**Function:** Identifies the bits of 32-bit data in order from bit 31 and finds the number of the first bit that is a 1.

**Table 4.11 FIND1 Arguments**

	Contents	Storage Location	Data Length (Bytes)
Input	32-bit data	ER0	4
Output	Bit number (bit 31–bit 0)	R1L	1



**Figure 4.19 Changes in Internal Registers and Flag Changes for FIND1**

Program memory (bytes)
14
Data memory (bytes)
0
Stack (bytes)
0
Number of states
398
Re-entrant
Yes
Relocation
Yes
Interrupts during execution
Yes

Caution: The number of states given in the programming specifications is the value when the 32-bit data is H'00000000.

**Figure 4.20 Programming Specifications**

### 4.7.1 Description of Functions

Arguments are as follows:

- ER0: Sets the 32-bit data.
- R1L: Sets the number of the first bit found to have a 1 (bit 31 to bit 0).

Figure 4.21 is an example of execution of the software FIND1. When the input arguments are set as shown, the number of the first bit with a 1 is set in R1L.

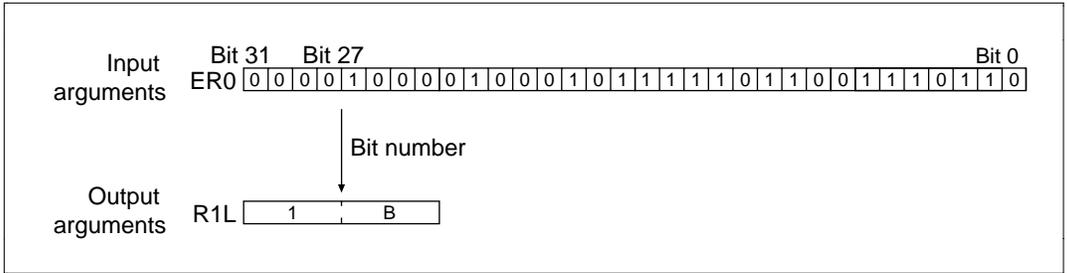


Figure 4.21 Executing FIND1

### 4.7.2 Cautions for Use

When the 32-bit data is H'00000000, H'FF is set as the bit number (R1L).

### 4.7.3 Description of Data Memory

No data memory is used by FIND1.

#### 4.7.4 Examples of Use

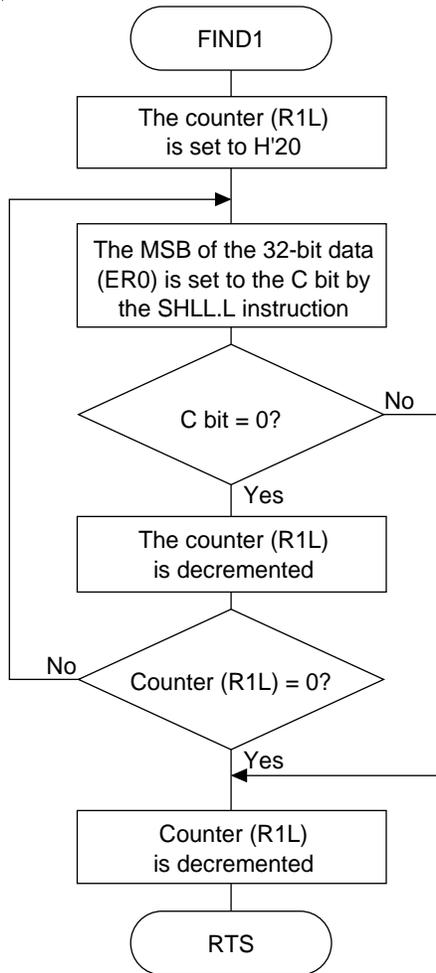
After setting the 32-bit data, do a subroutine call of FIND1.

**Table 4.12 Block Transfer Example (FIND1)**

Label	Instruction	Action				
WORK 1	.RES. L 1	Reserves the data memory area that sets the 32-bit data in the user program.				
WORK 2	.RES. B 1	Reserves the data memory area that sets the number of the bit that has the first 1.				
	MOV. L @WORK1,ER0	Sets the 32-bit data set in the user program as an input argument.				
	<table border="1"><tr><td></td><td>JSR</td><td>@FIND1</td><td></td></tr></table>		JSR	@FIND1		Subroutine call of FIND1
	JSR	@FIND1				
	MOV. B R0H,@WORK2	Stores the number of the first bit set in the output argument that has a 1 in the data memory area of the user program.				

#### 4.7.5 Principles of Operation

- The SHLL.L instruction stores the bits of 32-bit data in the C bit in order from bit 31 in order to identify the bits.
- When the C bit becomes 1, the counter for finding the bit number (R1L) is decremented and FIND1 ends.



**Figure 4.22 FIND1 Flowchart**

**4.7.6 Program Listing**

## 4.8 64-Bit Binary Addition

MCU: H8/300H Series

**Label Name:** ADD

**Functions Used:** ADD.L Instruction

**Function:** Does binary addition in the format: Summand (signed 64 bits) + addend (signed 64 bits) = sum (signed 64 bits).

**Table 4.13 ADD Arguments**

	<b>Contents</b>	<b>Storage Location</b>	<b>Data Length (Bytes)</b>
Input	Bottom 32 bits of summand (signed 64 bits)	ER1	4
	Top 32 bits of addend (signed 64 bits)	ER2	4
	Bottom 32 bits of addend (signed 64 bits)	ER3	4
Output	Top 32 bits of sum (signed 64 bits)	ER0	4
	Bottom 32 bits of sum (signed 64 bits)	ER1	4
	Existence of carrying (yes = 0, no = 1)	C flag (CCR)	1

	31	16	15	8	7	0		
ER0	Top 32 bits of summand, top 32 bits of sum							
ER1	Bottom 32 bits of summand, bottom 32 bits of sum							
ER2	Top 32 bits of addend							
ER3	Bottom 32 bits of addend							
ER4								
ER5								
ER6								
ER7(SP)								

I	U	H	U	N	Z	V	C	
—	—	↕	—	↕	↕	0	↕	— : No change ↕ : Changes 0 : Locked to 0 1 : Locked to 1

**Figure 4.23 Changes in Internal Registers and Flag Changes for ADD**

Program memory (bytes)
18
Data memory (bytes)
0
Stack (bytes)
0
Number of states
26
Re-entrant
Yes
Relocation
Yes
Interrupts during execution
Yes

**Figure 4.24 Programming Specifications**

### 4.8.1 Description of Functions

Arguments are as follows:

- ER0: Sets the top 32-bits of the summand (signed 64 bits) as an input argument. Sets the top 32 bits of the sum (signed 64 bits) as an output argument.
- ER1: Sets the bottom 32-bits of the summand (signed 64 bits) as an input argument. Sets the bottom 32 bits of the sum (signed 64 bits) as an output argument.
- ER2: Sets the top 32-bits of the addend (signed 64 bits) as an input argument.
- ER3: Sets the bottom 32-bits of the addend (signed 64 bits) as an input argument.
- C flag (CCR): Indicates whether a carry has occurred after execution of ADD.
  - When C flag = 0: Indicates a carry has occurred.
  - When C flag = 1: Indicates no carry has occurred.

Figure 4.25 is an example of execution of the software ADD. When the input arguments are set as shown, the results of addition are set in ER0 and ER1.

### 4.8.2 Cautions for Use

Since the results of addition are set in the register used to set the summand, the summand is destroyed when ADD is executed. When you will still require the summand after executing ADD, save the summand elsewhere in memory beforehand.

### 4.8.3 Description of Data Memory

No data memory is used by ADD.

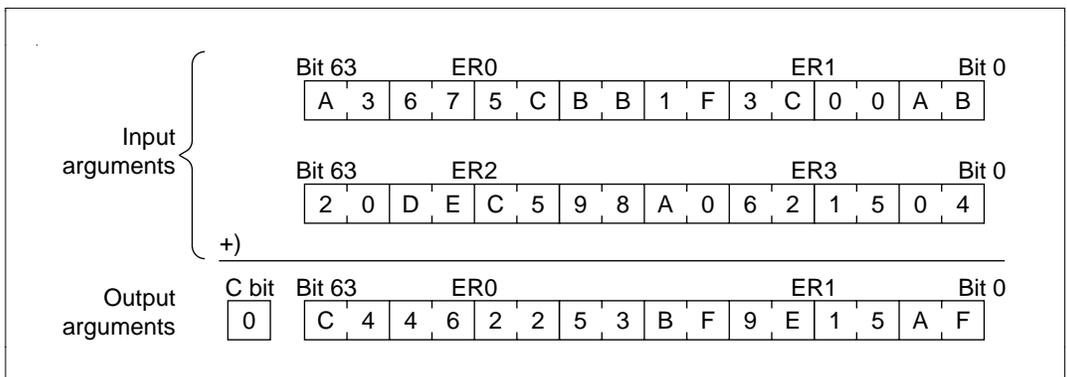


Figure 4.25 Executing ADD

#### 4.8.4 Examples of Use

After setting the summand and addend, does a subroutine call to ADD.

**Table 4.14 Block Transfer Example (ADD)**

Label	Instruction	Action
WORK 1	.RES. L 1	Reserves the data memory area that sets the top 32-bits of the summand (signed 64 bits) in the user program.
WORK 2	.RES. L 1	Reserves the data memory area that sets the bottom 32-bits of the summand (signed 64 bits) in the user program.
WORK 3	.RES. L 1	Reserves the data memory area that sets the top 32-bits of the addend (signed 64 bits) in the user program.
WORK 4	.RES. L 1	Reserves the data memory area that sets the bottom 32-bits of the addend (signed 64 bits) in the user program.
	MOV. L @WORK1,ER0	Set as the input argument the top 32-bits of the summand set in the user program.
	MOV. L @WORK2,ER1	Set as the input argument the bottom 32-bits of the summand set in the user program.
	MOV. L @WORK3,ER2	Set as the input argument the top 32-bits of the addend set in the user program.
	MOV. L @WORK4,ER3	Set as the input argument the bottom 32-bits of the addend set in the user program.
	⋮	Subroutine call to ADD.
	JSR @ADD	
	BCS OVER	When carrying occurs, the routine branches to the processing routine for carrying.
OVER	Processing routine for carrying over	

#### 4.8.5 Principles of Operation

- Bits 0–31 are added using the ADD.L instruction.
- Bits 32–63 are added in 1-byte units from the bottom using the addition instruction with carrying (ADDX.B), which can handle carrying. Since bits 48–55 are on the extended register, the addition instruction with carry is transferred into a usable general register and addition is then performed.

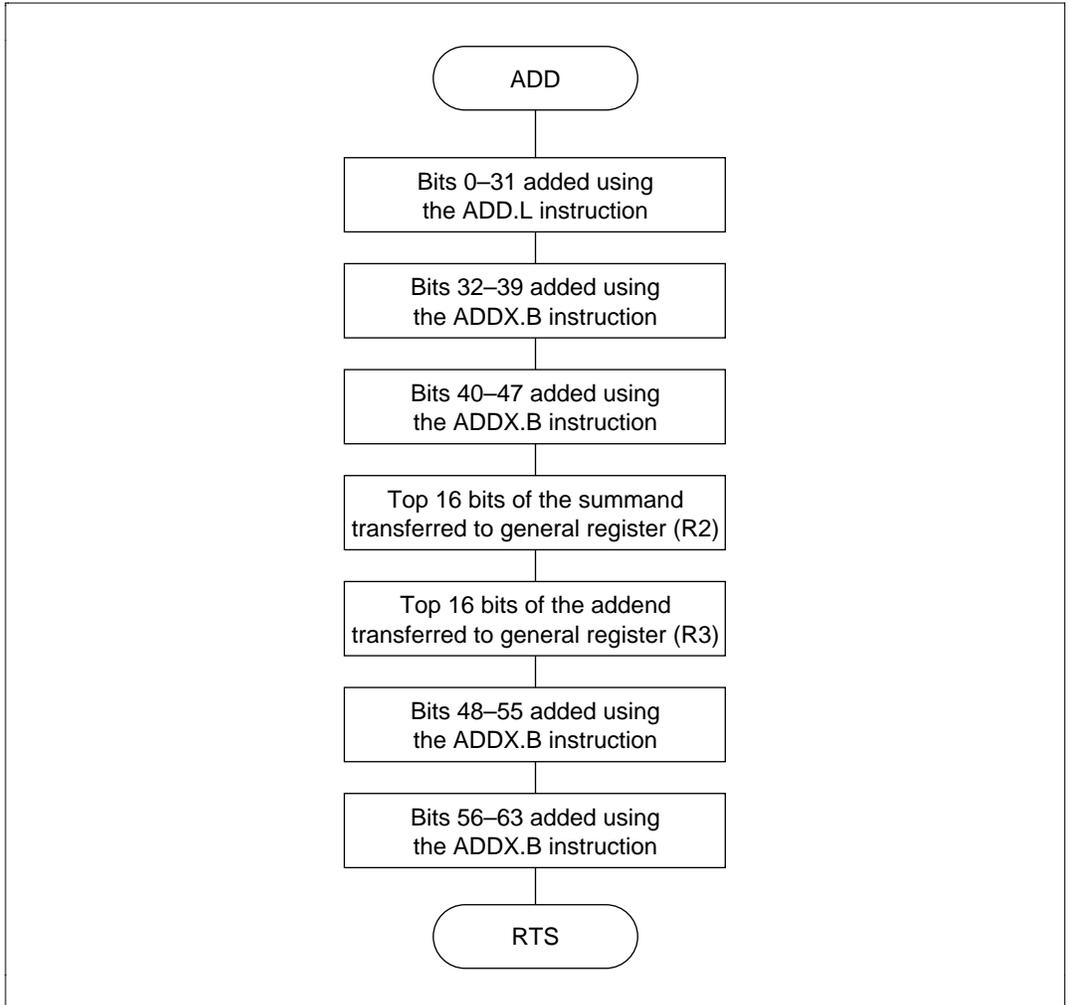


Figure 4.26 ADD Flowchart

## 4.8.6 Program Listing

## 4.9 64-Bit Binary Subtraction

MCU: H8/300H Series

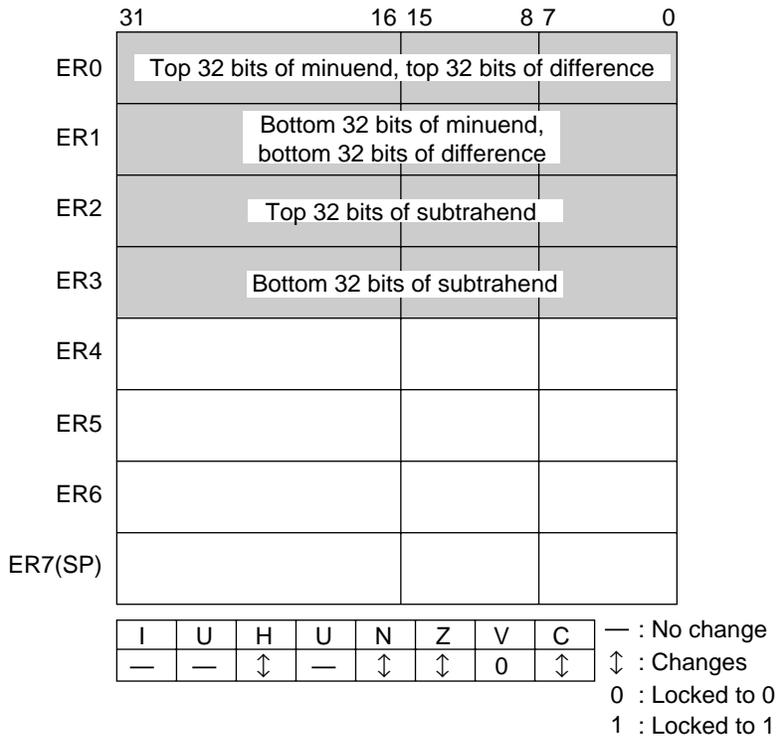
**Label Name:** SUB

**Functions Used:** SUB.L Instruction

**Function:** Does binary subtraction in the format: minuend (signed 64 bits) – subtrahend (signed 64 bits) = difference (signed 64 bits).

**Table 4.15 SUB Arguments**

	<b>Contents</b>	<b>Storage Location</b>	<b>Data Length (Bytes)</b>
Input	Top 32 bits of minuend (signed 64 bits)	ER0	4
	Bottom 32 bits of minuend (signed 64 bits)	ER1	4
	Top 32 bits of subtrahend (signed 64 bits)	ER2	4
	Bottom 32 bits of subtrahend (signed 64 bits)	ER3	4
Output	Top 32 bits of difference (signed 64 bits)	ER0	4
	Bottom 32 bits of difference (signed 64 bits)	ER1	4
	Existence of carrying	C flag (CCR)	1



**Figure 4.27 Changes in Internal Registers and Flag Changes for SUB**

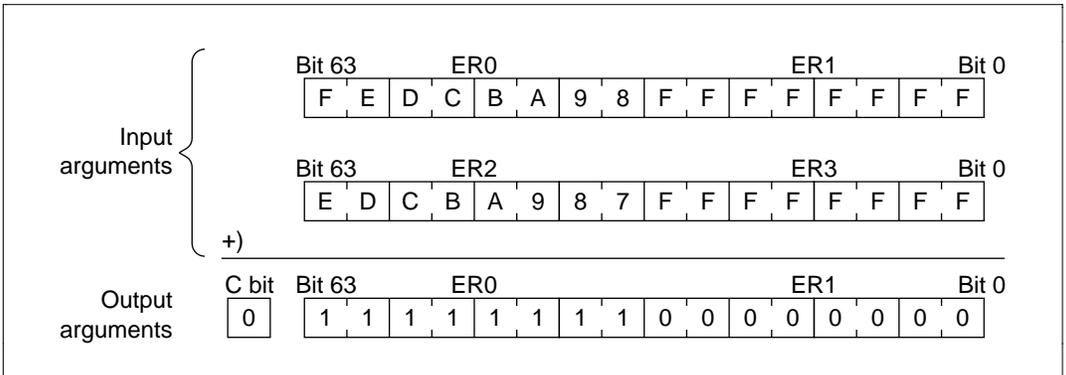
Program memory (bytes)
18
Data memory (bytes)
0
Stack (bytes)
0
Number of states
26
Re-entrant
Yes
Relocation
Yes
Interrupts during execution
Yes

**Figure 4.28 Programming Specifications**

### 4.9.1 Description of Functions

Arguments are as follows:

- ER0: Sets the top 32-bits of minuend (signed 64 bits) as an input argument. Sets the top 32 bits of the difference (signed 64 bits) as an output argument.
- ER1: Sets the bottom 32-bits of the minuend (signed 64 bits) as an input argument. Sets the bottom 32 bits of the difference (signed 64 bits) as an output argument.
- ER2: Sets the top 32-bits of the subtrahend (signed 64 bits) as an input argument.
- ER3: Sets the bottom 32-bits of the subtrahend (signed 64 bits) as an input argument.
- C flag (CCR): Indicates whether a borrow has occurred after execution of SUB.
  - When C flag = 1: Indicates a borrow has occurred.
  - When C flag = 0: Indicates no borrow has occurred.
- Figure 4.29 is an example of execution of the software SUB. When the input arguments are set as shown, the results of subtraction are set in ER0 and ER1.



**Figure 4.29 Executing SUB**

### 4.9.2 Cautions for Use

Since the results of subtraction are set in the register used to set the minuend, the minuend is destroyed after SUB is executed. When you will still require the minuend after executing SUB, save the minuend elsewhere in memory beforehand.

### 4.9.3 Description of Data Memory

No data memory is used by SUB.

#### 4.9.4 Examples of Use

After setting the subtrahend and minuend, does a subroutine call to SUB.

**Table 4.16 Block Transfer Example (SUB)**

Label	Instruction	Action
WORK 1	.RES. L 1	Reserves the data memory area that sets the top 32-bits of the minuend (signed 64 bits) in the user program.
WORK 2	.RES. L 1	Reserves the data memory area that sets the bottom 32-bits of the minuend (signed 64 bits) in the user program.
WORK 3	.RES. L 1	Reserves the data memory area that sets the top 32-bits of the subtrahend (signed 64 bits) in the user program.
WORK 4	.RES. L 1	Reserves the data memory area that sets the bottom 32-bits of the subtrahend (signed 64 bits) in the user program.
	MOV. L @WORK1,ER0	Set as the input argument the top 32-bits of the minuend set in the user program.
	MOV. L @WORK2,ER1	Set as the input argument the bottom 32-bits of the minuend set in the user program.
	MOV. L @WORK3,ER2	Set as the input argument the top 32-bits of the subtrahend set in the user program.
	MOV. L @WORK4,ER3	Set as the input argument the bottom 32-bits of the subtrahend set in the user program.
	⋮	Subroutine call to SUB.
	JSR @SUB	
	BCS OVER	When borrowing occurs, the routine branches to the processing routine for borrowing.
OVER	Processing routine for borrowing	

#### 4.9.5 Principles of operation

- Bits 0–31 are subtracted using the SUB.L instruction.
- Bits 32–63 are subtracted in 1-byte units from the bottom using the subtraction instruction with carrying (SUBX.B), which can handle borrowing. Since bits 48–55 are in the extended register, the subtraction instruction with borrow is transferred into the usable general register and subtraction is then performed.

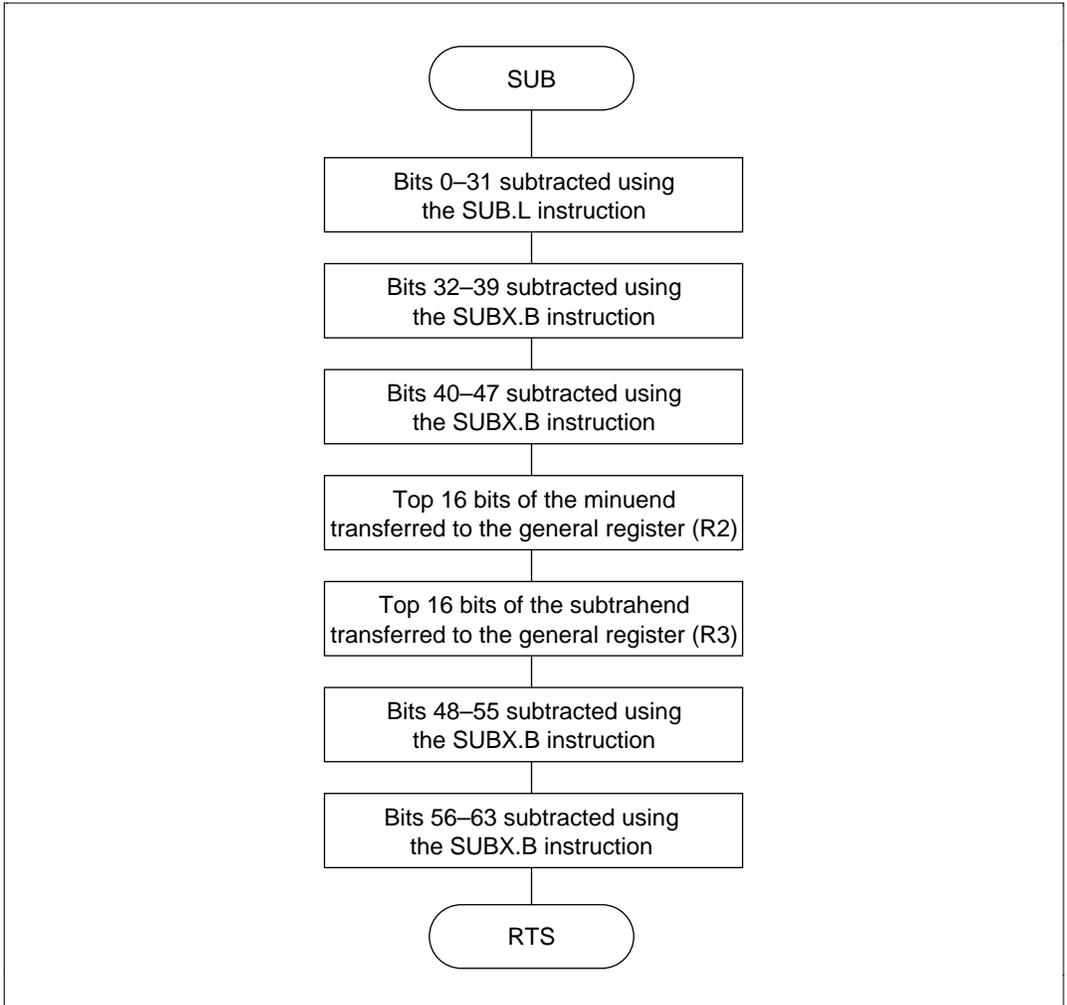


Figure 4.30 SUB Flowchart

## 4.9.6 Program Listing

## 4.10 Unsigned 32-Bit Binary Multiplication

MCU: H8/300H Series

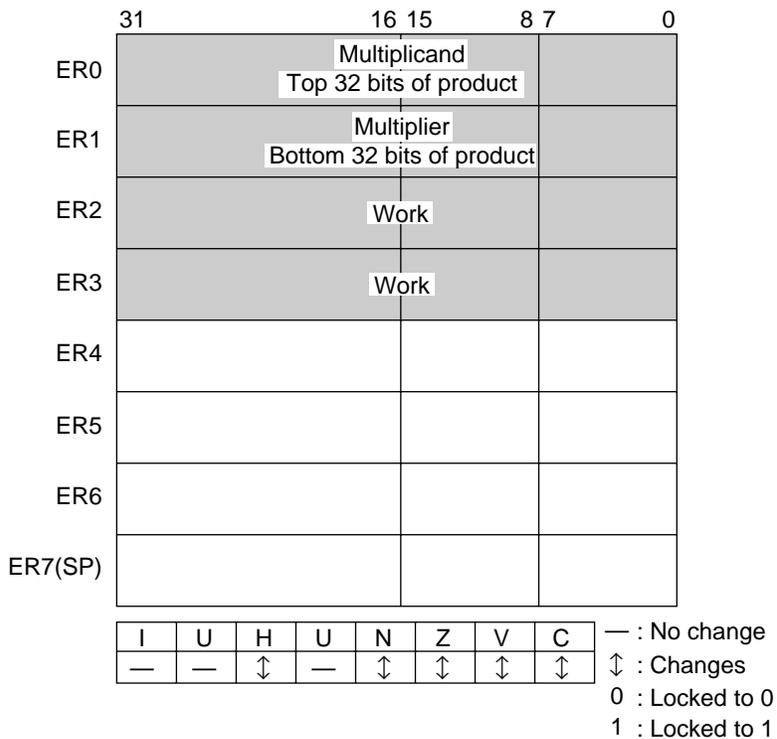
**Label Name:** MUL

**Functions Used:** MULXU.W Instruction

**Function:** Does multiplication in the format: Multiplicand (unsigned 32 bits) × multiplier (unsigned 32 bits) = product (unsigned 64 bits).

**Table 4.17 MUL Arguments**

	<b>Contents</b>	<b>Storage Location</b>	<b>Data Length (Bytes)</b>
Input	Multiplicand (unsigned 32 bits)	ER0	4
	Multiplier (unsigned 32 bits)	ER1	4
Output	Top 32 bits of product (unsigned 64 bits)	ER0	4
	Bottom 32 bits of product (unsigned 64 bits)	ER1	4



**Figure 4.31 Changes in Internal Registers and Flag Changes for MUL**

Program memory (bytes)
34
Data memory (bytes)
0
Stack (bytes)
0
Number of states
126
Re-entrant
Yes
Relocation
Yes
Interrupts during execution
Yes

Caution: The number of states in the programming specifications is the value when calculating as H'FFFFFFFF x H'FFFFFFFF.

**Figure 4.32 Programming Specifications**

### 4.10.1 Description of functions

Arguments are as follows:

- ER0: Sets the multiplicand (unsigned 32 bits) as an input argument. Sets the top 32 bits of the product (unsigned 64 bits) as an output argument.
- ER1: Sets the multiplier (unsigned 32 bits) as an input argument. Sets the bottom 32 bits of the product (unsigned 64 bits) as an output argument.
- Figure 4.33 is an example of execution of the software MUL. When the input arguments are set as shown, the product is set in ER0 and ER1.

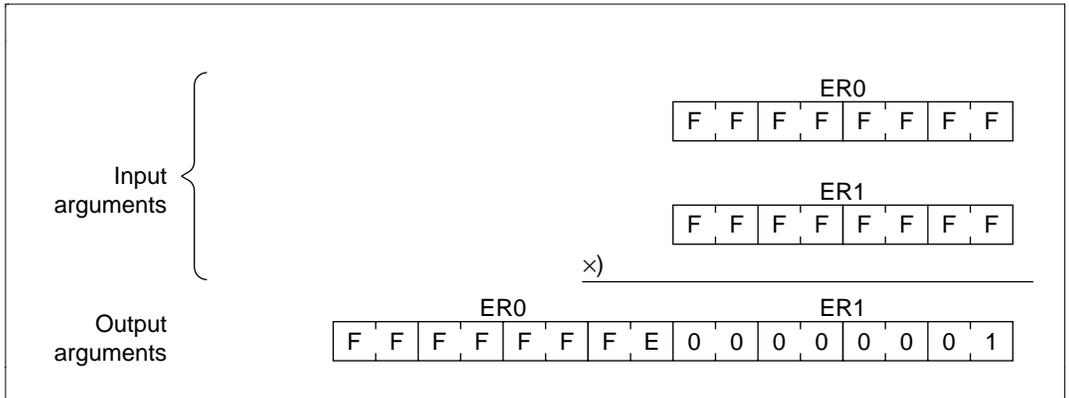


Figure 4.33 Executing MUL

### 4.10.2 Cautions for Use

Since the product is set in the register used to set the multiplicand and multiplier, the multiplicand and multiplier are destroyed after MUL is executed. When you will still require the multiplicand and multiplier after executing MUL, save them elsewhere in memory beforehand.

### 4.10.3 Description of Data Memory

No data memory is used by MUL.

#### 4.10.4 Examples of Use

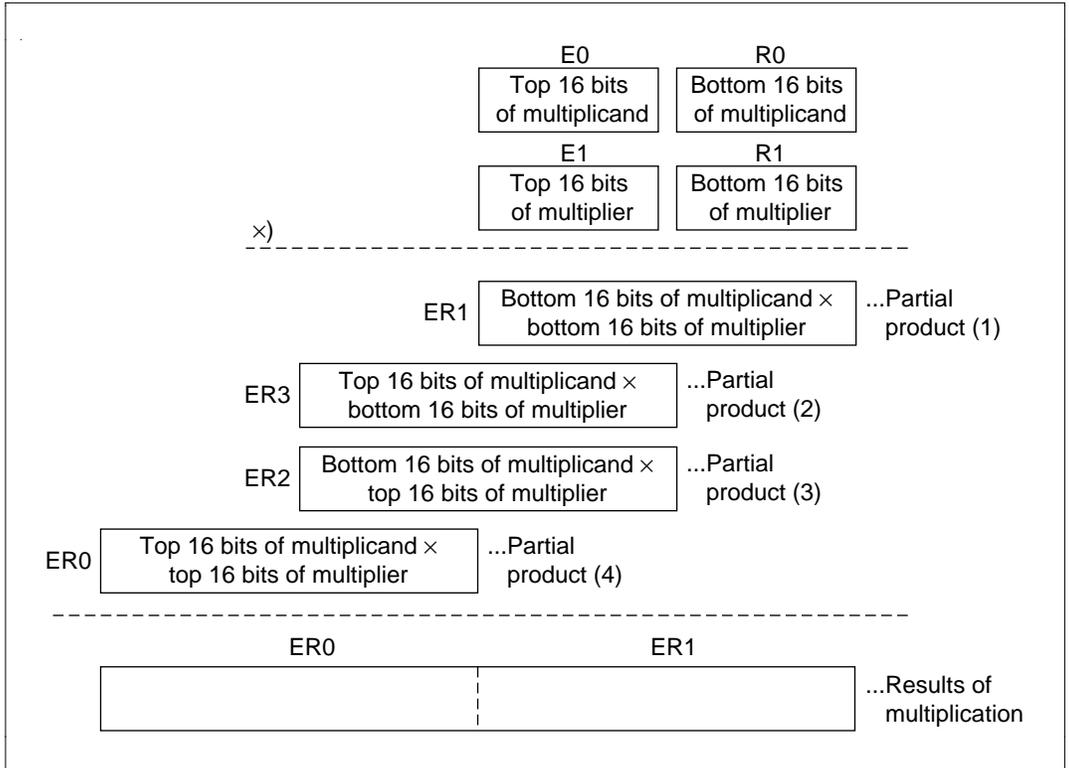
After setting the multiplicand and multiplier, do a subroutine call to MUL.

**Table 4.18 Block Transfer Example (MUL)**

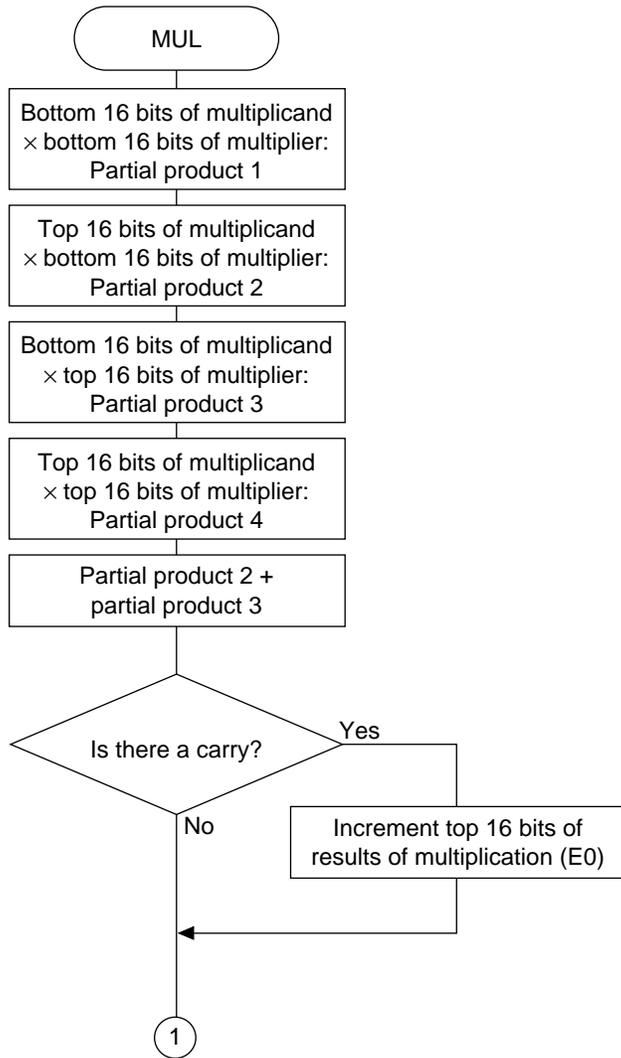
Label	Instruction	Action
WORK 1	.RES. L 1	Reserves the data memory area that sets the multiplicand (unsigned 32 bits) in the user program.
WORK 2	.RES. L 1	Reserves the data memory area that sets the multiplier (unsigned 32 bits) in the user program.
	MOV. L @WORK1,ER0	Sets as the input argument the 32-bit binary multiplicand set in the user program.
	MOV. L @WORK2,ER1	Sets as the input argument the 32-bit binary multiplier set in the user program.
	JSR @MUL	Subroutine call to MUL.

### 4.10.5 Principles of Operation

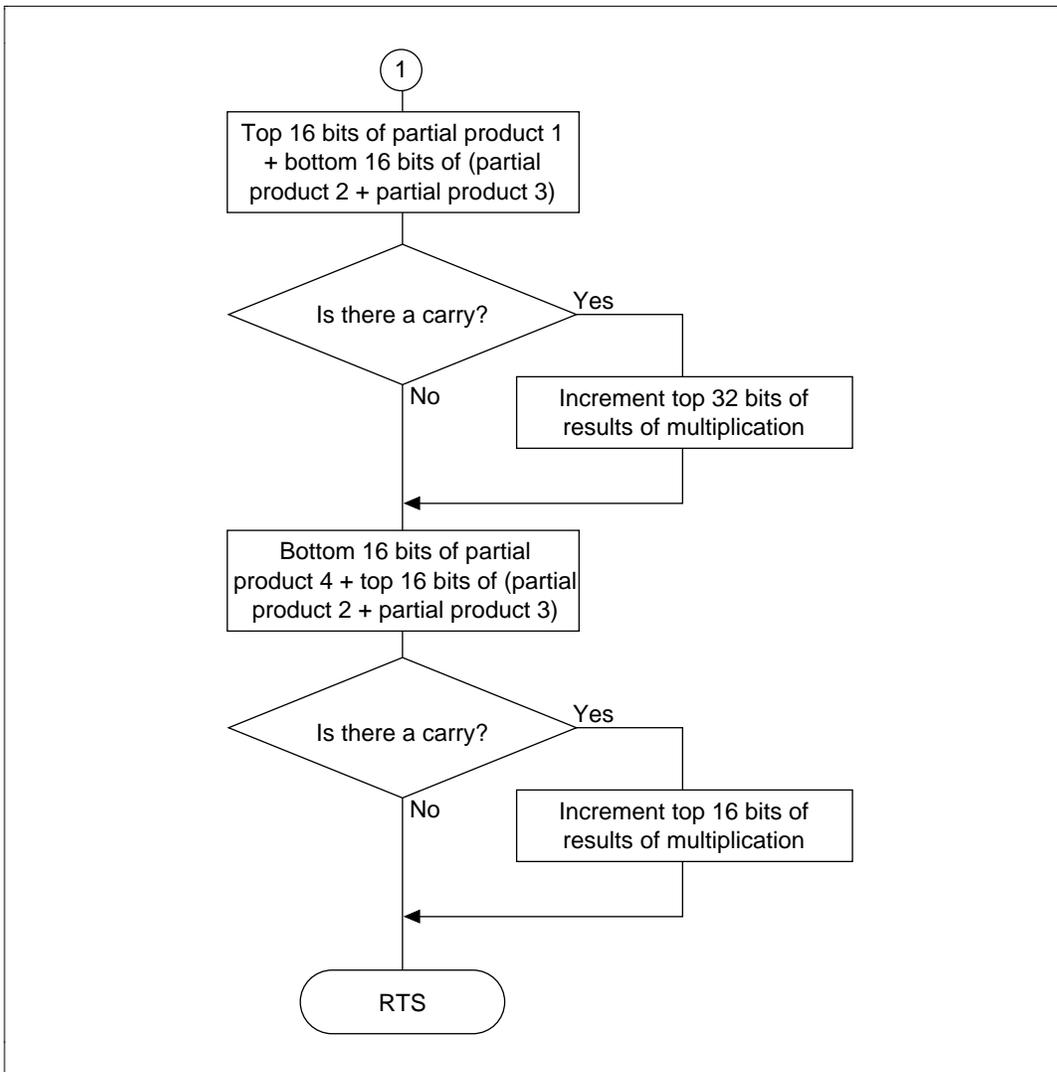
- The partial products of two 16-bit binary numbers are found using the multiplication instruction (MULXU.W) and the results of multiplication are then integrated to perform 32-bit binary multiplication, as shown in figure 4.34.



**Figure 4.34 Multiplication**



**Figure 4.35 MUL Flowchart**



**Figure 4.35 MUL Flowchart (cont)**

## 4.10.6 Program Listing

## 4.11 Unsigned 32-Bit Binary Division

MCU: H8/300H Series

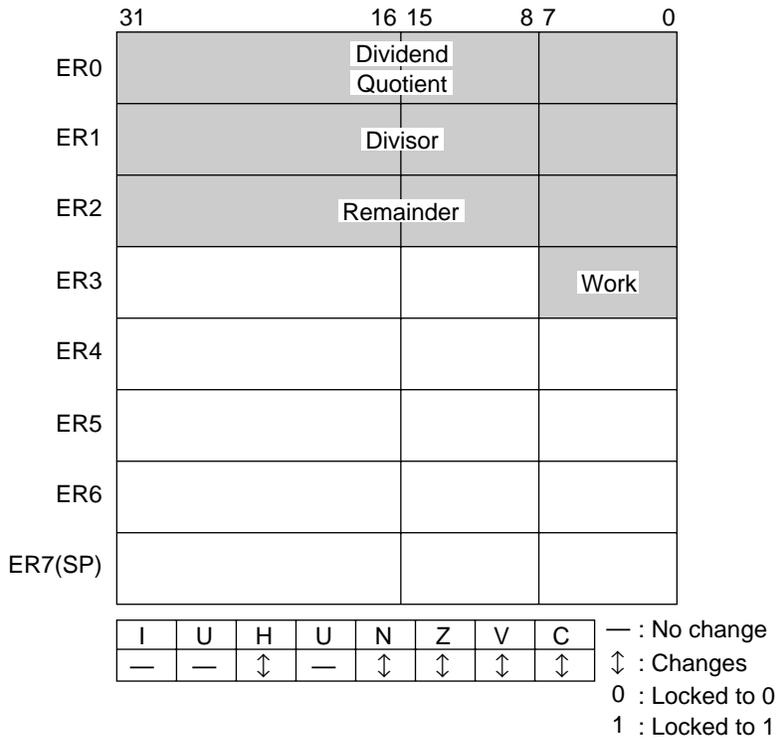
**Label Name:** DIV

**Functions Used:** SHLL.L Instruction, ROTXL.L Instruction

**Function:** Does division in the format: Dividend (unsigned 32 bits) / divisor (unsigned 32 bits) = quotient (unsigned 32 bits) ... remainder (unsigned 32 bits). Dividing by 0 sets the Z flag.

**Table 4.19 DIV Arguments**

	<b>Contents</b>	<b>Storage Location</b>	<b>Data Length (Bytes)</b>
Input	Dividend (unsigned 32 bits)	ER0	4
	Divisor (unsigned 32 bits)	ER1	4
Output	Quotient (unsigned 32 bits)	ER0	4
	Remainder (unsigned 32 bits)	ER2	4
	Presence of error (division by 0) (Yes, Z = 0; No, Z = 1)	Z flag (CCR)	1



**Figure 4.36 Changes in Internal Registers and Flag Changes for DIV**

Program memory (bytes)
30
Data memory (bytes)
0
Stack (bytes)
0
Number of states
762
Re-entrant
Yes
Relocation
Yes
Interrupts during execution
Yes

Caution: The number of states in the programming specifications is the value when calculating as  $H'FFFFFFF / H'1$ .

**Figure 4.37 Programming Specifications**

### 4.11.1 Description of Functions

Arguments are as follows:

- ER0: Sets the dividend (unsigned 32 bits) as an input argument. Sets the quotient (unsigned 32 bits) as an output argument.
- ER1: Sets the divisor (unsigned 32 bits) as an input argument.
- ER2: Sets the remainder (unsigned 32 bits) as an output argument.
- Z Flag (CCR): Indicates whether there are any errors (division by 0) after execution of DIV.
  - When Z flag = 1: Indicates that there is an error in the division executed.
  - When Z flag = 0: Indicates that there is no error in the division executed.

Figure 4.38 is an example of execution of the software DIV. When the input arguments are set as shown, the quotient is set in ER0 and the remainder is set in ER1.

With the software DIV, the first thing done is to determine if the divisor is 0 or nonzero; if it is 0, DIV ends.

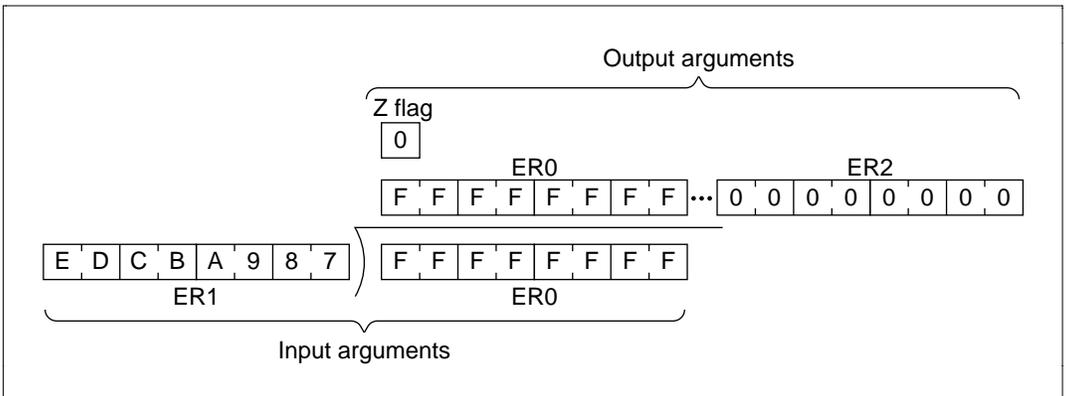


Figure 4.38 Executing DIV

### 4.11.2 Cautions for Use

Since the quotient is set in ER0, the dividend is destroyed after DIV is executed. When you will still require the dividend after executing DIV, save it elsewhere in memory beforehand.

### 4.11.3 Description of Data Memory

No data memory is used by DIV.

#### 4.11.4 Examples of Use

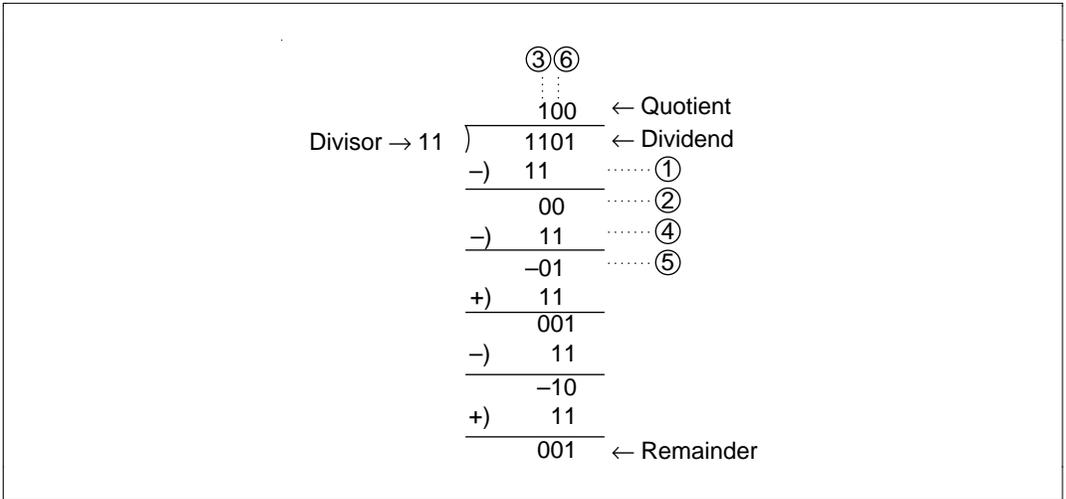
After setting the dividend and divisor, do a subroutine call to DIV.

**Table 4.20 Block Transfer Example (DIV)**

Label	Instruction	Action
WORK 1	.RES. L 1	Reserves the data memory area that sets the dividend (unsigned 32 bits) in the user program.
WORK 2	.RES. L 1	Reserves the data memory area that sets the divisor (unsigned 32 bits) in the user program.
	MOV. L @WORK1,ER0	Sets as the input argument the dividend (unsigned 32 bits) set in the user program.
	MOV. L @WORK2,ER1	Sets as the input argument the divisor (unsigned 32 bits) set in the user program.
	JSR @DIV	Subroutine call to DIV.

### 4.11.5 Principles of Operation

- Binary division finds the quotient and remainder by repeatedly subtracting. In figure 4.39, H'0D is divided by H'03 as an example of the division operation.



**Figure 4.39 Division**

- Detailed description of the program:
  - i. Sets the number of shifts in the counter R3L (which indicates the number of shifts).
  - ii. The dividend is shifted 1 bit to the left and the MSB loaded in the C bit is set in the LSB of ER2 (which stores the remainder).
  - iii. The divisor is subtracted from ER2. When the result of subtraction is positive, the LSB of ER0 is set (1 to 2 to 3 in figure 4.39). When the results of subtraction is negative, the LSB of ER0 is cleared and the divisor is added to the results of subtraction, returning it to the state prior to subtraction. ((4) to (5) to (6) in figure 4.39).
  - iv. The shift counter set in step (i) is decremented.
  - v. Steps (ii) through (iv) are repeated until the shift counter reaches -1.

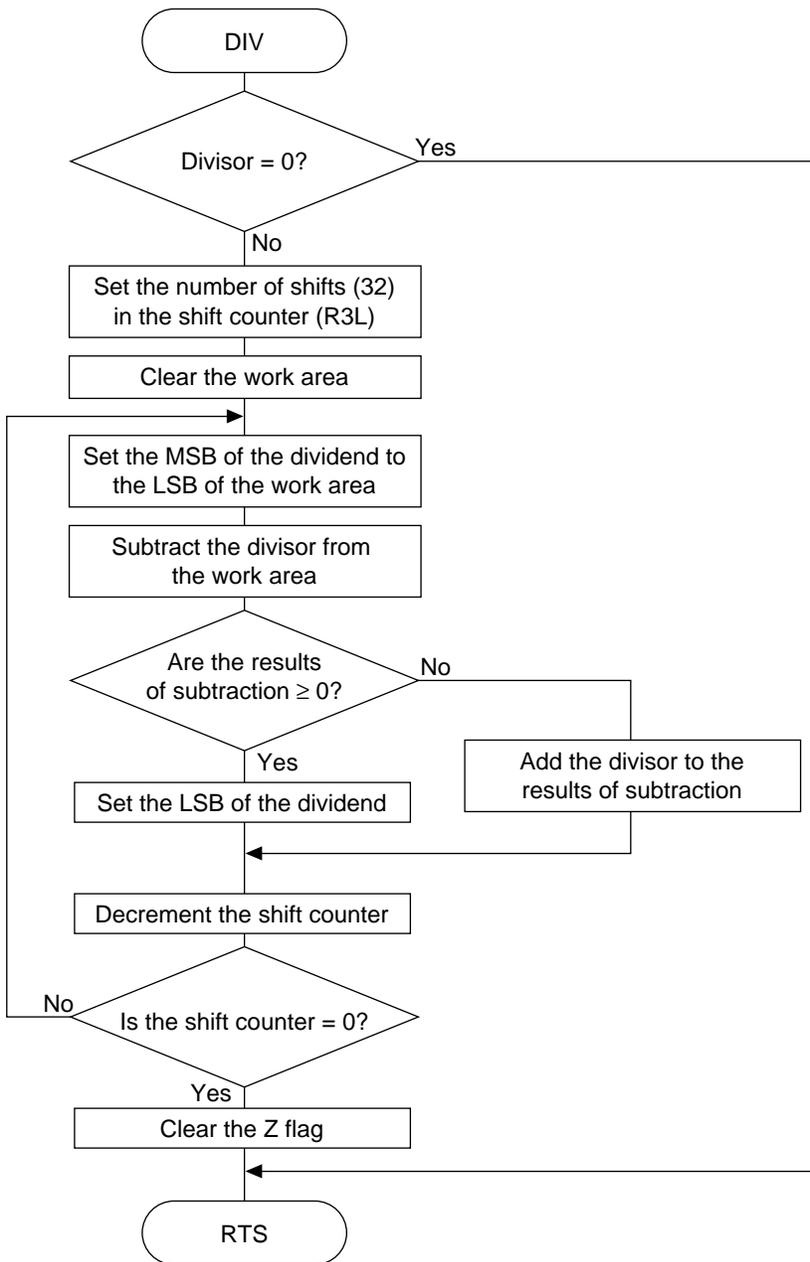


Figure 4.40 DIV Flowchart

## 4.11.6 Program Listing

## 4.12 Signed 16-Bit Binary Multiplication

MCU: H8/300H Series

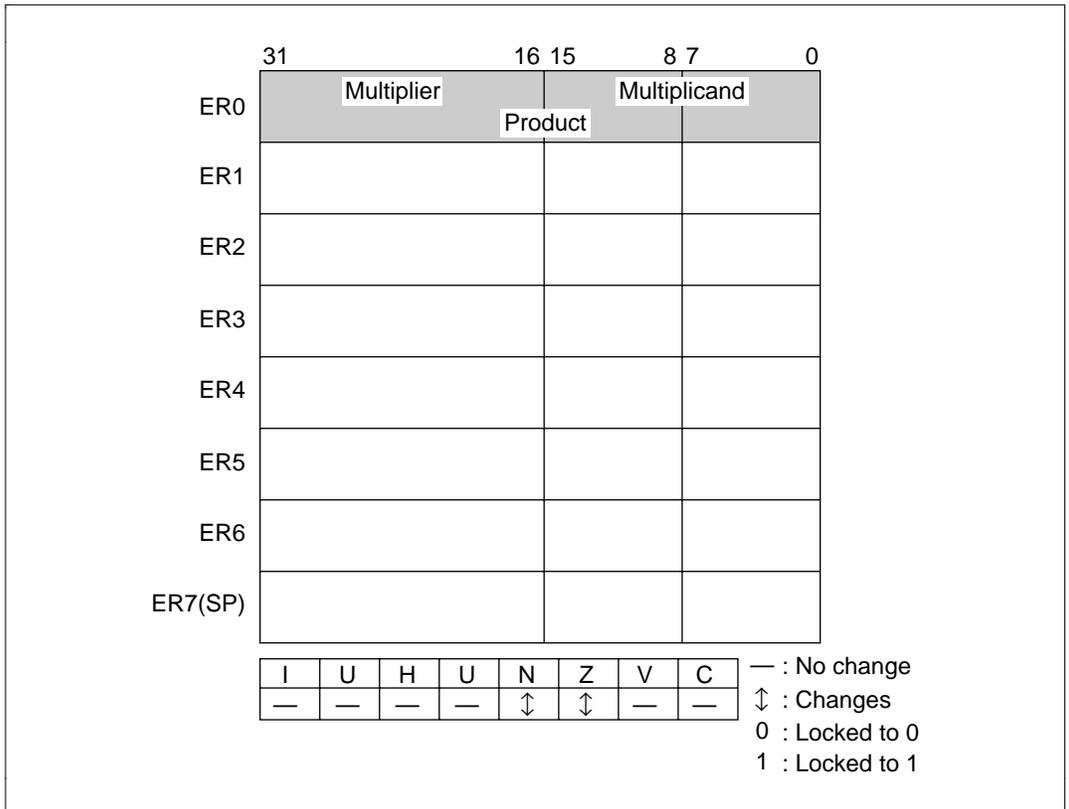
**Label Name:** MULXS

**Functions Used:** MULXS.W Instruction

**Function:** Does multiplication in the format: Multiplicand (signed 16 bits) × multiplier (signed 16 bits) = product (signed 32 bits).

**Table 4.21 MULXS Arguments**

	Contents	Storage Location	Data Length (Bytes)
Input	Multiplicand (signed 16 bits)	R0	2
	Multiplier (signed 16 bits)	E0	2
Output	Product (signed 32 bits)	ER0	4



**Figure 4.41 Changes in Internal Registers and Flag Changes for MULXS**

Program memory (bytes)
4
Data memory (bytes)
0
Stack (bytes)
0
Number of states
24
Re-entrant
Yes
Relocation
Yes
Interrupts during execution
Yes

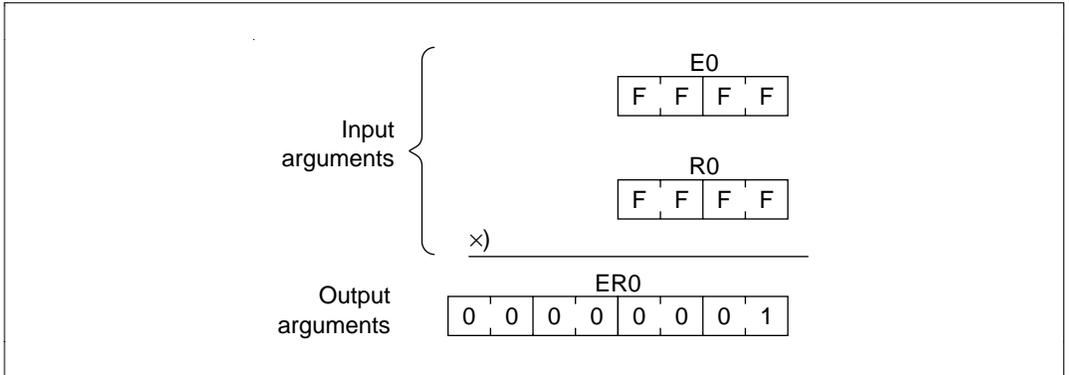
**Figure 4.42 Programming Specifications**

### 4.12.1 Description of Functions

Arguments are as follows:

- E0: Sets the multiplicand (signed 16 bits) as an input argument.
- R0: Sets the multiplier (signed 16 bits) as an input argument.
- ER0: Sets the product (signed 32 bits) as an output argument.

Figure 4.43 is an example of execution of the software MULXS.B When the input arguments are set as shown, the results of multiplication are set in ER0.



**Figure 4.43 Executing MULXS**

### 4.12.2 Cautions for Use

Since the results of multiplication are set in the register used to set the multiplicand and multiplier, the multiplicand and multiplier are destroyed after MULXS is executed. When you will still require the multiplicand and multiplier after executing MULXS, save them elsewhere in memory beforehand.

### 4.12.3 Description of Data Memory

No data memory is used by MULXS.

#### 4.12.4 Examples of Use

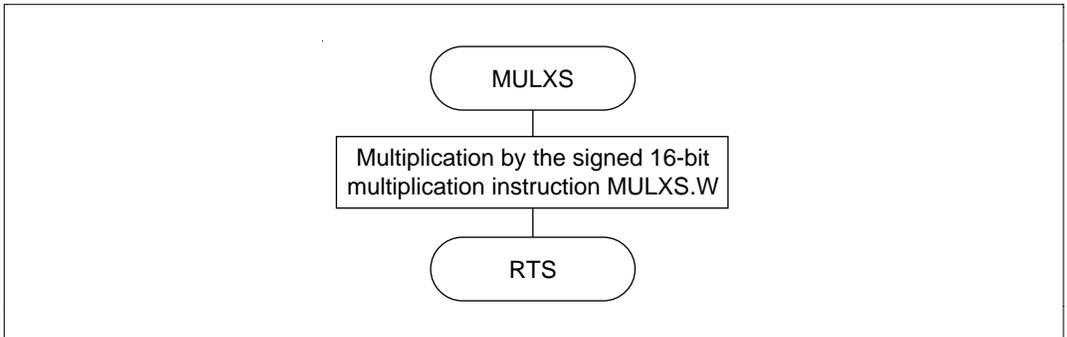
After setting the multiplicand and multiplier, do a subroutine call to MULXS.

**Table 4.22 Block Transfer Example (MULXS)**

Label	Instruction	Action
WORK 1	RES. W 1	Reserves the data memory area that sets the multiplicand (signed 16 bits) in the user program.
WORK 2	RES. W 1	Reserves the data memory area that sets the multiplier (signed 16 bits) in the user program.
	MOV. L @WORK1,R0	Sets as the input argument the 16-bit binary multiplicand set in the user program.
	MOV. L @WORK2,E0	Sets as the input argument the 16-bit binary multiplier set in the user program.
	JSR @MULXS	Subroutine call to MULXS.

#### 4.12.5 Principles of Operation

Use the signed 16-bit multiplication instruction MULXS.W.



**Figure 4.44 MULXS Flowchart**

## 4.12.6 Program Listing

## 4.13 Signed 32-Bit Binary Multiplication

MCU: H8/300H Series

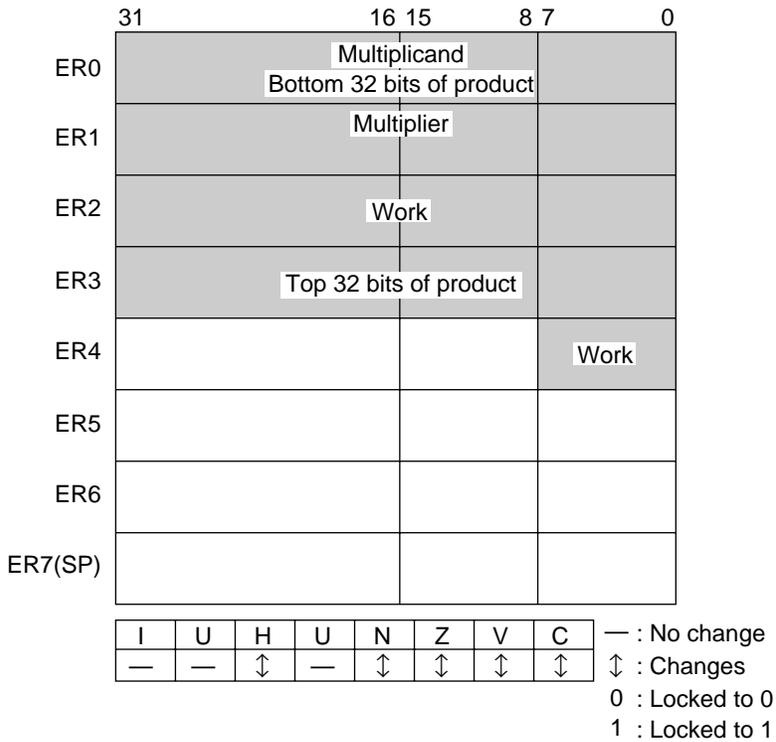
**Label Name:** MULS

**Functions Used:** MULXU.W Instruction

**Function:** Does binary multiplication in the format: Multiplicand (signed 32 bits) x multiplier (signed 32 bits) = product (signed 64 bits).

**Table 4.23 MULS Arguments**

	<b>Contents</b>	<b>Storage Location</b>	<b>Data Length (Bytes)</b>
Input	Multiplicand (signed 32 bits)	ER0	4
	Multiplier (signed 32 bits)	ER1	4
Output	Top 32 bits of product (signed 64 bits)	ER3	4
	Bottom 32 bits of product (signed 64 bits)	ER0	4



**Figure 4.45 Changes in Internal Registers and Flag Changes for Muls**

Program memory (bytes)
66
Data memory (bytes)
0
Stack (bytes)
0
Number of states
156
Re-entrant
Yes
Relocation
Yes
Interrupts during execution
Yes

Caution: The number of states in the programming specifications is the value when calculated as H'80000000 x H'7FFFFFFF.

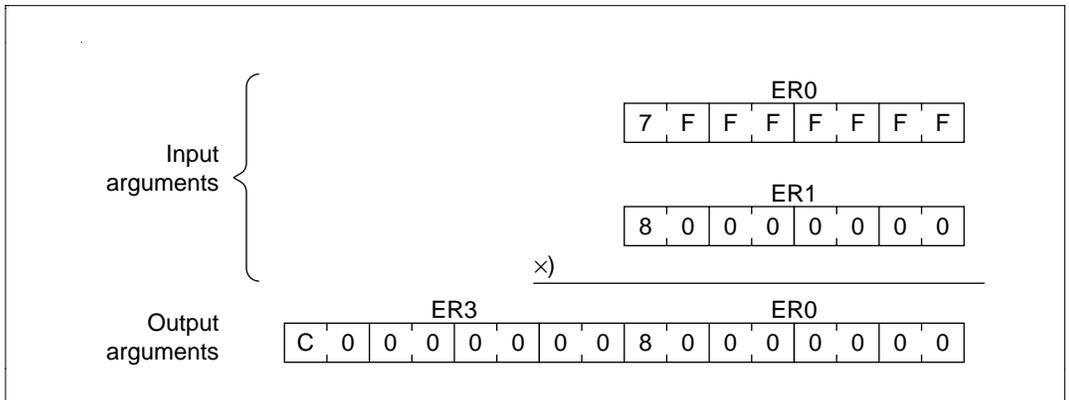
**Figure 4.46 Programming Specifications**

### 4.13.1 Description of Functions

Arguments are as follows:

- ER0: Sets the multiplicand (signed 32 bits) as an input argument. Sets the bottom 32 bits of the product (signed 64 bits) as an output argument.
- ER1: Sets the multiplier (signed 32 bits) as an input argument. Sets the bottom 32 bits of the product (signed 64 bits) as an output argument.
- Sets the top 32 bits of the product (signed 64 bits) as an output argument.

Figure 4.47 is an example of execution of the software MULS. When the input arguments are set as shown, the product is set in ER3 and ER0.



**Figure 4.47 Executing MULS**

### 4.13.2 Cautions for Use

Since the results of multiplication are set in the register used to set the multiplicand and multiplier, the multiplicand and multiplier are destroyed after MULS is executed. When you will still require the multiplicand and multiplier after executing MULS, save them elsewhere in memory beforehand.

### 4.13.3 Description of Data Memory

No data memory is used by MULS.

#### 4.13.4 Examples of Use

After setting the multiplicand and multiplier, do a subroutine call to MULS.

**Table 4.24 Block Transfer Example (MULS)**

<b>Label</b>	<b>Instruction</b>	<b>Action</b>				
WORK 1	.RES. L 1	Reserves the data memory area that sets the multiplicand (signed 32 bits) in the user program.				
WORK 2	.RES. L 1	Reserves the data memory area that sets the multiplier (signed 32 bits) in the user program.				
	MOV. L @WORK1,ER0	Sets as the input argument the multiplicand (signed 32 bits) set in the user program.				
	MOV. L @WORK2,ER1	Sets as the input argument the multiplier (signed 32 bits) set in the user program.				
	<table border="1"><tr><td></td><td>JSR</td><td>@MULS</td><td></td></tr></table>		JSR	@MULS		Subroutine call to MULS.
	JSR	@MULS				

### 4.13.5 Principles of Operation

- Negative multiplicands and multipliers are converted to positive.
- The product is found by taking the partial products ((1), (2), (3) and (4) in figure 4.48) and then accumulating the results of multiplication (figure 4.48 (5)). The partial products are found by using the signed multiplication instruction (MULXU.W) on two 16-bit binary numbers.
- The product is then converted to negative if the sign flag is 1, as shown in table 4.25.

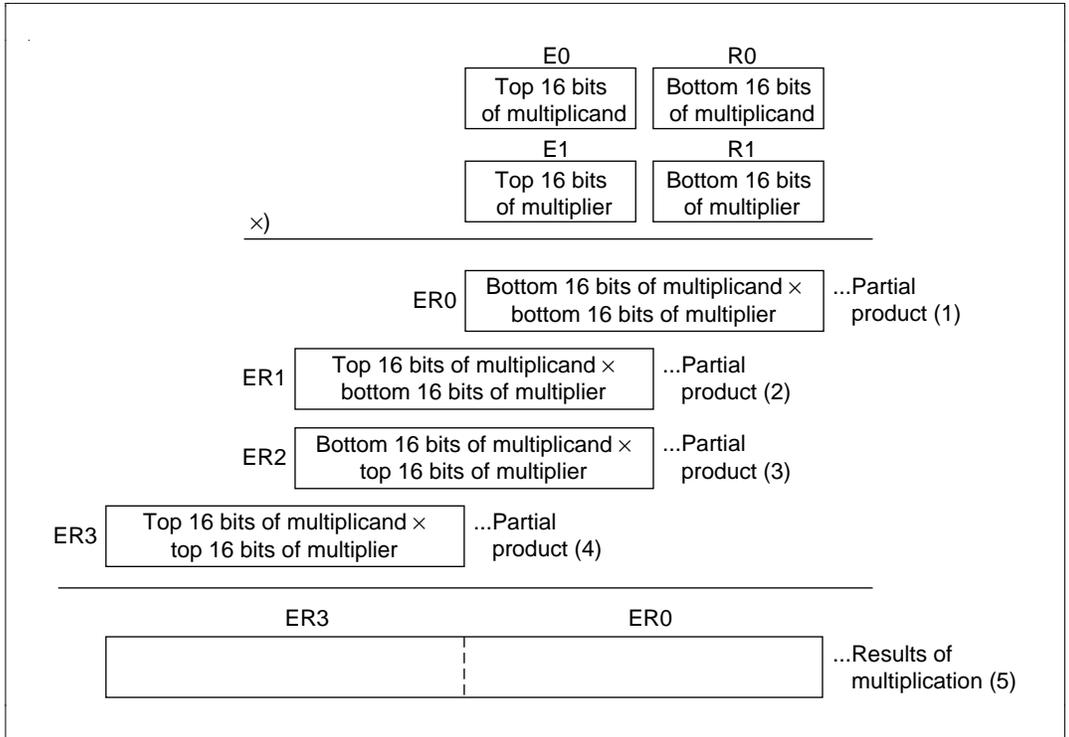
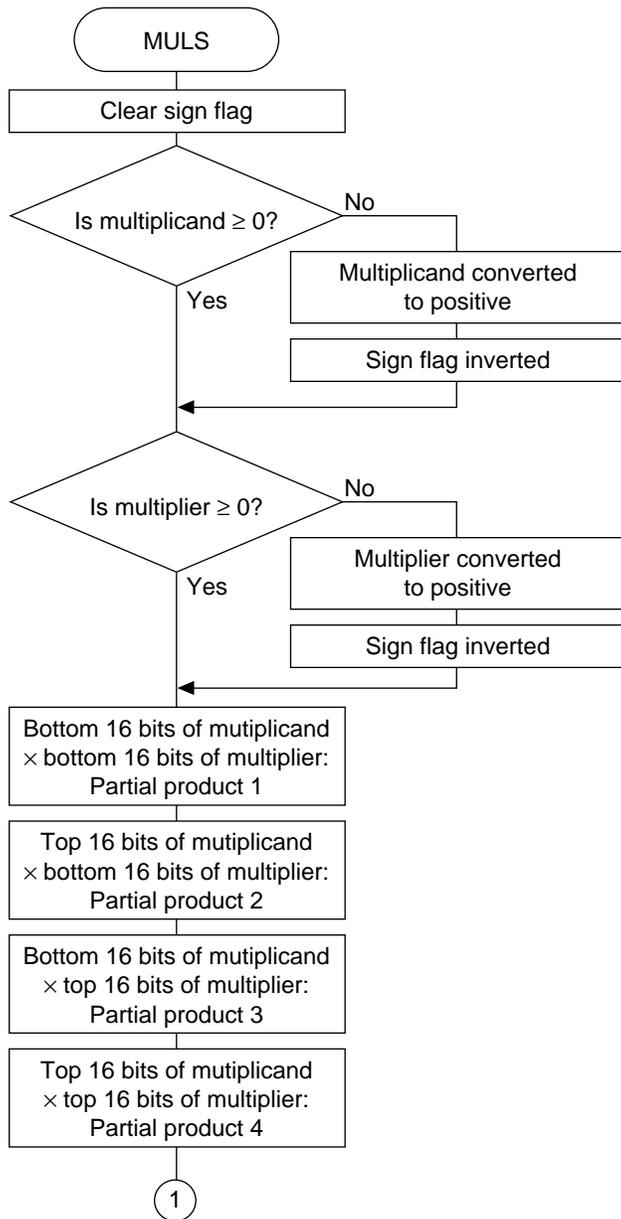


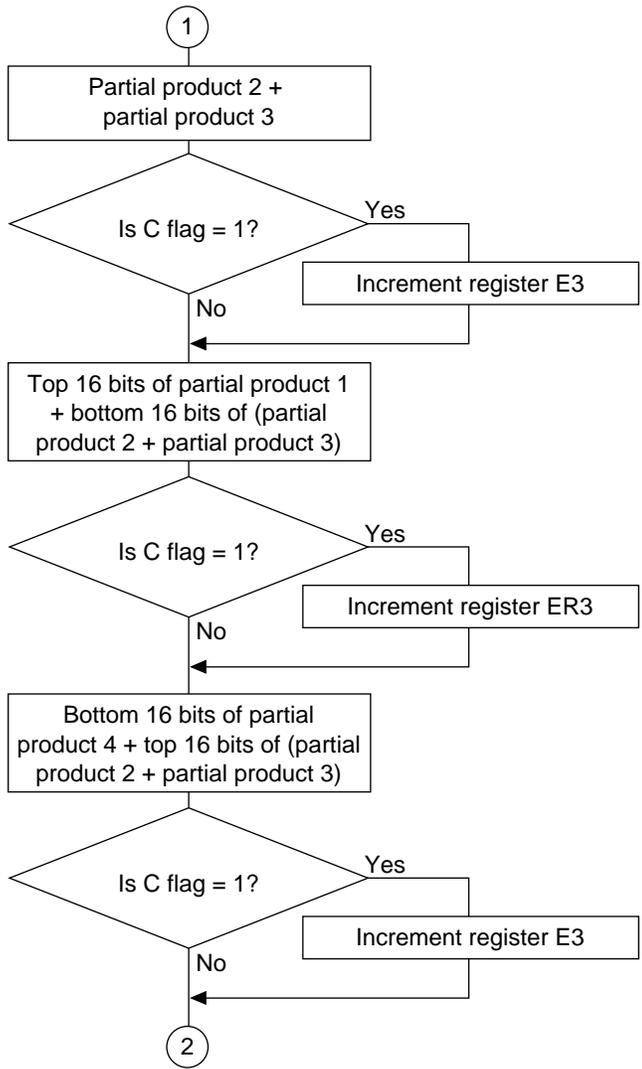
Figure 4.48 Multiplication

Table 4.25 Sign of Results of Multiplication and Sign Flag

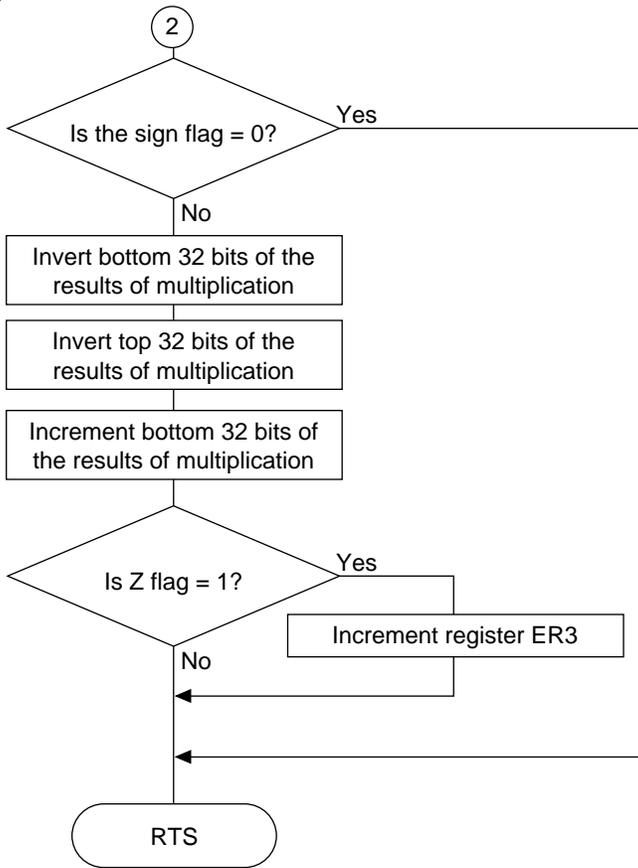
Multiplicand	Multiplier	Product	Sign Flag
Positive	Positive	Positive	0
	Negative	Negative	1
Negative	Positive	Negative	1
	Negative	Positive	0



**Figure 4.49 MULS Flowchart**



**Figure 4.49 MULS Flowchart (cont)**



**Figure 4.49 MULS Flowchart (cont)**

### 4.13.6 Program Listing

## 4.14 Signed 32-Bit Binary Division (16-Bit Divisor)

MCU: H8/300H Series

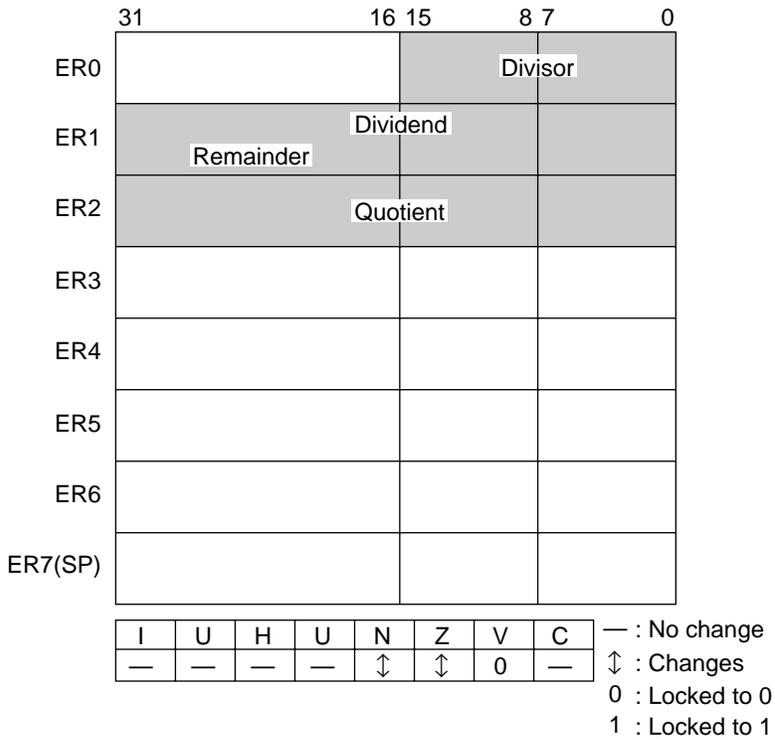
**Label Name:** DIVXS

**Functions Used:** DIVXS.W Instruction

**Function:** Does division in the format: Dividend (signed 32 bits) / divisor (signed 16 bits) = quotient (signed 32 bits) ... remainder (signed 16 bits).

**Table 4.26 DIVXS Arguments**

	<b>Contents</b>	<b>Storage Location</b>	<b>Data Length (Bytes)</b>
Input	Dividend (signed 32 bits)	ER1	4
	Divisor (signed 16 bits)	R0	2
Output	Quotient (signed 32 bits)	ER2	4
	Remainder (signed 16 bits)	E1	2
	Presence of error	Z flag (CCR)	1



**Figure 4.50 Changes in Internal Registers and Flag Changes for DIVXS**

Program memory (bytes)
26
Data memory (bytes)
0
Stack (bytes)
0
Number of states
76
Re-entrant
Yes
Relocation
Yes
Interrupts during execution
Yes

Caution: The number of states in the programming specifications is the value when calculated as H'80000000 / H7FFF'.

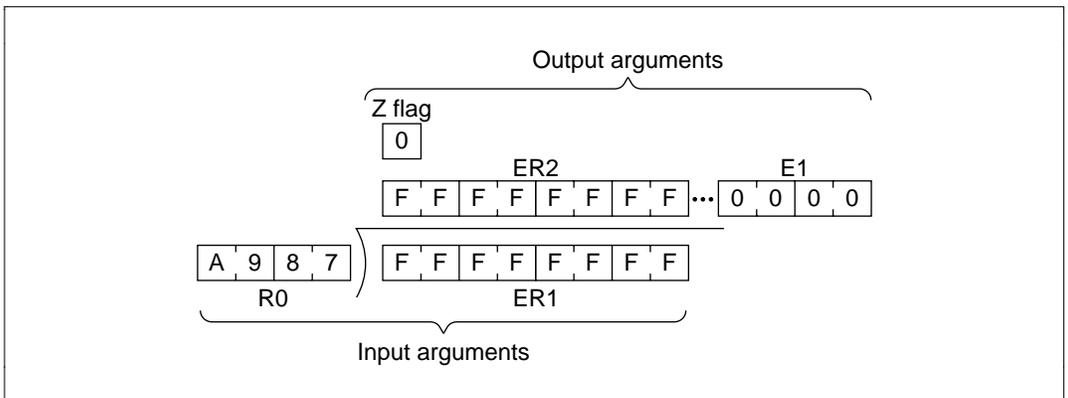
**Figure 4.51 Programming Specifications**

### 4.14.1 Description of Functions

Arguments are as follows

- R0: Sets the divisor (signed 16 bits) as an input argument.
- ER1: Sets the dividend (signed 32 bits) as an input argument.
- ER2: Sets the quotient (signed 32 bits) as an output argument.
- E1: Sets the remainder (signed 16 bits) as an output argument.
- Z Flag (CCR): Indicates whether there are any errors (division by 0) after execution of DIVXS.
  - When Z flag = 1: Indicates that there is an error in the division.
  - When Z flag = 0: Indicates that there is no error in the division.

Figure 4.52 is an example of execution of the software DIVXS. When the input arguments are set as shown, the quotient is set in ER2 and the remainder is set in E1.



**Figure 4.52 Executing DIVXS**

- With the software DIVXS, the first thing done is to determine if the divisor is 0 or nonzero; if it is 0, DIVXS ends.

### 4.14.2 Cautions for Use

Since the remainder is set in E1 and the bottom 16 bits of the quotient are set in R1, the dividend is destroyed after DIVXS is executed. When you will still require the dividend after executing DIVXS, save it elsewhere in memory beforehand.

### 4.14.3 Description of Data Memory

No data memory is used by DIVXS.

#### 4.14.4 Examples of Use

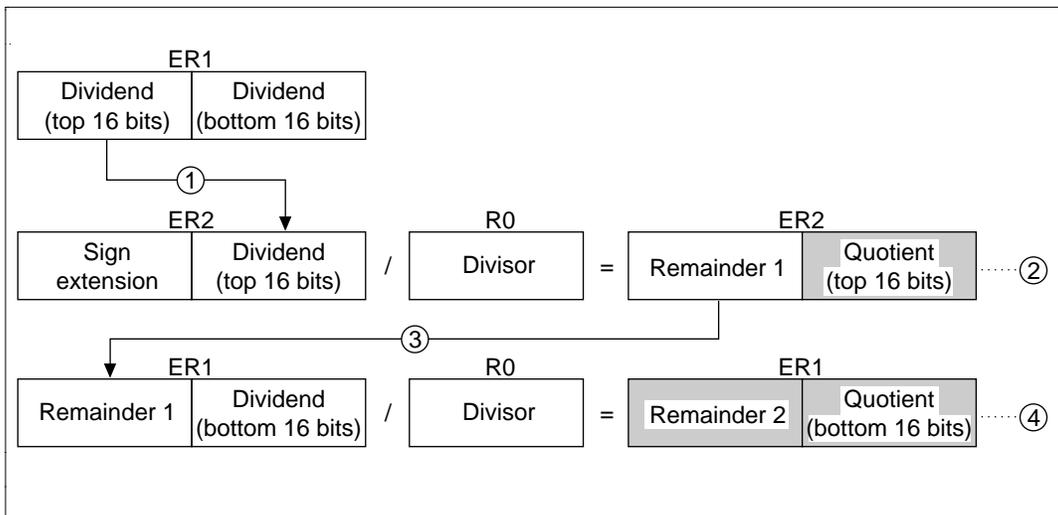
After setting the dividend and divisor as input arguments, do a subroutine call to DIVXS.

**Table 4.27 Block Transfer Example (DIVXS)**

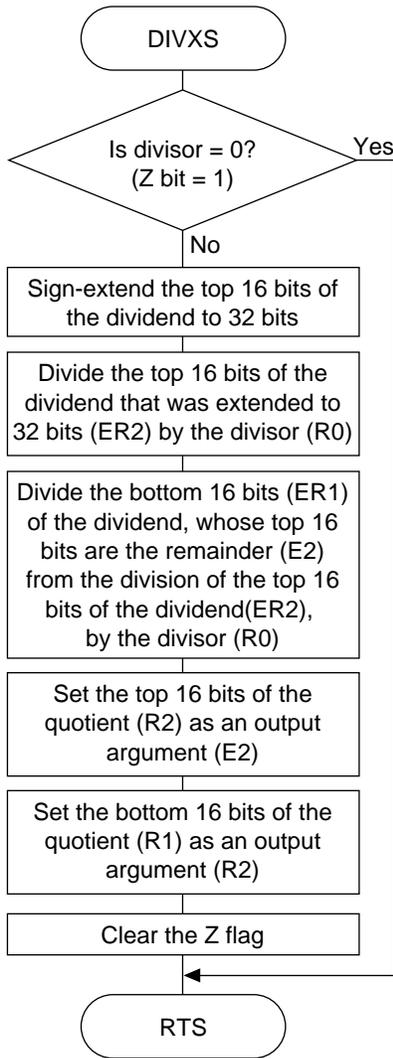
Label	Instruction	Action
WORK 1	.RES. L 1	Reserves the data memory area that sets the dividend (signed 32 bits) in the user program.
WORK 2	.RES. W 1	Reserves the data memory area that sets the divisor (signed 16 bits) in the user program.
	MOV. L @WORK1,ER1	Sets as the input argument the dividend set in the user program.
	MOV. L @WORK2,R0	Sets as the input argument the divisor set in the user program.
	JSR @DIVXS	Subroutine call to DIVXS.
	BEQ ERROR	When division by 0 is attempted, the program branches to the processing routine for errors.
	⋮	
ERROR	Processing routine for errors	

#### 4.14.5 Principles of Operation

- First, the program searches for zero-division errors. If there is such an error, the divisor is transferred to the register in which it is itself stored so that the resulting Z bit can be used to determine if the divisor is 0. If the Z bit is 1 (divisor = 0), DIVXS ends.
- When 32 bits is being divided by 16 bits using the signed division instruction (DIVXS.W), a quotient of 16 bits is found. The quotient will thus overflow when division such as H'FFFFFF/H'1 is performed. For that reason, a quotient of 32 bits is found using the following procedure.
  - The top 16 bits of the dividend are sent to R2 and sign-extended into 32 bits (figure 4.53 (1)).
  - The top 16 bits of the dividend are divided to obtain the top 16 bits of the quotient (ii) (figure 4.53 (2)).
  - The remainder of (ii) (remainder 1) is sent to R1 (figure 4.53 (3)).
  - Division is performed on the bottom 16 bits of the dividend to find the bottom 16 bits of the quotient and the remainder (remainder 2) (figure 4.53 (4)).



**Figure 4.53 Overflow Processing**



**Figure 4.54 DIVXS Flowchart**

### 4.14.6 Program Listing

## 4.15 Signed 32-Bit Binary Division (32-Bit Divisor)

MCU: H8/300H Series

**Label Name:** DIVS

**Functions Used:** SHLL.L Instruction, ROTL.L Instruction, NEG.L Instruction

**Function:** Does division in the format: Dividend (signed 32 bits) / divisor (signed 32 bits) = quotient (signed 32 bits) ... remainder (signed 32 bits).

**Table 4.28 DIVS Arguments**

	<b>Contents</b>	<b>Storage Location</b>	<b>Data Length (Bytes)</b>
Input	Dividend (signed 32 bits)	ER0	4
	Divisor (signed 32 bits)	ER1	4
Output	Quotient (signed 32 bits)	ER0	4
	Remainder (signed 32 bits)	ER2	4
	Presence of error	Z flag (CCR)	1

	31		16	15		8	7		0
ER0	Dividend				Quotient				
ER1	Divisor								
ER2	Remainder								
ER3					Work		Work		
ER4									
ER5									
ER6									
ER7(SP)									

I	U	H	U	N	Z	V	C
—	—	↕	—	↕	↕	↕	↕

— : No change  
 ↕ : Changes  
 0 : Locked to 0  
 1 : Locked to 1

**Figure 4.55 Changes in Internal Registers and Flag Changes for DIVS**

Program memory (bytes)
66
Data memory (bytes)
0
Stack (bytes)
0
Number of states
770
Re-entrant
Yes
Relocation
Yes
Interrupts during execution
Yes

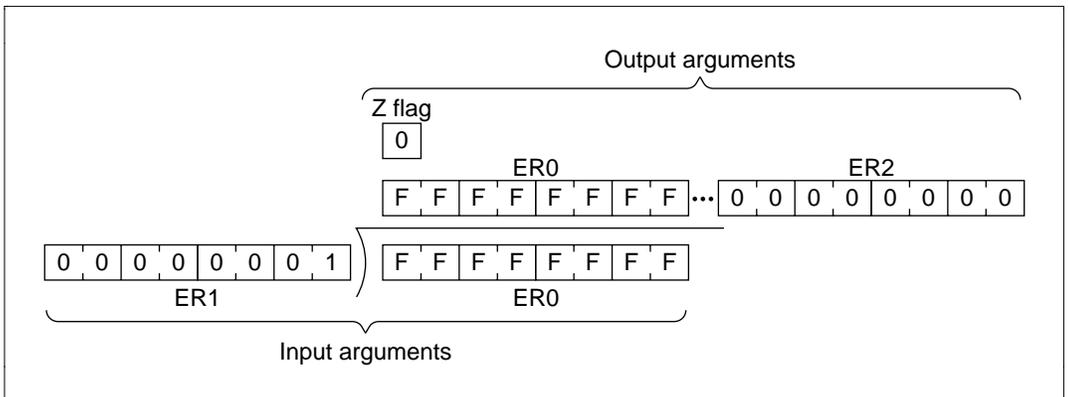
Caution: The number of states in the programming specifications is the value when calculated as H'80000000 / H7FFFFFFF.

**Figure 4.56 Programming Specifications**

### 4.15.1 Description of Functions

Arguments are as follows:

- ER0: Sets the dividend (unsigned 32 bits) as an input argument. Sets the quotient (unsigned 32 bits) as an output argument.
- ER1: Sets the divisor (unsigned 32 bits) as an input argument.
- ER2: Sets the remainder (unsigned 32 bits) as an output argument.
- Z Flag (CCR): Indicates whether there are any errors (division by 0) after execution of DIVS.
  - When Z flag = 1: Indicates that there is an error in the division.
  - When Z flag = 0: Indicates that there is no error in the division.
- Figure 4.57 is an example of execution of the software DIVS. When the input arguments are set as shown, the quotient is set in ER0 and the remainder is set in ER2.
- When the divisor is 0, DIVS ends immediately.



**Figure 4.57 Executing DIVS**

### 4.15.2 Cautions for Use

Since the quotient is set in ER0, the dividend is destroyed after DIVS is executed. When you will still require the dividend after executing DIVS, save it elsewhere in memory beforehand.

### 4.15.3 Description of Data Memory

No data memory is used by DIVS.

#### 4.15.4 Examples of Use

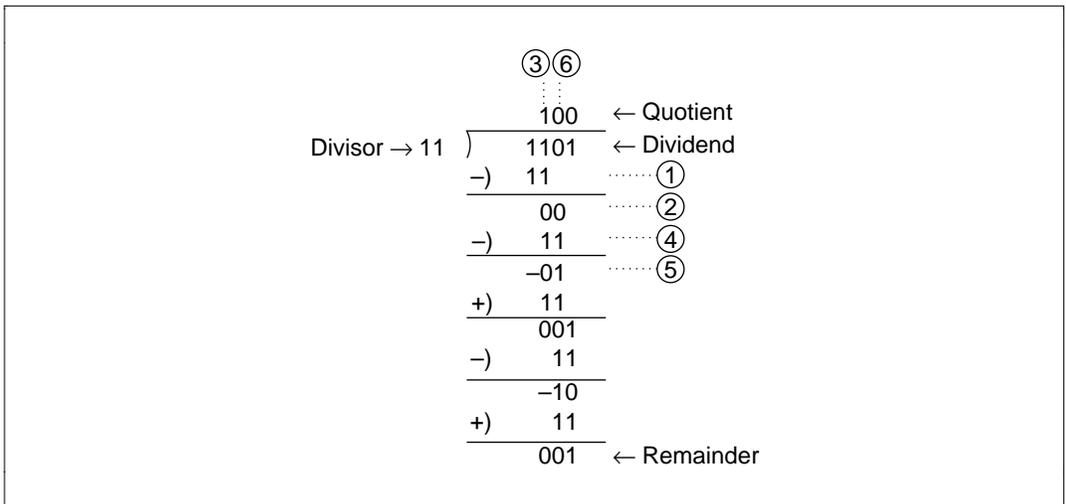
After setting the dividend and divisor, do a subroutine call to DIVS.

**Table 4.29 Block Transfer Example (DIVS)**

Label	Instruction	Action				
WORK 1	.RES. L 1	Reserves the data memory area that sets the dividend (signed 32 bits) in the user program.				
WORK 2	.RES. L 1	Reserves the data memory area that sets the divisor (signed 32 bits) in the user program.				
	MOV. L @WORK1,ER0	Sets as the input argument the dividend (signed 32 bits) set in the user program.				
	MOV. L @WORK2,ER1	Sets as the input argument the divisor (signed 32 bits) set in the user program.				
	<table border="1"><tr><td></td><td>JSR</td><td>@DIVS</td><td></td></tr></table>		JSR	@DIVS		Subroutine call to DIVS.
	JSR	@DIVS				

### 4.15.5 Principles of Operation

- Negative dividends and divisors are converted to positive.
- Division finds the quotient and remainder by repeatedly subtracting. In figure 4.58, H'0D is divided by H'03 as an example of the division operation.
  - i. Sets the number of shifts in the counter R3L (which indicates the number of shifts).
  - ii. The dividend is shifted 1 bit to the left and the MSB loaded in the C bit is set in the LSB of ER2 (which stores the remainder).
  - iii. The divisor is subtracted from ER2. When the result of subtraction is positive, the LSB of ER0 is set. ((1) to (2) to (3) in figure 4.58). When the results of subtraction is negative, the LSB of ER0 is cleared and the divisor is added to the results of subtraction, returning it to the state prior to subtraction. ((4) to (5) to (6) in figure 4.58).
  - iv. The shift counter set in step (i) is decremented.
  - v. Steps (ii) through (iv) are repeated until the shift counter reaches -1.
- The quotient and/or remainder is then converted to negative if the sign flag is 1, as shown in table 4.30.



**Figure 4.58 Division Example**

**Table 4.30 Sign of Results of Division and the Sign Flag**

Dividend	Divisor	Quotient	Remainder	Quotient Sign Flag	Remainder Sign Flag
Positive	Positive	Positive	Positive	0	0
	Negative	Negative	Positive	1	0
Negative	Positive	Negative	Negative	1	1
	Negative	Positive	Positive	0	0

## 4.15.6 Program Listing

## 4.16 8-Digit Decimal Addition

MCU: H8/300H Series

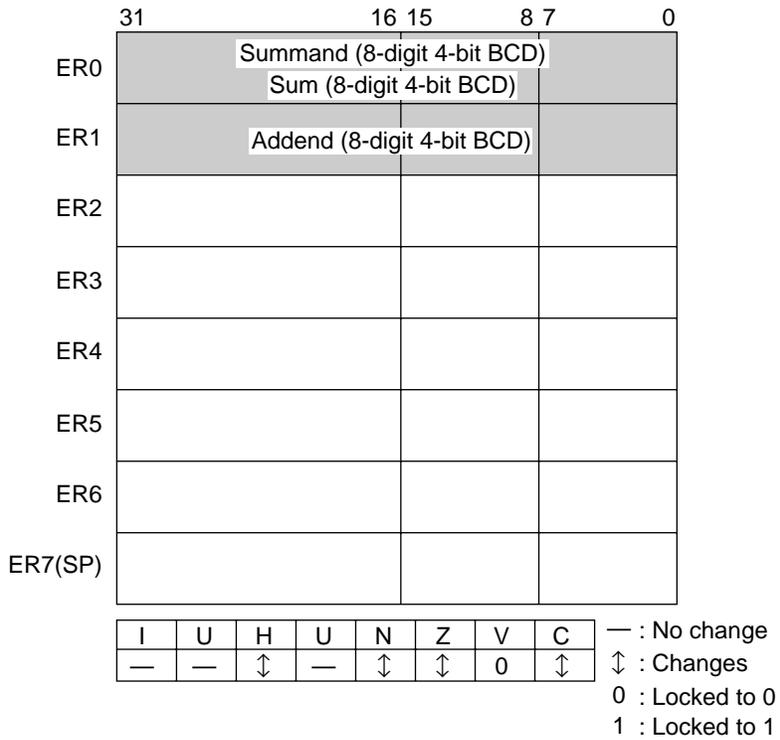
**Label Name:** ADDD

**Functions Used:** DAA.B Instruction

**Function:** Does addition in the format: Summand (8-digit 4-bit BCD) × addend (8-digit 4-bit BCD) = sum (8-digit 4-bit BCD).

**Table 4.31 ADDD Arguments**

	<b>Contents</b>	<b>Storage Location</b>	<b>Data Length (Bytes)</b>
Input	Summand (8-digit 4-bit BCD)	ER0	4
	Summand (8-digit 4-bit BCD)	ER1	4
Output	Sum (8-digit 4-bit BCD)	ER0	4
	Presence of carry (Yes, C = 1; No, C = 0)	C flag	1



**Figure 4.59 Changes in Internal Registers and Flag Changes for DIVS**

Program memory (bytes)
28
Data memory (bytes)
0
Stack (bytes)
0
Number of states
36
Re-entrant
Yes
Relocation
Yes
Interrupts during execution
Yes

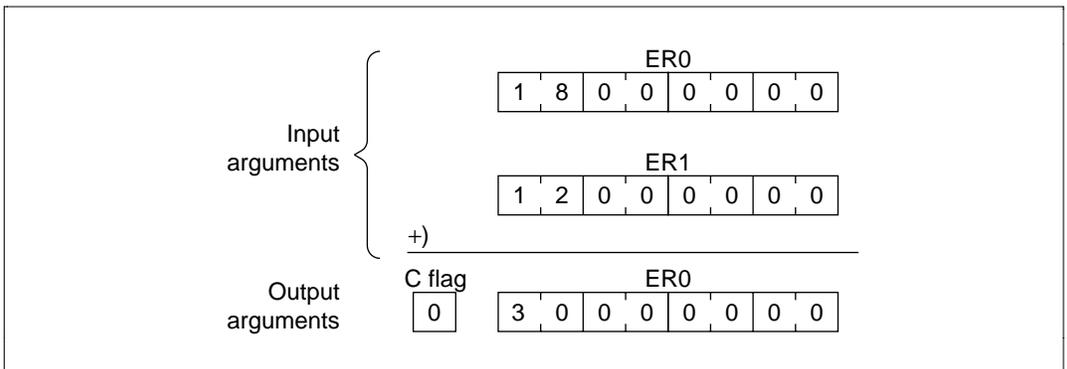
**Figure 4.60 Programming Specifications**

### 4.16.1 Description of Functions

Arguments are as follows:

- ER0: Sets the summand (8-digit 4-bit BCD) as an input argument. Sets the sum (8-digit 4-bit BCD) as an output argument.
- ER1: Sets the addend (8-digit 4-bit BCD) as an input argument.
- C flag (CCR): Indicates whether there is carrying after ADDD is executed.
  - C flag = 1: Indicates there is a carry.
  - C flag = 0: Indicates there is no carry.

Figure 4.61 is an example of execution of the software ADDD. When the input arguments are set as shown, the sum is set in ER0.



**Figure 4.61 Executing ADDD**

### 4.16.2 Cautions for Use

Since the results of addition are set in the register used to set the summand, the summand is destroyed after ADDD is executed. When you will still require the summand after executing ADDD, save it elsewhere in memory beforehand.

### 4.16.3 Description of Data Memory

No data memory is used by ADDD

#### 4.16.4 Examples of Use

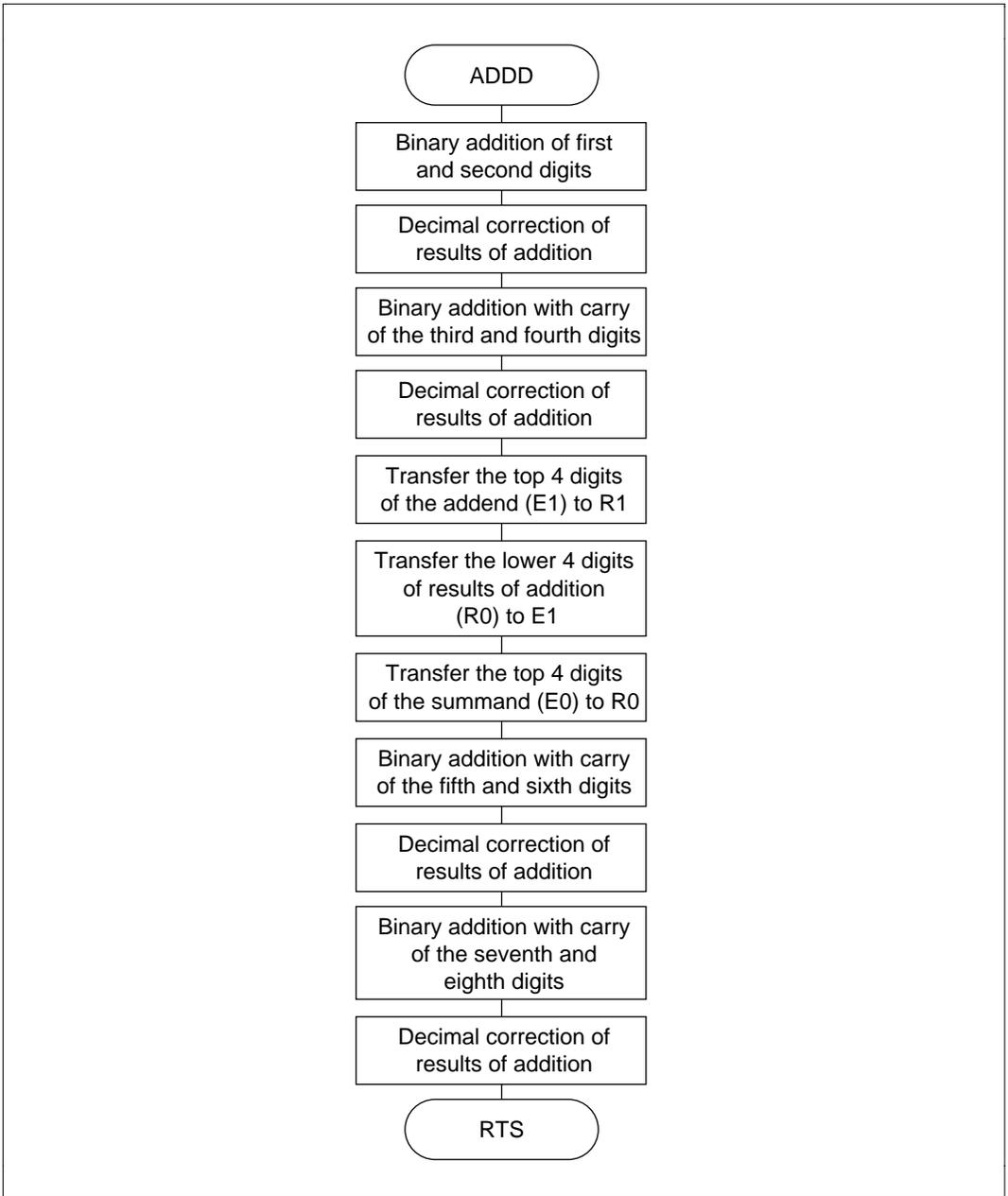
After setting the summand and addend, do a subroutine call to `ADDD`.

**Table 4.32 Block Transfer Example (ADDD)**

Label	Instruction	Action
WORK 1	<code>.RES. L 1</code>	Reserves the data memory area that sets the summand (8-digit 4-bit BCD) in the user program.
WORK 2	<code>.RES. L 1</code>	Reserves the data memory area that sets the addend (8-digit 4-bit BCD) in the user program.
	<code>MOV. L @WORK1,ER0</code>	Sets as the input argument the summand set in the user program.
	<code>MOV. L @WORK2,ER1</code>	Sets as the input argument the addend set in the user program.
	<code>...</code>	Subroutine call to <code>ADDD</code> .
	<code>JSR @ADDD</code>	
	<code>BCS OVER</code>	When the results of addition produce carrying, the program branches to the processing routine for carrying.
	<code>...</code>	
OVER	Processing routine for carrying over	
	<code>...</code>	

#### 4.16.5 Principles of Operation

- Binary addition occurs in 2-digit increments from the bottom and the results of addition are corrected into 2 digits of 4-bit BCD by the `DAA.B` instruction. This process is repeated four times.
- Addition of everything after the initial bottom 2 digits is performed by `ADDX.B` (addition with carrying instruction), since carrying occurs.
- In the extended register in which the upper four digits of the summand and addend are stored, the `DAA.B` and `ADDX.B` instructions cannot be used, so the upper 4 digits of the summand and addend are added after transfer to the general registers.



**Figure 4.62 ADDD Flowchart**

## 4.16.6 Program Listing

## 4.17 8-Digit Decimal Subtraction

MCU: H8/300H Series

**Label Name:** SUBD

**Functions Used:** DAS.B Instruction

**Function:** Does subtraction in the format: Minuend (8-digit 4-bit BCD) – subtrahend (8-digit 4-bit BCD) = difference (8-digit 4-bit BCD).

**Table 4.33 SUBD Arguments**

	<b>Contents</b>	<b>Storage Location</b>	<b>Data Length (Bytes)</b>
Input	Minuend (8-digit 4-bit BCD)	ER0	4
	Subtrahend (8-digit 4-bit BCD)	ER1	4
Output	Difference (8-digit 4-bit BCD)	ER0	4
	Presence of borrow (Yes, C = 1; No, C = 0)	C flag (CCR)	1

	31	16	15	8	7	0			
ER0	Minuend (8-digit 4-bit BCD)		Difference (8-digit 4-bit BCD)						
ER1	Subtrahend (8-digit 4-bit BCD)								
ER2									
ER3									
ER4									
ER5									
ER6									
ER7(SP)									
	I	U	H	U	N	Z	V	C	
	—	—	↕	—	↕	↕	0	↕	— : No change ↕ : Changes 0 : Locked to 0 1 : Locked to 1

**Figure 4.63 Changes in Internal Registers and Flag Changes for SUBD**

Program memory (bytes)
28
Data memory (bytes)
0
Stack (bytes)
0
Number of states
36
Re-entrant
Yes
Relocation
Yes
Interrupts during execution
Yes

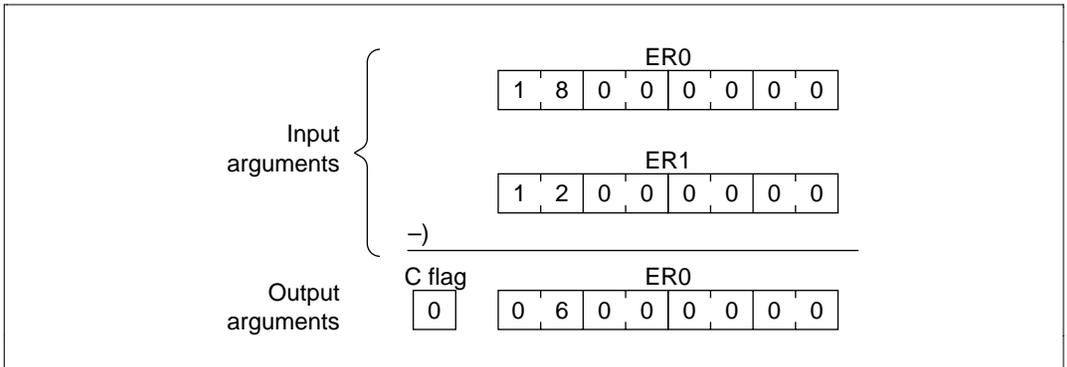
**Figure 4.64 Programming Specifications**

### 4.17.1 Description of Functions

Arguments are as follows:

- ER0: Sets the minuend (8-digit 4-bit BCD) as an input argument. Sets the difference (8-digit, 4-bit BCD) as an output argument.
- ER1: Sets the subtrahend (8-digit 4-bit BCD) as an input argument.
- C flag (CCR): Indicates whether there is borrowing after SUBD is executed.
  - C flag = 1: Indicates there is a borrow.
  - C flag = 0: Indicates there is no borrow.

Figure 4.65 is an example of execution of the software SUBD. When the input arguments are set as shown, the difference is set in ER0.



**Figure 4.65 Executing SUBD**

### 4.17.2 Cautions for Use

Since the results of subtraction are set in the register used to set the minuend, the minuend is destroyed after SUBD is executed. When you will still require the minuend after executing SUBD, save it elsewhere in memory beforehand.

### 4.17.3 Description of Data Memory

No data memory is used by SUBD.

#### 4.17.4 Examples of Use

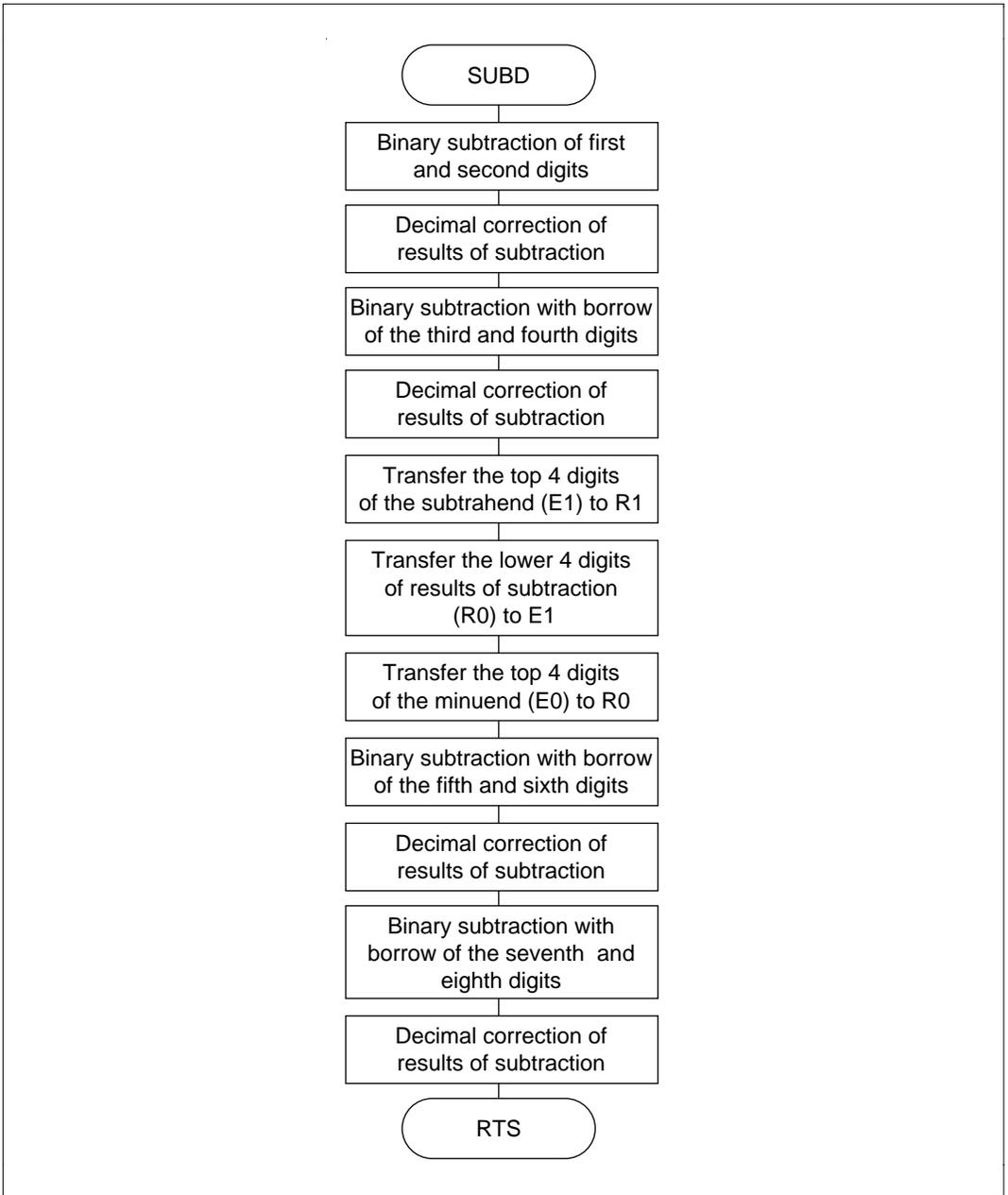
After setting the minuend and subtrahend, do a subroutine call to SUBD.

**Table 4.34 Block Transfer Example (SUBD)**

Label	Instruction	Action
WORK 1	.RES. L 1	Reserves the data memory area that sets the minuend (8-digit 4-bit BCD) in the user program.
WORK 2	.RES. L 1	Reserves the data memory area that sets the subtrahend (8-digit 4-bit BCD) in the user program.
	MOV. L @WORK1,ER0	Sets as the input argument the minuend set in the user program.
	MOV. L @WORK2,ER1	Sets as the input argument the subtrahend set in the user program.
	⋮	Subroutine call to SUBD.
	JSR @SUBD	
	BCS OVER	When the results of subtraction produce borrowing, the program branches to the processing routine for borrowing.
	⋮	
OVER	Processing routine for borrowing	
	⋮	

#### 4.17.5 Principles of Operation

- Binary subtraction occurs in 2-digit increments from the bottom and the results of subtraction are corrected into 2 digits of 4-bit BCD by the DAS.B instruction. This process is repeated four times.
- Subtraction of everything after the initial bottom 2 digits is performed by SUBX.B (subtraction with borrowing instruction), since borrowing occurs.
- In the extended register in which the upper four digits of the minuend and subtrahend are stored, the DAS.B and SUBX.B instructions cannot be used, so the upper 4 digits of the minuend and subtrahend are subtracted after transfer to the general registers.



**Figure 4.66 SUBD Flowchart**

### 4.17.6 Program Listing

## 4.18 Sum of Products

MCU: H8/300H Series

Label Name: SEKIWA

Functions Used: MULXU.W Instruction

**Function:** Does the following sum of products on unsigned 16-bit data  $a_n, b_n$  ( $n = 1, 2, \dots, n$ ) from data tables a and b. The maximum number of data n is 255.

$$\sum_{n=1}^n a_n b_n = a_1 b_1 + a_2 b_2 + \dots + a_n b_n$$

Table 4.35 SEKIWA Arguments

	Contents	Storage Location	Data Length (Bytes)
Input	Start address of data table a	ER0	4
	Start address of data table b	ER1	4
	Number of data n	R3H	1
Output	Results of sum of products (top 8 bits)	R3L	1
	Results of sum of products (bottom 32 bits)	ER2	4

	31	16	15	8	7	0
ER0	Start address of data table a					
ER1	Start address of data table b					
ER2	Results of sum of products (bottom 32 bits)					
ER3				Number of data n		Results of sum of products (top 8 bits)
ER4	Work					
ER5						
ER6						
ER7(SP)						

I	U	H	U	N	Z	V	C
—	—	↕	—	↕	1	↕	↕

— : No change  
 ↕ : Changes  
 0 : Locked to 0  
 1 : Locked to 1

**Figure 4.67 Changes in Internal Registers and Flag Changes for SUBD**

Program memory (bytes)
20
Data memory (bytes)
0
Stack (bytes)
0
Number of states
11234
Re-entrant
Yes
Relocation
Yes
Interrupts during execution
Yes

Caution: The number of states in the programming specifications is the value when the number of data n is H'FF.

**Figure 4.68 Programming Specifications**

### 4.18.1 Description of Functions

Arguments are as follows:

- ER0: Sets the start address of data table a (multiplicands) as an input argument.
- ER1: Sets the start address of data table b (multipliers) as an input argument.
- R3H: Sets the number as an input argument.
- R3L: Sets the top 8 bits of the result of the sum of products operation as an output argument.
- ER2: Sets the bottom 32 bits of the result of the sum of products operation as an output argument.

Figure 4.69 is an example of execution of the software. When the start address of data table a, start address of data table b, and number are set as shown, the top 8 bits of the result of the sum of products operation are set in R3L and bottom 32 bits of the result of the sum of products operation are set in ER2.

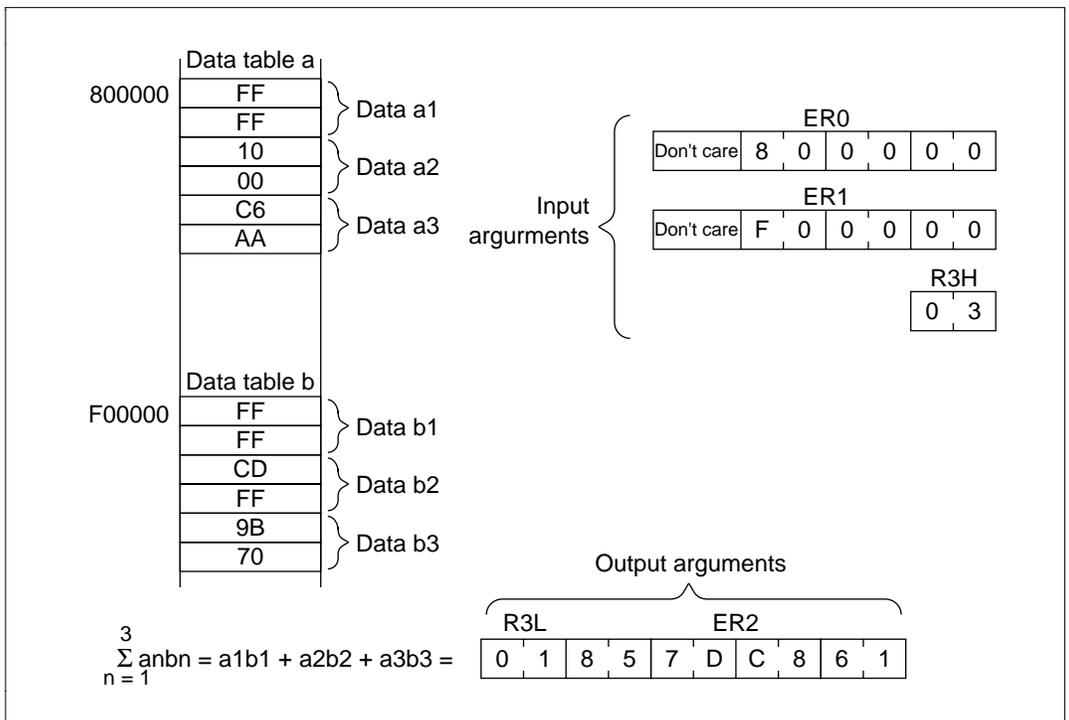


Figure 4.69 Executing SEKIWA

### 4.18.2 Cautions for Use

Since R0H is 1 byte, set data in the range H'01 ≤ R3H ≤ H'FF.

### 4.18.3 Description of Data Memory

No data memory is used by SEKIWA.

### 4.18.4 Examples of Use

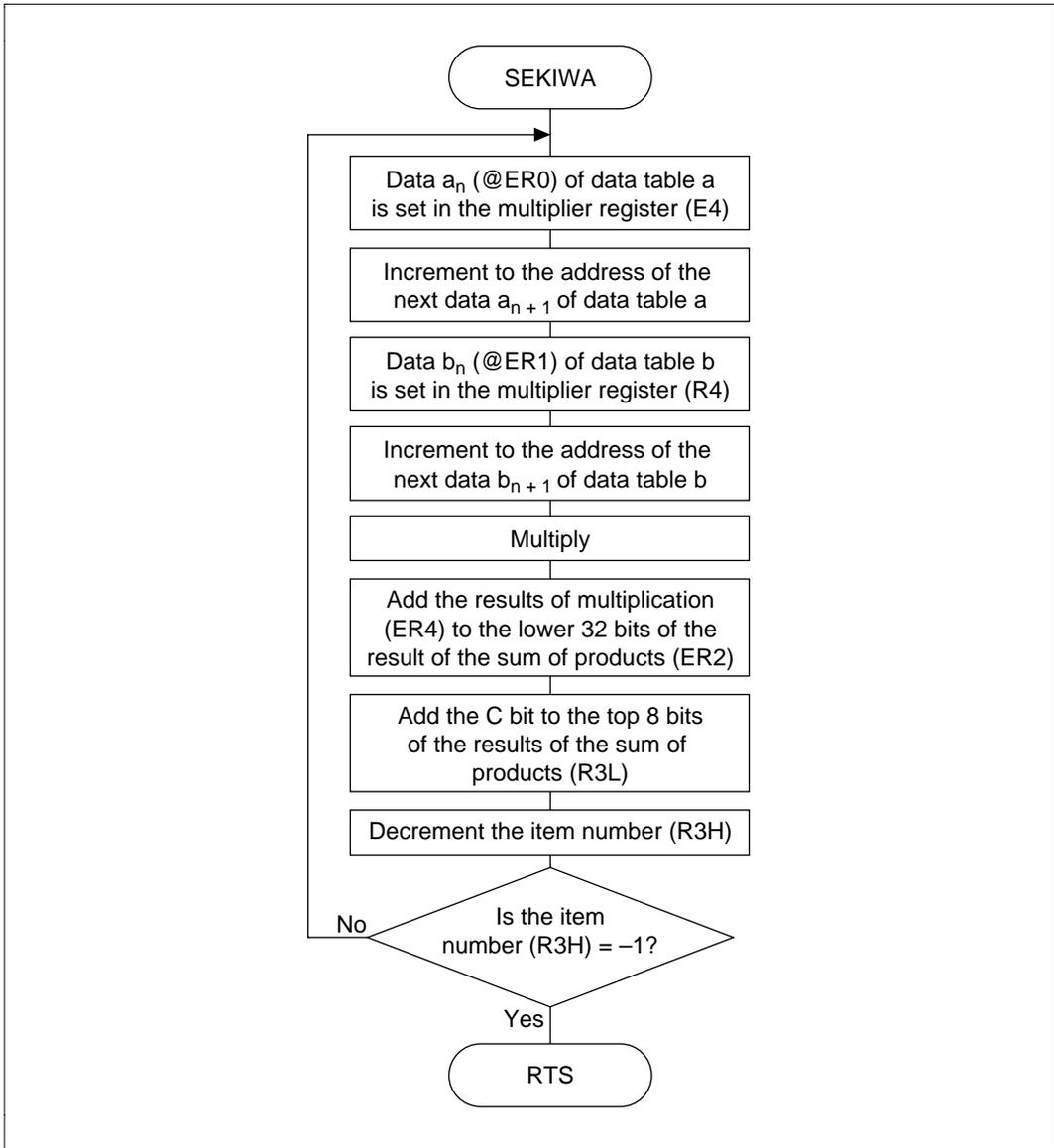
After setting the start address of data table a, start address of data table b and number, do a subroutine call to SEKIWA.

**Table 4.36 Block Transfer Example (SEKIWA)**

Label	Instruction	Action
WORK 1	.RES. L 1	Reserves the data memory area that sets the start address of data table a in the user program.
WORK 2	.RES. L 1	Reserves the data memory area that sets the start address of data table b in the user program.
WORK 3	.RES. B 1	Reserves the data memory area that sets the number in the user program.
	MOV. L @WORK1,ER0	Sets as the input argument the start address of data table a set in the user program.
	MOV. L @WORK2,ER1	Sets as the input argument the start address of data table b set in the user program.
	MOV. B @WORK3,R3H	Sets as the input argument the number set in the user program
	⋮	Subroutine call to SEKIWA.
	JSR @SEKIWA	

### 4.18.5 Principles of Operation

1. ER0 and ER1 are used as pointers to the addresses of the multiplicand (data table a) and multiplier (data table b) data. After the multiplicands and multipliers are set in E4 and R4 respectively, the program increments to the next data address by post-increment register indirect.
2. E4 and R4 are de-signed and multiplied.
3. The results of multiplication stored in ER4 are added to ER2, where the bottom 32 bits of the results of the sum of products are stored.
4. Because of carrying, addition of R3L, where the top 8 bits of the result of the sum of products is stored, uses addition with carrying.
5. R3H is decremented and the processes of steps 1 through 4 repeat until R3H = -1.



**Figure 4.70 SEKIWA Flowchart**

## 4.18.6 Program Listing

## 4.19 Sorting

MCU: H8/300H Series

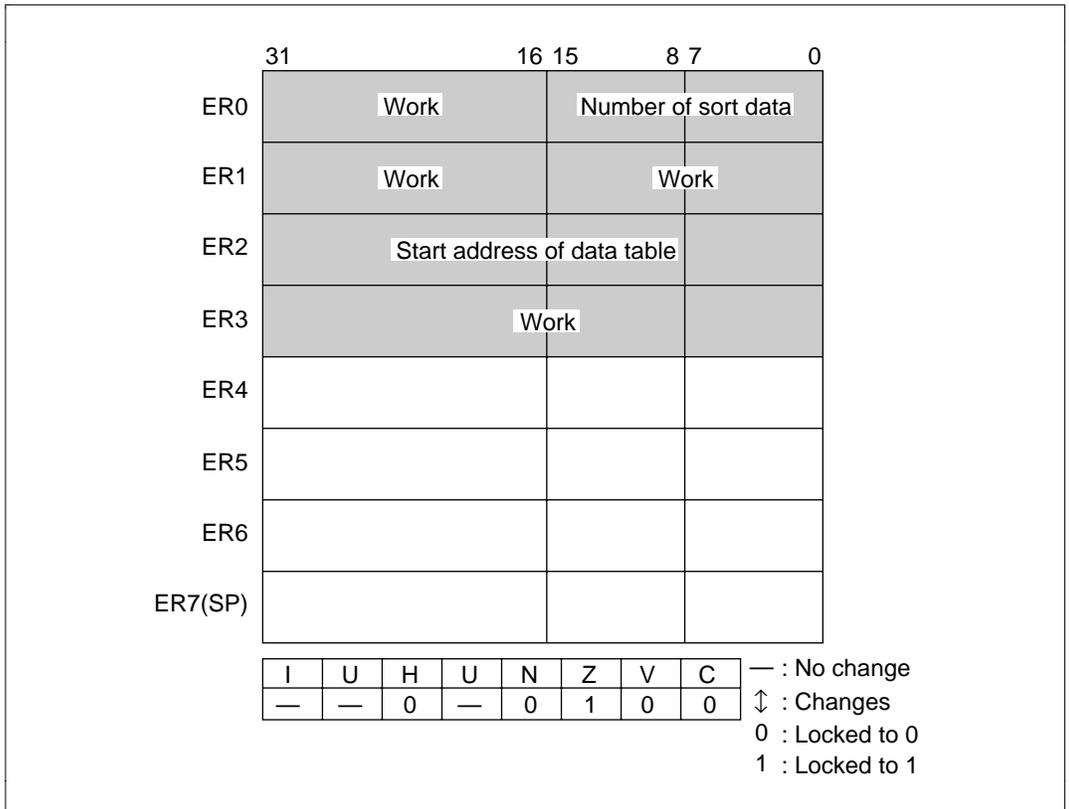
**Label Name:** SORT

**Functions Used:** Post-Increment Register Indirect, Pre-Decrement Register Indirect

**Function:** Sorts data (unsigned 16 bits) of the data table from largest to smallest. The maximum number of data is 65535.

**Table 4.37 SORT Arguments**

	Contents	Storage Location	Data Length (Bytes)
Input	Number of sort data	R0	2
	Start address of data table	ER2	4
Output	—	—	—



**Figure 4.71 Changes in Internal Registers and Flag Changes for SORT**

Program memory (bytes)
32
Data memory (bytes)
0
Stack (bytes)
0
Number of states
404
Re-entrant
Yes
Relocation
Yes
Interrupts during execution
Yes

Caution: The number of states in the programming specifications is the value when 5 words of data arranged smallest to largest is sorted into largest to smallest.

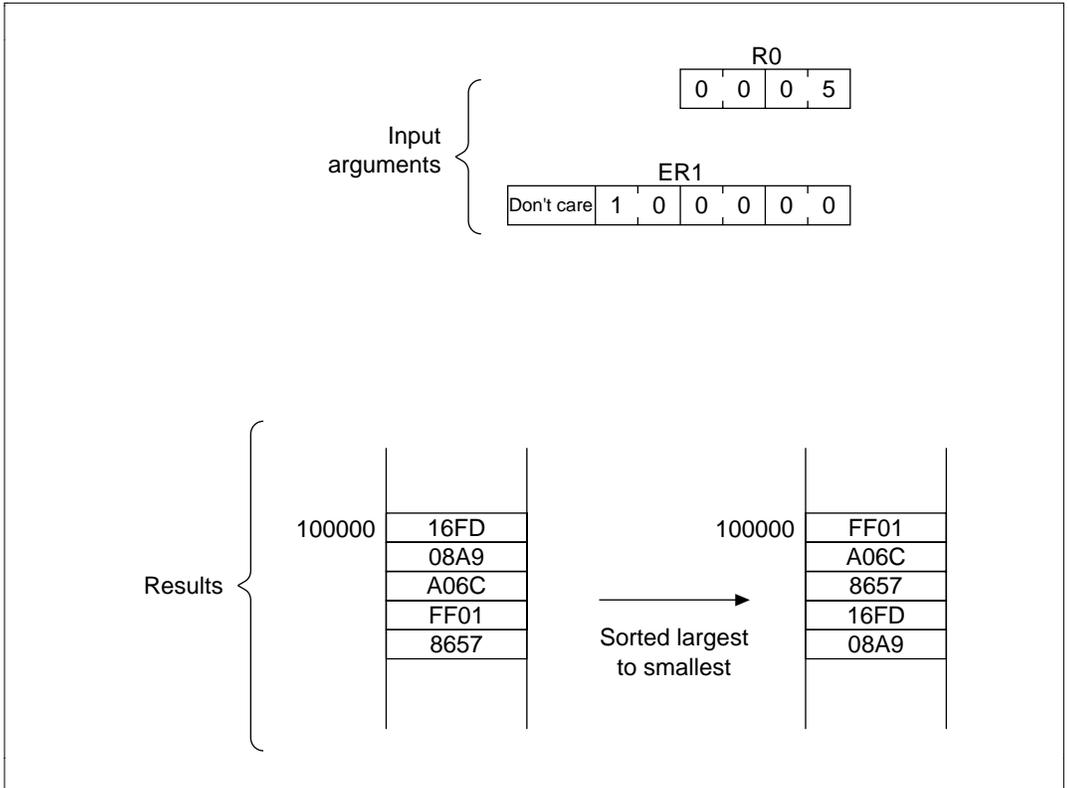
**Figure 4.72 Programming Specifications**

### 4.19.1 Description of Functions

Arguments are as follows:

- R0: Sets the number of sort data.
- ER1: Sets the start address of the data table.

Figure 4.73 is an example of execution of the SORT software. When the input arguments are set as shown, the data table data is sorted largest to smallest.



**Figure 4.73 Executing SORT**

### 4.19.2 Description of Data Memory

No data memory is used by SORT.

### 4.19.3 Examples of Use

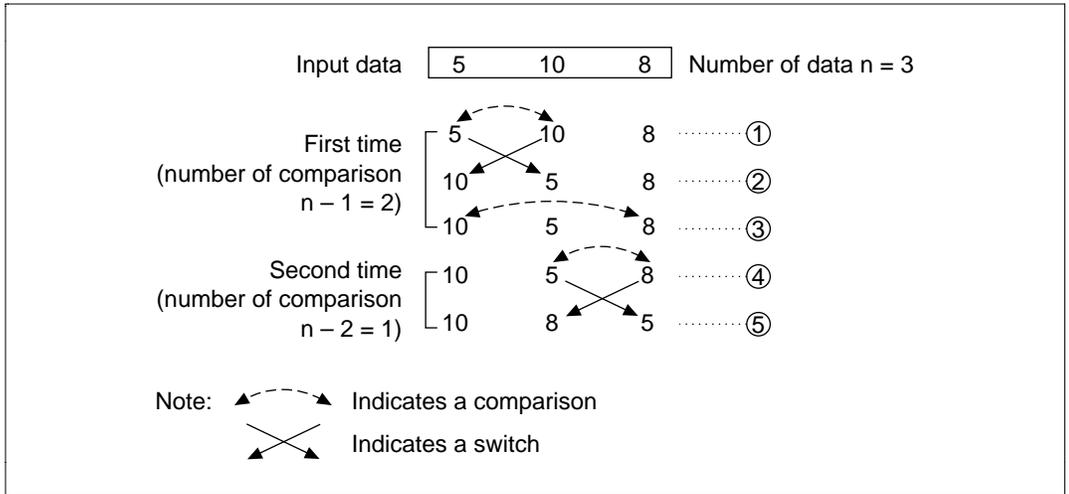
After setting the start address of the data table and the number of sort data, do a subroutine call to SORT.

**Table 4.38 Block Transfer Example (SORT)**

Label	Instruction	Action
WORK 1	.RES. W 1	Reserves the data memory area that sets the number of sort data in the user program.
WORK 2	.RES. L 1	Reserves the data memory area that sets the start address of the data table in the user program.
	MOV. W @WORK1,R0	Sets as the input argument the number of sort data set in the user program.
	MOV. L @WORK2,ER1	Sets as the input argument the start address of the data table set in the user program.
	⋮	Subroutine call of SORT.
	JSR @SORT	

#### 4.19.4 Principles of Operation

Figure 4.74 shows an example of sorting 3 items of data from largest to smallest.

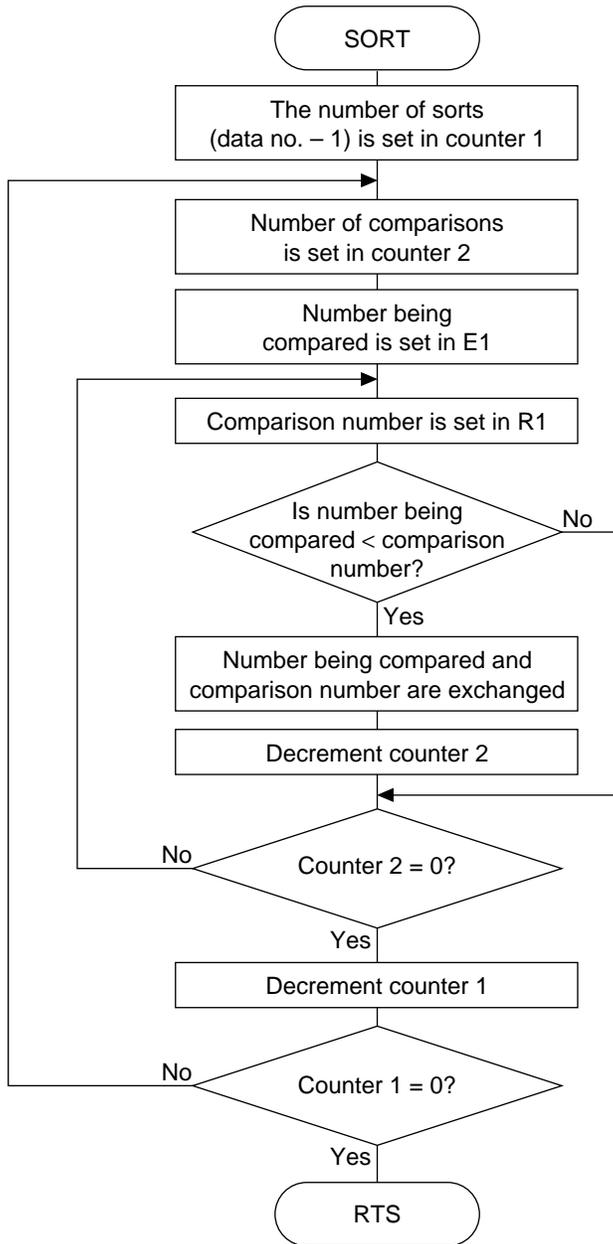


**Figure 4.74 Sorting Example**

1. Selects the largest of the 3 input data and places it at the far left ((1), (2) and (3) in figure 4.74).
2. Selects the largest data from second to left to the end and places it at the second place from left ((4) and (5) in figure 4.74).

#### 4.19.5 Processing Method in Program

1. The number being compared (reference data) is set to E1 and the comparison number is set to R1; the comparison is then done. Since the data being compared is supposed to be the larger of the two numbers, the data are switched whenever the comparison number is larger.
2. ER3 is used as a pointer to the address of the comparison number. Using the post-increment register indirect method, the pointer is incremented to the address where the next comparison number is stored.
3. E0 is used as the counter that counts the number of comparisons done between data to find the largest item in the group of data. Each time a comparison is completed, E0 is decremented and the process repeats until E0 becomes 0.
4. ER2 is used as the pointer that indicates the address of the memory that stores the next largest value. Using the post-increment register indirect method, ER2 is incremented to the address that stores the next maximum value.
5. R0 is used as the counter that counts the number of determinations of the maximum value. Each time a maximum value is determined, R0 is decremented and the process repeats until R0 becomes 0.



**Figure 4.75 SORT Flowchart**

## 4.19.6 Program Listing



# Appendix A Instruction Set

**Table A.1 Operation Symbols**

Symbol	Description
PC	Program counter
SP	Stack pointer (ER7)
CCR	Condition code register
Z	Zero flag of condition code register
C	Carry flag of condition code register
Rs, Rd, Rn	General registers <data> (8 bits: R0H/R0L–R7H/R7L and 16 bits: R0–R7, E0–E7)
ERs, ERd	General registers <address> (24 bits: ER0–ER7), <data> (32 bits: ER0–ER7)
d:8, d:16, d:24	Displacement: 8 bits/16 bits/24 bits
#xx:2/3/8/16/32	Immediate data: 2 bits/3 bits/8 bits/16 bits/32 bits
→	Left end operand transferred to right end operand
+	Add operands of both sides
-	Subtract right end operand from left end operand
×	Multiply both operands
÷	Divide left end operand by right end operand
^	AND of both operands
∨	OR of both operands
⊕	Exclusive OR of both end operands
	Logical complement ( complement of 1)
() <>	Description of execution address of operand

**Table A.2 Condition Code Symbols**

Symbol	Description
↕	Changes with the results of operation
*	Undetermined. Value not guaranteed.
0	Always cleared to 0.
-	No effect on operation.

- Notes:
1. (The number of execution states is the value when the operation code and operand data is in the 2-cycle area that is word accessible, such as on-chip RAM.)
  2. For a word-size operation: When there is a carry or borrow to or from bit 11, this bit is set to 1; otherwise, it is cleared to 0.
  3. For a longword size operation: When there is a carry or borrow to or from bit 27, this bit is set to 1; otherwise, it is cleared to 0.
  4. When the operation result is 0, the value prior to the operation is held; otherwise, it is cleared to 0.
  5. Set to 1 when the results of correction causes a carry; otherwise, the value prior to the operation is held.
  6. The number of execution states is  $4n+8$  when the value set for R4L (for EEPMOV.B) or R4 (for EEPMOV.W) is  $n$ .
  7. Do not use the E clock synchronous transfer instruction with the H8/3003.

## A1 Number of Execution States

The number of execution states for the instruction set is the value when the operation code and operand data is in the 2-cycle area that is word accessible, such as on-chip RAM. Operation code resides in external memory, but its attributes (byte/word access, 2/3 state access, wait/not wait, number of waits) can be set with the bus controller and wait state controller. The attributes of the on-chip peripheral modules are fixed and come in two types: 3-state word access modules and 3-state byte access modules. These combinations increase the number of execution states by the number of states indicated in the following table.

**Table A.3 Increase in Number of Execution States by Operand Data**

Access Conditions	Data Type	Increase in Number of Execution States
External address (2-state byte access)	Byte	0
	Word	2
External address/on-chip RAM (2-state word access)	Byte	0
	Word	0
On-chip peripheral module (3-state byte access)	Byte	1
	Word	4
On-chip peripheral module (3-state word access)	Byte	1
	Word	1
External address (3-state byte access $m$ cycle wait)	Byte	$1 + m$
	Word	$4 + 2m$
External address (3-state word access $m$ cycle wait)	Byte	$1 + m$
	Word	$1 + m$

**Table A.4 Increase in Number of Execution States by Operand Code**

Access Conditions	Instruction Length (Byte)	Increase in Number of Execution States				
		2	4	6	8	10
External address (2-state byte access)	Nonbranch	2	4	6	8	10
	Branch	4	6	-	-	-
External address/on-chip RAM (2-state word access)	Nonbranch	0	0	0	0	0
	Branch	0	0	-	-	-
External address (3-state byte access m cycle wait)	Nonbranch	4 + 2m	8 + 4m	12 + 6m	16 + 8m	20 + 10m
	Branch	8 + 4m	12 + 6m	-	-	-
External address (3-state word access m cycle wait)	Nonbranch	1 + m	2 + 2m	3 + 3m	4 + 4m	5 + 5m
	Branch	2 + 2m	3 + 3m	-	-	-

**Table A.5 Instruction List**

Mnemonic	Op. Sz.	Operation	Addressing Mode/ Instruction Length								Condition Code				No. of Execution States			
			#xx:	Rn	@ERn	@(d: , ERn)	@-ERn/@ERN+	@aa:	@(d: , PC)	@@aa	Implied	I	UI	H		N	Z	V
Data transfer instr.	MOV.B #xx:8, Rd	B	#xx:8→Rd8	2										↓	↓	0	—	2
	MOV.B Rs,Rd	B	Rs8→Rd8	2										↓	↓	0	—	2
	MOV.B @ERs,Rd	B	@ERs→Rd8		2									↓	↓	0	—	4
	MOV.B @(d:16,ERs)Rd	B	@(d:16,ERs)→Rd8			4								↓	↓	0	—	6
	MOV.B @(d:24,ERs),Rd	B	@(d:24,ERs)→Rd8			8								↓	↓	0	—	10
	MOV.B @ERs+,Rd	B	@ERs→Rd8, ERs+1→ERs				2							↓	↓	0	—	6
	MOV.B @aa:8,Rd	B	@aa:8→Rd8					2						↓	↓	0	—	4
	MOV.B @aa:16,Rd	B	@aa:16→Rd8						4					↓	↓	0	—	6
	MOV.B @aa:24,Rd	B	@aa:24→Rd8							6				↓	↓	0	—	8
	MOV.B Rs,@ERd	B	Rs8→@ERd		2									↓	↓	0	—	4
	MOV.B Rs,@(d:16,ERd)	B	Rs8→@(d:16,ERd)			4								↓	↓	0	—	6
	MOV.B Rs,@(d:24,ERd)	B	Rs8→@(d:24,ERd)			8								↓	↓	0	—	10
	MOV.B Rs,@-ERd	B	ERd-1→ERd, Rs8→@ERd				2							↓	↓	0	—	6
	MOV.B Rs,@aa:8	B	Rs8→@aa:8					2						↓	↓	0	—	4
	MOV.B Rs,@aa:16	B	Rs8→@aa:16						4					↓	↓	0	—	6
	MOV.B Rs,@aa:24	B	Rs8→@aa:24							6				↓	↓	0	—	8
	MOV.W#xx:16, Rd	W	#xx:16→Rd16	4										↓	↓	0	—	4
	MOV.W Rs,Rd	W	Rs16→Rd16		2									↓	↓	0	—	2
	MOV.W @ERs,Rd	W	@ERs→Rd16			2								↓	↓	0	—	4
	MOV.W @(d:16,ERs),Rd	W	@(d:16,ERs)→Rd16				4							↓	↓	0	—	6
MOV.W @(d:24,ERs),Rd	W	@(d:24,ERs)→Rd16				8							↓	↓	0	—	10	
MOV.W @ERs+,Rd	W	@ERs→Rd16, ERs+2→ERs					2						↓	↓	0	—	6	
MOV.W @aa:16,Rd	W	@aa:16→Rd16						4					↓	↓	0	—	6	







**Table A.5 Instruction List (cont)**

Mnemonic	Op. Sz.	Operation	#xx:	Rn	@ERn	@ (d: , ERn)	@-ERn/@ERN+	@aa:	@ (d: , PC)	@@aa	Implied	Condition Code							No. of Execution States
												I	UI	H	N	Z	V	C	
Logical op. instr.	AND.B #xx:8,Rd	B	Rd8^#xx:8→Rd8	2								—	—	—	↓	↓	0	—	2
	AND.B Rs,Rd	B	Rd8^Rs8→Rd8	2								—	—	—	↓	↓	0	—	2
	AND.W #xx:16,Rd	W	Rd16^#xx:16→RD16	4								—	—	—	↓	↓	0	—	4
	AND.W Rs,Rd	W	Rd16^Rs16→Rd16	2								—	—	—	↓	↓	0	—	2
	AND.L #xx:32,ERd	L	ERd32^#xx:32→ERd32	6								—	—	—	↓	↓	0	—	6
	AND.L ERs,ERd	L	ERd32^ERs32→ERd32	4								—	—	—	↓	↓	0	—	4
	OR.B #xx:8,Rd	B	Rd8v#xx:8→Rd8	2								—	—	—	↓	↓	0	—	2
	OR.B Rs,Rd	B	Rd8vRs8→Rd8	2								—	—	—	↓	↓	0	—	2
	OR.W #xx:16,Rd	W	Rd16v#xx:16→Rd16	4								—	—	—	↓	↓	0	—	4
	OR.W Rs,Rd	W	Rd16vRs16→Rd16	2								—	—	—	↓	↓	0	—	2
	OR.L #xx:32,ERd	L	ERd32v#xx:32→ERd32	6								—	—	—	↓	↓	0	—	6
	OR.L ERs,ERd	L	ERd32vERs32→ERd32	4								—	—	—	↓	↓	0	—	4
	XOR.B #xx:8,Rd	B	Rd8@#xx:8→Rd8	2								—	—	—	↓	↓	0	—	2
	XOR.B Rs,Rd	B	Rd8@Rs8→Rd8	2								—	—	—	↓	↓	0	—	2
	XOR.W #xx:16,Rd	W	Rd16@#xx:16→Rd16	4								—	—	—	↓	↓	0	—	4
	XOR.W Rs,Rd	W	Rd16@Rs16→Rd16	2								—	—	—	↓	↓	0	—	2
	XOR.L #xx:32,ERd	L	ERd32@#xx:32→ERd32	6								—	—	—	↓	↓	0	—	6
	XOR.L ERs,ERd	L	ERd32@ERs32→ERd32	4								—	—	—	↓	↓	0	—	4
NOT.B Rd	B	Rd8→Rd8	2								—	—	—	↓	↓	0	—	2	
NOT.W Rd	W	Rd16→Rd16	2								—	—	—	↓	↓	0	—	2	
NOT.L ERd	L	ERd32→ERd32	2								—	—	—	↓	↓	0	—	2	
Shift instr.	SHAL.B Rd	B	Rd8 left arithmetic shift→Rd8	2							—	—	—	↓	↓	↓	↓	2	
	SHAL.W Rd	W	Rd16 left arithmetic shift→Rd16	2							—	—	—	↓	↓	↓	↓	2	
	SHALL ERd	L	ERd32 left arithmetic shift→ERd32	2							—	—	—	↓	↓	↓	↓	2	
	SHAR.B Rd	B	Rd8 right arithmetic shift→Rd8	2							—	—	—	↓	↓	0	↓	2	





**Table A.5 Instruction List (cont)**

Mnemonic	Op. Sz.	Operation	#xx:	Addressing Mode/ Instruction Length							Condition Code							No. of Execution States	
				Rn	@ERn	@(d: , ERn)	@-ERn/@ERN+	@aa:	@(d: , PC)	@@aa	Implied	I	UI	H	N	Z	V		C
Bit man. instr. (cont)	BILD #xx:3, @aa:8	B	(#xx:3 of @aa:8) →C						4										6
	BST #xx:3,Rd	B	C→(#xx:3 of Rd8)	2															2
	BST #xx:3, @ERd	B	C→(#xx:3 of @ERd)		4														8
	BST #xx:3, @aa:8	B	C→(#xx:3 of @aa:8)						4										8
	BIST #xx:3,Rd	B	C→(#xx:3 of Rd8)	2															2
	BIST #xx:3, @ERd	B	C→(#xx:3 of @ERd)		4														8
	BIST #xx:3, @aa:8	B	C→(#xx:3 of @aa:8)						4										8
	BAND #xx:3, Rd	B	C^(#xx:3 of Rd8) →C	2															2
	BAND #xx:3, @ERd	B	C^(#xx:3 of @ERd) →C		4														6
	BAND #xx:3, @aa:8	B	C^(#xx:3 of @aa:8) →C						4										6
	BIAND #xx:3, Rd	B	C^(#xx:3 of Rd8) →C	2															2
	BIAND #xx:3, @ERd	B	C^(#xx:3 of @ERd) →C		4														6
	BIAND #xx:3, @aa:8	B	C^(#xx:3 of @aa:8) →C						4										6
	BOR #xx:3,Rd	B	Cv(#xx:3 of Rd8) →C	2															2
	BOR #xx:3, @ERd	B	Cv(#xx:3 of @ERd) →C		4														6
	BOR #xx:3, @aa:8	B	Cv(#xx:3 of @aa:8) →C						4										6
	BIOR #xx:3,Rd	B	Cv(#xx:3 of Rd8) →C	2															2
	BIOR #xx:3, @ERd	B	Cv(#xx:3 of @ERd) →C		4														6
	BIOR #xx:3, @aa:8	B	Cv(#xx:3 of @aa:8) →C						4										6
	BXOR #xx:3, Rd	B	C⊕ (#xx:3 of Rd8) →C	2															2
	BXOR #xx:3, @ERd	B	C⊕ (#xx:3 of @ERd) →C		4														6
	BXOR #xx:3, @aa:8	B	C⊕ (#xx:3 of @aa:8) →C						4										6
	BIXOR #xx:3, Rd	B	C⊕ (#xx:3 of Rd8) →C	2															2

**Table A.5 Instruction List (cont)**

Mnemonic	Op. Sz.	Operation	Addressing Mode/ Instruction Length								Condition Code							No. of Execution States
			#xx:	Rn	@ERn	@(d: , ERn)	@-ERn/@ERN+	@aa:	@(d: , PC)	@@aa	Implied	I	UI	H	N	Z	V	
Bit man. instr. (cont)	BIXOR	B	C $\oplus$ (#xx:3 of @ERd) $\rightarrow$ C		4												$\uparrow$	6
	BIXOR	B	C $\oplus$ (#xx:3 of @aa:8) $\rightarrow$ C					4									$\uparrow$	6
Branch instr.	Bcc d:8	—	if condition is true, then PC $\leftarrow$ PC+d:8 else next						2									4
	Bcc d:16	—	If condition is true, then PC $\leftarrow$ PC+d:16 else next						4									6
	JMP @ERn	—	PC $\leftarrow$ ERn		2													4
	JMP @aa:24	—	PC $\leftarrow$ aa:24					4										6
	JMP @@aa:8(normal)	—	PC $\leftarrow$ (@aa:8)16							2								8
	JMP @@aa:8(advanced)	—	PC $\leftarrow$ (@aa:8)24							2								10
	BSR d:8 (normal)	—	SP-2 $\rightarrow$ SP, PC16 $\rightarrow$ @SP PC $\leftarrow$ PC+d:8						2									6
	BSR d:8 (advanced)	—	SP-4 $\rightarrow$ SP, PC24 $\rightarrow$ @SP PC $\leftarrow$ PC+d:8						2									8
	BSR d:16 (normal)	—	SP-2 $\rightarrow$ SP, PC16 $\rightarrow$ @SP PC $\leftarrow$ PC+d:16						4									6
	BSR d:16 (advanced)	—	SP-4 $\rightarrow$ SP, PC24 $\rightarrow$ @SP PC $\leftarrow$ PC+d:16						4									8
	JSR @ERn (normal)	—	SP-2 $\rightarrow$ SP, PC16 $\rightarrow$ @SP PC $\leftarrow$ ERn		2													6
	JSR @ERn (advanced)	—	SP-4 $\rightarrow$ SP, PC24 $\rightarrow$ @SP PC $\leftarrow$ ERn		2													8
	JSR @aa:24 (normal)	—	SP-2 $\rightarrow$ SP, PC16 $\rightarrow$ @SP PC $\leftarrow$ aa:24						4									8
	JSR @aa:24 (advanced)	—	SP-4 $\rightarrow$ SP, PC24 $\rightarrow$ @SP PC $\leftarrow$ aa:24						4									10
	JSR @@aa:8 (normal)	—	SP-2 $\rightarrow$ SP, PC16 $\rightarrow$ @SP PC $\leftarrow$ (@aa:8)16							2								8
	JSR @@aa:8 (advanced)	—	SP-4 $\rightarrow$ SP, PC24 $\rightarrow$ @SP PC $\leftarrow$ (@aa:8)24							2								12
RTS (normal)	—	PC $\leftarrow$ (@SP)16 SP+2 $\rightarrow$ SP								2							8	

**Table A.5 Instruction List**

Mnemonic	Op. Sz.	Operation	#xx:	Addressing Mode/ Instruction Length							Condition Code							No. of Execution States		
				Rn	@ERn	@(d: , ERn)	@-ERn/@ERN+	@aa:	@(d: , PC)	@@aa	Implied	I	UI	H	N	Z	V		C	
System control instr.	RTS (advanced)	—									2	—	—	—	—	—	—	—	—	10
	RTE	—									2	↑	↑	↑	↑	↑	↑	↑	↑	10
	TRAPA #xx:2	—									2	1	—	—	—	—	—	—	—	14
	SLEEP	—										—	—	—	—	—	—	—	—	2
	NOP	—									2	—	—	—	—	—	—	—	—	2
	LDC #xx:8,CCR	B	#xx:8→CCR	2								↓	↓	↓	↓	↓	↓	↓	↓	2
	LDC Rs,CCR	B	Rs8→CCR		2							↓	↓	↓	↓	↓	↓	↓	↓	2
	LDC @ERs, CCR	W	@ERs(even)→CCR			4						↓	↓	↓	↓	↓	↓	↓	↓	6
	LDC @ (d:16,ERs),CCR	W	@(d:16,ERs)(even)→CCR				6					↓	↓	↓	↓	↓	↓	↓	↓	8
	LDC @ (d:24,ERs),CCR	W	@(d:24,ERs)(even)→CCR					10				↓	↓	↓	↓	↓	↓	↓	↓	12
	LDC @ERs+, CCR	W	@ERs(even)→CCR,ERs+2→ERs						4			↓	↓	↓	↓	↓	↓	↓	↓	8
	LDC @aa:16, CCR	W	@aa:16(even)→CCR							6		↓	↓	↓	↓	↓	↓	↓	↓	8
	LDC @aa:24, CCR	W	@aa:24(even)→CCR								8	↓	↓	↓	↓	↓	↓	↓	↓	10
	STC CCR,Rd	B	CCR→Rd8		2							—	—	—	—	—	—	—	—	2
	STC CCR, @ERd	W	CCR→@ERd (even)			4						—	—	—	—	—	—	—	—	6
	STC CCR, @(d:16,ERd)	W	CCR→@(d:16,ERd)(even)				6					—	—	—	—	—	—	—	—	8
	STC CCR, @(d:24,ERd)	W	CCR→@(d:24,ERd)(even)					10				—	—	—	—	—	—	—	—	12
	STC CCR, @-ERd	W	ERd-2→ERd, CCR→@ERd (even)						4			—	—	—	—	—	—	—	—	8
	STC CCR, @aa:16	W	CCR→@aa:16 (even)							6		—	—	—	—	—	—	—	—	8
	STC CCR, @aa:24	W	CCR→@aa:24 (even)								8	—	—	—	—	—	—	—	—	10
	ANDC #xx:8, CCR	B	#xx:8^CCR→CCR 2									↓	↓	↓	↓	↓	↓	↓	↓	2
	ORC #xx:8,CCR	B	#xx:8∨CCR→CCR 2									↓	↓	↓	↓	↓	↓	↓	↓	2
	XORC #xx:8, CCR	B	#xx:8⊕CCR→CCR 2									↓	↓	↓	↓	↓	↓	↓	↓	2

# Appendix B Assembler Control Instruction Functions

## B.1 .CPU

Specifies the CPU.

### Format:

Label	Operation	Operand
x	.CPU	CPU type

Note: CPU type: {300HA | 300HN | 300 | 300L}

**Description:** Specifies the CPU that the source program to be assembled is for. The assembler assembles it for the specified CPU.

CPU types are as follows:

- 300HA H8/300H advanced mode
- 300HN H8/300H normal mode
- 300 H8/300
- 300L H8/300L

When this control instruction is omitted, 300HA is set.

This control instruction should be stated at the start of the source program. If there is nothing at the start of the source program except the control instruction for the assembler list, an error will result.

This control instruction is valid only once. It is valid when there is no /CPU command line option specified.

### Example:

```
.CPU: 300HA

.SECTION A, CODE, ALIGN = 2
MOV.W R0, R1
MOV.W R0, R2
```

Assembles for H8/300H, advanced mode.

## B.2 .SECTION

Declares the section.

### Format:

Label	Operation	Operand
x	.SECTION	Section name [, section attributes [, format type]] type

Note: Section attributes: {CODE | DATA | STACK | COMMON | DUMMY}

Format type: {LOCATE = start address|ALIGN = boundary adjust number}

**Description:** Declares the start and restart of the section.

- Section start: Starts the section and sets the section name, section attributes and type of format.
  - Section name: Specifies the section name. Section names are written the same as symbol names. Case is not distinguished.
  - Section attributes: Sets the section attributes. Section attributes are as follows:

CODE: Code section

DATA: Data section

STACK: Stack section

COMMON: Common section

DUMMY: Dummy section

When no attribute is specified, CODE is set.

- Format type: Sets the format type:

LOCATE = start address                      Absolute addressing

ALIGN = boundary adjust number      Relative addressing

When no format is specified, ALIGN = 2 is set.

With absolute addressing, the start address of the section is set. The start address is specified as a rear-referenced absolute value. The maximum start address values are as follows:

- H8/300H advanced mode: H'00FFFFFF
- H8/300H normal mode: H'0000FFFF
- H8/300: H'0000FFFF
- H8/300L: H'0000FFFF

Relative addressing sets the boundary adjust number of the section. With the linkage editor, the start address of the relative address section when linked to an object module is corrected to a multiple of the boundary adjust number. The boundary adjust number is specified as a rear-referenced absolute value. The boundary adjust number can be specified as a  $2^n$  value.

If no section is declared with this control instruction, the following is set as the default section.

```
.SECTION P, CODE, ALIGN=2
```

- Section restart: Restarts the section already existing in the source program. At section restart, the section name of the existing section is specified. The previously declared section attributes and formats are used.

Example:

```
.SECTION  A, CODE, ALIGN=2           ( 1 )
MOV.W    R0, R1
.SECTION  B, DATA, LOCATE=H'001000  ( 2 )
DATA1
.DATA.W   H'0001
.SECTION  A                           ( 3 )
MOV.W    R0, R3
```

- Starts section A. The section name is A, the section attribute is code section, the format type is relative address format, and the boundary adjust number is 2.
- Starts section B. The section name is B, the section attribute is data section, the format type is absolute address format, and the start address is H'001000.
- Restarts section A.

## B.3 .EQU

Sets the symbol value.

### Format:

Label	Operation	Operand
Symbol name	.EQU	Number

**Description:** Sets a value for the symbol. The value is set as a rear-referenced absolute value or a rear-referenced address value. The symbol value defined by this control instruction cannot be changed.

### Example:

```
SYM1 .EQU 1
SYM2 .EQU 2
.SECTION A, CODE, ALIGN = 2
MOV.B    #SYM1:8, R0L... Same as MOV.B    #1:8, R0L
MOV.B    #SYM2:8, R1L... Same as MOV.B    #2:8, R1L
```

Sets 1 for SYM1 and 2 for SYM2.

## B.4 .ORG

Sets the location counter value.

### Format:

Label	Operation	Operand
x	.ORG	Location counter value

**Description:** Changes the location counter value in the section to the specified value.

The location counter value is specified as a rear-referenced absolute value or as a rear-referenced address value of the section itself. The maximum location counter values are as follows.

H8/300H advanced mode: H'00FFFFFF

H8/300H normal mode: H'0000FFFF

H8/300: H'0000FFFF

H8/300L: H'0000FFFF

When specified in the absolute address section, the location counter value specified must be a value after the start address of the section. When this control instruction is specified in the absolute address section, the set location counter value becomes an absolute address; when specified in the relative address section, it becomes a relative address.

### Example:

```
.SECTION A, DATA, ALIGN = 2
DATA1
.DATA.W    H'0001
.DATA.W    H'0002
.ORG H'000100    (1)
DATA2
.DATA.W    H'0003
.DATA.W    H'0004
```

(1) The location counter value is changed to the relative H'000100 address for A.

## B.5 .DATA

Reserves integer data.

### Format:

Label	Operation	Operand
x	.DATA [- s]	Integer data [, integer data ...]

Note: s (size): {B|W|L}

**Description:** Reserves integer data according to the size specified.

The sizes are as follows.

- B: Byte (1 byte)
- W: Word (2 byte)
- L: Longword (4 bytes)

When not specified, B is set.

The following integer data values can be specified according to size.

- B: -128 to 255
- W: -32,768 to 65,535
- L: -2,147,483,648 to 4,294,967,295

Example:

```
.SECTION A, DATA, ALIGN = 2
.DATA.W    H'0102, H'0304
.DATA.B    H'05, H'06, H'07, H'08
```

Data is reserved as follows:

01	02	03	04	05	06	07	08
----	----	----	----	----	----	----	----

## B.6 .RES

Reserves the integer data region.

### Format:

Label	Operation	Operand
[Symbol name]	.RES [. s]	Number of regions

Note: s (size): {B|W|L}

**Description:** Reserves integer data regions. A region of exactly the size specified for the integer data region is ensured.

The sizes are as follows:

- B: Byte (1 byte)
- W: Word (2 byte)
- L: Longword (4 bytes)

When not specified, B is set.

The number of regions is specified as a rear-referenced absolute value. Any number higher than 1 can be specified.

### Example:

```
.SECTION A, DATA, ALIGN = 2  
.RES.W      10  
.RES.B      255
```

A 20 byte region and a 255 byte region are kept.

## B.7 .END

End of source program.

### Format:

Label	Operation	Operand
x	.END	[Execution start address]

**Description:** Indicates the end of the source program. When this control instruction appears, the assembler quits assembling. The execution start address allows you to specify the address used when the simulation is started on a simulation debugger. The code section address is set for the execution start address. The execution start address is specified as an absolute value or address value.

### Example:

```
.CPU 300HA
.OUTPUT  DBG
:
.SECTION A, CODE, ALIGN = 2
START
MOV.L    #0:32, ER0
MOV.L    #1:32, ER1
MOV.L    #2:32, ER2
BRA      START:8
;
.END     START
```

In the simulation debugger, the simulation starts from the START address.