SuperH RISC Engine
SH7000 Series CPU

Application Note

HITACHI

First Edition
July 1995
Hitachi Micro Systems, Incorporated

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.

Introduction

The SH7000 series are new generation single-chip RISC (Reduced instruction set computer) microcomputers that achieve high-performance operation owing to a RISC-type CPU. The series also integrates peripheral functions required for system configuration and realizes the low power consumption needed for portable computer applications.

The SH7000 CPU has a RISC-type instruction set. Most instructions can be executed in one clock cycle, which greatly improves instruction execution speed. In addition, the 32-bit internal bus architecture enhances data processing capability.

This application note details the SH7000 series CPU architecture, and gives examples of software applications that take advantage of the CPU's features. Use this manual to familiarize yourself with addressing modes, instruction functions, etc.

Section 1 CPU Architecture

1.1 Features

The SH7000 series CPU has a RISC-type instruction set. Basic instructions are executed in one system clock cycle, which greatly improves instruction execution speed. In addition, the 32-bit internal data path boosts data processing capability.

Table 1.1 SH7000 Series CPU Features

Item	Features
Architecture	• Hitachi original architecture • 32-bit internal data path
General register machine	• General registers: 32 bit × 16 • Control registers: 32 bit × 3 • System registers: 32 bit × 4
Instruction set	• RISC type instruction set: — 16-bit fixed-length instructions for improved code efficiency — Load-store architecture (basic arithmetic operations are executed between registers) — Delayed conditional/unconditional branch instructions reduce pipeline disruption — C language-oriented instruction set
Instruction execution time	Basic instructions are executed in one clock cycle (one cycle = 50 ns during 20-MHz operation)
Address space	Architecture supports 4 Gbyte addresses
On-chip multiplier	Multiplication operations ($16 \times 16 \rightarrow 32$) executed in 1–3 cycles, ($16 \times 16 + 42 \rightarrow 42$) multiplication/accumulation executed in 2–3 cycles
Pipeline	5-stage pipeline
Processing states	• Program execution • Exception processing • Bus release • Reset • Power-down
Power-down state	• Sleep mode • Standby mode

1.2 Register Organization

The SH7000 series CPU registers are shown below. There are sixteen 32-bit general registers, three 32-bit control registers, and four 32-bit system registers.

1.2.1 General Registers

There are sixteen general registers (Rn), which are 32 bits in length. General registers are used for data processing and address calculation. Several instructions are restricted to the use of general register R0. R15 is used as a hardware stack pointer (SP) during exception processing, so care must be taken when using this register (figure 1.1).

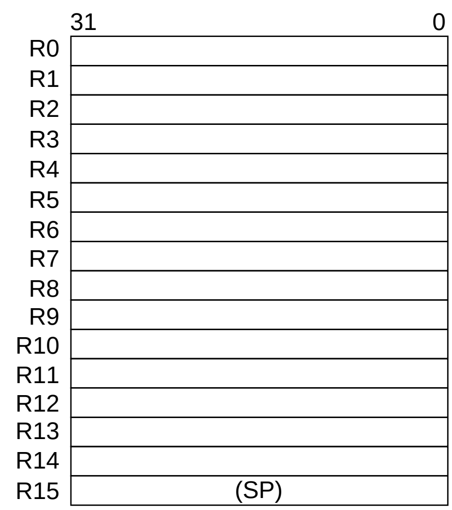


Figure 1.1 General Registers

1.2.2 Control Registers

Status Register (SR): The status register (SR) is 32 bits in length and indicates CPU processing states (figure 1.2). The functions of each bit in the SR are given in table 1.2.

Table 1.2 Status Register Bit Functions

Bit No.	Bit Name	Description
0	T	Indicates true (1) and false (0) for the following instructions: MOVT, CMP/cond, TAS, TST, BT, BF, SETT, CLRT. Indicates carry, borrow, overflow, and underflow with the following instructions: ADDV, ADDC, SUBV, SUBC, NEGC, DIV0U, DIV0S, DIV1, SHAR, SHAL, SHLR, SHLL, ROTR, ROTL, ROTCR, ROTCL.
1	S	Used with MAC instruction.
2–3	—	Reserved bits. Cannot be written to, and always read 0.
4	I0	Interrupt mask bit.
5	I1	
6	I2	
7	I3	
8	Q	Used with DIV0U, DIV0S, and DIV1 instructions.
9	M	
10–31	—	Reserved bits. Cannot be written to, and always read 0.

```
        Status register (SR)
31--------------10 9 8 7 6 5 4 3 2 1 0
| - - - - - - - - - - - - - M Q I3 I2 I1 I0 - - S T |
```

Figure 1.2 Status Register

Global Base Register (GBR): Indicates the base address of the indirect GBR addressing mode (figure 1.3).

```
     Global base register (GBR)
31                               0
|                                |
```

Figure 1.3 Global Base Register

Vector Base Register (VBR): Indicates the base address of the exception processing vector table (figure 1.4).

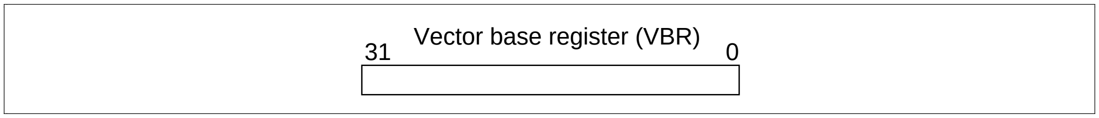


Figure 1.4 Vector Base Register

1.2.3 System Registers

Multiply and Accumulate Registers: Store the results of multiply and accumulate operations. The lower 10 bits of MACH are valid, and are sign-extended during read (figure 1.5).

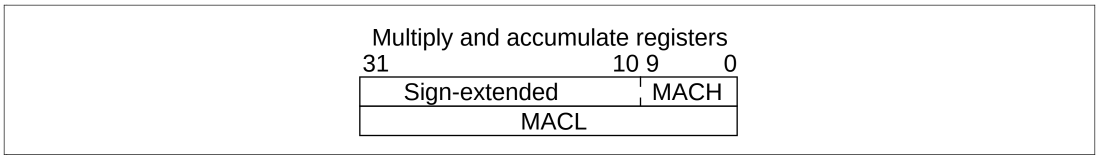


Figure 1.5 Multiply and Accumulate Registers

Procedure Register (PR): Stores a return address from a subroutine procedure (figure 1.6).

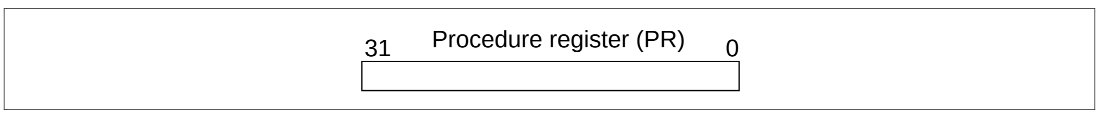


Figure 1.6 Procedure Register

Program Counter (PC): Indicates the fourth byte (second instruction) after the current execution (figure 1.7).

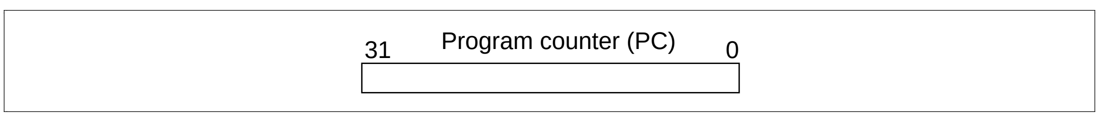


Figure 1.7 Program Counter

1.3 Data Formats

The SH7000 series CPU can handle byte (8 bits), word (16 bits), and longword (32 bits). The data formats in general registers and memory are given below.

1.3.1 Data Format in General Registers

The data length of general registers is always longword. Byte data and word data is sign-extended to a longword for storage in general registers (figure 1.8).

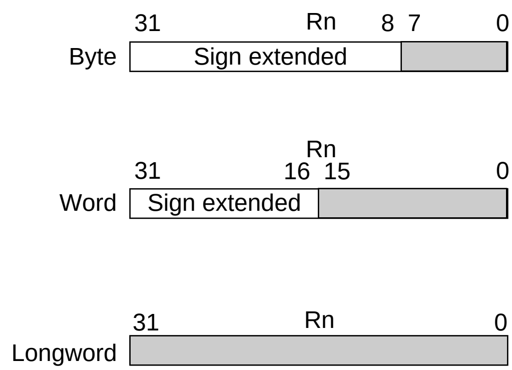


Figure 1.8 General Register Data Format

1.3.2 Data Format in Memory

Memory data formats are classified into byte, word and longword. Byte data can be accessed from any address, word data from address 2n, and longword from address 4n. If you attempt to access word data starting from an address other than 2n, or longword data starting from an address other than 4n, an address error will occur. In such cases, the data accessed cannot be guaranteed (figure 1.9).

More information on address errors is available in the appropriate SH hardware manual.

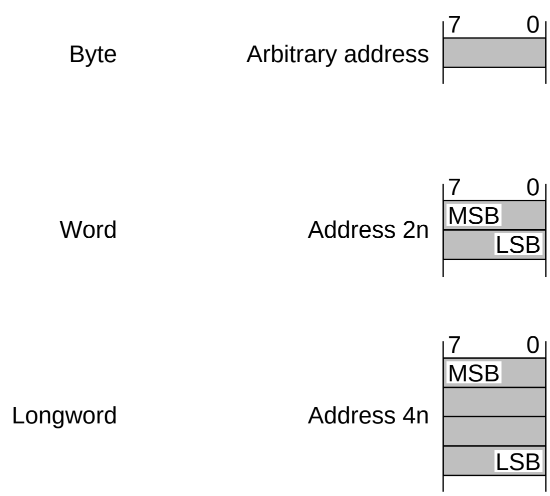


Figure 1.9 Memory Data Format

1.4 Instructions

1.4.1 Instruction Features

Instructions are RISC-type. Their features are detailed in this section.

16-Bit Fixed Length: All instructions are 16 bits long. This improves program code efficiency.

One Instruction/Cycle: Basic instructions can be executed in one cycle using the pipeline system. Instructions are executed in 50 ns at 20 MHz.

Data Length: The longword is the standard data length. Memory can be accessed in bytes, words or longwords. Constant data is sign-extended for arithmetic operations, zero-extended for logic operations, and handled as longword data (table 1.3).

Table 1.3 Sign Extension of Word Data

SH7000 Series CPU		Description	Example of Conventional CPU	
MOV. W	@(disp, PC), R1	Data is sign-extended to 32 bits, and R1 becomes H'00001234.	ADD. W	#H'1234,R0
ADD	R1,R0	Data is next operated on by an ADD instruction.		
	↓			
.data.w	H'1234			

Note: The address of the constant data is accessed by @(disp,PC).

Load-Store Architecture: Basic operations are executed between registers. For operations involving memory, data is loaded to the registers and executed (load-store architecture). Instructions such as AND that manipulate bits, however, are executed directly in memory.

Delayed Branch Instructions: Unconditional branch instructions are delayed. Pipeline disruption during branching is reduced by first executing the instruction that follows the branch instruction and then branching (table 1.4).

Table 1.4 Delayed Branch Instructions

SH7000 Series CPU		Description	Example of Conventional CPU	
BRA	TRGET	Executes an ADD before branching to TRGET	ADD.W	R1,R0
ADD	R1, R0		BRA	TRGET

Multiply/Accumulate Operations: Multiplication operations (16 × 16 → 32) are executed in 1–3 cycles, and (16×16 + 42 → 42) multiplication/accumulation operations are executed in 2–3 cycles by the 5-stage pipeline system and on-chip multiplier.

T Bit: The T bit in the status register (SR) changes according to the result of the comparison, and in turn is the condition (true/false) that determines if the program will branch. The number of instructions altering the T bit is kept to a minimum to improve processing speed (table 1.5).

Table 1.5 T Bit

SH7000 Series CPU		Description	Example of Conventional CPU	
CMP/GE	R1,R0	T bit is set when R0 ≥ R1.	CMP. W	R1,R0
BT	TRGET0	Program branches to TRGET0 when R0 ≥ R1.	BEG	TRGET0
BF	TRGET1	Program branches to TRGET1 when R0 < R1.	BLT	TRGET1
ADD	#1,R0	T bit is not changed by ADD.	SUB. W	#1,R0
CMP/EQ	#0,R0	T bit is set when R0 = 0.	BEQ	TRGET
BT	TRGET	Program branches when R0 = 0.		

Constant Data: Byte constant data is located in instruction code. Word or longword constant data is not input via instruction codes but is stored in a memory table. The memory table is accessed by a constant data transfer instruction (MOV) using the PC-relative addressing mode with displacement (table 1.6).

Table 1.6 Constant Data Accessing

Classification	SH 7000 Series CPU		Example of Conventional CPU	
8-bit constant	MOV	#H'12,R0	MOV. B	#H'12,R0
16-bit constant	MOV. W	@(disp,PC),R0	MOV. W	#H'1234,R0
		↓		
	.data.w	H'1234		
32-bit constant	MOV. L	@(disp,PC),R0	MOV. W	#H'12345678,R0
		↓		
	.data.w	H'12345678		

Note: The address of the constant data is accessed by @(disp,PC).

Absolute Address: When data is accessed by absolute address, the value already in the absolute address is placed in the memory table. Loading constant data when the instruction is executed transfers that value to the register, and the data is accessed in the register indirect addressing mode (table 1.7).

Table 1.7 Absolute Address Accessing

Classification	SH 7000 Series CPU		Example of Conventional CPU	
Absolute address	MOV. L	@(disp, PC), R1	MOV. B	#H'12345678,R0
	MOV. B	@R1, R0		
		↓		
	.data.l	H'12345678		

16-Bit/32-Bit Displacement: When data is accessed by 16-bit or 32-bit displacement, the preexisting displacement value is placed in the memory table. Loading immediate data when the instruction is executed transfers that value to the register, and the data is accessed in the indirect indexed register addressing mode (table 1.8).

Table 1.8 Displacement Accessing

Classification	SH 7000 Series CPU		Example of Conventional CPU	
16-bit displacement	MOV. W	@(disp,PC),R0	MOV. W	@(H'1234,R1),R2
	MOV. W	@(R0,R1),R2		
		↓		
	.DATA.W	H'1234		

1.4.2 Addressing Modes

The SH7000 series CPU supports the addressing modes shown in table 1.9. The addressing mode which can be used changes according to the instruction.

Table 1.9 SH7000 Series CPU Addressing Modes

Addressing Mode	Specified Object	Comments
Register direct	Register	—
Immediate	8-bit constant data	—
Register indirect	Address	—
Post-increment register indirect		—
Pre-decrement register indirect		—
Register indirect addressing with displacement		—
Register indirect indexed		—
Indirect GBR addressing with displacement		On-chip peripheral module register address
Indirect indexed GBR		On-chip peripheral module register address
PC-relative		Branch destination address
PC-relative with displacement		16- or 32-bit constant data address

Register Direct Mode: Used in specifying general registers R0–R15. Care should be taken since several instructions are restricted to the use of general register R0. Notation: Rn (figure 1.10).

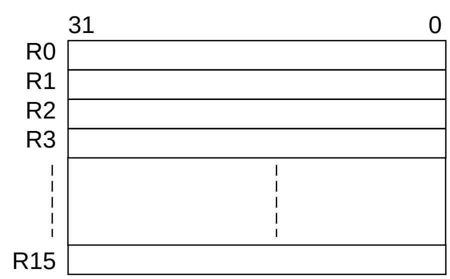


Figure 1.10 Register Direct Addressing Mode

Immediate Data Mode: Used in specifying 8-bit constant data located in the second byte of the instruction code. 8-bit constant data is extended according to the instruction as shown in table 1.10. Notation: #imm: 8.

Table 1.10 Immediate Data Extension

Instruction	Extension method
TST, AND, OR, XOR	Zero-extended to 32 bits
MOV, ADD, CMP/EQ	Sign-extended to 32 bits
TRAPA	Zero-extended to 32 bits and quadrupled

Register Indirect Mode: The contents of general register Rn are the effective address. Address setting to Rn according to data length in memory is shown in figure 1.11 and table 1.11. Notation: @Rn.

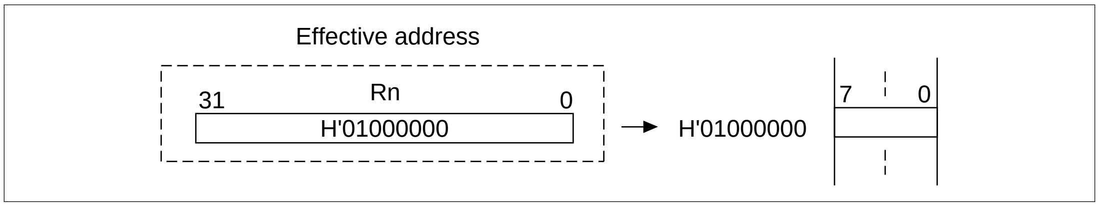


Figure 1.11 Register Indirect Address Setting Mode

Table 1.11 Register Indirect Addressing

Access Data Length	Address
Byte	Optional address
Word	Address 2n
Longword	Address 4n

Post-Increment Register Indirect and Pre-Decrement Register Indirect Modes: Specifies data addresses in stack or data tables.

- Post-increment register indirect mode

  The contents of register Rn is the effective address. After the address is specified, the accessed memory data length (byte = 1, word = 2, longword = 4) is automatically added to the contents of register Rn. This addressing mode is used for the return of data from the stack and for accessing data in a data table (figure 1.12). Notation: @Rn+.

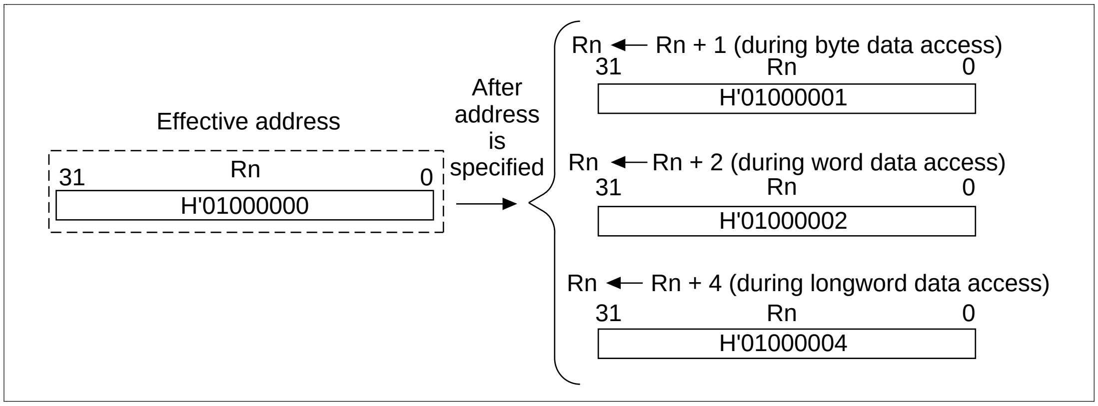


Figure 1.12 Post-Increment Register Indirect Mode

- Pre-decrement register indirect mode

  The content of register Rn is the effective address. Before specifying the address, the memory data length for access (byte = 1, word = 2, longword = 4) is automatically subtracted from the content of register Rn. This addressing mode is supported only by a destination operand and is used in saving data to the stack (figure 1.13). Notation: @-Rn.

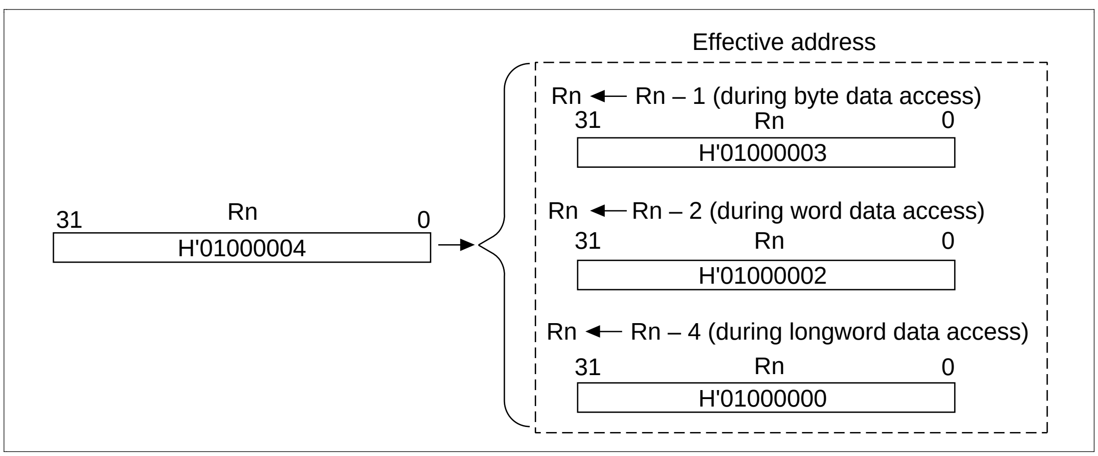


Figure 1.13 Pre-Decrement Register Indirect Mode

Register Indirect with Displacement and Register Indirect Indexed Modes: Add the relative interval with the specified destination address to the base address (contents of register Rn) in order to determine the effective address (figure 1.14). The contents of register Rn are the base address. Register Rn does not change due to addition of the relative interval (figure 1.15).

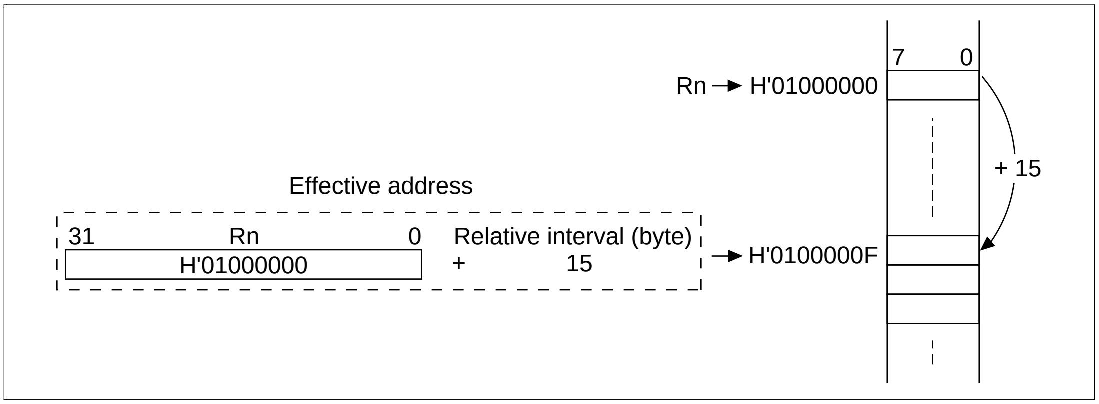


Figure 1.14 Effective Address for Register Indirect with Displacement and Register Indirect Indexed Modes

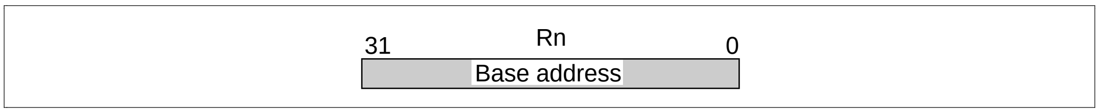


Figure 1.15 Base Address for Register Indirect with Displacement and Register Indirect Indexed Modes

- Register indirect with displacement mode
  The relative interval is indicated by a 4-bit displacement. After zero-extension, the 4-bit displacement remains the same for a byte operation, is doubled for a word operation, and quadrupled for a longword operation. The range of relative intervals is shown in table 1.12.
  Notation: @(disp:4,Rn)

Table 1.12 Access Data Relative Intervals

Access Data Length	Relative Interval (bytes)
Byte	0 to +15
Word	Doubled 0 to +30
Longword	Quadrupled 0 to +60

- Register indirect indexed mode

  The relative interval is set in a general register. Note that the general register must be R0. The relative interval is used when address access is unattainable with a 4-bit displacement (figure 1.16). Notation: @(R0,Rn).

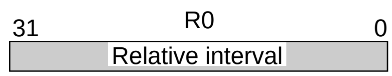


Figure 1.16 Register Indirect Indexed Relative Interval

Indirect GBR with Displacement and Indirect Indexed GBR Modes: Used to specify the addresses of on-chip peripheral modules. As shown in figure 1.17, these addressing modes add the relative interval with the specified destination address to the GBR base address to determine the effective address. Indirect GBR with displacement supports the MOV instruction, and indirect indexed GBR supports the logic operation instructions ADD, OR, NOT, and TST. The value of the relative interval plus base address is the effective address.

The base address is the contents of GBR. The contents of GBR do not change due to addition of the relative interval (figure 1.18).

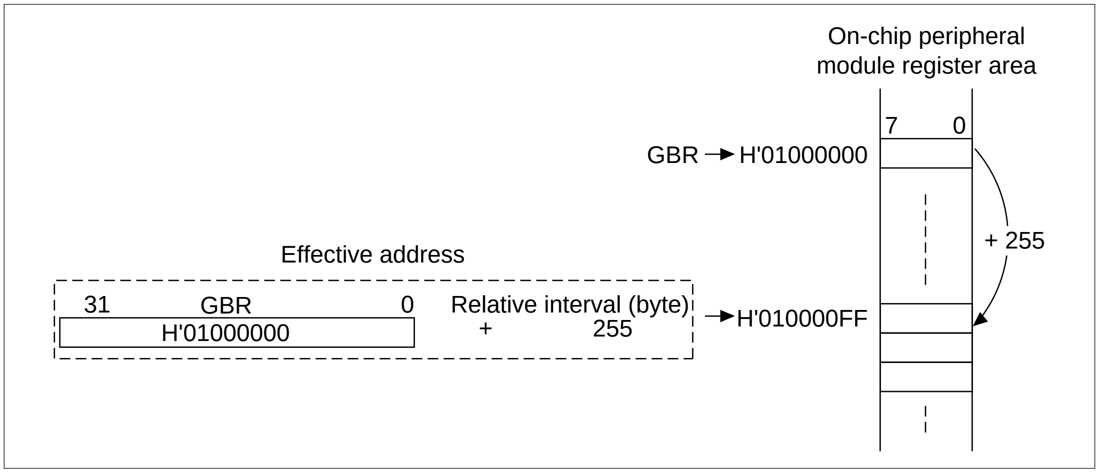


Figure 1.17 Effective Address for Indirect GBR with Displacement and Indirect Indexed GBR Modes

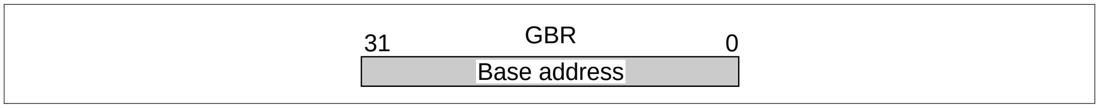


Figure 1.18 Base Address for Indirect GBR with Displacement and Indirect Indexed GBR Modes

- Indirect GBR with displacement mode

  The relative interval is indicated by an 8-bit displacement. After zero-extension, the 8-bit displacement remains the same for a byte operation, is doubled for a word operation, and quadrupled for a longword operation. Relative intervals range as shown in table 1.13. Notation: @(disp:8,GBR)

Table 1.13 Access Data Relative Intervals

Access Data Length	Relative Interval (bytes)
Byte	0 to +255
Word	Doubled 0 to +510
Longword	Quadrupled 0 to +1020

- Indirect indexed GBR mode

  The relative interval is set in a general register. Note that the general register must be R0. The relative interval is used when address access is unattainable with an 8-bit displacement (figure 1.19). Notation: @(R0,GBR)

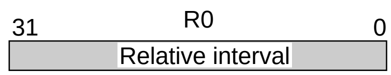


Figure 1.19 Indirect Indexed GBR Relative Interval

PC-Relative and PC-Relative with Displacement Modes: Used in specifying addresses of the program area. As shown in figure 1.20, these addressing modes add the relative interval with the specified destination address to the contents of the PC to determine the effective address. PC-relative is used in specifying the branch destination address for branch instructions BF, BT, BRA, and BSR. PC-relative with displacement is used in specifying the addresses of 16-bit or 32-bit constant data located in the program area.

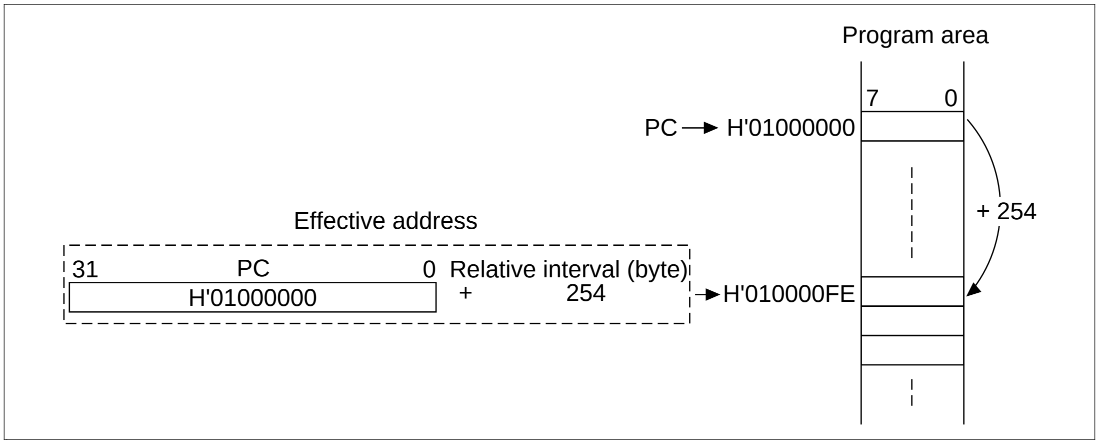


Figure 1.20 Effective Address for PC-Relative and PC-Relative with Displacement Modes

- PC-relative mode

  The base address is the contents of the PC (address after the second instruction). The PC contents become the branch address by adding the relative interval (figure 1.21).

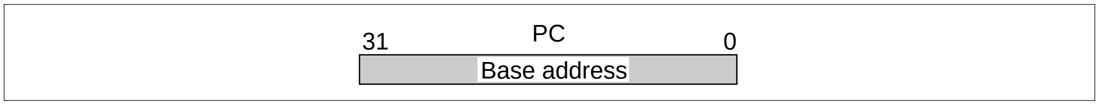


Figure 1.21 Base Address for PC-Relative Mode

- Notation: disp: 8

  The relative interval is indicated by an 8-bit displacement. The 8-bit displacement is doubled after sign-extension, so the relative interval range becomes –256 to +254.
- Notation: disp: 12

  The relative interval is indicated by a 12-bit displacement. The 12-bit displacement is doubled after sign-extension, so the relative interval range becomes –4096 to +4094.
- PC-relative with displacement mode
- Notation: @(disp:8,PC)

  The base address is the contents of the PC (address after the second instruction). The PC contents are not changed due to addition of the relative interval. If the access data length is longword, the base address is the PC contents with the lower two bits masked (figure 1.22).

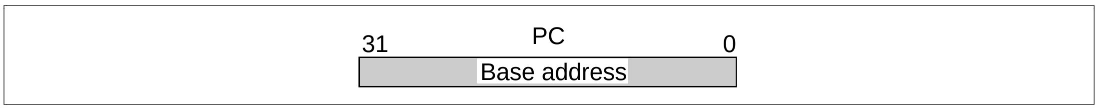


Figure 1.22 Base Address for PC-Relative with Displacement Mode

The relative interval is indicated by an 8-bit displacement. After zero-extending the 8-bit displacement, it is doubled for a word operation, and quadrupled for a longword operation. Relative intervals range as shown in table 1.14.

Table 1.14 Access Data Relative Intervals

Access Data Length	Relative Interval (bytes)
Word	Doubled 0 to +510
Longword	Quadrupled 0 to +1020

1.4.3 Instruction Types

The instruction set is classified according to functions in table 1.15. See section 1.4.4, Description of Instructions, for more information on each instruction.

Table 1.15 Classification of Instructions

Classification	Operation code	Function
Data transfer	MOV	Data transfer
	MOVA	Effective address transfer
	MOVT	T bit transfer
	SWAP	Swap of upper and lower bytes
	XTRCT	Extraction of middle of connected registers
Arithmetic operations	ADD	Binary addition
	ADDC	Binary addition with carry
	ADDV	Binary addition with overflow check
	CMP/cond	Comparison
	DIV1	Division
	DIV0S	Initialization of signed division
	DIV0U	Initialization of unsigned division
	EXTS	Sign extension
	EXTU	Zero extension
	MAC	MULTIPLY/accumulate
	MULS	Signed multiplication
	MULU	Unsigned multiplication
	NEG	Negation
	NEGC	Negation with borrow
	SUB	Binary subtraction
	SUBC	Binary subtraction with borrow
	SUBV	Binary subtraction with underflow
Logic operations	AND	Logical AND
	NOT	Bit inversion
	OR	Logical OR
	TAS	Memory test and bit set
	TST	Logical AND and T bit set
	XOR	Exclusive OR

Table 1.15 Classification of Instructions (cont)

Classification	Operation code	Function
Shift	ROTL	One-bit left rotation
	ROTR	One-bit right rotation
	ROTCL	One-bit left rotation with T bit
	ROTCR	One-bit right rotation with T bit
	SHAL	One-bit arithmetic left shift
	SHAR	One-bit arithmetic right shift
	SHLL	One-bit logical left shift
	SHLLn	n-bit logical left shift
	SHLR	One-bit logical right shift
	SHLRn	n-bit logical right shift
Branch	BF	Conditional branch (T = 0)
	BT	Conditional branch (T = 1)
	BRA	Unconditional branch
	BSR	Branch to subroutine procedure
	JMP	Unconditional branch
	JSR	Branch to subroutine procedure
	RTS	Return from subroutine procedure
System control	CLRT	Clear T bit
	CLRMAC	Clear MAC register
	LDC	Load to control register
	LDS	Load to system register
	NOP	No operation
	RTE	Return from exception processing
	SETT	Set T bit
	SLEEP	Shift into power-down mode
	STC	Store from control register
	STS	Store from system register
	TRAPA	Trap exception processing

Table 1.16 Data Transfer Instruction: MOV

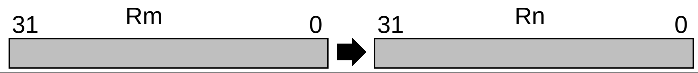
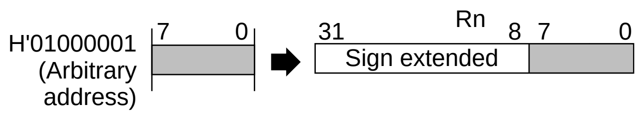
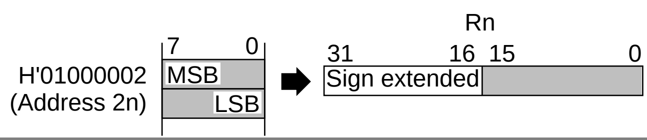
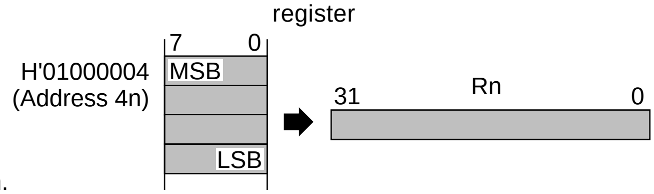
Operand Size	Source Operand	Destination Operand	Description
—	Rm (R0–R15)	Rn (R0–R15)	Transfers general register Rm contents to general register Rn. 
B	@Rm @Rm+ @(disp:4, Rm) @(R0, Rm) @(disp:8, GBR)	Rn; instruction restricted to R0 when the source operand is @(disp:4, Rm) or @(disp:8, GBR)	Transfers byte data in memory (optional address) to general register Rn. Byte data is sign-extended to 32-bit data in general register Rn. 
W			Transfers word data in memory (address 2n) to general register Rn. Word data is sign-extended to 32-bit data in general register Rn. 
L		Rn(R0–R15)	Transfers longword data in memory (address 4n) to general register  Rn.

Table 1.16 Data Transfer Instruction: MOV (cont)

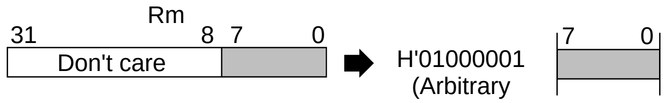
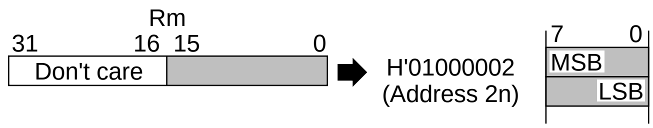
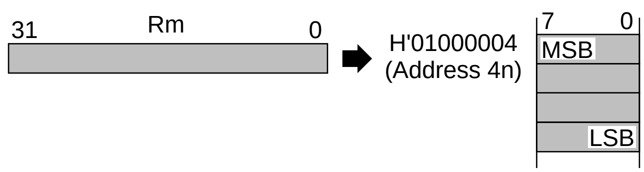
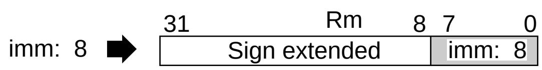
Operand Size	Source Operand	Destination Operand	Description
B	Rm; instruction is restricted to R0 when the destination operand is @(disp: 4, Rn) or @(disp: 8, GBR)	@Rn @-Rn @(disp: 4, Rn) @(R0,Rn) @(disp: 8, GBR)	Transfer of lower 8 bits of data in general register Rm to memory (arbitrary address). 
W			Transfer of lower 16 bits of data in general register Rm to memory (address 2n). 
L	Rm (R0–R15)		Transfer of contents of general register Rm to memory (address 4n). 
—	#imm: 8	Rn (R0–R15)	Transfer of 8-bit constant data to general register Rn. 8-bit constant data sign-extended to 32-bit data in general register Rn. 

Table 1.17 Data Transfer Instructions: MOVA, MOVT

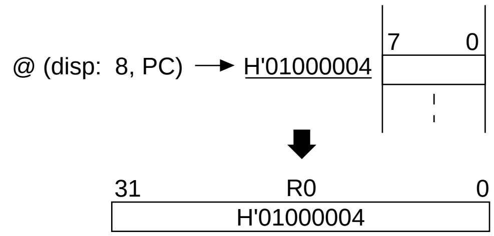
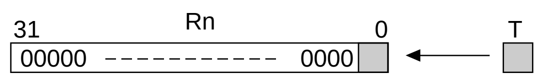
Operand Size	Source Operand	Destination Operand	Description
—	@(disp:8, PC) or symbol	Limited to R0	Mnemonic: MOVA. Stores address (@(disp:8,PC)) in general register R0. With processing using a data table, the data table start address need not be held in the constant data (32 bits). 
—	—	Rn (R0–R15)	Mnemonic: MOVT. Transfers T bit contents to bit 0 of general register Rn. Bits 1–31 become 0. 

Table 1.18 Data Transfer Instruction: SWAP

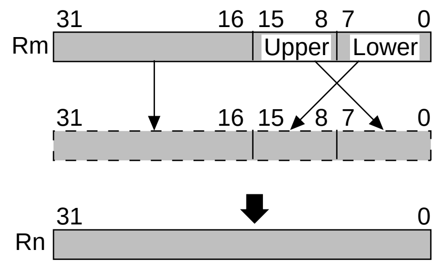
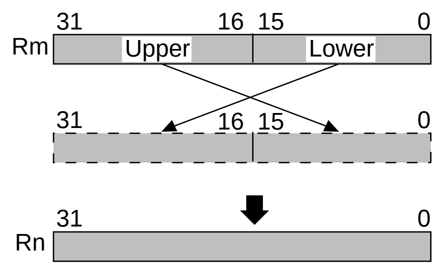
Operand Size	Source Operand	Destination Operand	Description
B	Rm (R0–R15)	Rn (R0–R15)	Leaves upper 16 bits of register Rm unchanged, swaps the upper and lower bytes of the lower 16 bits, and stores the contents in register Rn. 
W			Swaps the upper and lower sixteen bits of register Rm and stores the contents in register Rn. 

Table 1.19 Data Transfer Instruction: XTRCT

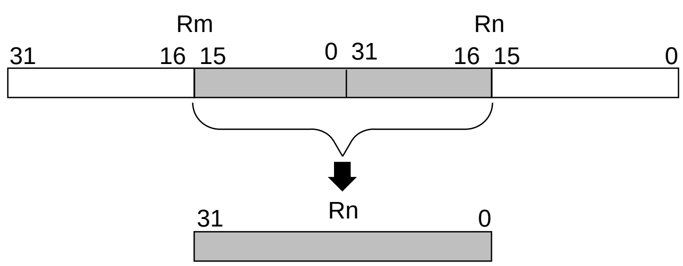
Operand Size	Source Operand	Destination Operand	Description
—	Rm (R0–R15)	Rn (R0–R15)	Transfers the middle 64 bits, comprising Rm's upper 32 bits and Rn's lower 32 bits, to Rn. 

Table 1.20 Arithmetic Operation Instruction: ADD

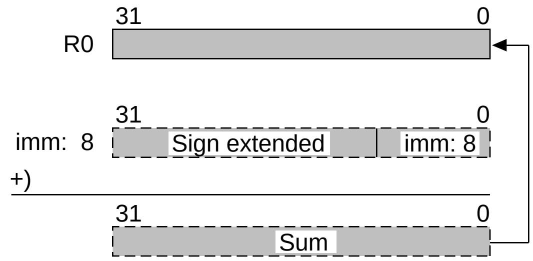
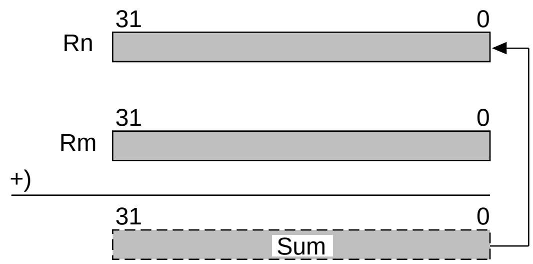
Operand Size	Source Operand	Destination Operand	Description
—	#imm: 8	limited to R0	Adds 8-bit constant data to contents of general register R0. The result of addition is returned to R0. 8-bit constant data is sign-extended in the range –128 to +127, so this instruction can be used for subtraction also. 
	Rm (R0–R15)	Rn (R0–R15)	Adds contents of register Rm to Rn. The sum is returned to Rn. 

Table 1.21 Arithmetic Operation Instruction: ADDC

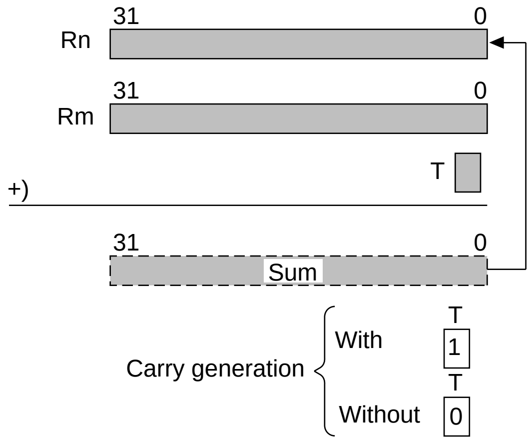
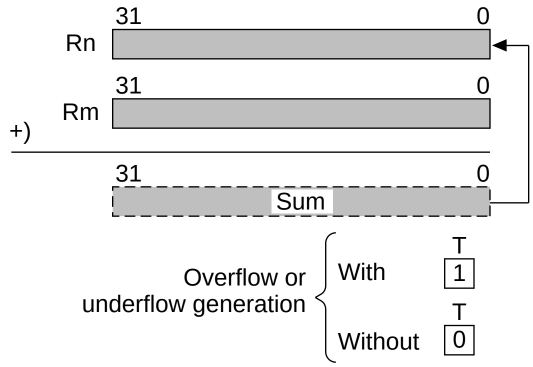
Operand Size	Source Operand	Destination Operand	Description
—	Rm (R0–R15)	Rn (R0–R15)	Adds the contents of register Rm and the T bit contents to general register Rn. The sum is returned to Rn. The presence/absence of a carry is indicated in the T bit (with carry: T = 1, without carry: T = 0). Used in additions that exceed 32 bits. 

Table 1.22 Arithmetic Operation Instruction: ADDV

Operand Size	Source Operand	Destination Operand	Description
—	Rm (R0–R15)	Rn (R0–R15)	Adds the contents of register Rm to the contents of Rn. The sum is returned to Rn. Used in the addition of signed values.



Note: The presence or absence of overflow or underflow is indicated in the T bit (with: T = 1, without: T = 0). Use table 1.23 to determine whether underflow or overflow has occurred.

Table 1.23 ADDV Overflow/Underflow

Register Rn (Augend)	Register Rm (Addend)	Register Rm (Result)	Overflow/Underflow
Positive	Positive	Positive	—
		Negative	Overflow
	Negative	Positive	—
		Negative	—
Negative	Positive	Positive	—
		Negative	—
	Negative	Positive	Underflow
		Negative	—

Table 1.24 Arithmetic Operation Instructions: CMP/cond

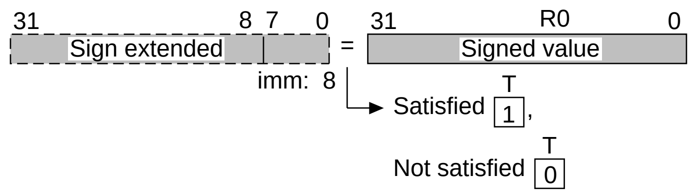
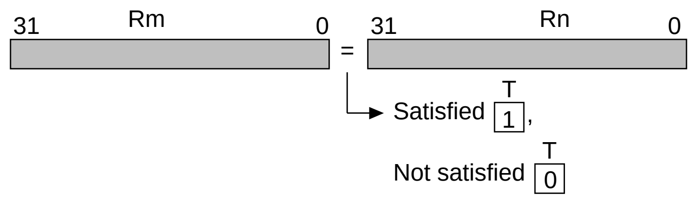
Operand Size	Source Operand	Destination Operand	Description
—	#imm: 8	Limited to R0	Mnemonic: CMP/EQ Comparison with the condition imm:8 (–128 to +127) = R0 (signed value) and result indication in the T bit. 
	Rm (R0–R15)	Rn (R0–R15)	Mnemonic: CMP/EQ Comparison with the condition Rm = Rn, and result indication in the T bit. 

Table 1.24 Arithmetic Operation Instructions: CMP/cond (cont)

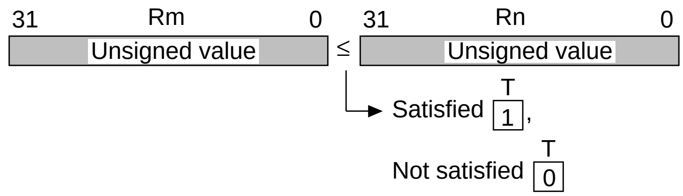
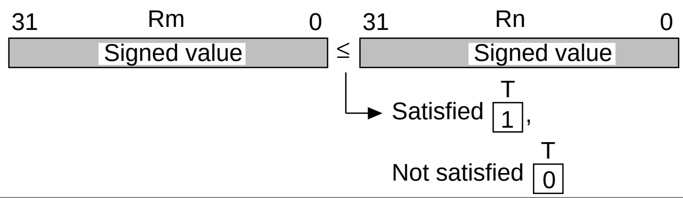
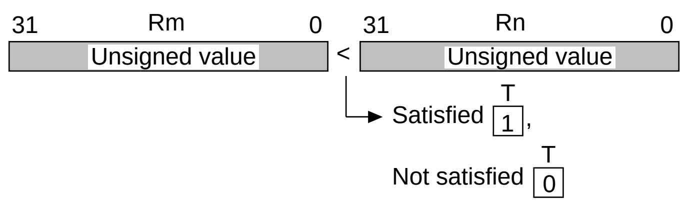
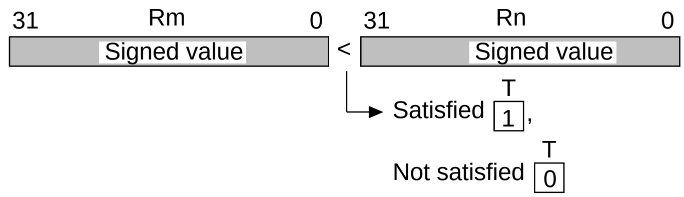
Operand Size	Source Operand	Destination Operand	Description
—	Rm (R0–R15) (cont)	Rn (R0–R15) (cont)	Mnemonic: CMP/HS Comparison with the condition Rm (unsigned) ≤ Rn (unsigned), and result indication in the T bit. 
			Mnemonic: CMP/GE Comparison with the condition Rm (signed) ≤ Rn (signed) and result indication in T bit. 
			Mnemonic: CMP/HI Comparison with the condition Rm (unsigned) < Rn (unsigned), and result indication in T bit. 

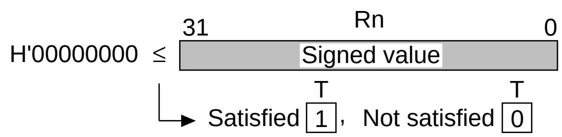
Table 1.24 Arithmetic Operation Instructions: CMP/cond (cont)

<table>
<tr><th>Operand Size</th><th>Source Operand</th><th>Destination Operand</th><th>Description</th></tr>
<tr><td rowspan="4">—</td><td rowspan="4">Rm (R0–R15) (cont)</td><td rowspan="4">Rn (R0–R15) (cont)</td><td>Mnemonic: CMP/GT

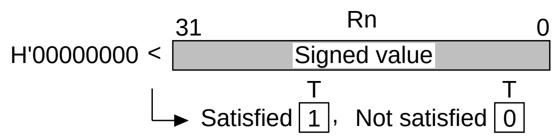
Comparison with the condition Rm (signed) < Rn (signed), and result indication in T bit.


</td></tr>
<tr><td>Mnemonic: CMP/PZ

Comparison with the condition 0 ≤ Rn (signed), and result indication in T bit.


</td></tr>
<tr><td>Mnemonic: CMP/PL

Comparison with the condition 0 < Rn (signed), and result indication in T bit.


</td></tr>
<tr><td>Mnemonic: CMP/STR

Comparison of Rm (HH) = Rn (HH'), Rm (LH) = Rn (LH'), Rm(HL) = Rn (HL') and Rm (LL) = Rn (LL'). If one of the conditions is satisfied, the T bit is set. If none of the conditions are satisfied, the T bit is cleared.

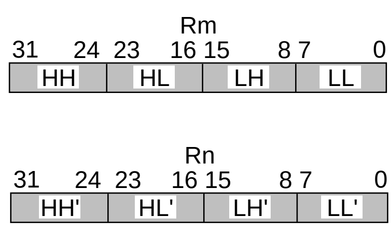

</td></tr>
</table>

Table 1.25 Arithmetic Operation Instructions: DIV0S, DIV0U, DIV1

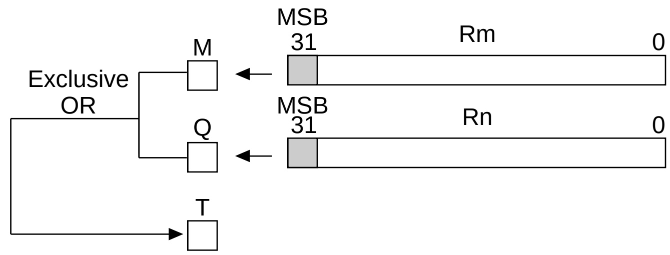
Operand Size	Source Operand	Destination Operand	Description
—	—	—	Mnemonic: DIV0U Initialization of unsigned division. Clears the M bit, Q bit, and T bit of the SR. M Q T $\boxed{0}$ $\boxed{0}$ $\boxed{0}$
	Rm (R0–R15)	Rn (R0–R15)	Mnemonic: DIV0S Initialization of signed division. Stores the contents of the MSB of register Rm (divisor) in the M bit, the contents of register Rn's (dividend) MSB in the Q bit, and stores the sign of the quotient from an exclusive OR of M bit and Q bit in the T bit. 

Table 1.25 Arithmetic Operation Instructions: DIV0S, DIV0U, DIV1 (cont)

Operand Size	Source Operand	Destination Operand	Description
—	Rm	Rn	Mnemonic: DIV1 Performs one-step division of register Rn’s 32-bit contents (dividend) by the contents of Rm (divisor). One-step division shifts dividend one bit left, subtracts the absolute value of the divisor (addition when the dividend is negative), and determines the quotient (1 digit) in the T bit depending on the negative/positive result. As shown here, this instruction repeatedly divides through the number of bits of the divisor. Do not rewrite the M, Q, or T bits during this repetition. Refer to the software application examples for details of the division sequence.

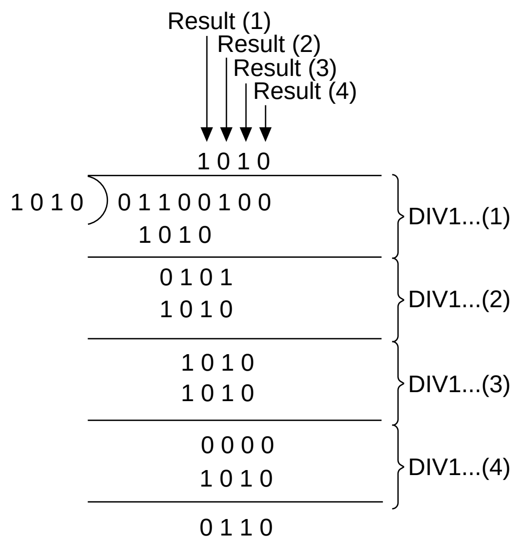


Table 1.26 Arithmetic Operation Instruction: EXTU

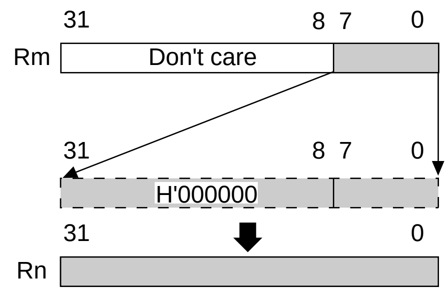
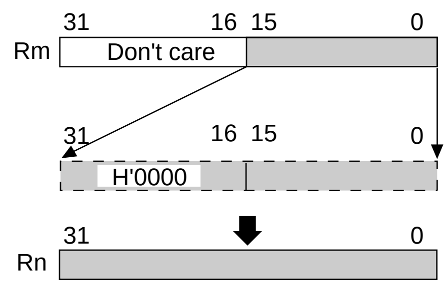
<table>
<tr><th>Operand Size</th><th>Source Operand</th><th>Destination Operand</th><th>Description</th></tr>
<tr><td>B</td><td>Rm (R0–R15)</td><td>Rn (R0–R15)</td><td>Stores Rm register lower 8 bits, zero-extended to 32 bits, in Rn.

</td></tr>
<tr><td>W</td><td></td><td></td><td>Stores Rm register lower 16 bits, zero-extended to 32 bits, in Rn.

</td></tr>
</table>

Table 1.27 Arithmetic Operation Instruction: EXTS

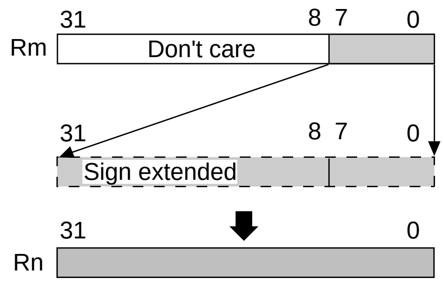
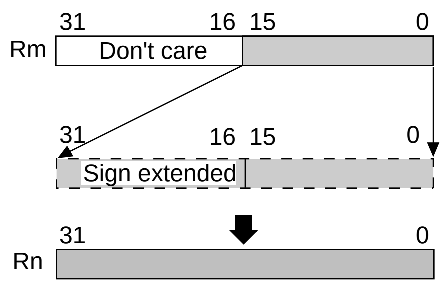
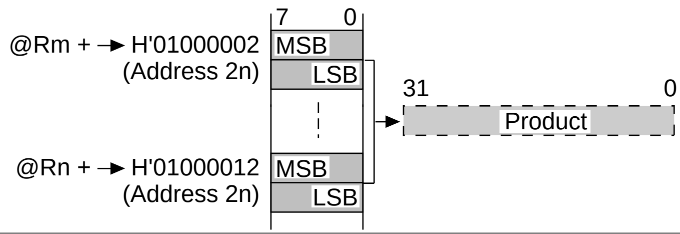
Operand Size	Source Operand	Destination Operand	Description
B	Rm (R0–R15)	Rn (R0–R15)	Sign extends register Rm’s lower 8 bits to 32 bits (transfers contents of bit 7 to bits 8–31) and stores contents in register Rn. 
W			Sign extends register Rm’s lower 16 bits to 32 bits (transfers contents of bit 7 to bits 8–31) and stores contents in register Rn. 

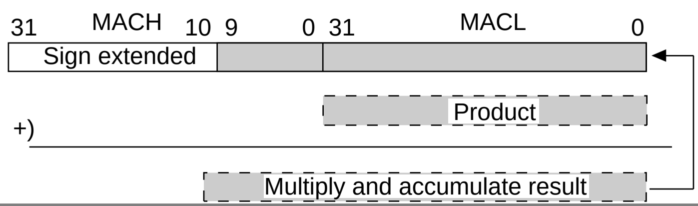
Table 1.28 Arithmetic Operation Instruction: MAC

Operand Size	Source Operand	Destination Operand	Description
W	@Rm+	@Rn+	Multiplies @Rm+ memory contents (16 bits) by @Rn+ memory contents (16 bits), adds signed result to MAC, and returns sum to MAC. Note that range of results varies with SR's S bit contents.



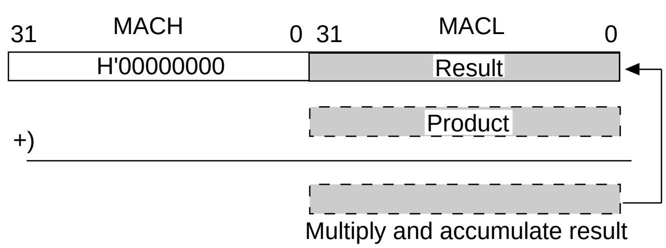
When the S bit = 0:

Uses MACH/MACL. Stores lower 32 bits of result in MACL and upper 10 bits in MACH. Upper 22 bits of MACH is sign extended.

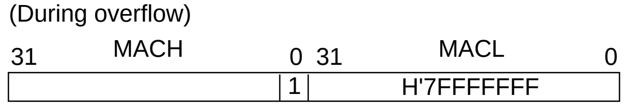


When S bit = 1 (saturation operation):

Only MACL register is valid, and result is limited to H'80000000–H'7FFFFFFF. Also, if an overflow or underflow is generated, MACH's LSB is set to 1, and H'7FFFFFFF is stored in the MACL for an overflow, and H'80000000 is stored for an underflow.



(During overflow)



(During underflow)

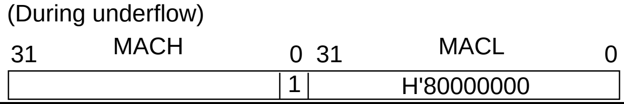


Table 1.29 Arithmetic Operation Instructions: MULU, MULS

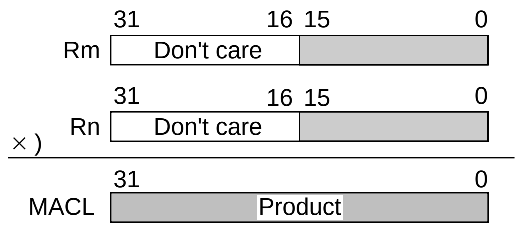
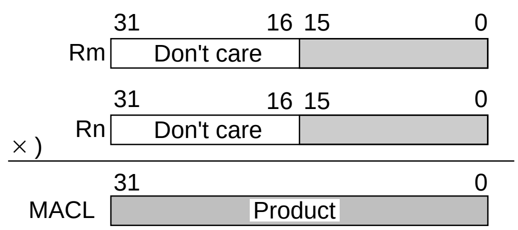
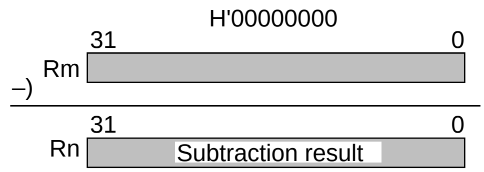
Operand Size	Source Operand	Destination Operand	Description
—	Rm (R0–R15)	Rn (R0–R15)	Mnemonic: MULU Performs unsigned multiplication of the contents of the lower 16 bits of register Rm and lower 16 bits of register Rn, and stores the multiplication result (32 bits) in the MACL. The contents of registers Rm and Rn do not change due to multiplication. 
			Mnemonic: MULS Performs signed multiplication of the contents of the lower 16 bits of register Rm and lower 16 bits of register Rn, and stores the multiplication result (32 bits) in the MACL. The contents of registers Rm and Rn do not change due to multiplication. 

Table 1.30 Arithmetic Operation Instruction: NEG, NEGC

<table>
<tr><th>Operand Size</th><th>Source Operand</th><th>Destination Operand</th><th>Description</th></tr>
<tr><td>—</td><td>Rm (R0–R15)</td><td>Rn (R0–R15)</td><td>

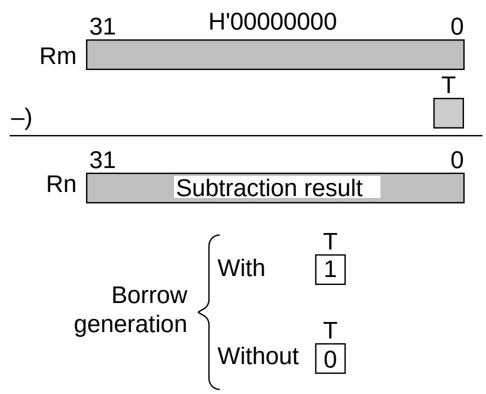
Mnemonic: NEG

Subtracts the contents of register Rm from H'00000000 and stores the result in Rn. Used in negation of data of 32 bits or less.



Mnemonic: NEGC

Subtracts the contents of register Rm and the T bit contents from H'00000000 and stores the result in Rn. Also indicates whether a borrow was generated in the T bit (with borrow: T=1, without borrow: T=0). Used in negation of data exceeding 32 bits.



</td></tr>
</table>

Table 1.31 Arithmetic Operation Instructions: SUB, SUBC, SUBV

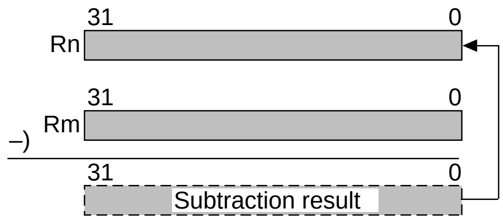
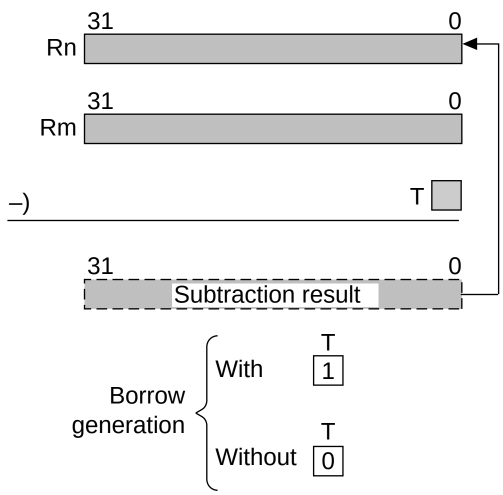
Operand Size	Source Operand	Destination Operand	Description
—	Rm (R0–R15)	Rn (R0–R15)	Mnemonic: SUB Subtracts the contents of register Rm from the contents of register Rn and returns the subtraction result to register Rn. 
			Mnemonic: SUBC Subtracts the contents of register Rm and the T bit contents from the contents of register Rn. Returns the subtraction result to register Rn. Also indicates presence/absence of a borrow in the T bit (with borrow: T = 1, without borrow: T =0). This instruction is used in subtraction of data that exceed 32 bits. 

Table 1.31 Arithmetic Operation Instructions: SUB, SUBC, SUBV (cont)

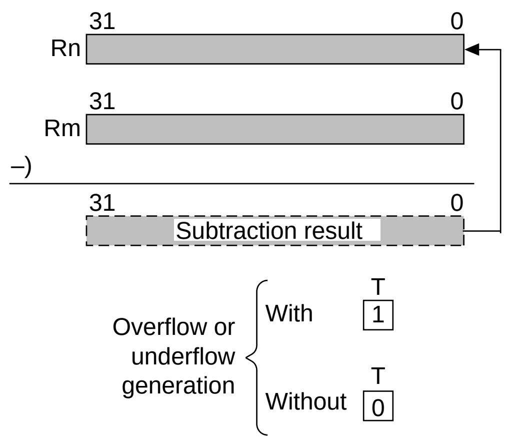
Operand Size	Source Operand	Destination Operand	Description
—	Rm (R0–R15) (cont)	Rn (R0–R15) (cont)	Mnemonic: SUBV Subtracts the contents of register Rm from the contents of register Rn and returns the subtraction result to register Rn. Also indicates presence or absence of an overflow or underflow in the T bit (with: T = 1; without: T = 0). Determine whether overflow or underflow has occurred using table 1.32. This instruction is used in the subtraction of signed values. 

Table 1.32 SUBV Overflow/Underflow

Register Rn (Minuend)	Register Rm (Subtracter)	Register Rm (Result)	Overflow/Underflow
Positive	Positive	Positive	—
		Negative	—
	Negative	Positive	Overflow
		Negative	—
Negative	Positive	Positive	Underflow
		Negative	—
	Negative	Positive	—
		Negative	—

Table 1.33 Logic Operation Instructions: AND, OR, XOR

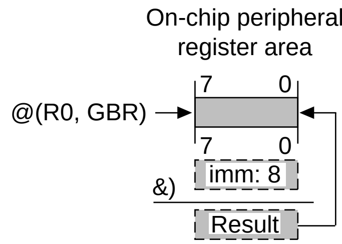
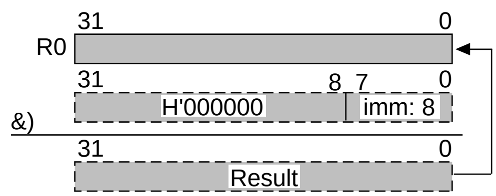
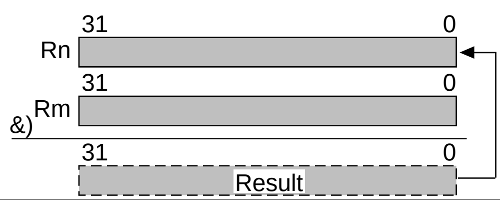
Operand Size	Source Operand	Destination Operand	Description
B	#imm: 8	@(R0,GBR)	Performs logic operations on 8-bit constant data with @(R0,GBR) memory contents (8 bits) and returns results to @(R0,GBR). Convenient for use in bit manipulation (bit set, bit clear, and bit inversion) of on-chip peripheral module register area. 
—	#imm: 8	Limited to R0	Performs logic operations of register R0 contents and 8-bit constant data (zero-extended) and returns the result to R0. 
	Rm (R0–R15)	Rn (R0–R15)	Performs logic operations on contents of Rm and Rn and returns result to Rn. 

Table 1.34 Logic Operation Instruction: NOT

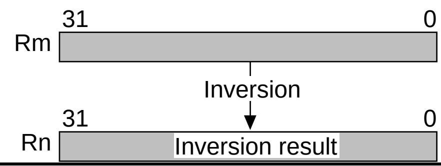
Operand Size	Source Operand	Destination Operand	Description
—	Rm (R0–R15)	Rn (R0–R15)	Performs bit inversion on register Rm and stores result in Rn. 

Table 1.35 Logic Operation Instruction: TST

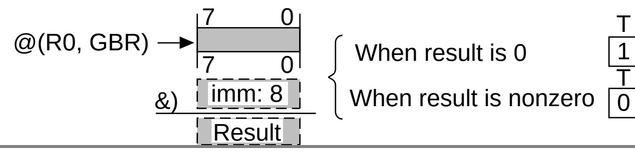
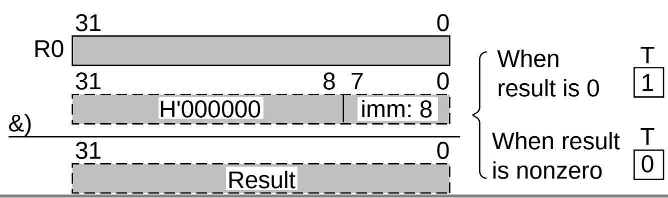
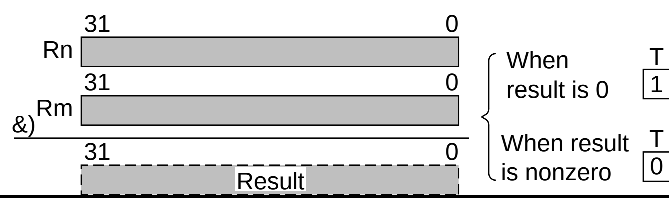
Operand Size	Source Operand	Destination Operand	Description
B	#imm: 8	@(R0, GBR)	Logically ANDs 8-bit constant data and @(R0,GBR) memory contents, sets T bit when result is 0, and clears T bit when result is nonzero. Result is not returned anywhere. Convenient for use in bit testing of on-chip peripheral module registers. 
—	#imm: 8	Limited to R0	Logically ANDs 8-bit constant data (zero-extended) and contents of register R0, sets T bit when result is 0, and clears T bit when result is nonzero. The result is not returned anywhere. 
	Rm (R0–R15)	Rn (R0–R15)	Logically ANDs contents of Rm and Rn, sets T bit when result is 0, and clears T bit when result is nonzero. Result is not returned. 

Table 1.36 Logic Operation Instruction: TAS

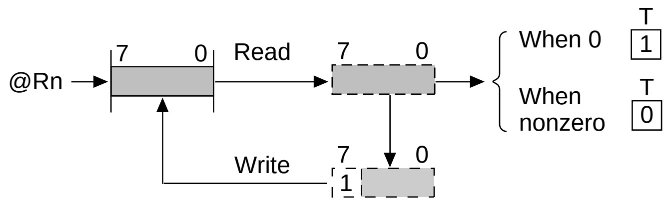
Operand Size	Source Operand	Destination Operand	Description
—	—	@Rn	Reads the @Rn memory contents (8 bits), sets the T bit when 0 and clears the T bit when not zero. Then sets bit 7 and writes it in. The bus is not released during execution of this instruction. 

Table 1.37 Shift Instructions: ROTL, ROTR, ROTCL, ROTCR

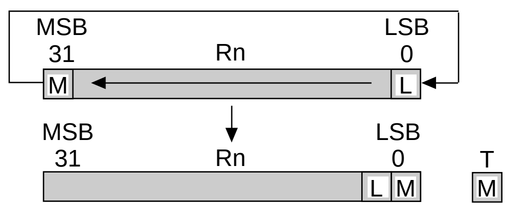
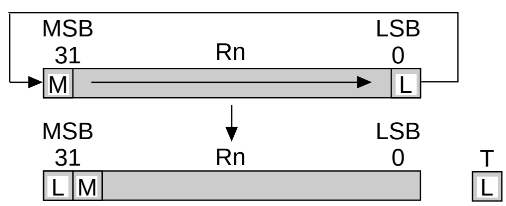
Operand Size	Source Operand	Destination Operand	Description
—	—	Rn (R0–R15)	Mnemonic: ROTLRotates the contents of register Rn one bit to the left. Also sets the T bit to the MSB shifted out by the rotation. 
			Mnemonic: ROTR Rotates the contents of register Rn one bit to the right. Also sets the T bit to the LSB shifted out by the rotation. 

Table 1.37 Shift Instructions: ROTL, ROTR, ROTCL, ROTCR (cont)

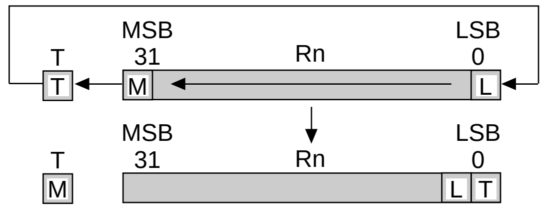
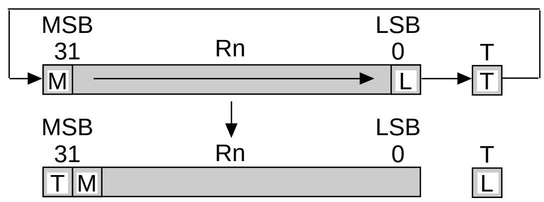
Operand Size	Source Operand	Destination Operand	Description
—	—	Rn (R0–R15) (cont)	Mnemonic: ROTCL Rotates contents of Rn including the T bit contents one bit left. 
			Mnemonic: ROTCR Rotates the contents of Rn including the T bit contents one bit to the right. 

Table 1.38 Shift Instructions: SHAL, SHAR, SHLL, SHLR

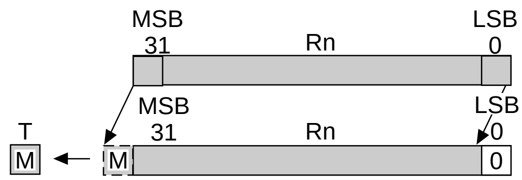
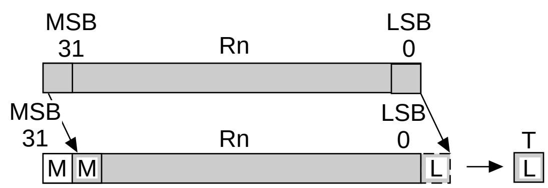
Operand Size	Source Operand	Destination Operand	Description
—	—	Rn (R0–R15) (cont)	Mnemonic: SHAL Shifts the contents of register Rn one bit arithmetically to the left. Stores the contents of the shifted-out MSB in the T bit and stores a 0 in the empty LSB. This instruction is used in left shifting of signed values. 
			Mnemonic: SHAR Shifts the contents of register Rn one bit arithmetically to the right. Stores the contents of the shifted-out LSB in the T bit and stores the MSB contents in the available MSB space. Used in right shifting of signed values. 

Table 1.38 Shift Instructions: SHAL, SHAR, SHLL, SHLR (cont)

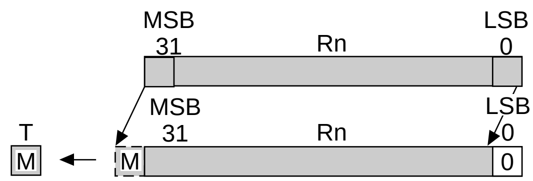
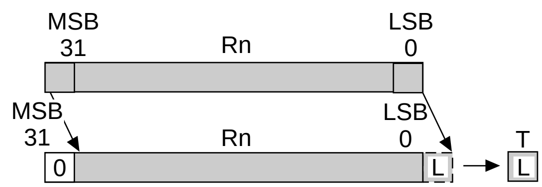
Operand Size	Source Operand	Destination Operand	Description
—	—	Rn (R0–R15)	Mnemonic: SHLL Shifts the contents of register Rn one bit arithmetically to the left. Stores the contents of the shifted-out MSB in the T bit and stores a 0 in the empty LSB. 
			Mnemonic: SHLR Shifts the contents of register Rn one bit arithmetically to the right. Stores the contents of the shifted-out LSB in the T bit and stores a 0 in the empty MSB. 

Table 1.39 Shift Instructions: SHLL2, SHLR2, SHLL8, SHLR8, SHLL16, SHLR16

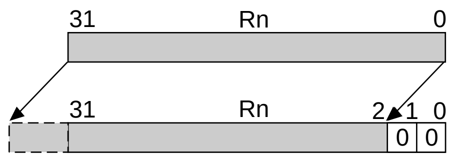
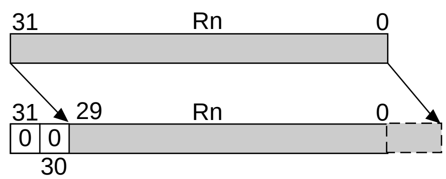
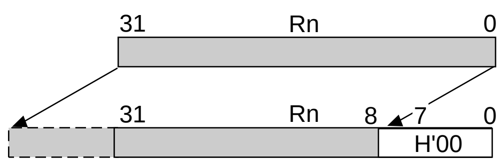
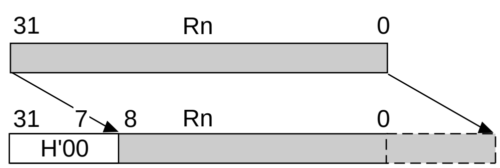
Operand Size	Source Operand	Destination Operand	Description
—	—	Rn (R0–R15)	Mnemonic: SHLL2 Shifts the contents of register Rn two bits logically to the left. Discards the shifted-out bits and stores a 0 in the empty bits. 
			Mnemonic: SHLR2 Shifts the contents of register Rn two bits logically to the right. Discards the shifted-out bits and stores a 0 in the empty bits. 
			Mnemonic: SHLL8 Shifts the contents of register Rn eight bits logically to the left. Discards the shifted-out bits and stores a 0 in the empty bits. 
			Mnemonic: SHLR8 Shifts the contents of register Rn eight bits logically to the right. Discards the shifted-out bits and stores a 0 in the empty bits. 

Table 1.39 Shift Instructions: SHLL2, SHLR2, SHLL8, SHLR8, SHLL16, SHLR16 (cont)

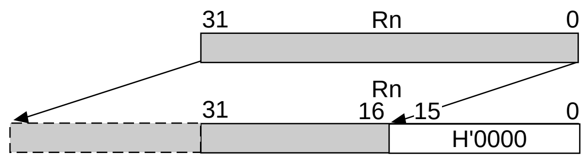
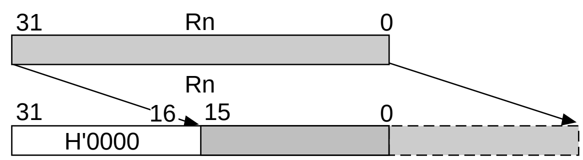
Operand Size	Source Operand	Destination Operand	Description
—	—	Rn (R0–R15) (cont)	Mnemonic: SHLL16 Shifts the contents of register Rn sixteen bits logically to the left. Discards the shifted-out bits and stores a 0 in the empty bits. 
			Mnemonic: SHLR16 Shifts the contents of register Rn sixteen bits logically to the right. Discards the shifted-out bits and stores a 0 in the empty bits. 

Table 1.40 Branch Instructions: BF, BT, BRA, BSR, JMP, JSR, RTS

Mnemonic	Operand	Description
BF	disp: 8 or branch destination symbol	Conditional branch instruction that accesses the T bit. Transfers the branch address (disp:8) to the PC when T = 0, and is inactive when T = 1. When the branch destination is unattainable, this instruction must be combined with BRA or JMP. When T = 0 Branch destination address (disp: 8) → PC (31–0) When T = 1 NOP
BT		Conditional branch instruction that accesses the T bit. Remains inactive when T = 0, and transfers the branch address (disp:8) to the PC when T = 1. When the branch destination is unattainable, this instruction must be combined with BRA or JMP. When T = 0 NOP When T = 1 Branch destination address (disp: 8) → PC (31–0)
BRA	disp:12 or branch destination symbol	Unconditional branch instruction. Transfers the branch destination address (disp:12) to the PC. Use the JMP instruction when the branch destination is unattainable. BRA is a delayed branch instruction. Branch destination address (disp: 12) → PC (31–0)
BSR	disp:12 or branch destination symbol of subroutine	Branch to subroutine instruction. Saves the contents of the PC (return address from subroutine) to the PR, and transfers the subroutine start address (disp:12) to the PC. The return from subroutine address is two instructions from this instruction. Use the JSR instruction when the branch destination is unattainable. BSR is a delayed branch instruction. PC (31–0) → PR (31–0) Return from subroutine destination address ↑ Subroutine start address (disp: 12)

Table 1.40 Branch Instructions: BF, BT, BRA, BSR, JMP, JSR, RTS (cont)

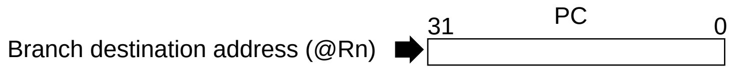
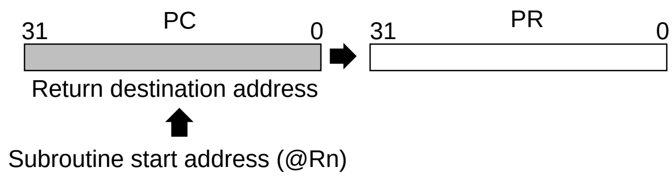
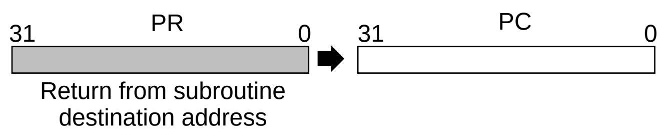
Mnemonic	Operand	Description
JMP	@Rn	Unconditional branch instruction. Transfers the branch destination address (@Rn) to the PC. JMP is a delayed branch instruction. 
JSR	@Rn	Branch to subroutine instruction. Saves the contents of the PC (return from subroutine address) to the PR, and transfers the subroutine start address (@Rn) to the PC. The return from subroutine address is two instructions from the current instruction. JSR is a delayed branch instruction. 
RTS	—	Return from subroutine instruction. Transfers the contents of the PR (return from subroutine address) to the PC. RTS is a delayed branch instruction. 

Table 1.41 System Control Instructions: RTE, CLRMAC, SLEEP, NOP, CLRT, SETT

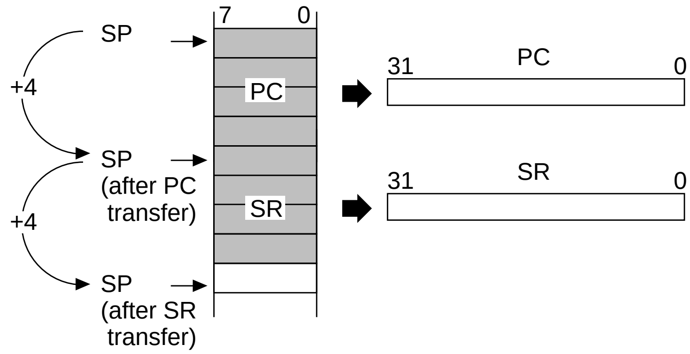
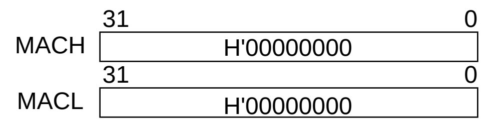
Operand Size	Source Operand	Destination Operand	Description
—	—	—	Mnemonic: RTE Return from interrupt routine instruction. Returns PC and SR from stack. RTE is a delayed branch instruction. 
			Mnemonic: CLRMAC Clears MACH and MACL. 
			Mnemonic: SLEEP Puts CPU into power-down mode. In power-down mode, the CPU's internal status is maintained, execution of the next instruction is halted, and the CPU waits for an interrupt request. The CPU is released from power-down mode on generation of the interrupt request.
			Mnemonic: NOP Increments (+2) the PC only and has no effect on the CPU's internal state.
			Mnemonic: CLRT. Clears the T bit. T $\boxed{0}$
			Mnemonic: SETT. Sets the T bit. T $\boxed{1}$

Table 1.42 System Control Instruction: LDC

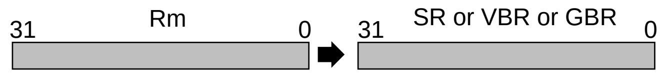
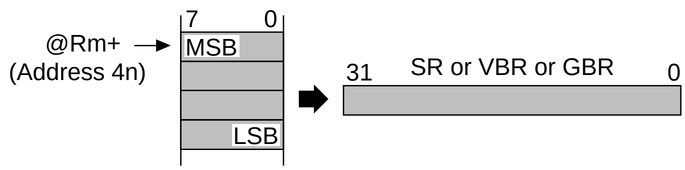
Operand Size	Source Operand	Destination Operand	Description
—	Rm (R0–R15)	SR, VBR, GBR	Transfers the contents of register Rm to a control register (SR, VBR, or GBR). 
L	@Rn+		Transfers @Rn+ memory contents to a control register (SR, VBR, or GBR). Used in resetting control registers. 

Table 1.43 System Control Instruction: STC

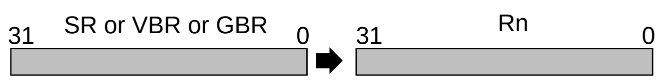
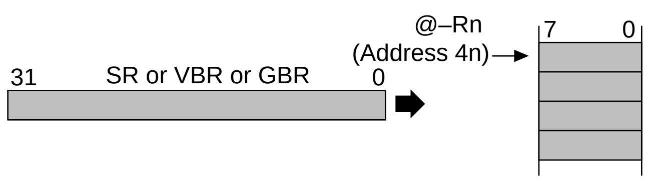
Operand Size	Source Operand	Destination Operand	Description
—	SR, VBR, GBR	Rm (R0–R15)	Transfers the contents of a control register (SR, VBR, or GBR) to general register Rn. 
L		@–Rn	Transfers the contents of a control register (SR, VBR, or GBR) to @Rn+ memory. Used in saving control registers. 

Table 1.44 System Control Instruction: LDS

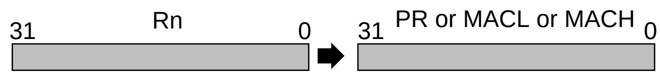
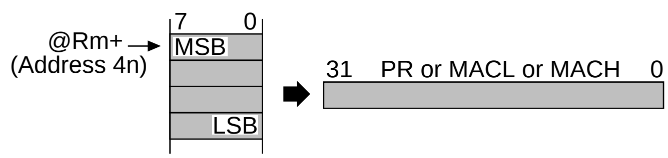
Operand Size	Source Operand	Destination Operand	Description
—	Rm (R0–R15)	PR, MACL, MACH	Transfers the contents of general register Rm to a system register (PR, MACL, or MACH). Note that the contents of Rm's lower 10 bits are sign extended for storage in MACH. 
L	@Rm+		Transfers the contents of @Rn+ memory to a system register (PR, MACL, or MACH). Used in restoring system registers. 

Table 1.45 System Control Instruction: STS

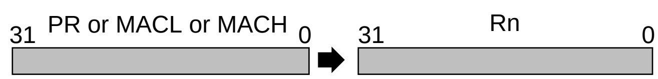
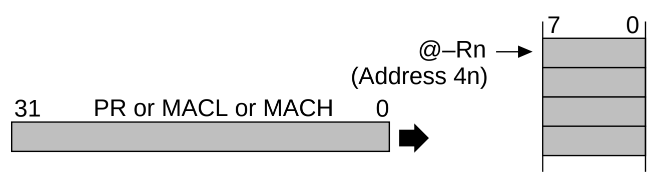
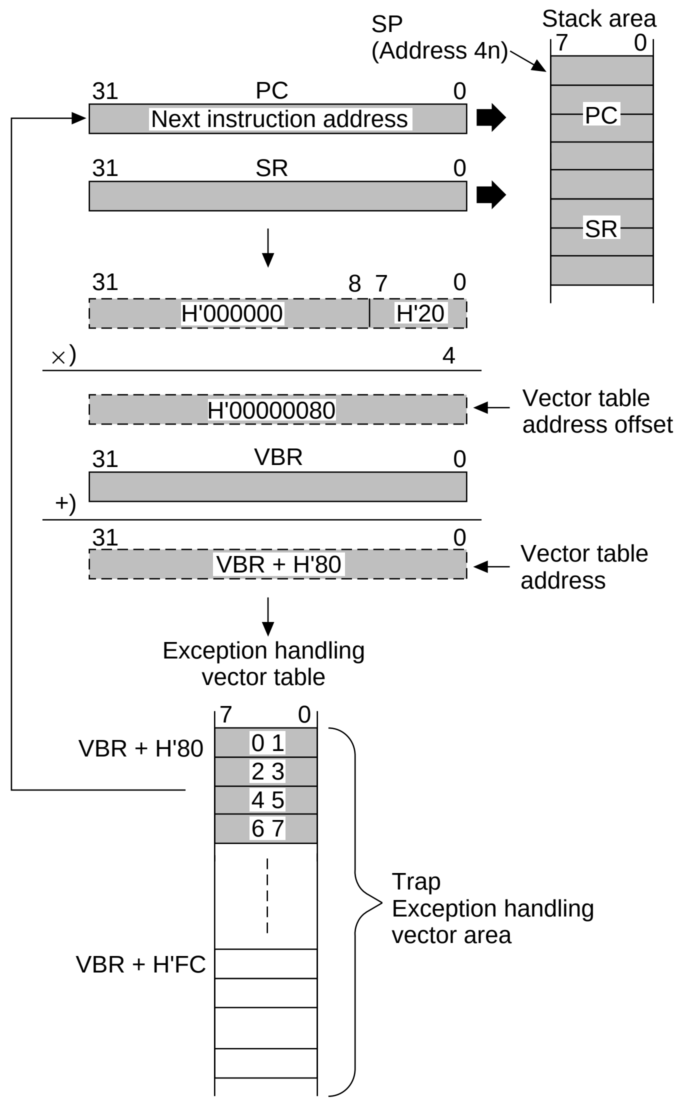
Operand Size	Source Operand	Destination Operand	Description
—	PR, MACL, MACH	Rm (R0–R15)	Transfers the contents of a system register (PR, MACL, or MACH) to general register Rn. 
L		@-Rn	Transfers the contents of a system register (PR, MACL, or MACH) to @Rn+ memory. Used in saving system registers. 

Table 1.46 System Control Instruction: TRAPA

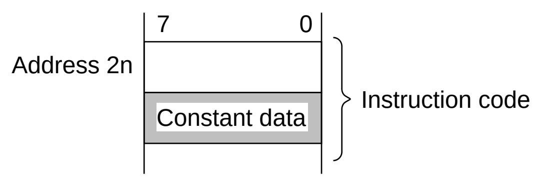
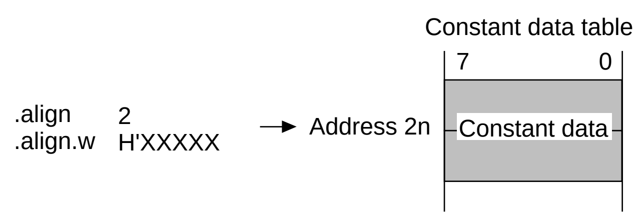
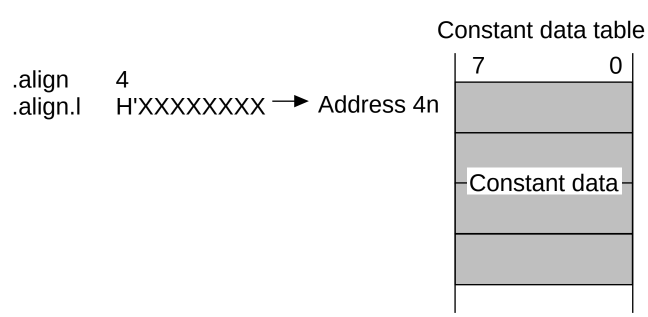
Operand Size	Source Operand	Destination Operand	Description
—	—	#imm: 8	Branch to trap exception processing instruction: saves PC (next instruction address) and SR contents to stack, adds 8-bit constant data to VBR after zero extension and quadrupling (vector table address offset), and transfers address indicated by sum (vector table address) to PC. Used with RTE for system calls.



1.5 Constant Data Table Placement Method

Because the SH7000 CPU instruction code is 16-bit fixed length, 16-bit and 32-bit constant data cannot be arranged in instruction code in the same way as 8-bit constant data. Therefore, a 16-bit and 32-bit constant data table is used. In this system, 16-bit and 32-bit constant data is accessed from the tables using PC-relative with displacement addressing mode. Table 1.47 shows where constant data is located.

Table 1.47 Placement of Constant Data

Constant Data Length	Constant Data Addressing Mode	Placement
8 bit	Immediate	8-bit constant data is placed in the second byte of instruction code. 
16 bit	PC-relative with displacement	16-bit constant data is placed in address 2n of the constant data table. 
32 bit		32-bit constant data is placed in address 4n of the constant data table. 

An unconditional branch instruction is needed when a constant data table is placed between processes as shown in figure 1.24. Effective programming is possible when the table is placed directly after a subroutine or repeat process, as shown in figures 1.25 and 1.26.

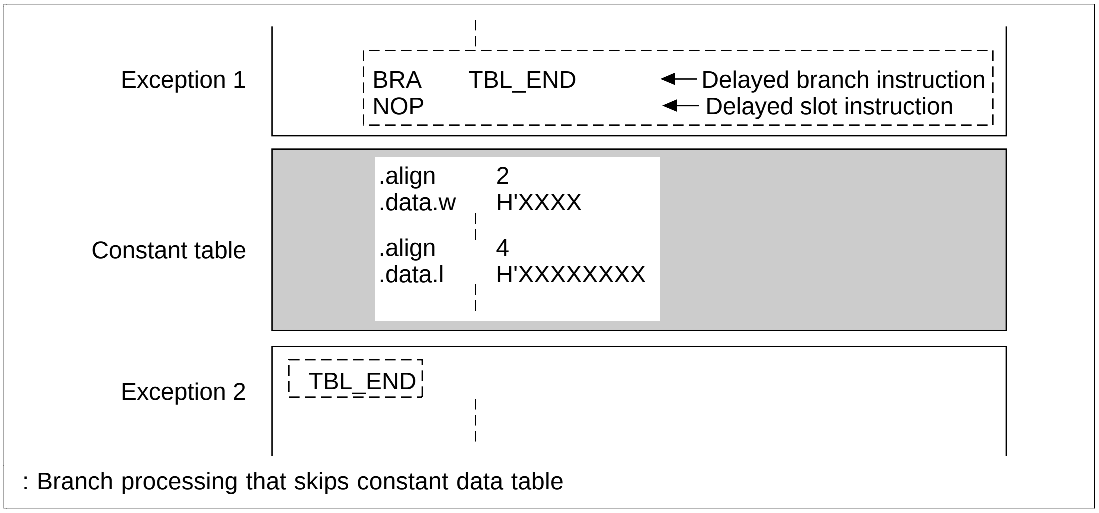


Figure 1.24 Constant Data Table Layout Location, 1

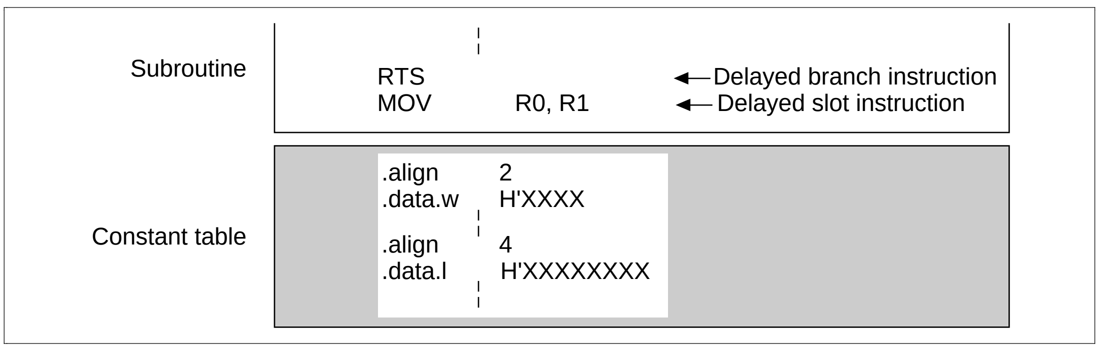


Figure 1.25 Constant Data Table Layout Location, 2

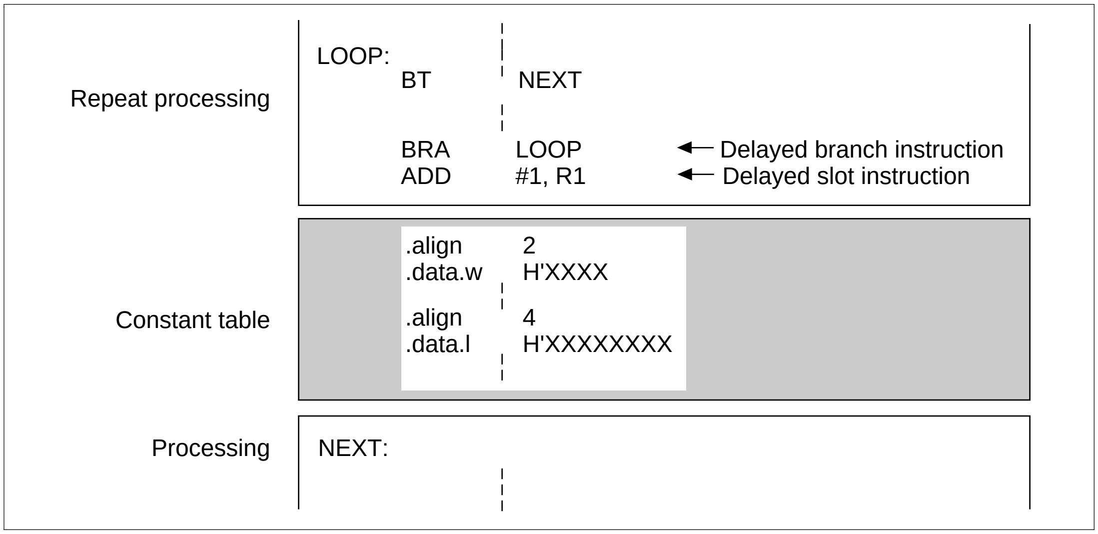


Figure 1.26 Constant Data Table Layout Location, 3

1.6 Delayed Branch Instructions

The unconditional branch instructions (BRA, BSR, JMP, JSR, RTS, and RTE) are delayed branch instructions which help to reduce pipeline disruption during branching. Delayed branch instructions cause a branch immediately after the next instruction (delay slot instruction) is executed (table 1.48).

Delay slot instructions become errors or warnings in the case of delayed branch instructions or BF, BT, TRAPA, MOVA, MOV, and @(disp,PC)Rn. For more information, see Precautions for Delayed Branch Instructions in *SH Series Cross-Assembler User's Manual*.

Table 1.48 Source Program Description

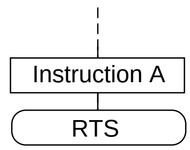
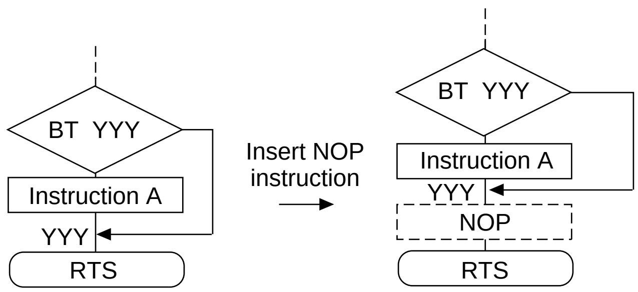
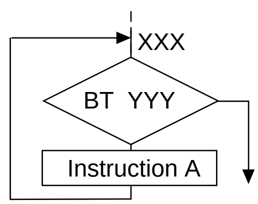
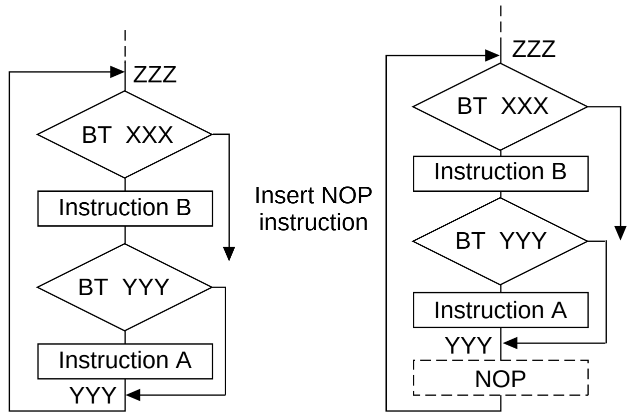
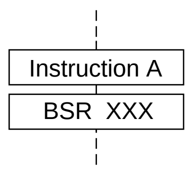
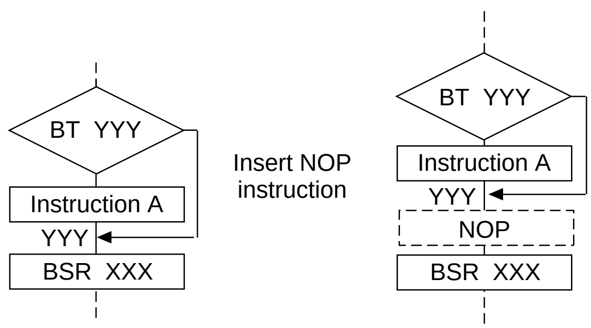
Delayed Branch Instruction	Description	Source Program
RTS, RTE	Replaces the immediately preceding instruction in the circumstances shown. 	↓ RTS Instruction A ← Delay slot instruction
	Inserts a NOP as a dummy instruction before replacement if an RTS (or RTE) is the branch destination. 	↓ BT YYY Instruction A YYY RTS NOP ← Delay slot instruction
JMP, BRA	Replaces the immediately preceding instruction in the circumstances shown. 	↓ BT YYY BRA XXX Instruction A ← Delay slot instruction

Table 1.48 Source Program Description (cont)

Delayed Branch Instruction	Description	Source Program
MP, BRA (cont)	Inserts a NOP as a dummy instruction immediately before replacement when BRA (or JMP) is the branch destination. 	↓ ZZZ BT XXX Instruction B BT YYY Instruction A YYY BRA ZZZ NOP← Delay slot instruction
JSR, BSR	Replaces the immediately preceding instruction in the circumstances shown. 	↓ BSR XXX Instruction A ← Delay slot instruction ↓
	Inserts a NOP immediately before BSR (or JSR) before replacement when BSR (or JSR) is the branch destination. 	↓ BT YYY Instruction A YYY BSR XXX NOP← Delay slot instruction ↓

1.7 Access Method for On-Chip Peripheral Module Register Area

The SH7000 series CPU has a global base register (GBR) that sets the base address for the on-chip peripheral module register area. This allows calculation of the on-chip peripheral module register address, using the offset value to the base address set in the GBR. Addressing modes are indirect GBR with displacement, which supports the MOV instruction, and indirect indexed GBR, which supports logic operation instructions for data transfers and bit manipulation.

Sections 1.7.1 and 1.7.2 detail data transfer and bit manipulation when H'5FFFF00 is set in the GBR.

1.7.1 Data Transfer

The on-chip peripheral module register address specification for data transfer varies depending on offsets of –256 to –129, –128 to –1, and 0 to +255 for each area (figure 1.27). There is also an access size restriction on certain registers. For more information see the SH-2 hardware manual.

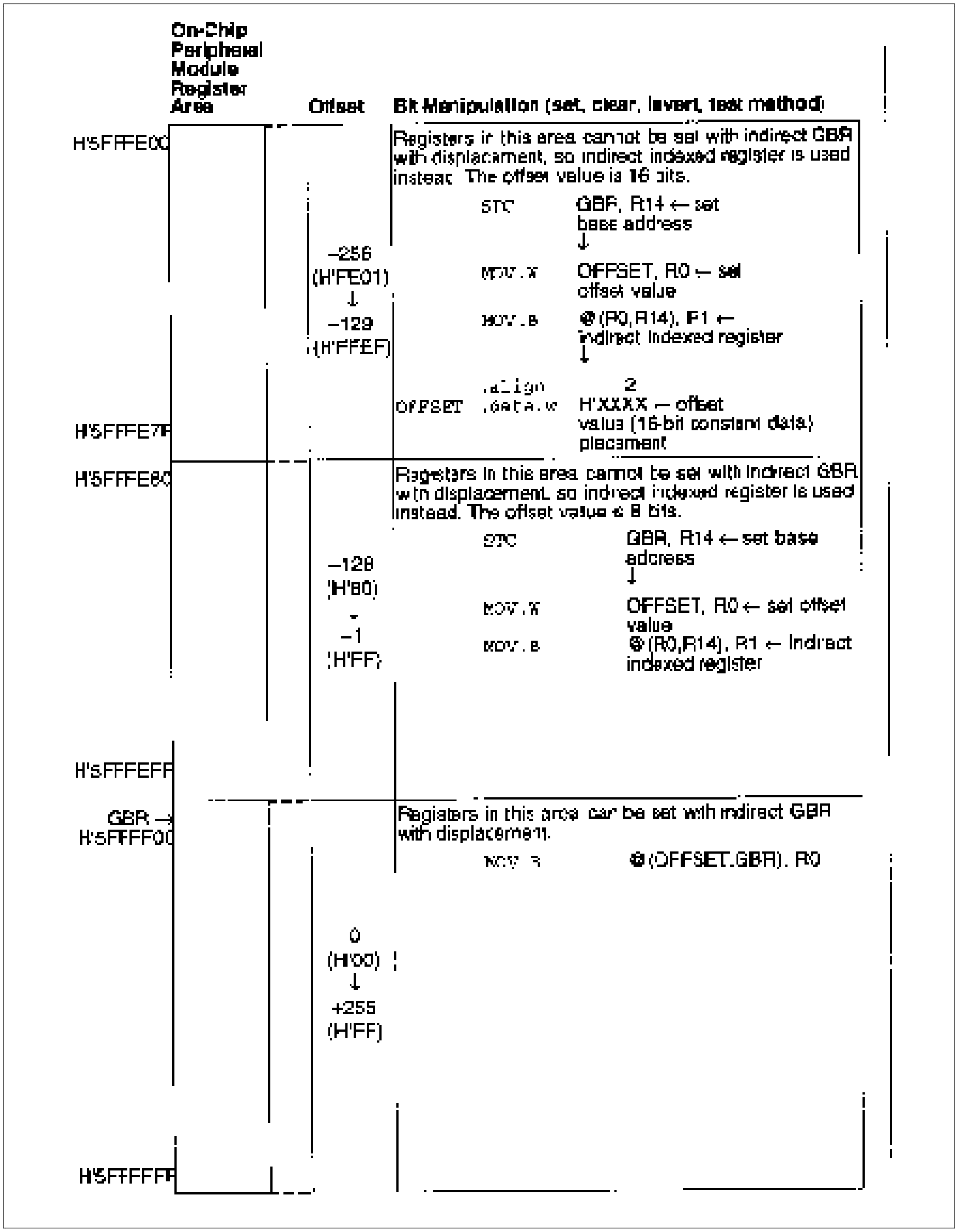


Figure 1.27 On-Chip Peripheral Module Data Transfer Method

1.7.2 Bit Manipulation

Indirect indexed GBR is used for setting the address of on-chip peripheral module registers for bit manipulation. Note that the offset value data length varies with the areas –256 to –129, –128 to –1, and 0 to +255 (figure 1.28).

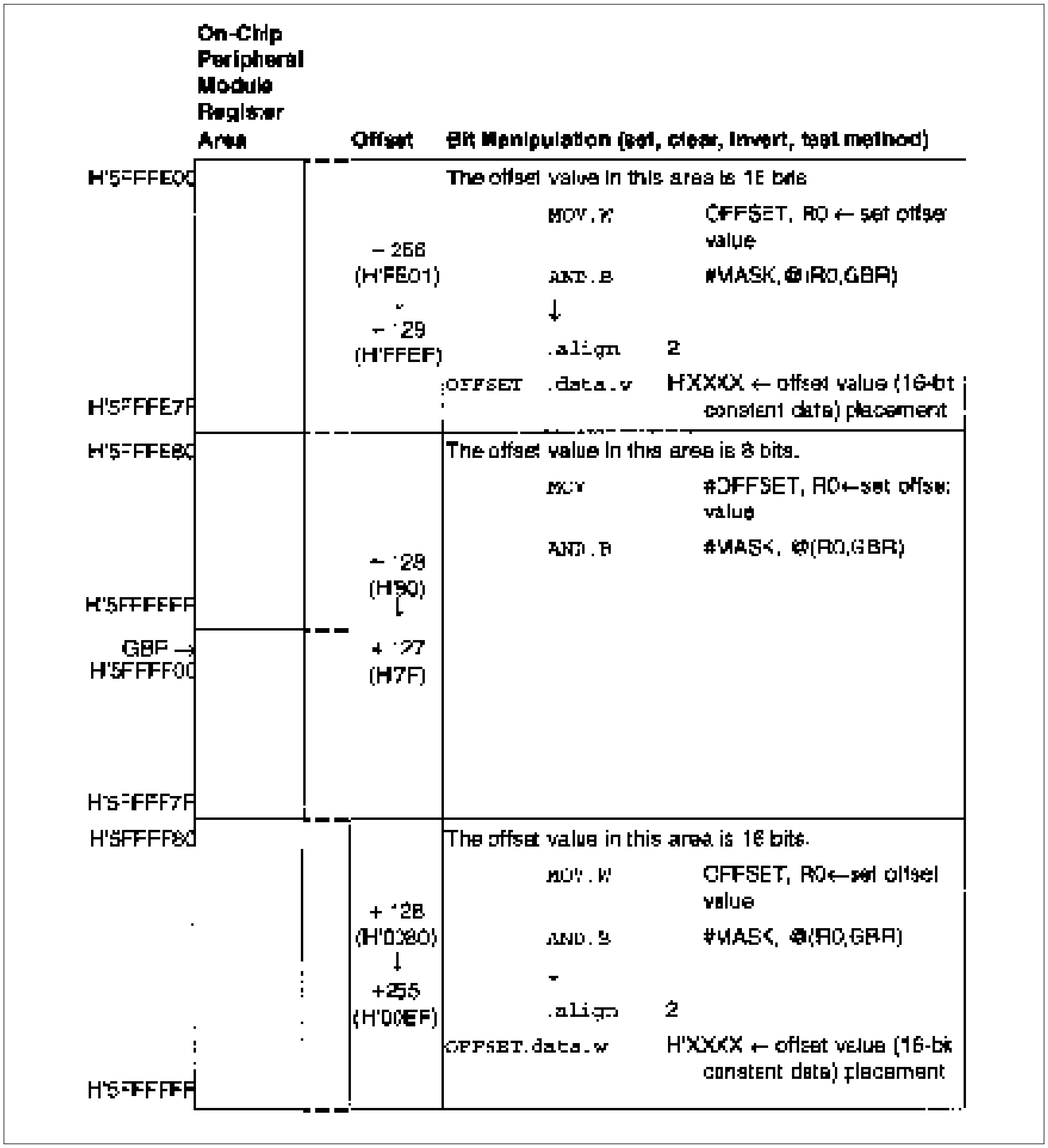


Figure 1.28 On-Chip Peripheral Module Register Bit Manipulation Method

1.8 Precautions for Subroutine Calls

When subroutine calls are made using the JSR instruction or BSR instruction, the return address from the subroutine is automatically saved in the PR register. Unlike the stack area, only one return address can be stored in the PR register. Therefore, if another subroutine is called during one subroutine, the return address of the calling subroutine will be destroyed. It is necessary, therefore, to save and restore the PR contents (figure 1.29). The system control instructions, STS and LDS, are used in saving and restoring. Memory (stack area, etc.) or general registers may be specified as the save destination.

PR save and restore must also be performed if a subroutine call is made during an interrupt routine, as in figure 1.29.

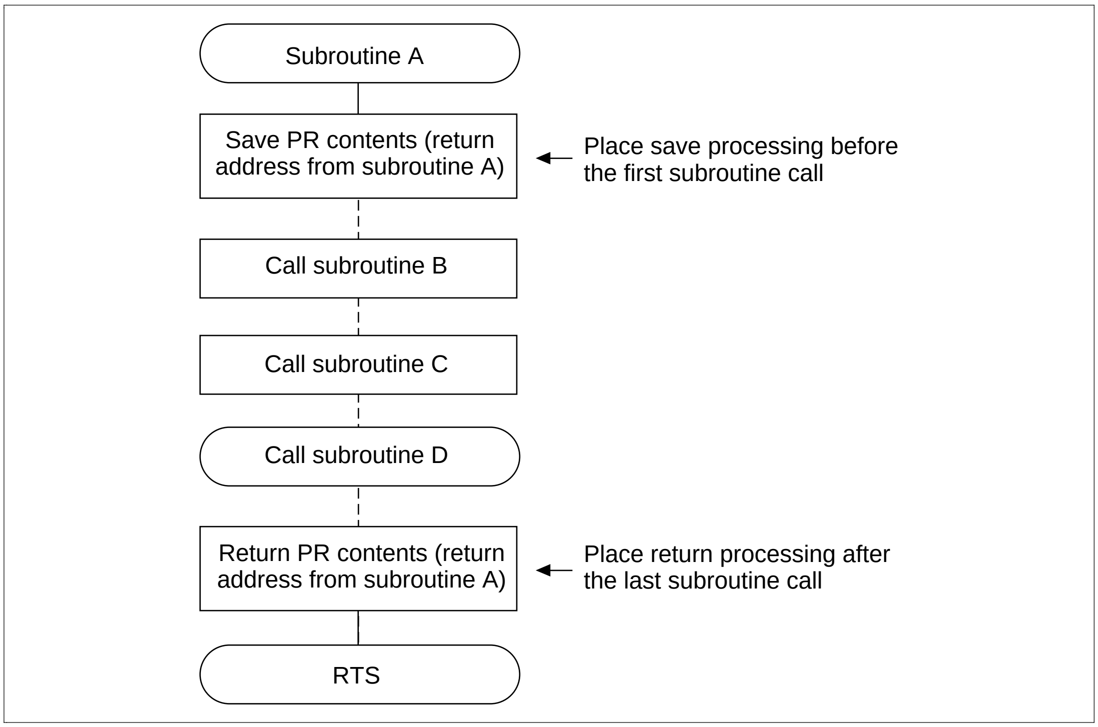


Figure 1.29 PR Save and Restore

Section 2 Load Module Conversion Procedure

The following is the load module conversion procedure for the SH7000 series CPU (figures 2.1 through 2.4).

1. Assembler source program creation using an arbitrary editor (MIFES, etc.).

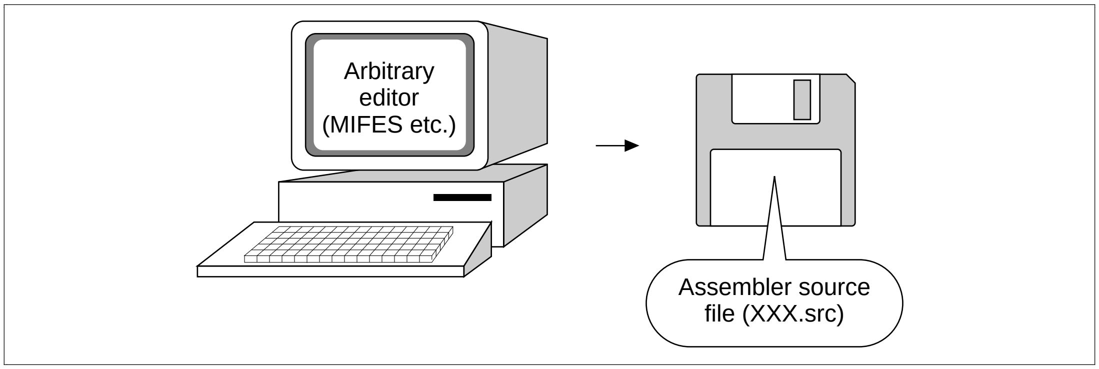


Figure 2.1 Assembler Source Program Creation

2. Conversion of assembler source program to object module using assembler (ASMSH.EXE).

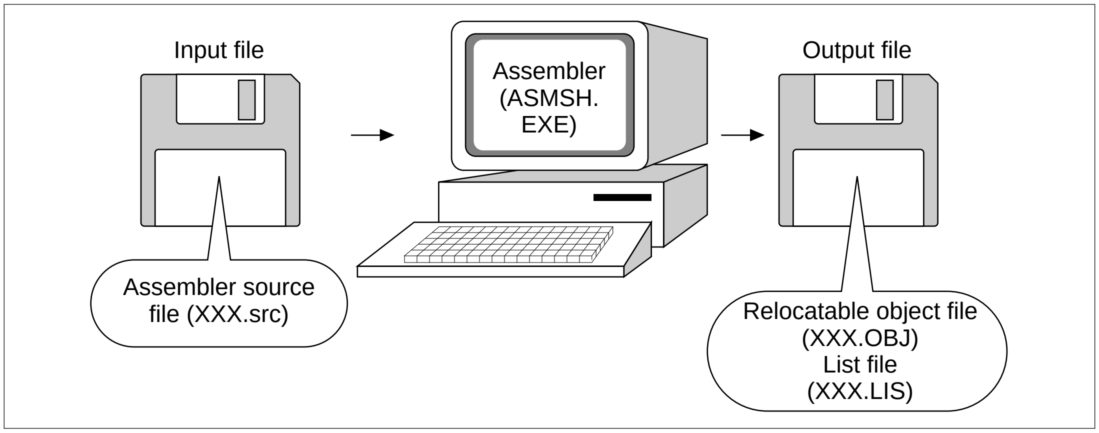


Figure 2.2 Conversion of Assembler Source Program to Object Module

3. Conversion of object module to load module using linkage editor (LINK.EXE). Use version 5.0 or later of the linkage editor.

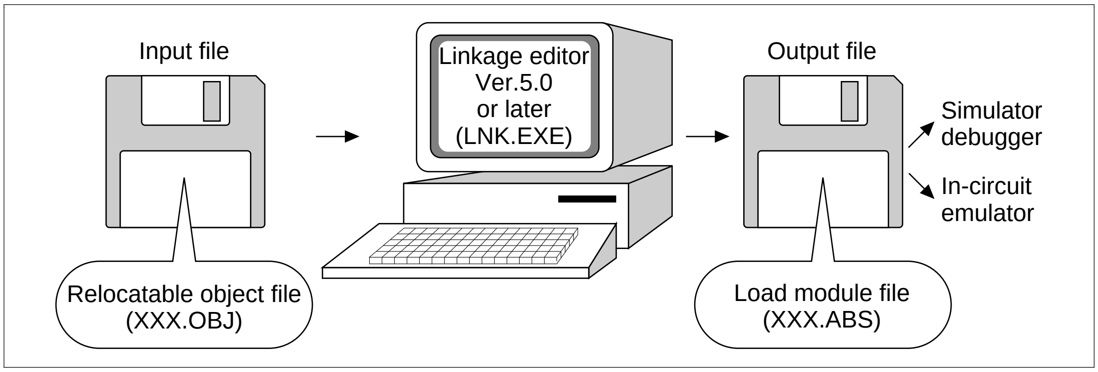


Figure 2.3 Conversion of Object Module to Load Module

4. Conversion of load module to s-type format load module using load module converter (CNVS.EXE).

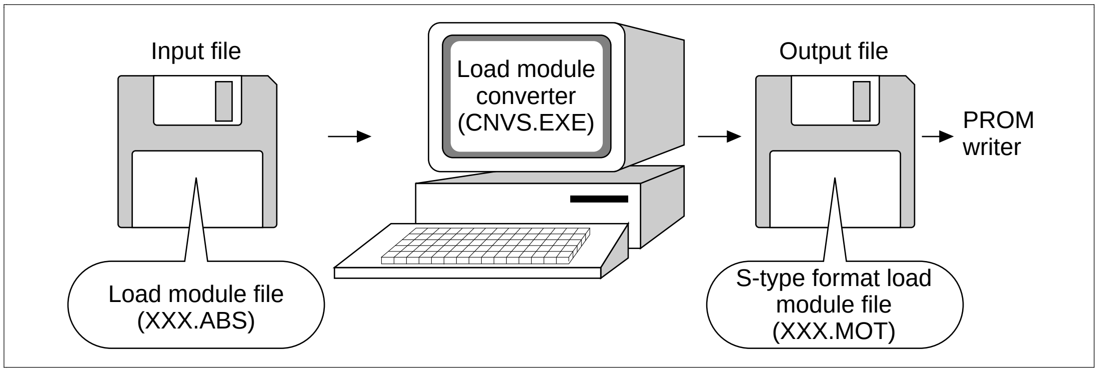


Figure 2.4 Load Module Conversion to S-Type Format

Section 3 Application Examples

Table 3.1 Application Examples

Application	Label	Function
Block transfer (4 bytes unaligned)	MOVE	MOV.B instruction, post-increment register indirect, Register indirect with displacement
Block transfer (4 bytes aligned)	MOVE4	MOV.L instruction, post-increment register indirect, Register indirect with displacement
32-bit data multi-bit shift (arithmetic right shift)	SHARN	SHLR2 instruction, SHLR8 instruction, SHLR16 instruction
32-bit data multi-bit shift (logical right shift)	SHLRN	SHLR2 instruction, SHLR8 instruction, SHLR16 instruction
32-bit data multi-bit shift (logical left shift)	SHLLN	SHLL2 instruction, SHLL8 instruction, SHLL16 instruction
32 bit data first find 1	FIND1	SHLL instruction
64 bit + 64 bit = 64 bit (unsigned)	ADDU64	ADDC instruction
64 bit + 64 bit = 64 bit (signed)	ADDS64	ADDV instruction
32 bit × 32 bit = 64 bit (unsigned)	MULU32	MULU instruction, SWAP instruction
32 bit × 32 bit = 64 bit (signed)	MULS32	MULU instruction, SWAP instruction, NEGC instruction
Quotient of 32 bit ÷ 32 bit (unsigned)	DIVU32Q	DIV0U instruction, DIV1 instruction
Remainder of 32 bit ÷ 32 bit (unsigned)	DIVU32R	DIV0U instruction, DIV1 instruction
Quotient of 32 bit ÷ 32 bit (signed)	DIVS32Q	DIV0S instruction, DIV1 instruction
Remainder of 32 bit ÷ 32 bit (signed)	DIVS32R	DIV0S instruction, DIV1 instruction
Affine transform	AFIN	MAC.W instruction post-increment register indirect

In the following sections, the entry point label name given for each software example is to be used for subroutine calls when using the example just as a subroutine.

The state of the T bit after running each software example is indicated as follows:

- No change: T bit contents are preserved through the execution of the software example.
- Change: T bit contents are destroyed due to execution of software example.
- Fixed: set to 0 or 1 after execution of the software example.

Program specifications for each example are listed as follows:

- Program memory (bytes): Indicates the volume of ROM used by the software example.
- Data memory (bytes): Indicates the volume of RAM used by the software example.
- Stack (bytes): Indicates the volume of stack used by the software example.

This does not include the stack volume for subroutine calls in a user program. The number of bytes indicated in the stack section is needed for running the software example, so make sure enough stack volume is reserved for the data memory volume.

- Number of states: Indicates the number of running states of the software example when run by the simulator debugger.
- Reentrant: Indicates whether software has a construction that allows running simultaneously with several resident programs.
- Relocation: Indicates whether the software example operates normally in whichever memory space it is located.
- Intermediate interrupt: Indicates whether the software example resumes normal operation after an interrupt routine is executed during its run. If the software example does not allow this (disabled), be sure to disable interrupts before calling the software example.

3.1 MOVE: Block Transfer (4 Bytes Unaligned)

- Instruction: MOV.B
- Addressing modes: Post-increment register indirect
  Register indirect with displacement
- Function: Transfers block data. Enables setting of any start address for block data transfer source area and transfer destination area, and any bytes of block data.

Table 3.2 MOVE Arguments

	Contents	Storage Location	Data Length (Bytes)
Input	Number of transfer bytes	R0	4
	Transfer destination start address	R1	4
	Transfer source start address	R2	4
Output	—	—	—

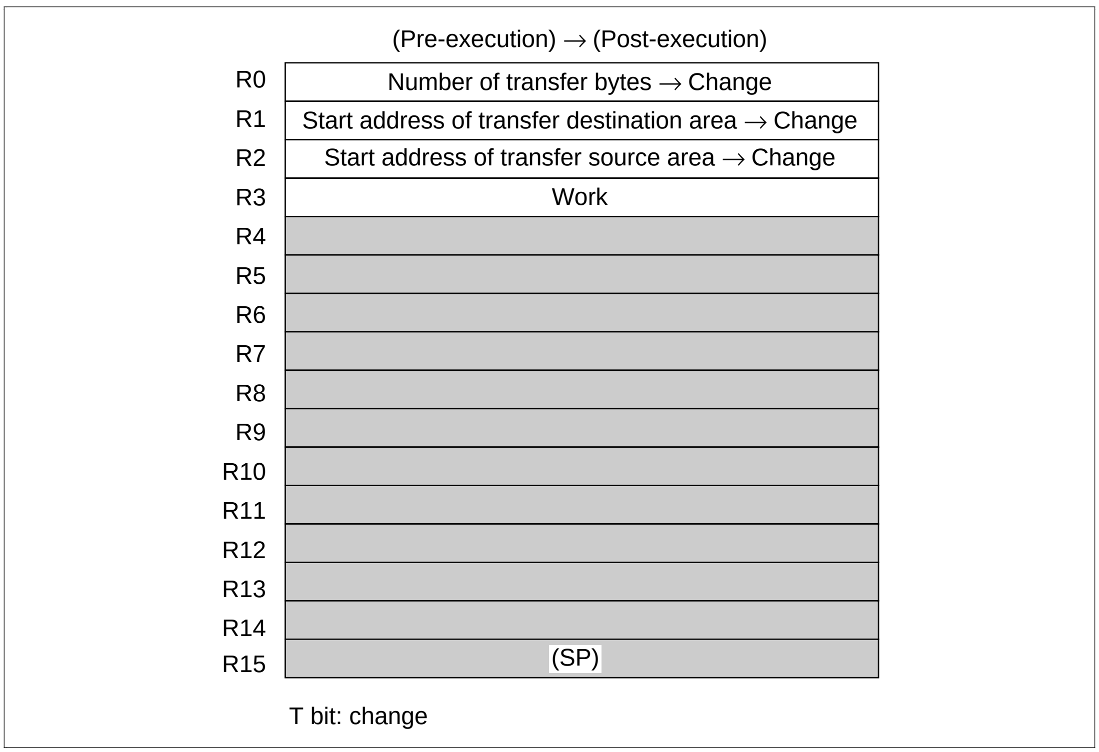


Figure 3.1 MOVE Internal Register Change and Flag Change

Table 3.3 MOVE Programming Specifications

Item	Value/State
Program memory (bytes)	142
Data memory (bytes)	0
Stack (bytes)	4
Number of states	429
Reentrant	Enabled
Relocation	
Intermediate interrupt	

- Precaution: The number of states in the programming specifications is the value when the number of transfer bytes is 100.

3.1.1 MOVE Arguments

- R0: Sets the number of transfer bytes (any bytes) as the input argument. Beware of hardware constraints.
- R1: Sets the transfer destination area start address (any address) as the input argument.
- R2: Sets the transfer source area start address (any address) as the input argument.

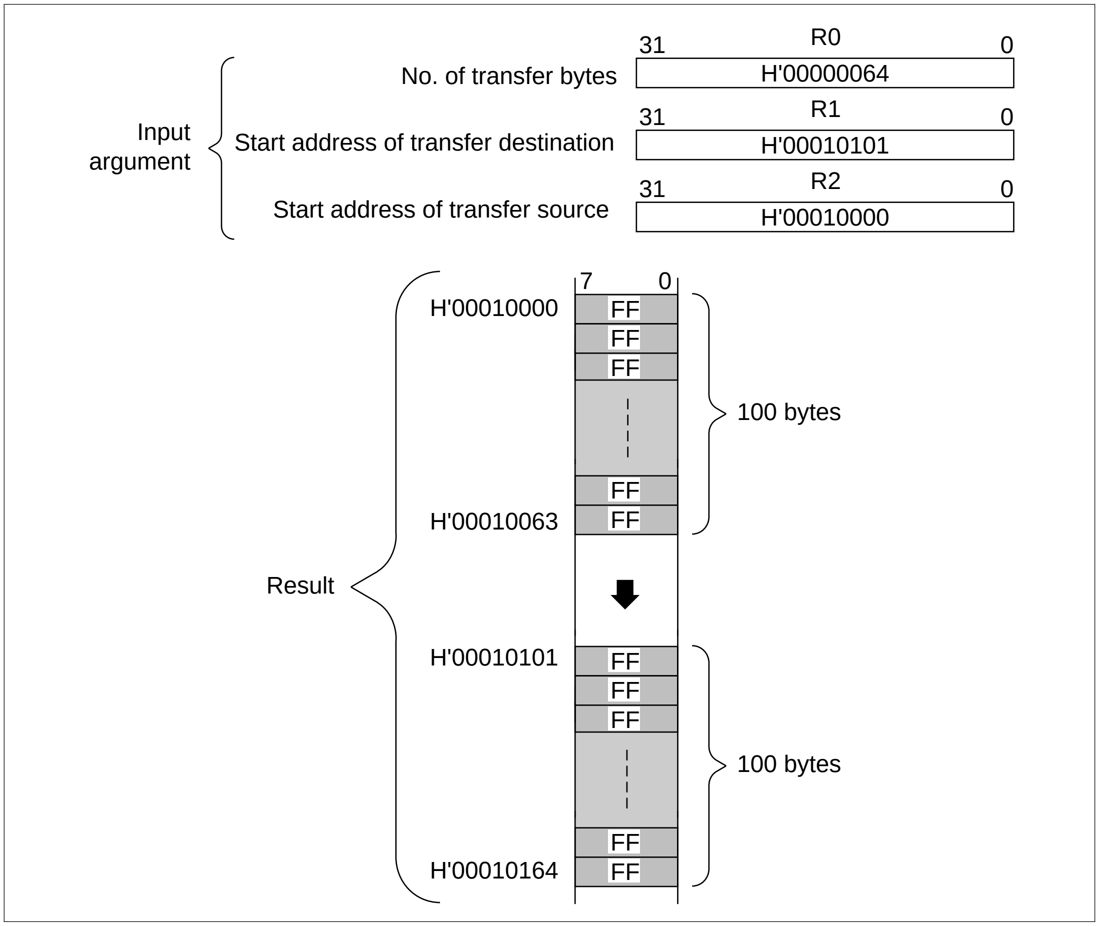


Figure 3.2 MOVE Execution Example

3.1.2 MOVE Precautions for Use

Set the input argument so that the transfer source and transfer destination areas do not overlap. If the two areas do overlap, the transfer source data in the overlapping portion is destroyed (figure 3.3).

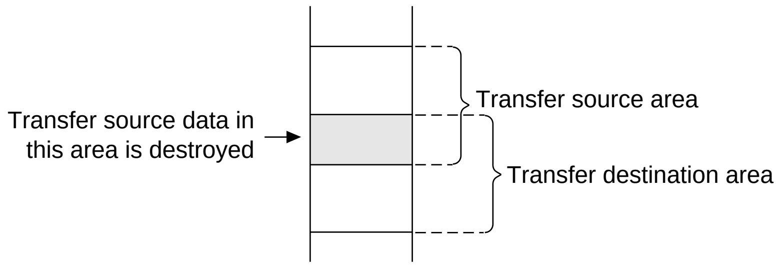


Figure 3.3 Block Transfer when Data Areas Overlap (MOVE)

R0, R1, and R2, which respectively contain the number of transfer bytes, transfer destination area start address, and transfer source area start address, have their contents changed by execution of MOVE. Be sure to save the number of transfer bytes and start addresses of transfer destination and source beforehand, if this data is also required after a MOVE execution.

3.1.3 MOVE RAM Use

RAM is not used with MOVE.

3.1.4 Example of MOVE Use

Make a subroutine call to MOVE after setting the transfer source area start address, transfer destination area start address, and number of transfer bytes in the input argument.

MOV.L	DATA1,R0	Set number of transfer bytes in the inputargument(R0)
MOV.L	DATA2,R1	Set the transfer destination area start address in the input argument (R1)
BSR	MOVE	Subroutine call MOVE
MOV.L	DATA3,R2	Set the transfer source area start address in the input argument (R2)

```
          ↓
        .align    4
DATA1   .data.l   H'00000064
DATA2   .data.l   H'00010101
DATA3   .data.l   H'00010000
```

3.1.5 MOVE Operation

Since transfer source and transfer destination addresses are arbitrary addresses (4 bytes unaligned), transfer is in single byte units from transfer source to destination.

Post-increment register indirect (@R2+) is used in specifying the transfer source address. The transfer source address is incremented automatically +1 at a time.

Register indirect with displacement is used in specifying the transfer destination address. Displacement is 0–15, so the transfer destination address must be incremented +16 for every 15 bytes of transfer. In other cases, increment processing is not required.

Set the transfer source start address (R2) + number of transfer bytes (R0) in R3. After setting, R0, containing the number of transfer bytes, is used as a data transfer work register. After sending transfer source data to R0, R2 ≤ R3 testing is performed. When this condition is satisfied (R2 ≤ R3), data in R0 is the data within the transfer source area, so data is transferred to the destination. When the condition is not satisfied (R2 > R3), data in R0 is outside the transfer source area, so transfer is halted.

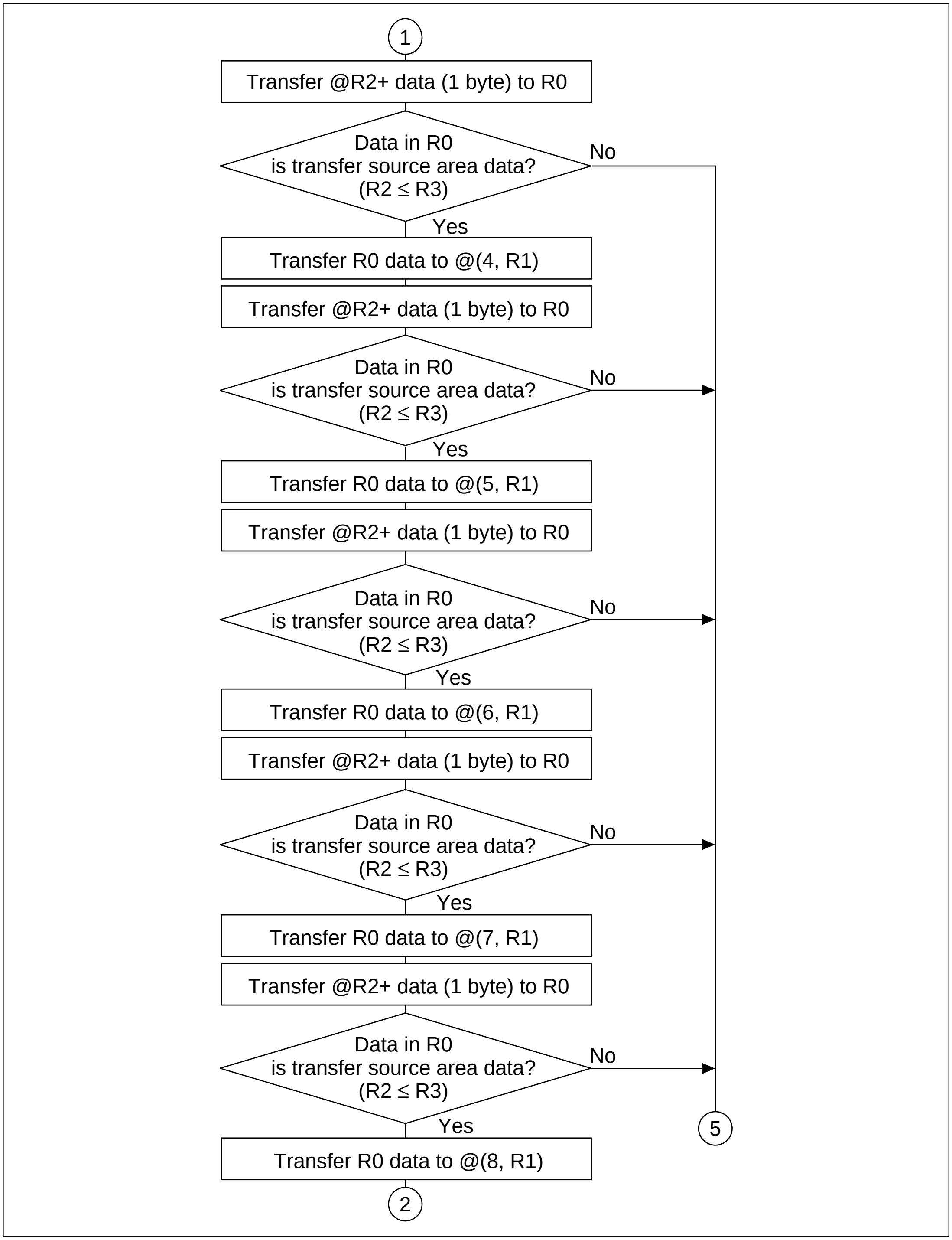


Figure 3.5 MOVE Flowchart

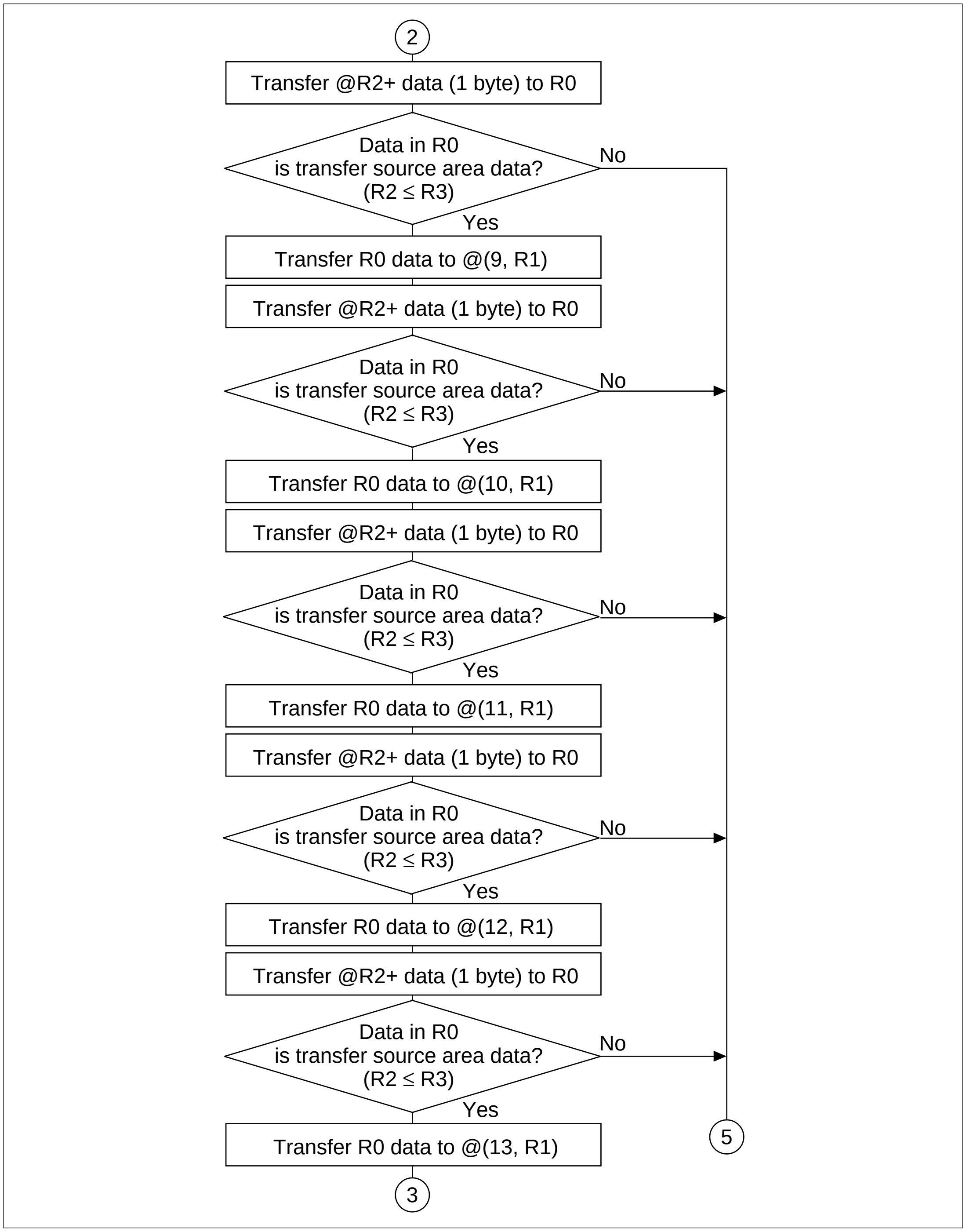


Figure 3.5 MOVE Flowchart (cont)

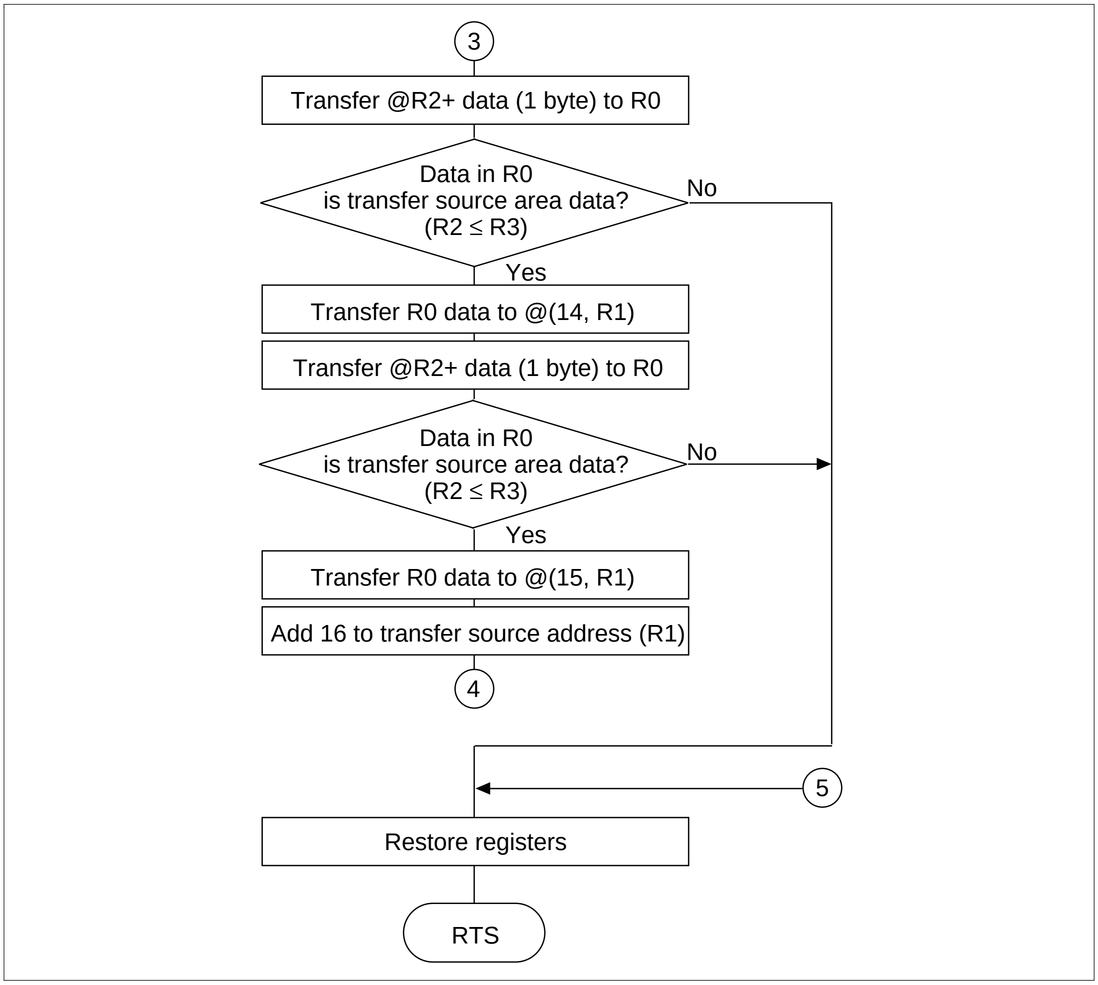


Figure 3.5 MOVE Flowchart (cont)

3.1.6 MOVE Program Listing

```
PROGRAM NAME: MOVING MEMORY BLOCKS (MOVE)
ENTRY:        R0      (NUMBER OF TRANSFER)
              R1      (DESTINATION ADDRESS)
              R2      (SOURCE ADDRESS)
RETURNS:          NOTHING
1                     1                       ;
2                     2                       ;
3                     3                       ;
4                     4                       ;
5                     5                       ;
6                     6                       ;
7                     7                       ;
8                     8                       ;
9                     9                       ;
10                    10                      ;
11                    11                      ;
12                    12                      ;
13  00001000          13  .SECTION A,CODE,LOCATE=H'1000
14  00001000          14  MOVE .EQU $         ;Entry point
15  00001000   2F36   15    MOV.L R3,@-R15    ;Escape register
16  00001002   6323   16    MOV   R2,R3       ;
17  00001004   330C   17    ADD   R0,R3       ;
18  00001006          18  MOVE1               ;
19  00001006   6024   19    MOV.B  @R2+,R0    ;Load source data
20  00001008   3322   20    CMP/HS R2,R3      ;R2 ≤ R3 ?
21  0000100A   8B3E   21    BF     MOVE_END   ;No
22  0000100C   2100   22    MOV.B  R0,@R1     ;Yes →Store source data
23  0000100E          23  MOVE2               ;
24  0000100E   6024   24    MOV.B  @R2+,R0    ;Load source data
25  00001010   3322   25    CMP/HS R2,R3      ;R2 ≤ R3?
26  00001012   8B3A   26    BF     MOVE_END   ;No
27  00001014   8011   27    MOV.B  R0,@(1,R1) ;Yes →Store source data
28  00001016          28  MOVE3               ;
29  00001016   6024   29    MOV.B  @R2+,R0    ;Load source data
30  00001018   3322   30    CMP/HS R2,R3      ;R2 ≤ R3?
31  0000101A   8B36   31    BF     MOVE_END   ;
```

```
32  0000101C   8012   32    MOV.B  R0,@(2,R1) ;Yes →Store source data
33  0000101E          33  MOVE4               ;
34  0000101E   6024   34    MOV.B  @R2+,R0    ;Load source data
35  00001020   3322   35    CMP/HS R2,R3      ;R2 ≤ R3?
36  0001022    8B32   36    BF     MOVE_END   ;No
37  00001024   8013   37    MOV.B  R0,@(3,R1) ;Yes →Store source data
38  00001026          38  MOVE5               ;
39  00001026   6024   39    MOV.B  @R2+,R0    ;Load source data
40  00001028   3322   40    CMP/HS R2,R3      ;R2 ≤ R3?
41  0000102A   8B2E   41    BF     MOVE_END   ;No
42  0000102C   8014   42    MOV.B  R0,@(4,R1) ;Yes →Store source data
43  0000102E          43  MOVE6               ;
44  0000102E   6024   44    MOV.B  @R2+,R0    ;Load source data
45  00001030   3322   45    CMP/HS R2,R3      ;R2 ≤ R3?
46  00001032   8B2A   46    BF     MOVE_END   ;No
47  00001034   8015   47    MOV.B  R0,@(5,R1) ;Yes →Store source data
48  00001036          48  MOVE7               ;
49  00001036   6024   49    MOV.B  @R2+,R0    ;Load source data
50  00001038   3322   50    CMP/HS R2,R3      ;R2 ≤ R3?
51  0000103A   8B26   51    BF     MOVE_END   ;No
52  0000103C   8016   52    MOV.B  R0,@(6,R1) ;Yes →Store source data
53  0000103E          53  MOVE8               ;
54  0000103E   6024   54    MOV.B  @R2+,R0    ;Load source data
55  00001040   3322   55    CMP/HS R2,R3      ;R2 ≤ R3 ?
56  00001042   8B22   56    BF     MOVE_END   ;No
57  00001044   8017   57    MOV.BR0,@(7,R1)   ;Yes →Store source data
58  00001046          58  MOVE9               ;
59  00001046   6024   59    MOV.B  @R2+,R0    ;Load source data
60  00001048   3322   60    CMP/HS R2,R3      ;R2 ≤ R3?
61  0000104A   8B1E   61    BF     MOVE_END   ;No
62  0000104C   8018   62    MOV.B R0,@(8,R1)  ;Yes →Store source data
63  0000104E          63  MOVE10              ;
64  0000104E   6024   64    MOV.B  @R2+,R0    ;Load source data
65  00001050   3322   65    CMP/HS R2,R3      ;R2 ≤ R3 ?
66  00001052   8B1A   66    BF     MOVE_END   ;No
67  00001054   8019   67    MOV.B  R0,@(9,R1) ;Yes →Store source data
68  00001056          68  MOVE11              ;
```

```
69  00001056   6024   69    MOV.B  @R2+,R0    ;Load source data
70  00001058   3322   70    CMP/HS R2,R3      ;R2 ≤ R3 ?
71  0000105A   8B16   71    BF     MOVE_END   ;No
72  0000105C   801A   72    MOV.B R0,@(10,R1) ;Yes →Store source data
73  0000105E          73  MOVE12              ;
74  0000105E   6024   74    MOV.B  @R2+,R0    ;Load source data
75  00001060   3322   75    CMP/HS R2,R3      ;R2 ≤ R3 ?
76  00001062   8B12   76    BF     MOVE_END   ;No
77  00001064   801B   77    MOV.B R0,@(11,R1) ;Yes →Store source data
78  00001066          78  MOVE13              ;
79  00001066   6024   79    MOV.B  @R2+,R0    ;Load source data
80  00001068   3322   80    CMP/HS R2,R3      ;R2 ≤ R3 ?
81  0000106A   8B0E   81    BF     MOVE_END   ;No
82  0000106C   801C   82    MOV.B R0,@(12,R1) ;Yes →Store source data
83  0000106E          83  MOVE14              ;
84  0000106E   6024   84    MOV.B  @R2+,R0    ;Load source data
85  00001070   3322   85    CMP/HS R2,R3      ;R2 ≤ R3 ?
86  00001072   8B0A   86    BF     MOVE_END   ;No
87  00001074   801D   87    MOV.B R0,@(13,R1) ;Yes →Store source data
88  00001076          88  MOVE15              ;
89  00001076   6024   89    MOV.B  @R2+,R0    ;Load source data
90  00001078   3322   90    CMP/HS R2,R3      ;R2 ≤ R3 ?
91  0000107A   8B06   91    BF     MOVE_END   ;No
92  0000107C   801E   92    MOV.B R0,@(14,R1) ;Yes →Store source data
93  0000107E          93  MOVE16              ;
94  0000107E   6024   94    MOV.B  @R2+,R0    ;Load source data
95  00001080   3322   95    CMP/HS R2,R3      ;R2 ≤ R3 ?
96  00001082   8B02   96    BF     MOVE_END   ;No
97  00001084   801F   97    MOV.B R0,@(15,R1) ;Yes →Store source data
98                    98                      ;
99  00001086   AFBE   99    BRA    MOVE1      ;
100 00001088   7110   100   ADD    #D'16,R1   ;R1 ← R1 + 16
101 0000108A          101  MOVE_END           ;
102 0000108A   000B   102    RTS              ;
103 0000108C   63F6   103    MOV.L @R15+,R3   ;Return register
104                   104    .END
*****TOTAL  ERRORS 0
```

```
*****TOTAL  WARNINGS 0
```

3.2 MOVE4: Block Transfer (4 Bytes Aligned)

- Instruction: MOV.L
- Addressing modes: post-increment register indirect
  register indirect with displacement
- Function: Transfers block data. Note that the block data transfer source area and transfer destination area address is address 4n, and that block data is limited to 4n bytes.

Table 3.4 MOVE4 Arguments

	Contents	Storage Location	Data Length (Bytes)
Input	Number of transfer bytes (4n bytes)	R0	4
	Transfer destination start address (address 4n)	R1	4
	Transfer source start address (address 4n)	R2	4
Output	—	—	—

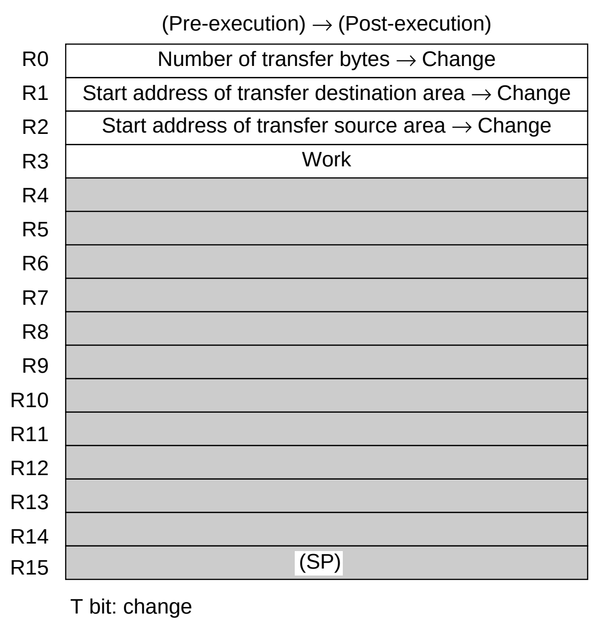


Figure 3.6 MOVE4 Internal Register Change and Flag Change

Table 3.5 MOVE4 Programming Specifications

Item	Value/State
Program memory (bytes)	142
Data memory (bytes)	0
Stack (bytes)	4
Number of states	114
Reentrant	Enabled
Relocation	
Intermediate interrupt	

- Precautions: The number of states in the programming specifications is the value when the number of transfer bytes is 100.

3.2.1 MOVE4 Arguments

- R0: Holds the number of transfer bytes (4n bytes) as the input argument. Beware of hardware constraints.
- R1: Holds the transfer destination area start address (address 4n) as the input argument.
- R2: Holds the transfer source area start address (address 4n) as the input argument.

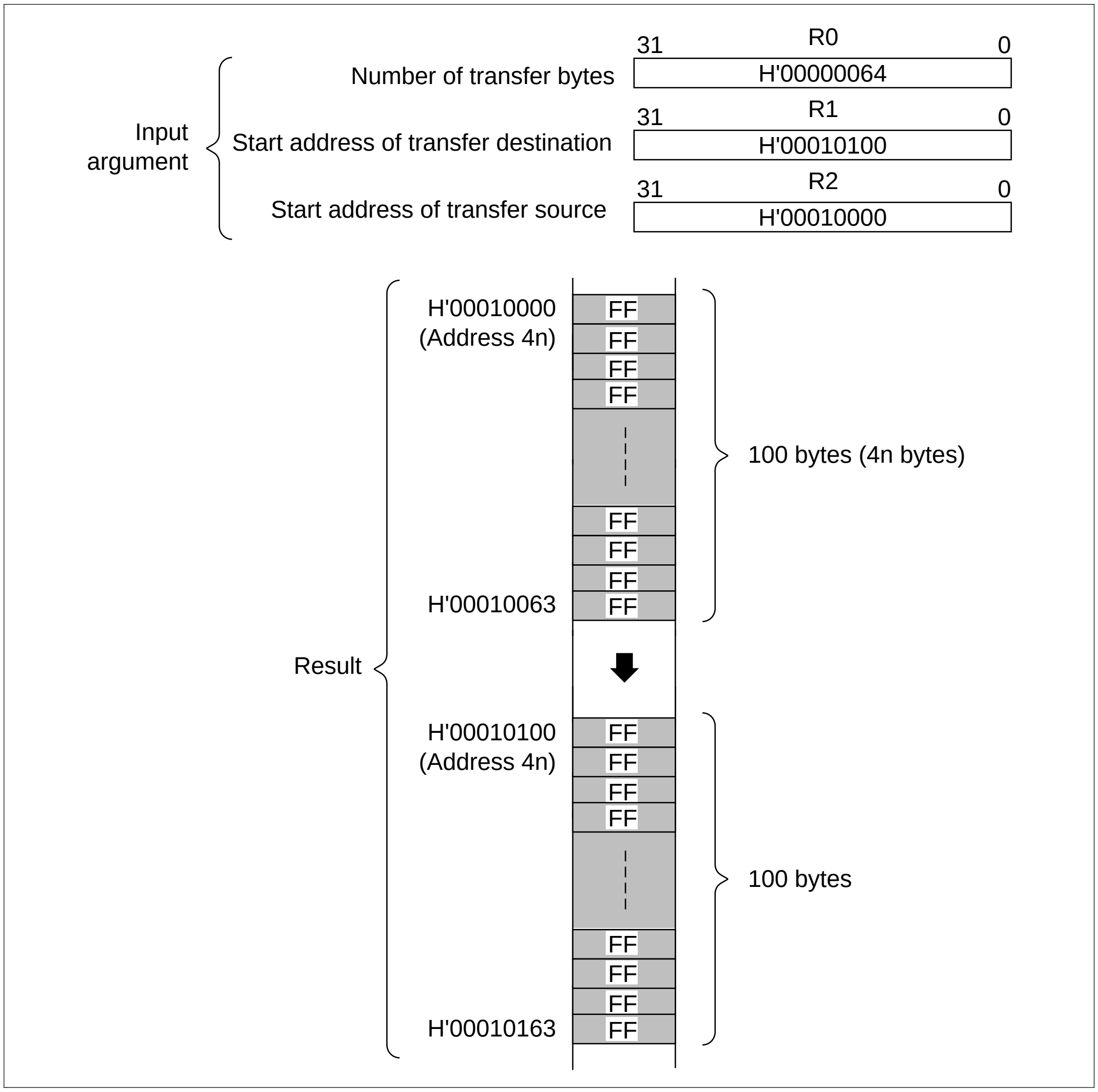


Figure 3.7 MOVE4 Execution Example

3.2.2 Precautions for MOVE4 Use

Set the input argument so that the transfer source and transfer destination areas do not overlap. If the two areas do overlap, the transfer source data in the overlapping portion will be destroyed (figure 3.8).

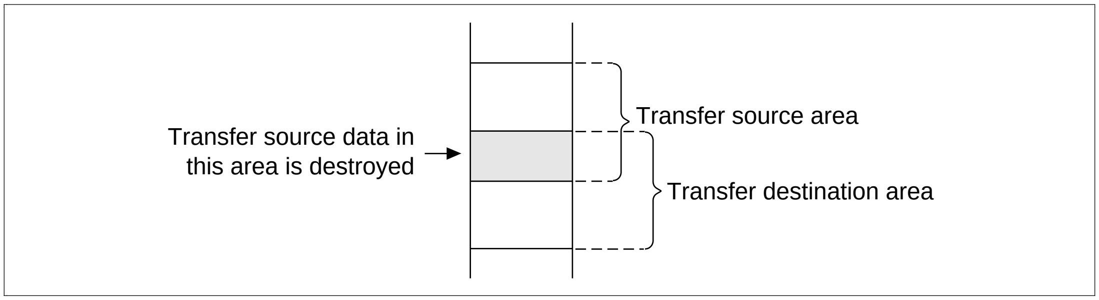


Figure 3.8 Block Transfer when Data Areas Overlap (MOVE4)

R0, R1, and R2,which contain, respectively, the number of transfer bytes, transfer destination area start address, and transfer source area start address, have their contents changed by execution of MOVE. Be sure, therefore, to save the number of transfer bytes and start addresses of transfer destination and source beforehand if these data are also required after a MOVE execution.

3.2.3 MOVE4 RAM Use

RAM is not used with MOVE4.

3.2.4 Example of MOVE4 Use

Make a subroutine call to MOVE4 after setting the transfer source area start address, transfer destination area start address, and number of transfer bytes in the input arguments.

`MOV.L`	`DATA1,R0`	Set number of transfer bytes in the input argument (R0)
`MOV.L`	`DATA2,R1`	Set the transfer destination area start address in the input argument (R1)
`BSR`	`MOVE4`	Subroutine call MOVE4
`MOV.L`	`DATA3,R2`	Set the transfer source area start address in the input argument (R2)

```
           ↓
        .align    4
DATA1   .data.l   H'00000064
DATA2   .data.l   H'00010100
DATA3   .data.l   H'00010000
```

3.2.5 MOVE4 Operation

Since transfer source and transfer destination addresses are both address 4n (4 bytes aligned), transfer is in 4-byte units from transfer source to destination.

Post-increment register indirect (@R2+) is used in specifying the transfer source address. The transfer source address is incremented automatically +4 at a time.

Register indirect with displacement is used in specifying the transfer destination address. Displacement is 0–60, so the transfer destination address must be incremented +64 for every 60 bytes of transfer. In other cases, increment processing is not required.

Set the transfer source start address (R2) + number of transfer bytes (R0) in R3. After setting, R0, which contains the number of transfer bytes, is used as a data transfer work register. After sending transfer source data to R0, R2 ≤ R3 testing is performed. When this condition is satisfied (R2 ≤ R3), data in R0 is the data within the transfer source area, so data is transferred to the destination. When the condition is not satisfied (R2 > R3), data in R0 is outside the the transfer source area, so transfer is halted.

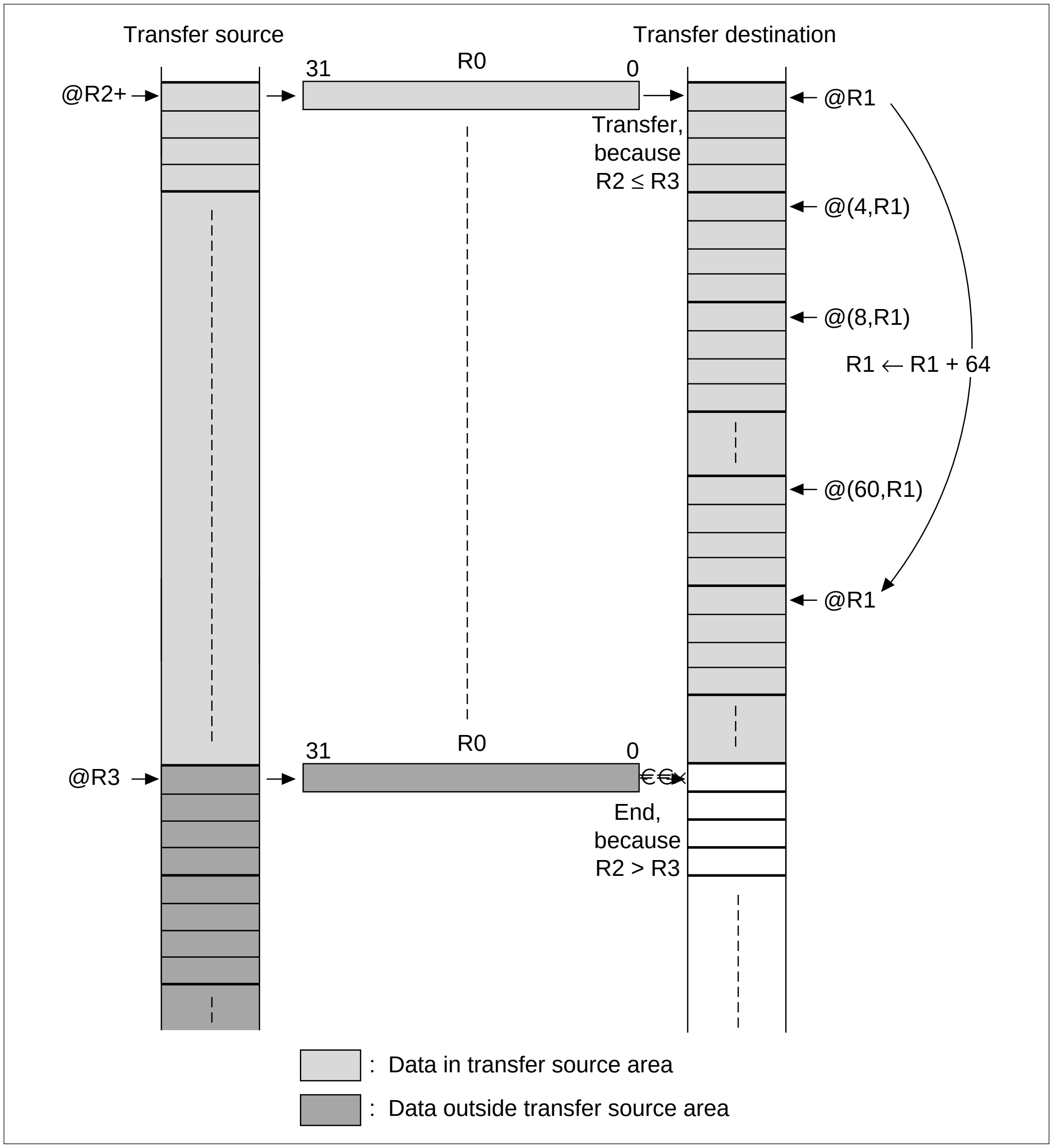


Figure 3.9 MOVE4 Data Transfer Method

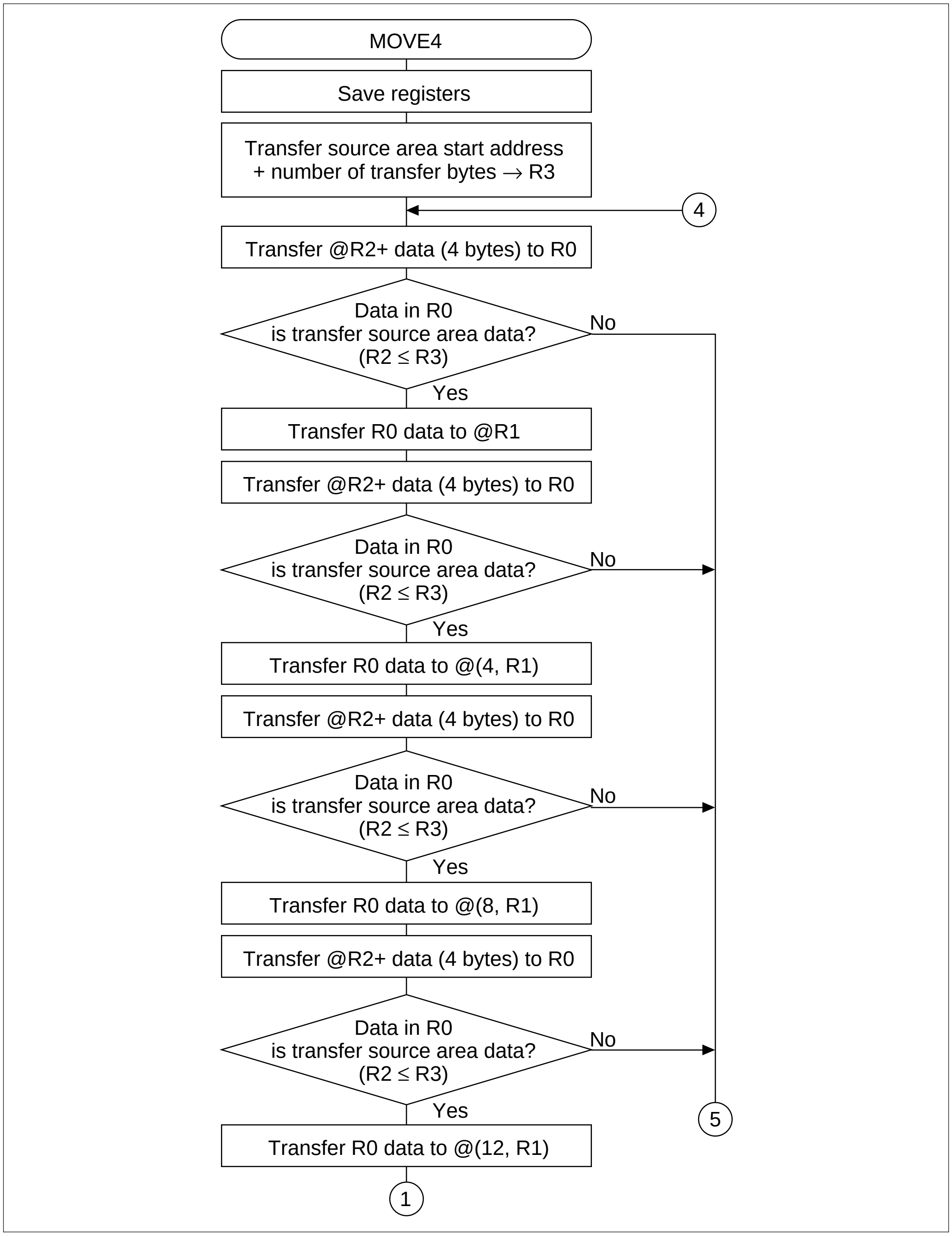


Figure 3.10 MOVE4 Flowchart

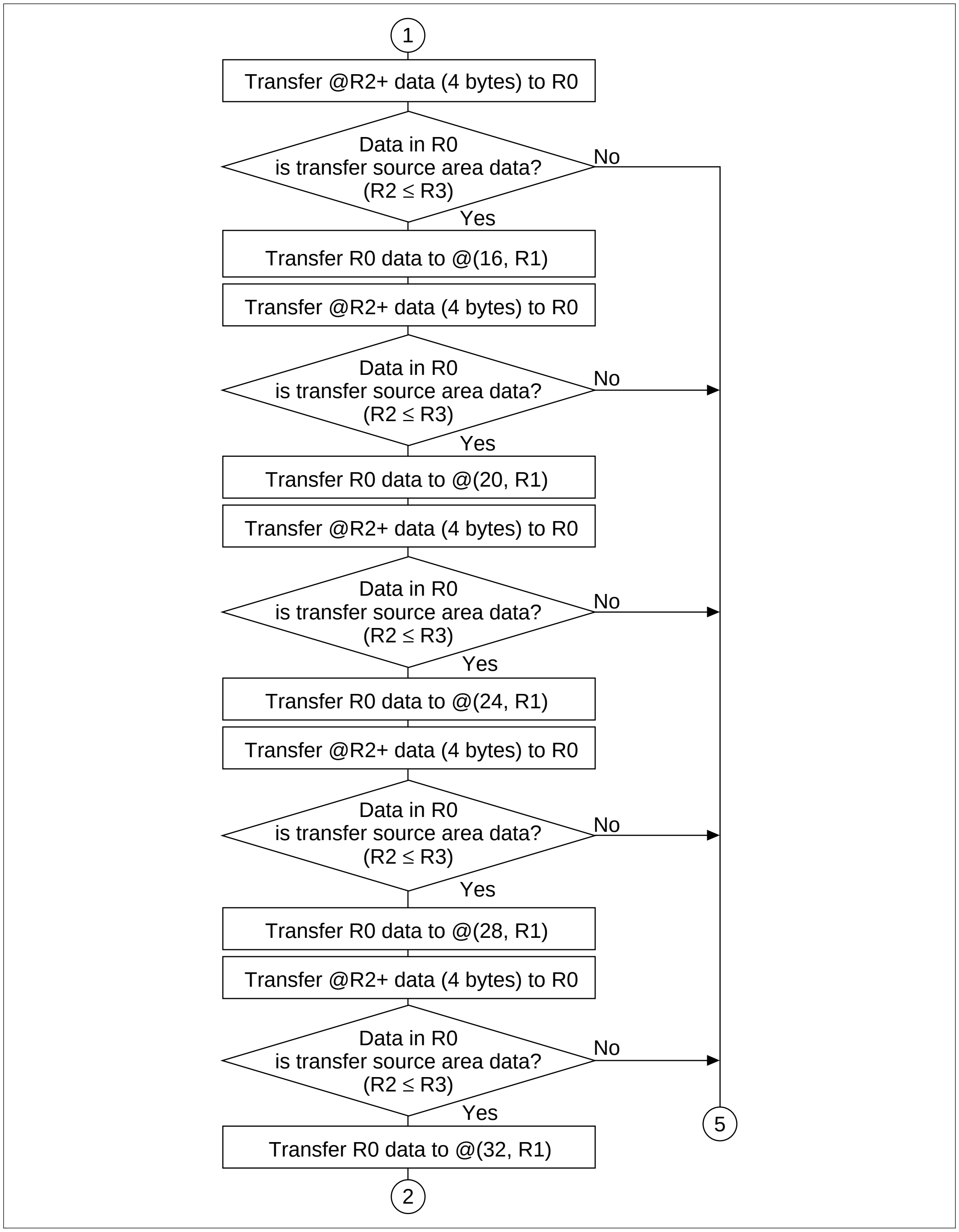


Figure 3.10 MOVE4 Flowchart (cont)

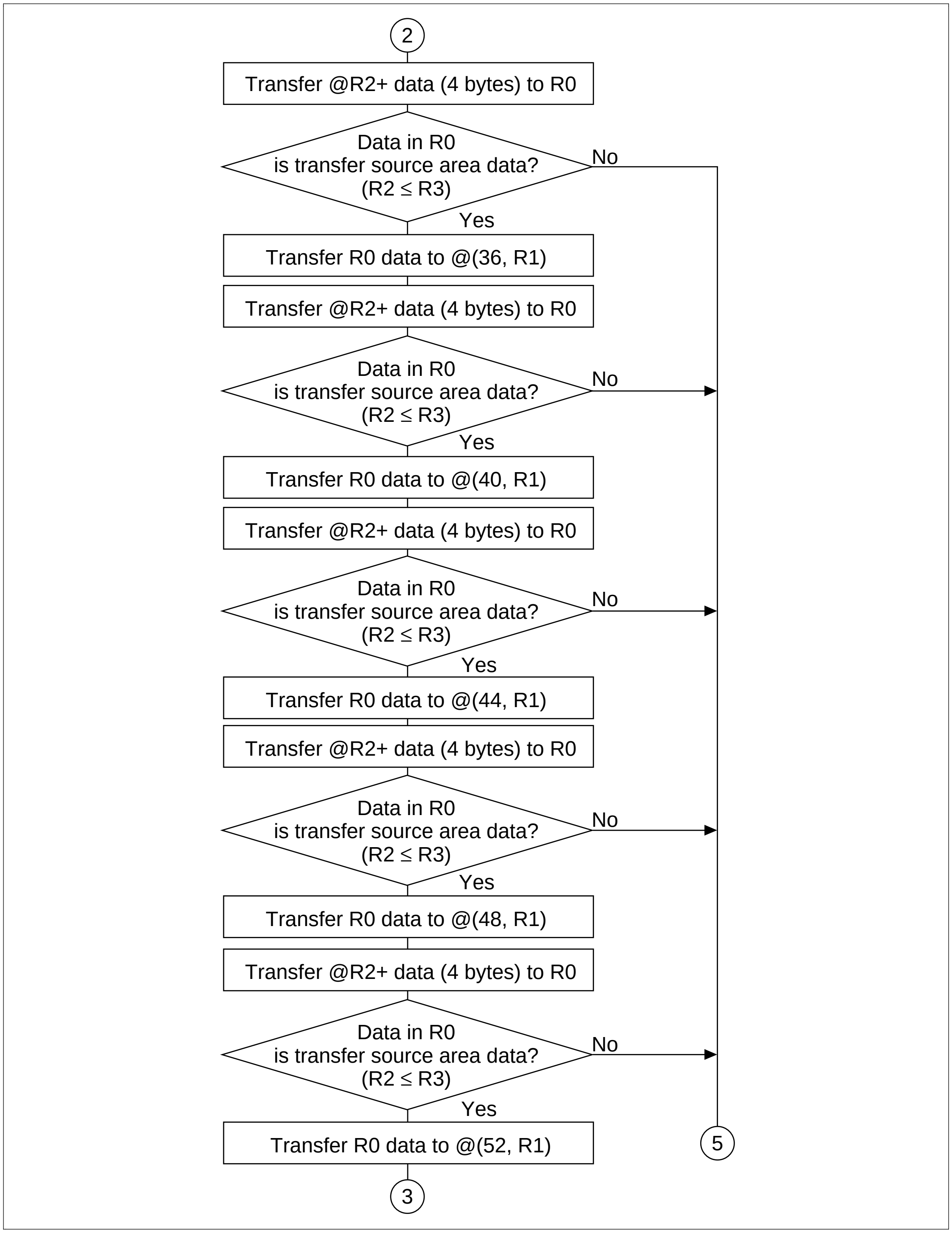


Figure 3.10 MOVE4 Flowchart (cont)

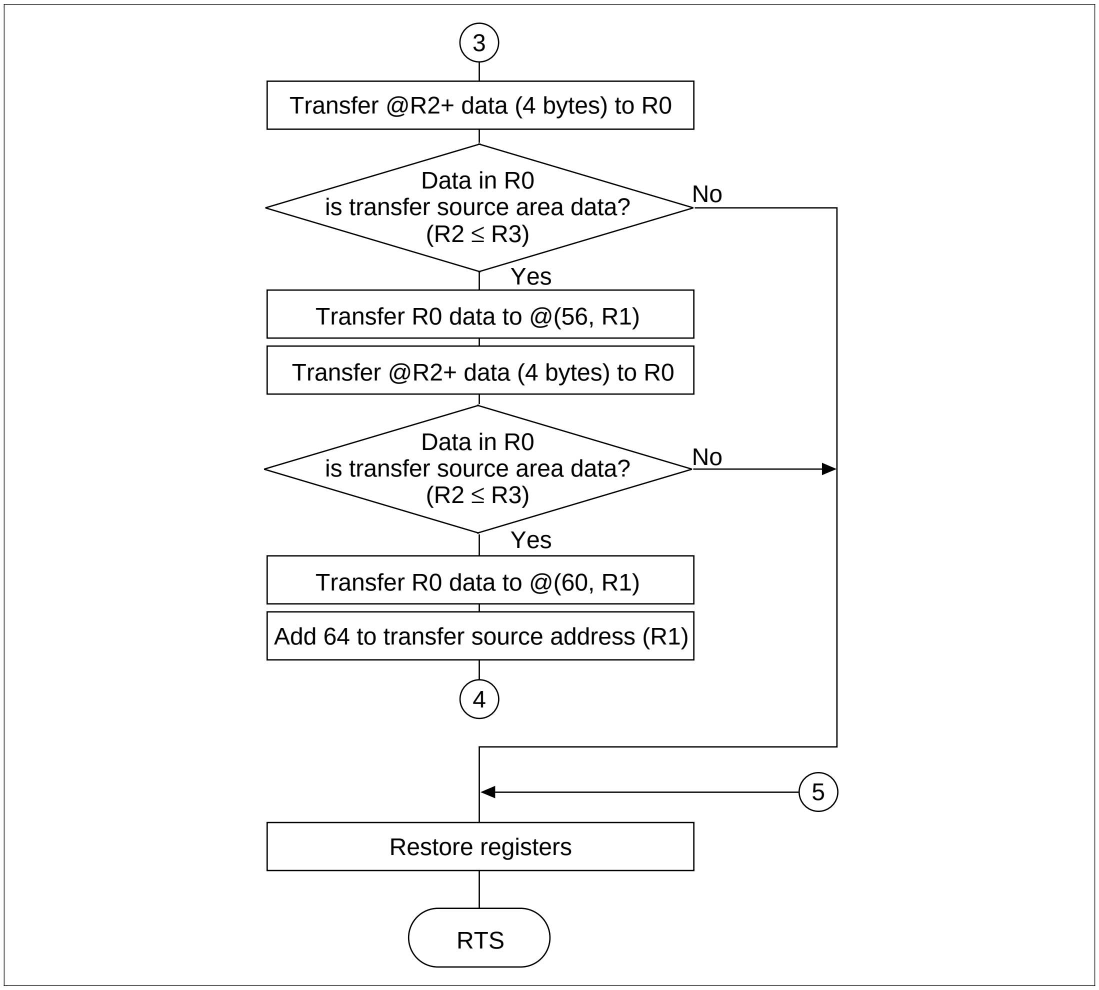


Figure 3.10 MOVE4 Flowchart (cont)

3.2.6 MOVE4 Program Listing

```
NAME:         MOVING MEMORY BLOCKS (MOVE4)
ENTRY:        R0      (NUMBER OF TRANSFER)
              R1      (DESTINATION ADDRESS)
              R2      (SOURCE ADDRESS)
RETURNS: NOTHING
                                           ;
                                           ;
                                           ;
                                           ;
                                           ;
                                           ;
                                           ;
                                           ;
                                           ;
                                           ;
                                           ;
                                           ;
00001000             .SECTION  A,CODE,LOCATE=H'1000
00001000             MOVE4 .EQU $          ;Entry point
00001000   2F36        MOV.L R3@-R15       ;Escape register
00001002   6323        MOV   R2,R3         ;
00001004   330C        ADD   R0,R3         ;
00001006             MOVE41                ;
00001006   6026        MOV.L  @R2+,R0      ;Load source data
00001008   3322        CMP/HS R2,R3        ;R2 ≤ R3?
0000100A   8B3E        BF     MOVE4_END    ;No
0000100C   2102        MOV.L  R0,@R1       ;Yes →Store source data
0000100E             MOVE42                ;
0000100E   6026        MOV.L  @R2+,R0      ;Load source data
00001010   3322        CMP/HS R2,R3        ;R2 ≤ R3?
00001012   8B3A        BF     MOVE4_END    ;No
00001014   1101        MOV.LR0 @(4,R1)     ;Yes →Store source data
00001016             MOVE43                ;
00001016   6026        MOV.L  @R2+,R0      ;Load source data
00001018   3322        CMP/HS R2,R3        ;R2 ≤ R3?
0000101A   8B36        BF     MOVE4_END    ;No
```

```
32  0000101C   1102   32    MOV.L  R0@(8,R1)  ;Yes →Store source data
33  0000101E          33  MOVE44              ;
34  0000101E   6026   34    MOV.L  @R2+,R0    ;Load source data
35  00001020   3322   35    CMP/HS R2,R3      ;R2 ≤ R3?
36  00001022   8B32   36    BF     MOVE4_END  ;No
37  00001024   1103   37    MOV.L R0@(12,R1)  ;Yes →Store source data
38  00001026          38  MOVE45              ;
39  00001026   6026   39    MOV.L  @R2+,R0    ;Load source data
40  00001028   3322   40    CMP/HS R2,R3      ;R2 ≤ R3?
41  0000102A   8B2E   41    BF     MOVE4_END  ;No
42  0000102C   1104   42    MOV.L R0,@(16,R1) ;Yes →Store source data
43  0000102E          43  MOVE46              ;
44  0000102E   6026   44    MOV.L  @R2+,R0    ;Load source data
45  00001030   3322   45    CMP/HS R2,R3      ;R2 ≤ R3?
46  00001032   8B2A   46    BF     MOVE4_END  ;No
47  00001034   1105   47    MOV.L R0,@(20,R1) ;Yes →Store source data
48  00001036          48  MOVE47              ;
49  00001036   6026   49    MOV.L  @R2+,R0    ;Load source data
50  00001038   3322   50    CMP/HS R2,R3      ;R2 ≤ R3?
51  0000103A   8B26   51    BF     MOVE4_END  ;No
52  0000103C   1106   52    MOV.L R0,@(24,R1) ;Yes →Store source data
53  0000103E          53  MOVE48              ;
54  0000103E   6026   54    MOV.L  @R2+,R0    ;Load source data
55  00001040   3322   55    CMP/HS R2,R3      ;R2 ≤ R3?
56  00001042   8B22   56    BF     MOVE4_END  ; No
57  00001044   1107   57    MOV.L R0,@(28,R1) ;Yes →Store source data
58  00001046          58  MOVE49              ;
59  00001046   6026   59    MOV.L  @R2+,R0    ;Load source data
60  00001048   3322   60    CMP/HS R2,R3      ; R2 ≤ R3?
61  0000104A   8B1E   61    BF     MOVE4_END  ;No
62  0000104C   1108   62    MOV.L R0,@(32,R1) ;Yes →Store source data
63  0000104E          63  MOVE410             ;
64  0000104E   6026   64    MOV.L  @R2+,R0    ;Load source data
65  00001050   3322   65    CMP/HS R2,R3      ;R2 ≤ R3?
66  00001052   8B1A   66    BF     MOVE4_END  ;No
67  00001054   1109   67    MOV.L R0,@(36,R1) ;Yes →Store source data
68  00001056          68  MOVE411             ;
```

```
00001056   6026      MOV.L  @R2+,R0    ;Load source data
00001058   3322      CMP/HS R2,R3      ;R2 ≤ R3?
0000105A   8B16      BF     MOVE4_END  ; No
0000105C   110A      MOV.L R0,@(40,R1) ;Yes →Store source data
0000105E           MOVE412             ;
0000105E   6026      MOV.L  @R2+,R0    ;Load source data
00001060   3322      CMP/HS R2,R3      ;R2 ≤ R3?
00001062   8B12      BF     MOVE4_END  ;No
00001064   110B      MOV.L R0,@(44,R1) ;Yes →Store source data
00001066           MOVE413             ;
00001066   6026      MOV.L  @R2+,R0    ;Load source data
00001068   3322      CMP/HS R2,R3      ;R2 ≤ R3?
0000106A   8B0E      BF     MOVE4_END  ;No
0000106C   110C      MOV.L R0,@(48,R1) ;Yes →Store source data
0000106E           MOVE414             ;
0000106E   6026      MOV.L  @R2+,R0    ;Load source data
00001070   3322      CMP/HS R2,R3      ;R2 ≤ R3?
00001072   8B0A      BF     MOVE4_END  ;No
00001074   110D      MOV.L R0,@(52,R1) ;Yes →Store source data
00001076           MOVE415             ;
00001076   6026      MOV.L  @R2+,R0    ;Load source data
00001078   3322      CMP/HS R2,R3      ;R2 ≤ R3?
0000107A   8B06      BF     MOVE4_END  ;No
0000107C   110E      MOV.L R0,@(56,R1) ;Yes →Store source data
0000107E           MOVE416             ;
0000107E   6026      MOV.L  @R2+,R0    ;Load source data
00001080   3322      CMP/HS R2,R3      ;R2 ≤ R3?
00001082   8B02      BF     MOVE4_END  ;No
00001084   110F      MOV.L R0,@(60,R1) ;Yes →Store source data
                                       ;
00001086   AFBE      BRA  MOVE41       ;
00001088   7140      ADD  #D'64,R1     ;R1 ← R1 + 64
0000108A            MOVE4_END          ;
0000108A  000B       RTS               ;
0000108C  63F6       MOV.L  @R15+,R3   ;Return register
                     .END
*****TOTAL  ERRORS 0
```

```
*****TOTAL  WARNINGS 0
```

3.3 SHARN: Multi-Bit Shift of 32-Bit Data (Arithmetic Right Shift)

- Instruction: SHLR2, SHLR8, SHLR16
- Function: Multi-bit (0–31) arithmetic right shift of 32-bit data.

Table 3.6 SHARN Arguments

	Contents	Storage Location	Data Length (Bytes)
Input	Number of shifts (0–31)	R0	4
	32-bit data before shift	R1	4
Output	32-bit data after shift	R1	4

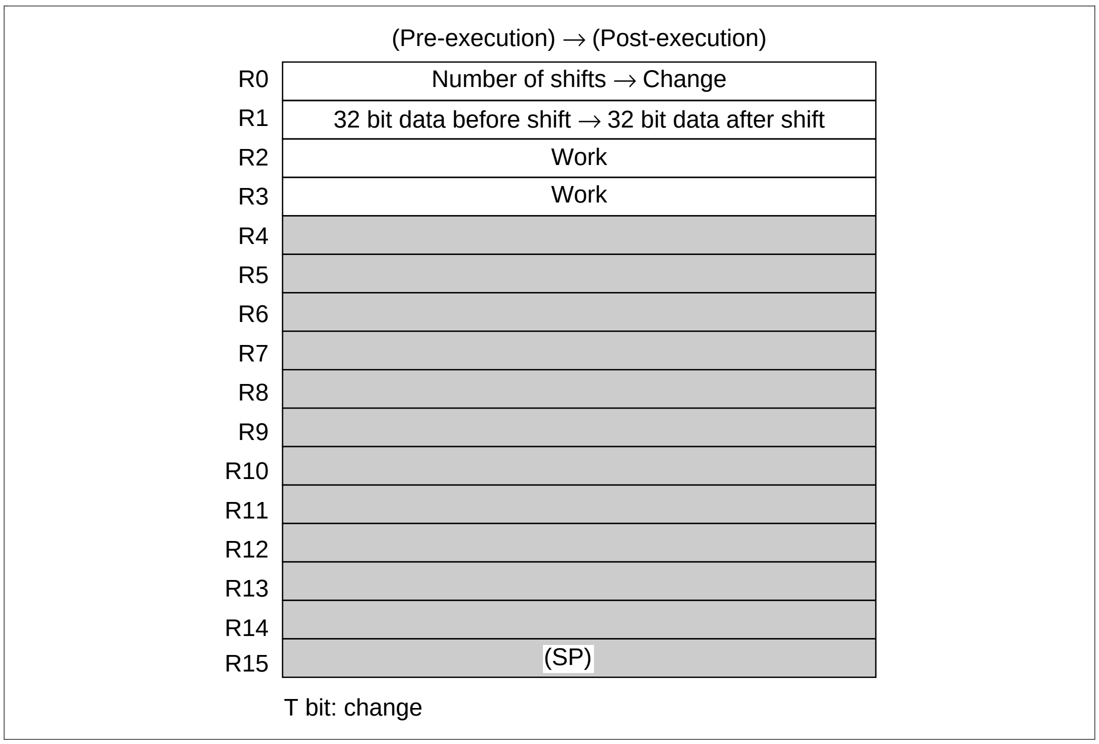


Figure 3.11 SHARN Internal Register Change and Flag Change

Table 3.7 SHARN Programming Specifications

Item	Value/State
Program memory (bytes)	74
Data memory (bytes)	0
Stack (bytes)	8
Number of states	38
Reentrant	Enabled
Relocation	
Intermediate interrupt	

- Precautions: The number of states in the programming specifications is the value for a 31-bit shift.

3.3.1 SHARN Arguments

- R0: Holds the number of shifts (0–31) as the input argument.
- R1: Holds the 32-bit data before shift as the input argument.
  Holds the 32-bit data after shift as the output argument.

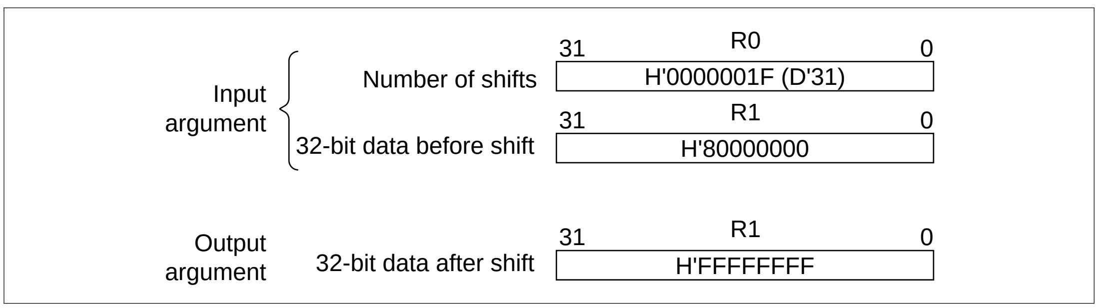


Figure 3.12 SHARN Execution Example

3.3.2 Precautions for SHARN Use

Since 32-bit data after shift is set in R1, which has the 32-bit data set before shift, the data before the shift is destroyed. Also, R0, which sets the number of shifts, has its contents changed by execution of SHARN.

If the 32-bit data before shift and the number of shifts are needed also after SHARN execution, be sure to save these data beforehand.

3.3.3 SHARN RAM Use

RAM is not used with SHARN.

3.3.4 Example of SHARN Use

Set the 32-bit data before shift and number of shifts in the input arguments, and make a subroutine call to SHARN.

`MOV #H'05,R0`	Set number of shifts in the input argument (R0)
`BSR SHARN`	Subroutine call SHARN
`MOV.L DATA,R1`	Set 32-bit data before shift in the input argument (R1)

```
        ↓
        .align  4
DATA    .data.l    H'80000000
```

3.3.5 SHARN Operation

SHARN tests each bit from bit 4 through bit 0 of the number of shift cycles set in R0, and if 1, uses SHLR16 to shift each bit 16 bits logically right, uses SHLR8 for an 8-bit logical right shift, SHLR2 for a 2-bit logical right shift, and SHLR for a 1-bit logical right shift (table 3.8).

Table 3.8 Number of Shifts and Instruction for Each Bit (SHARN)

Bit Number	Weighting	Instruction
Bit 4	$2^4 = 16$	SHLR16
Bit 3	$2^3 = 8$	SHLR8
Bit 2	$2^2 = 4$	SHLR2 (twice)
Bit 1	$2^1 = 2$	SHLR2
Bit 0	$2^0 = 1$	SHLR

Since 32-bit data before shift is shifted 16 bits, 8 bits, 2 bits, and 1 bit due to logical right shift instructions, when the MSB of 32-bit data before shift is 1, the vacated MSB after the shift becomes not 1 but 0.

Therefore, if R2, which contains H'FFFFFFFF, is shifted by the same shift number logically right as the 32-bit data before shift, when the MSB before shift is 1, at the end of the shift the MSB of the shift number portion is set to 1 by a logical OR with the inverted R2 value.

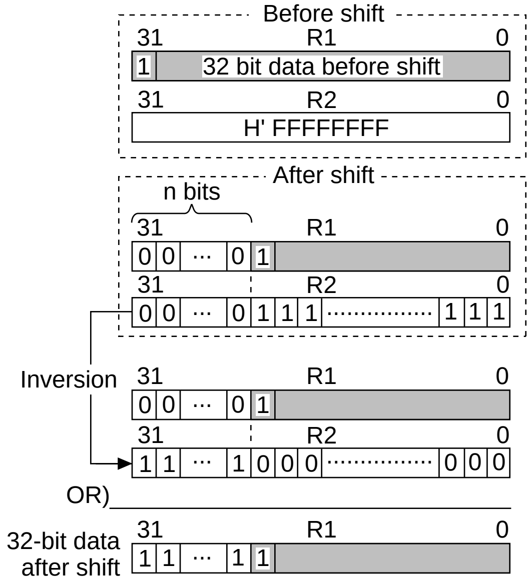


Figure 3.13 Multiple Bit Shift (SHARN)

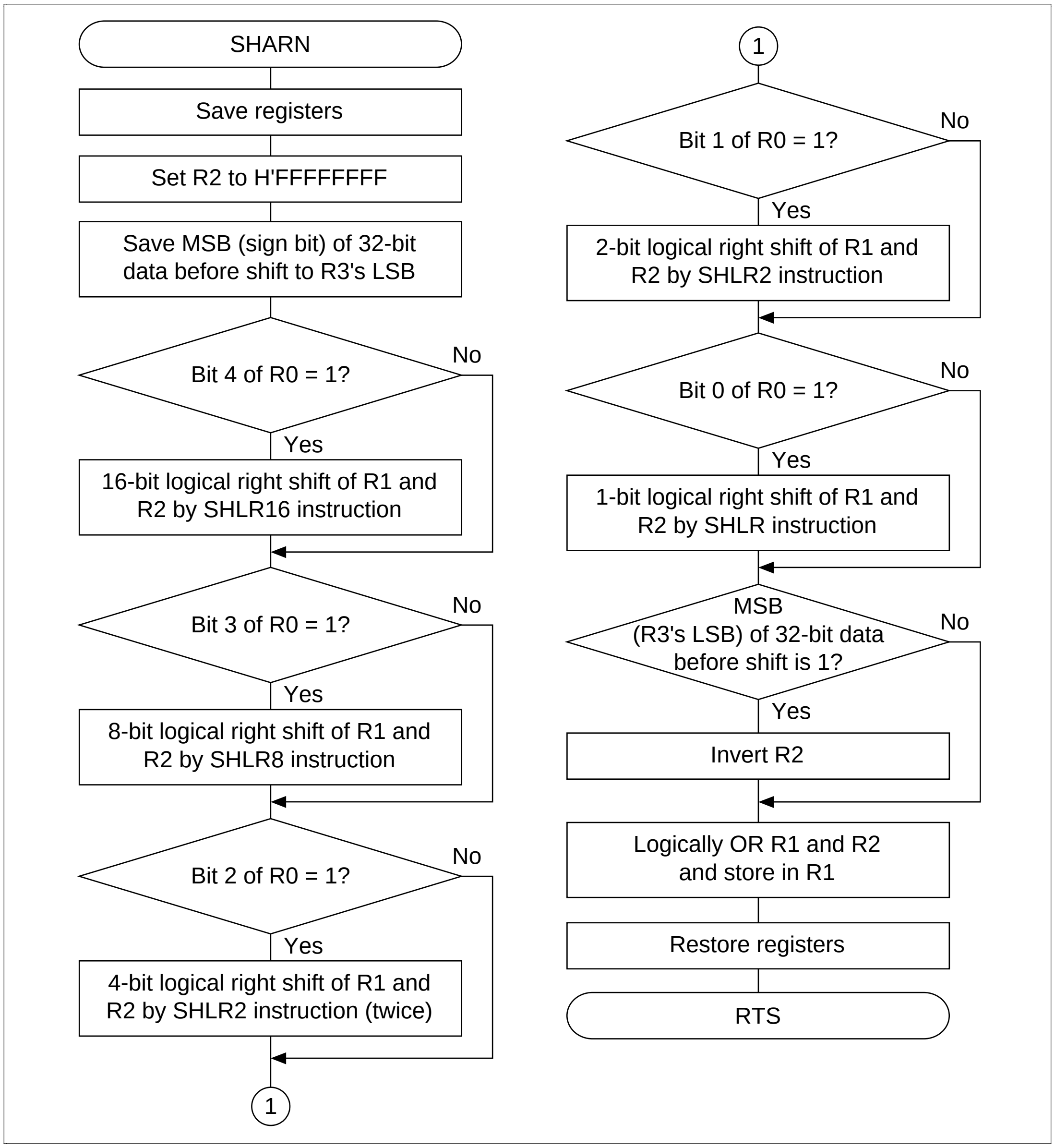


Figure 3.14 SHARN Flowchart

3.3.6 SHARN Program Listing

```
NAME:     n BITS SHIFT ARITHMETIC  RIGHT (SHARN)
ENTRY:    R0 (NUMBER OF BIT SHIFTED)
          R1 (32 BIT DATA)
RETURNS: R1 (SHIFT RESULT)
                                              ;
                                              ;
                                              ;
                                              ;
                                              ;
                                              ;
                                              ;
                                              ;
                                              ;
                                              ;
                                              ;
00001000          .SECTION  A,CODE,LOCATE=H'1000
00001000          SHARN .EQU  $               ;Entry point
00001000  2F26      MOV.L  R2,@-R15           ;Escape register
00001002  2F36      MOV.L  R3,@-R15           ;
00001004          SHARN1                      ;
00001004  3228      SUB    R2,R2              ;R2 ← H'FFFFFFFF
00001006  6227      NOT    R2,R2              ;
00001008          SHARN2                      ;
00001008  4104      ROTL   R1                 ;R3 ←MSB of 32 bit data
0000100A  0329      MOVT   R3                 ;
0000100C  4105      ROTR   R1                 ;
0000100E          SHARN3                      ;
0000100E  C810      TST #B'00010000,R0;Bit4 = 1?
00001010  8901      BT  SHARN4                ;No
00001012  4129      SHLR16 R1            ;16 bit shift logical right
00001014  4229      SHLR16 R2                 ;
00001016          SHARN4              ;
00001016  C808      TST #B'00001000,R0;Bit3 = 1?
00001018  8901      BT     SHARN5             ;No
0000101A  4119      SHLR8  R1            ;8 bit shift logical right
0000101C  4219      SHLR8  R2                 ;
```

```
33  0000101E          33  SHARN5               ;
34  0000101E   C804   34    TST #B'00000100,R0;Bit2 = 1?
35  00001020   8903   35    BT  SHARN6         ;No
36  00001022   4109   36    SHLR2  R1        ;4 bit shift logical right
37  00001024   4109   37    SHLR2  R1          ;
38  00001026   4209   38    SHLR2  R2          ;
39  00001028   4209   39    SHLR2  R2          ;
40  0000102A          40  SHARN6               ;
41  0000102A   C802   41    TST #B'00000010,R0;Bit1 = 1?
42  0000102C   8901   42    BT  SHARN7         ;No
43  0000102E   4109   43    SHLR2  R1        ;2 bit shift logical right
44  00001030   4209   44    SHLR2  R2          ;
45  00001032          45  SHARN7               ;
46  00001032   C801   46    TST #B'00000001,R0;Bit0 = 1?
47  00001034   8901   47    BT  SHARN8         ;No
48  00001036   4101   48    SHLR   R1        ;1 bit shift logical right
49  00001038   4201   49    SHLR   R2          ;
50  0000103A          50  SHARN8               ;
51  0000103A   6033   51    MOV R3,R0        ;
52  0000103C   C801   52    TST #B'00000001,R0;MSB of 32 bit data = 1?
53  0000103E   8901   53    BT  SHARN_END      ;No
54  00001040   6227   54    NOT R2,R2          ;
55  00001042   212B   55    OR  R2,R1          ;
56  00001044          56  SHARN_END            ;
57  00001044   63F6   57    MOV.L  @R15+,R3    ;Return register
58  00001046   000B   58    RTS                ;
59  00001048   62F6   59    MOV.L  @R15+,R2    ;
60                    60    .END
*****TOTAL  ERRORS 0
*****TOTAL  WARNINGS 0
```

3.4 SHLRN: Multi-Bit Shift of 32-Bit Data (Logical Right Shift)

- Instructions: SHLR2, SHLR8, SHLR16
- Function: Multi-bit (0–31) logical right shift of 32-bit data.

Table 3.9 SHLRN Arguments

	Contents	Storage Location	Data Length (Bytes)
Input	Number of shifts (0–31)	R0	4
	32-bit data before shift	R1	4
Output	32-bit data after shift	R1	4

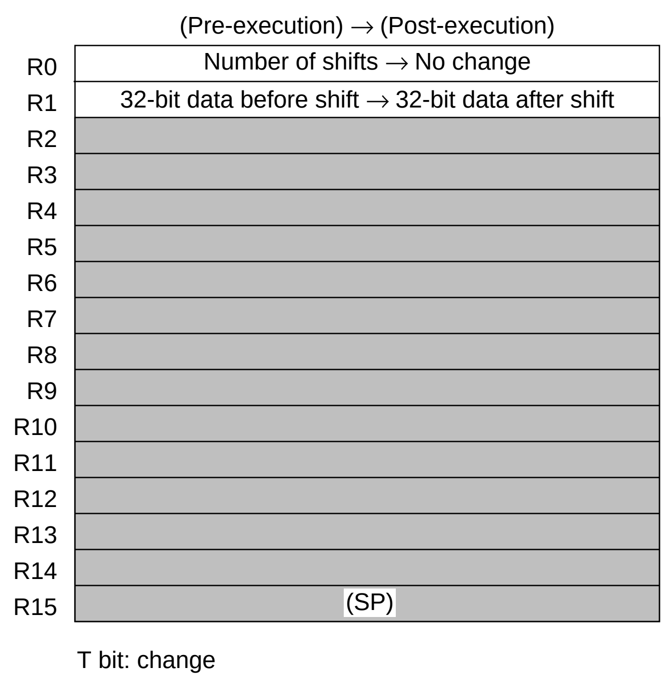


Figure 3.15 SHLRN Internal Register Change and Flag Change

Table 3.10SHLRN Programming Specifications

Item	Value/State
Program memory (bytes)	36
Data memory (bytes)	0
Stack (bytes)	0
Number of states	19
Reentrant	Enabled
Relocation	
Intermediate interrupt	

- • Precautions: The number of states in the programming specifications is the value at the time of a 31-bit shift.

3.4.1 SHLRN Arguments

- R0: Holds the number of shifts (0–31) as the input argument.
- R1: Holds the 32-bit data before shift as the input argument.
  Holds the 32-bit data after shift as the output argument.

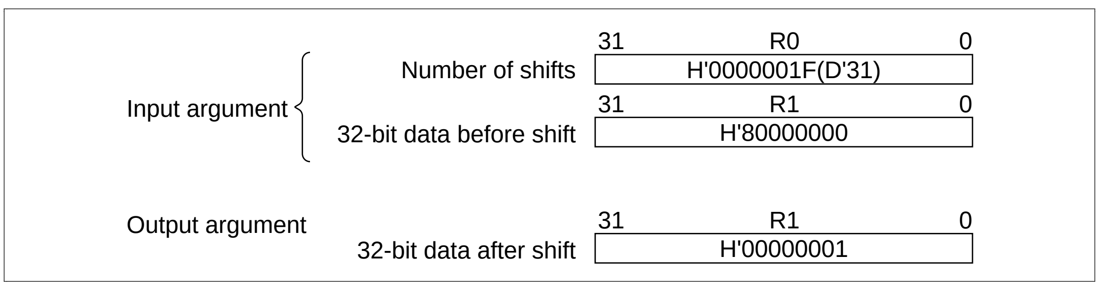


Figure 3.16 SHLRN Execution Example

3.4.2 Precautions for SHLRN Use

Since 32-bit data after shift is set in R1, which sets the 32-bit data before shift, the data before the shift is destroyed.

If the 32-bit data before shift is needed also after SHLRN execution, be sure to save the 32-bit data beforehand.

3.4.3 SHLRN RAM Use

RAM is not used with SHLRN.

3.4.4 Example of SHLRN Use

Set the number of shifts and 32-bit data before shift in the input argument, and make a subroutine call to SHLRN.

`MOV #H'05, R0`	Set number of shifts in the input argument (R0)
`BSR SHLRN`	Subroutine call SHLRN
`MOV.L DATA, R1`	Set 32-bit data before shift in the input argument (R1)

```
           ↓
        .align  4
DATA    .data.l    H'80000000
```

3.4.5 SHLRN Operation

SHLRN tests each bit from bit 4 through bit 0 of the number of shifts set in R0, and if 1, uses SHLR16 to shift the weighting of each bit 16 bits logically right, uses SHLR8 for an 8-bit logical right shift, SHLR2 for a 2-bit logical right shift, and SHLR for a 1-bit logical right shift (table 3.11).

Table 3.11 Number of Shifts and Instruction Used for Each Bit (SHLRN)

Bit Number	Weighting	Instruction Used
Bit 4	$2^4 = 16$	SHLR16
Bit 3	$2^3 = 8$	SHLR8
Bit 2	$2^2 = 4$	SHLR2 (twice)
Bit 1	$2^1 = 2$	SHLR2
Bit 0	$2^0 = 1$	SHLR

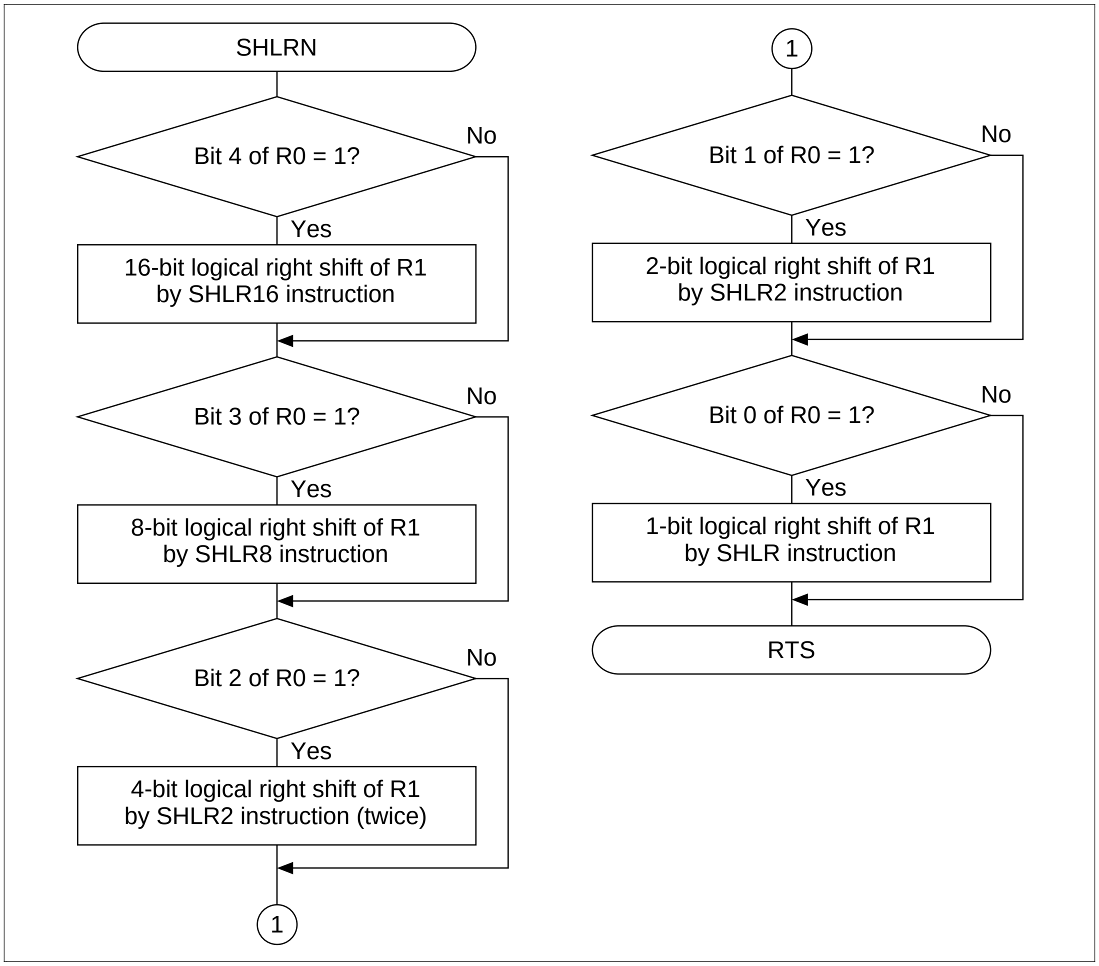


Figure 3.17 SHLRN Flowchart

3.4.6 SHLRN Program Listing

```
NAME:     n BITS SHIFT LOGICAL RIGHT (SHLRN)
ENTRY:    R0 (NUMBER OF BIT SHIFTED)
          R1 (32 BIT DATA)
RETURNS:  R1 (SHIFT RESULT)
1                      1                       ;
2                      2                       ;
3                      3                       ;
4                      4                       ;
5                      5                       ;
6                      6                       ;
7                      7                       ;
8                      8                       ;
9                      9                       ;
10                     10                      ;
11                     11                      ;
12  00001000           12  .SECTION A,CODE,LOCATE=H'1000
13  00001000           13  SHLRN  .EQU  $      ;Entry point
14  00001000           14  SHLRN1              ;
15  00001000   C810    15    TST #B'00010000,R0;Bit 4 = 1?
16  00001002   8900    16    BT  SHLRN2        ;No
17  00001004   4129    17    SHLR16 R1      ;16 bit shift logical right
18  00001006           18  SHLRN2              ;
19  00001006   C808    19    TST #B'00001000,R0;Bit 3 = 1?
20  00001008   8900    20    BT  SHLRN3        ; No
21  0000100A   4119    21    SHLR8 R1       ;8 bit shift logical right
22  0000100C           22  SHLRN3              ;
23  0000100C   C804    23    TST #B'00000100,R0;Bit 2 = 1?
24  0000100E   8901    24    BT  SHLRN4        ; No
25  00001010   4109    25    SHLR2 R1        ;4 bit shift logical right
26  00001012   4109    26    SHLR2 R1          ;
27  00001014           27  SHLRN4              ;
28  00001014   C802    28    TST #B'00000010,R0;Bit 1 = 1?
29  00001016   8900    29    BT  SHLRN5        ; No
30  00001018   4109    30    SHLR2  R1       ;2 bit shift logical right
31  0000101A           31  SHLRN5              ;
32  0000101A   C801    32    TST #B'00000001,R0;Bit 0 = 1?
```

```
33  0000101C   8900   33    BT  SHLRN_END     ; No
34  0000101E   4101   34    SHLR R1         ;1 bit shift logical right
35  00001020          35  SHLRN_END           ;
36  00001020   000B   36    RTS                 ;
37  00001022   0009   37    NOP                 ;
38                    38    .END
*****TOTAL  ERRORS 0
*****TOTAL  WARNINGS 0
```

3.5 SHLLN: Multi-Bit Shift of 32-Bit Data (Logical Left Shift)

- Instructions: SHLL2, SHLL8, SHLL16
- Function: Multi-bit (0–31) logical left shift of 32-bit data.

Table 3.12 SHLLN Arguments

	Contents	Storage Location	Data Length (Bytes)
Input	Number of shifts (0–31)	R0	4
	32-bit data before shift	R1	4
Output	32-bit data after shift	R1	4

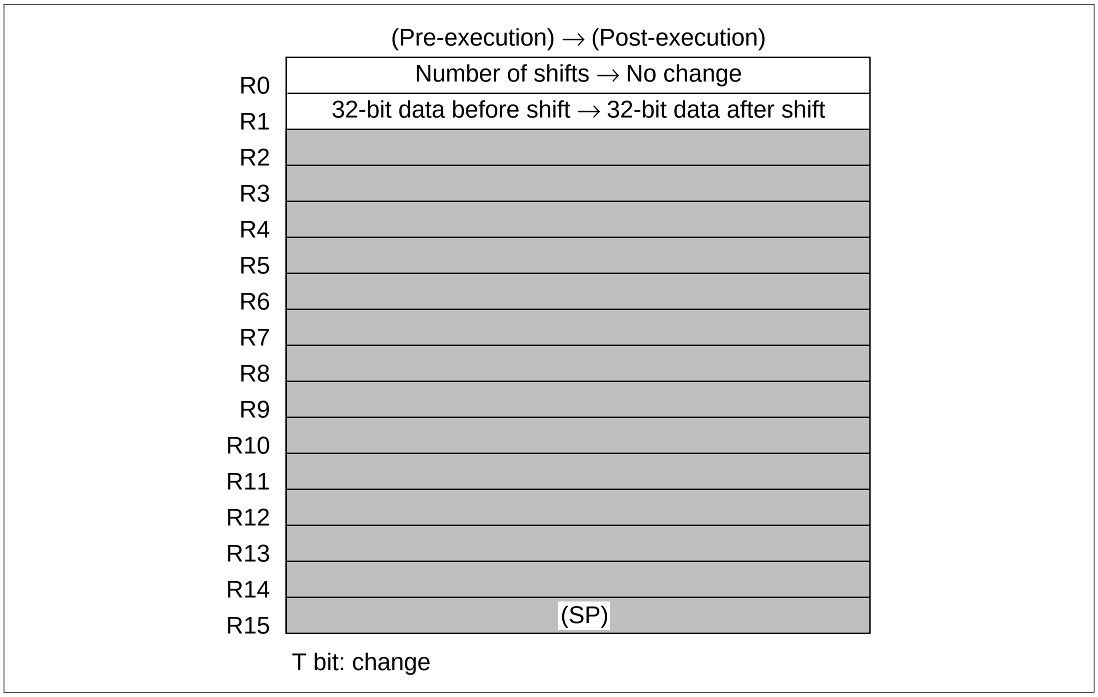


Figure 3.18 SHLLN Internal Register Change and Flag Change

Table 3.13 SHLLN Programming Specifications

Item	Value/State
Program memory (bytes)	36
Data memory (bytes)	0
Stack (bytes)	0
Number of states	19
Reentrant	Enabled
Relocation	
Intermediate interrupt	

- Precautions: The number of states in the programming specifications is the value at the time of a 31-bit shift.

3.5.1 SHLLN Arguments

- R0: Holds the number of shifts (0–31) as the input argument.
- R1: Holds the 32-bit data before shift as the input argument.
  Holds the 32-bit data after shift as the output argument.

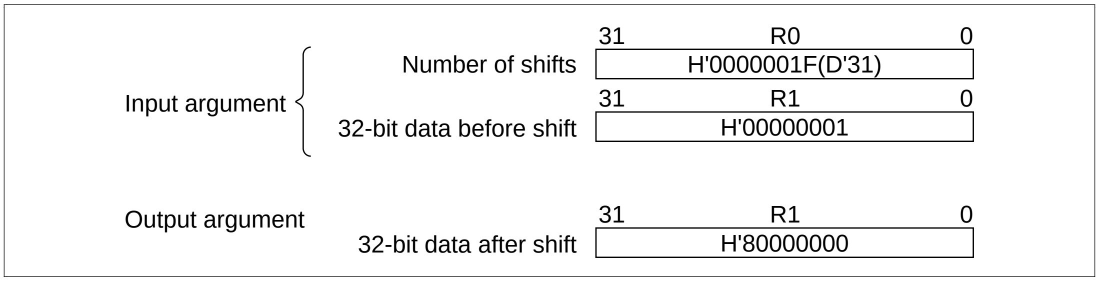


Figure 3.19 SHLLN Execution Example

3.5.2 Precautions for SHLLN Use

Since 32-bit data after shift is set in R1, which sets the 32-bit data before shift, the data before the shift is destroyed.

If the 32-bit data before shift is needed also after SHLLN execution, be sure to save the 32-bit data beforehand.

3.5.3 SHLLN RAM Use

RAM is not used with SHLLN.

3.5.4 Example of SHLLN Use

Set the number of shifts and 32-bit data before shift in the input argument, and subroutine call SHLLN.

```
      MOV     #H'05,R0          Set number of shifts in the input argument (R0)
      BSR     SHLLN             Subroutine call SHLLN
      MOV.L   DATA,R1           Set 32-bit data before shift in the input argument
                                (R1)
          ↓
       .align  4
DATA   .data.l   H'00000001
```

3.5.5 SHLLN Operation

SHLLN tests each bit from bit 4 through bit 0 of the number of shifts set in R0, and if 1, uses SHLL16 to shift the weighting of each bit 16 bits logically left, uses SHLL8 for an 8-bit logical left shift, SHLL2 for a 2-bit logical left shift, and SHLL for a 1-bit logical left shift (table 3.14).

Table 3.14 Number of Shifts and Instruction for Each Bit (SHLLN)

Bit Number	Weighting	Instruction
Bit 4	$2^4 = 16$	SHLL16
Bit 3	$2^3 = 8$	SHLL8
Bit 2	$2^2 = 4$	SHLL2 (twice)
Bit 1	$2^1 = 2$	SHLL2
Bit 0	$2^0 = 1$	SHLL

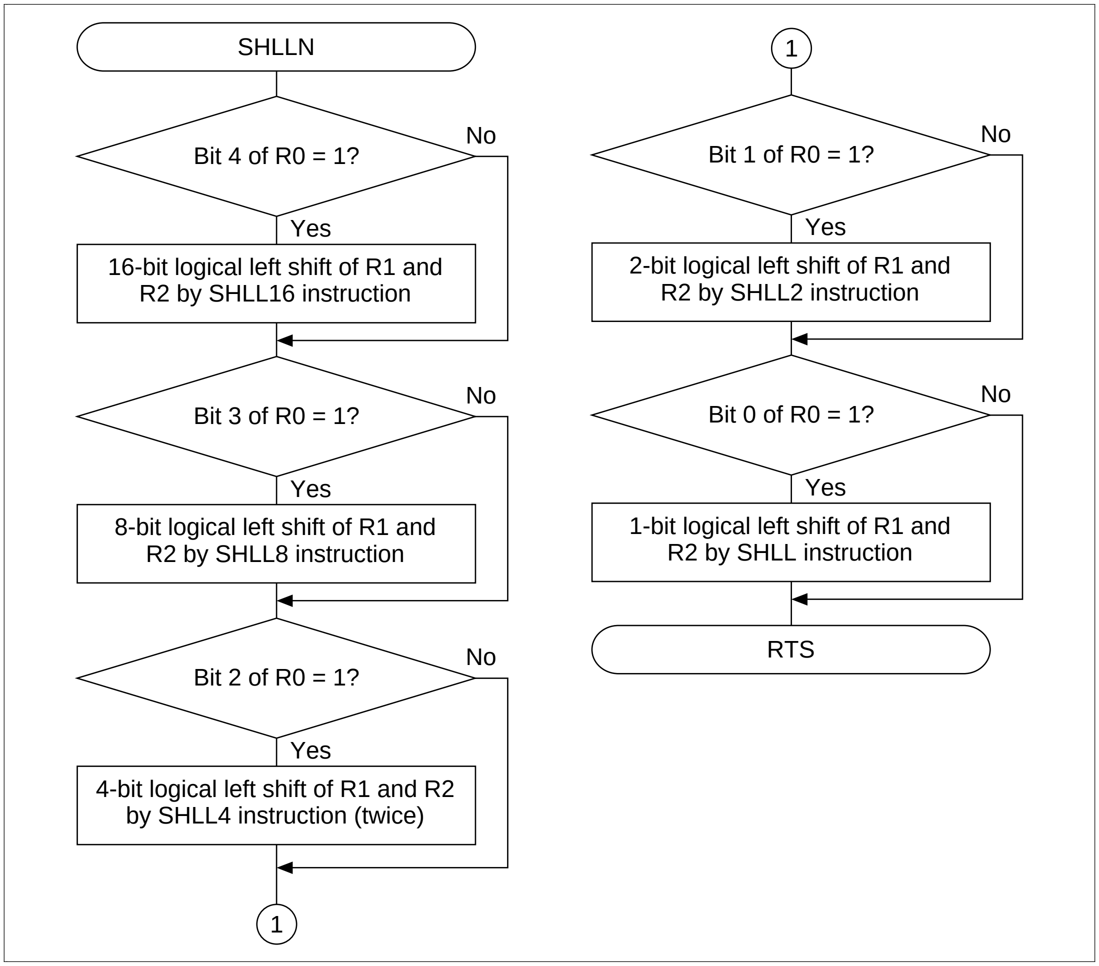


Figure 3.20 SHLLN Flowchart

3.5.6 SHLLN Program Listing

```
NAME:      n BITS SHIFT LOGICAL LEFT (SHLLN)
ENTRY:     R0 (NUMBER OF BIT SHIFTED)
           R1 (32 BIT DATA)
RETURNS:   R1 (SHIFT RESULT)
1                     1                        ;
2                     2                        ;
3                     3                        ;
4                     4                        ;
5                     5                        ;
6                     6                        ;
7                     7                        ;
8                     8                        ;
9                     9                        ;
10                    10                       ;
11                    11                       ;
12  00001000          12  .SECTION  A,CODE,LOCATE=H'1000
13  00001000          13  SHLLN  .EQU  $       ;Entry point
14  00001000          14  SHLLN1               ;
15  00001000   C810   15    TST #B'00010000,R0;Bit 4 = 1?
16  00001002   8900   16    BT  SHLLN2         ; No
17  00001004   4128   17    SHLL16  R1       ;16 bit shift logical left
18  00001006          18  SHLLN2               ;
19  00001006   C808   19    TST #B'00001000,R0;Bit 3 = 1?
20  00001008   8900   20    BT  SHLLN3         ;No
21  0000100A   4118   21    SHLL8  R1         ;8 bit shift logical left
22  0000100C          22  SHLLN3               ;
23  0000100C   C804   23    TST #B'00000100,R0;Bit 2 = 1?
24  0000100E   8901   24    BT  SHLLN4         ; No
25  00001010   4108   25    SHLL2  R1         ;4 bit shift logical left
26  00001012   4108   26    SHLL2  R1          ;
27  00001014          27  SHLLN4               ;
28  00001014   C802   28    TST #B'00000010,R0;Bit 1 = 1?
29  00001016   8900   29    BT  SHLLN5         ;No
30  00001018   4108   30    SHLL2  R1         ;2 bit shift logical left
31  0000101A          31  SHLLN5               ;
32  0000101A   C801   32    TST #B'00000001,R0;Bit 0 = 1?
```

```
33  0000101C   8900   33    BT  SHLLN_END     ;No
34  0000101E   4100   34    SHLL  R1         ;1 bit shift logical left
35  00001020          35  SHLLN_END           ;
36  00001020   000B   36    RTS               ;
37  00001022   0009   37    NOP               ;
38                    38    .END
*****TOTAL  ERRORS 0
*****TOTAL  WARNINGS 0
```

3.6 FIND1: Find First 1 in 32-Bit Data

- Instruction: SHLL
- Function: Tests each bit in sequence from the MSB, and determines the number of the bit (0–31) in which the first 1 occurs.

Table 3.15 FIND1 Arguments

	Contents	Storage Location	Data Length (Bytes)
Input	32-bit data for detection	R0	4
Output	Number of first detected 1 bit (0–31)	R1	4

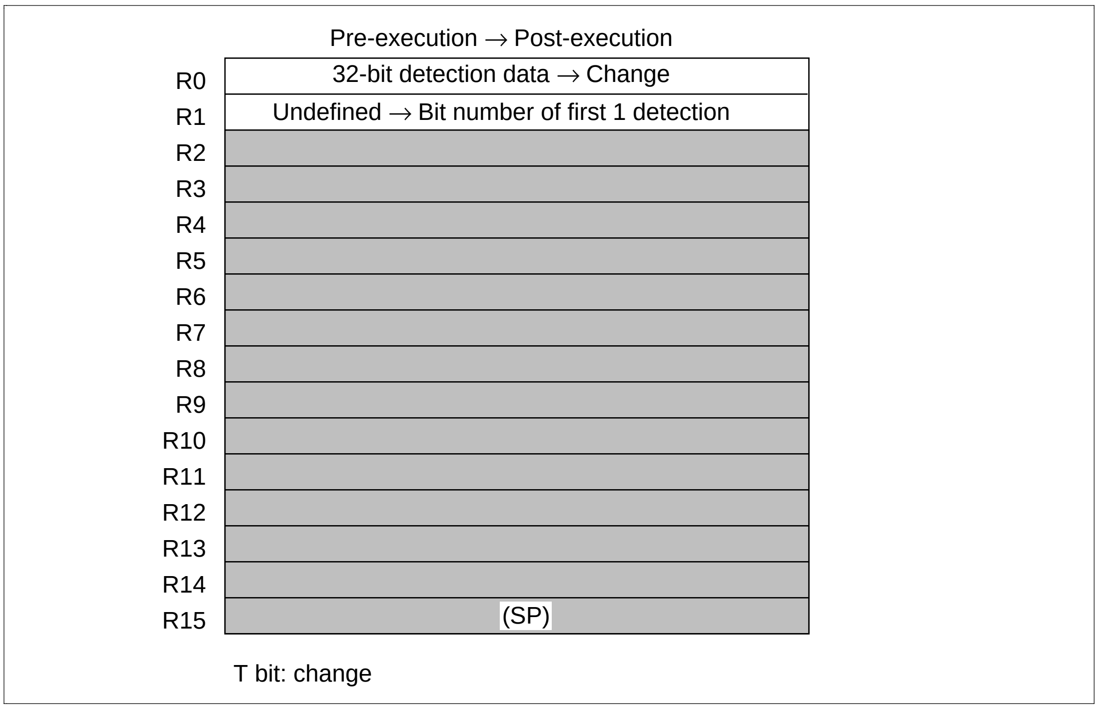


Figure 3.21 FIND1 Internal Register Change and Flag Change

Table 3.16 FIND1 Programming Specifications

Item	Value/State
Program memory (bytes)	16
Data memory (bytes)	0
Stack (bytes)	0
Number of states	29
Reentrant	Enabled
Relocation	
Intermediate interrupt	

- Precautions: The number of states in the programming specifications is the value when 32-bit data is H'10000000.

3.6.1 FIND1 Arguments

- R0: Holds the 32-bit data for detection as the input argument.
- R1: Holds the bit number (0–31) of the first detected 1 as the output argument.

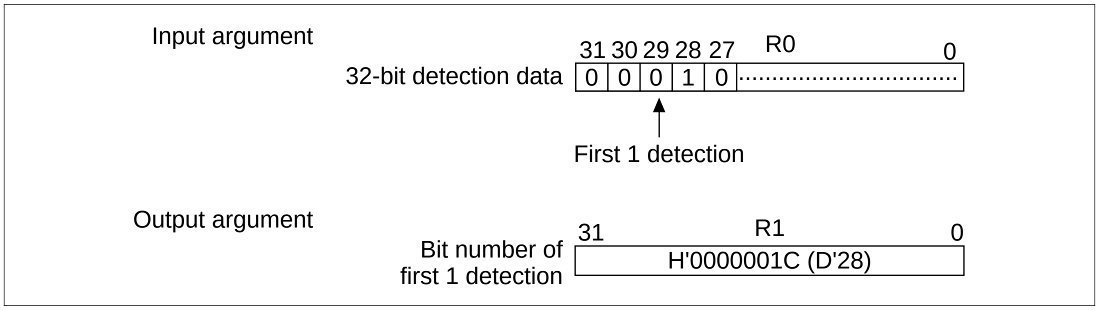


Figure 3.22 FIND1 Execution Example

3.6.2 Precautions for FIND1 Use

The contents of R0, which sets the 32-bit data for detection, are changed due to FIND1 execution. Be sure to save the 32-bit data beforehand if this data is also needed after execution of FIND1.

3.6.3 FIND1 RAM Use

RAM is not used by FIND1.

3.6.4 Example of FIND1 Use

Set the 32-bit data for detection in the input argument, and subroutine call FIND1.

```
        BSR     FIND1                Subroutine call FIND1.
        MOV.L   DATA, R0             Set 32-bit data for detection in input argument (R0)
            ↓
       .align    4
DATA   .data.l   H'12345678
```

3.6.5 FIND1 Operation

The SHLL instruction sets the T bit in sequence to the contents of bit 31 of the 32-bit data for detection and tests each bit.

R1 is used as a bit number pointer for bit testing. The first bit number (31) is set as the first value in R1 for bit discrimination. After bit testing, R1 is decremented by 1, and indicates the next bit number for judgment.

FIND1 ends when the first 1 is detected or when bit number (R1) < 0. When ending after 1 detection, R1 indicates the bit number of the first 1 detected. When ending after bit number (R1) < 0, the R1 contents become H'FFFFFFFF.

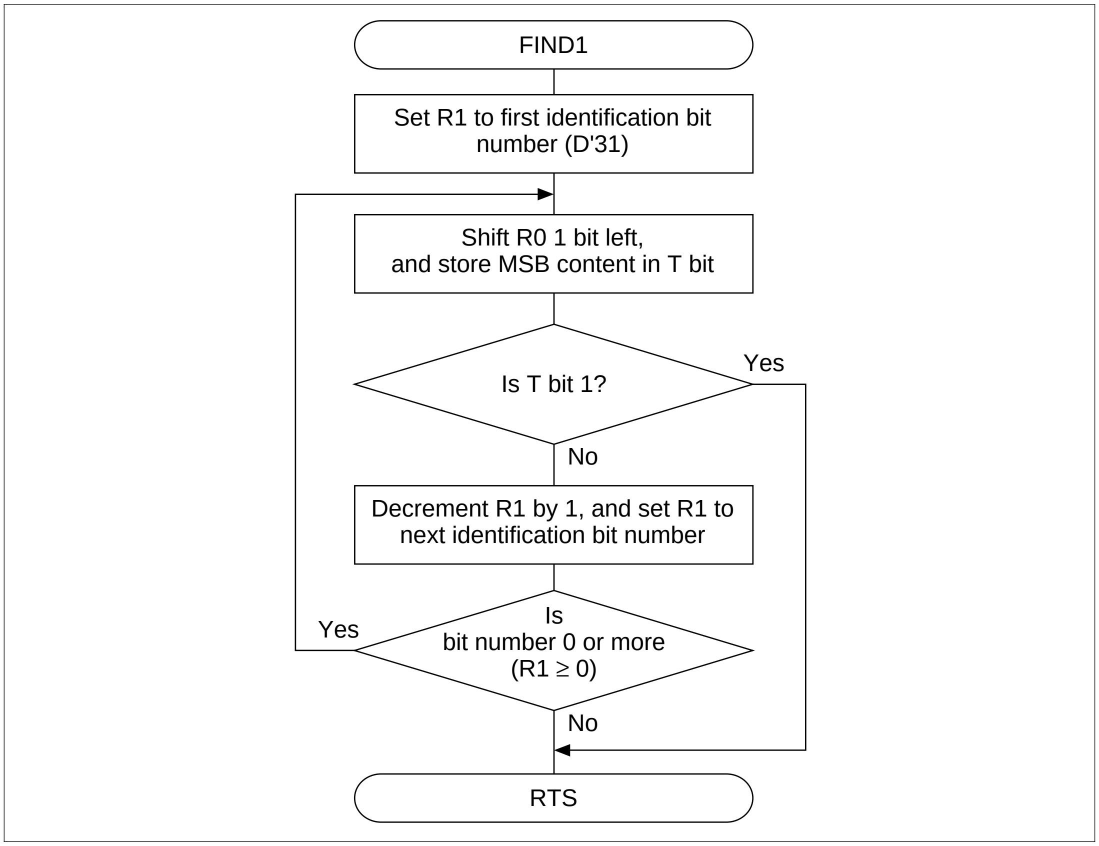


Figure 3.23 FIND1 Flowchart

3.6.6 FIND1 Program Listing

```
NAME:       FIND FIRST 1 (FIND1)
ENTRY:      R0 (32 BIT DATA)
RETURNS:    R1 (BIT NUMBER)
1                     1                       ;
2                     2                       ;
3                     3                       ;
4                     4                       ;
5                     5                       ;
6                     6                       ;
7                     7                       ;
8                     8                       ;
9                     9                       ;
10                    10                      ;
11  00001000          11  .SECTION  A,CODE,LOCATE=H'1000
12  00001000          12  FIND1  .EQU  $      ;Entry point
13  00001000   E11F   13    MOV  #D'31,R1     ;Initialize R1
14  00001002          14  FIND11              ;
15  00001002   4000   15    SHLL  R0          ;T bit = 1?
16  00001004   8902   16    BT  FIND_END      ;Yes
17  00001006   71FF   17    ADD  #H'FF,R1     ;Decrement bit number
18  00001008   4111   18    CMP/PZ  R1        ;Bit number ≥ 0?
19  0000100A   89FA   19    BT  FIND11        ;Yes
20  0000100C          20  FIND_END            ;
21  0000100C   000B   21    RTS               ;
22  0000100E   0009   22    NOP               ;
23                    23    .END
*****TOTAL  ERRORS 0
*****TOTAL  WARNINGS 0
```

3.7 ADDU64: 64 Bit + 64 Bit = 64 Bit (Unsigned)

- Instruction: ADDC
- Function: Adds the augend (unsigned 64 bits) and addend (unsigned 64 bits), and determines the sum (unsigned 64 bits). At this time, any carry generated is set in the T bit.

Table 3.17 ADDU64 Arguments

	Contents	Storage Location	Data Length (Bytes)
Input	Upper 32 bits of augend (unsigned 64 bits)	R0	4
	Lower 32 bits of augend (unsigned 64 bits)	R1	4
	Upper 32 bits of addend (unsigned 64 bits)	R2	4
	Lower 32 bits of addend (unsigned 64 bits)	R3	4
Output	Upper 32 bits of sum (unsigned 64 bits)	R0	4
	Lower 32 bits of sum (unsigned 64 bits)	R1	4
	With/without carry (with: T = 1, without: T = 0)	T bit (SR)	4

	(Pre-execution) → (Post-execution)
R0	Upper 32 bits of augend → Upper 32 bits of sum
R1	Lower 32 bits of augend → Lower 32 bits of sum
R2	Upper 32 bits of addend → No change
R3	Lower 32 bits of addend → No change
R4	
R5	
R6	
R7	
R8	
R9	
R10	
R11	
R12	
R13	
R14	
R15	(SP)

T bit: change

Figure 3.24 ADDU64 Internal Register Change and Flag Change

Table 3.18 ADDU64 Programming Specifications

Item	Value/State
Program memory (bytes)	8
Data memory (bytes)	0
Stack (bytes)	0
Number of states	5
Reentrant	Enabled
Relocation	
Intermediate interrupt	

3.7.1 ADDU64 Arguments

- R0: Holds the upper 32 bits of the augend (unsigned 64 bits) as the input argument.
  Holds the upper 32 bits of the sum (unsigned 64 bits) as the output argument.
- R1: Holds the lower 32 bits of the augend (unsigned 64 bits) as the input argument.
  Holds the lower 32 bits of the sum (unsigned 64 bits) as the output argument.
- R2: Holds the upper 32 bits of the addend (unsigned 64 bits) as the input argument.
  R3: Holds the lower 32 bits of the addend (unsigned 64 bits) as the input argument.
- T bit (SR): Indicates presence/absence of a carry after execution of ADDU64.
- T bit = 1: Carry.
- T bit = 0: No carry.

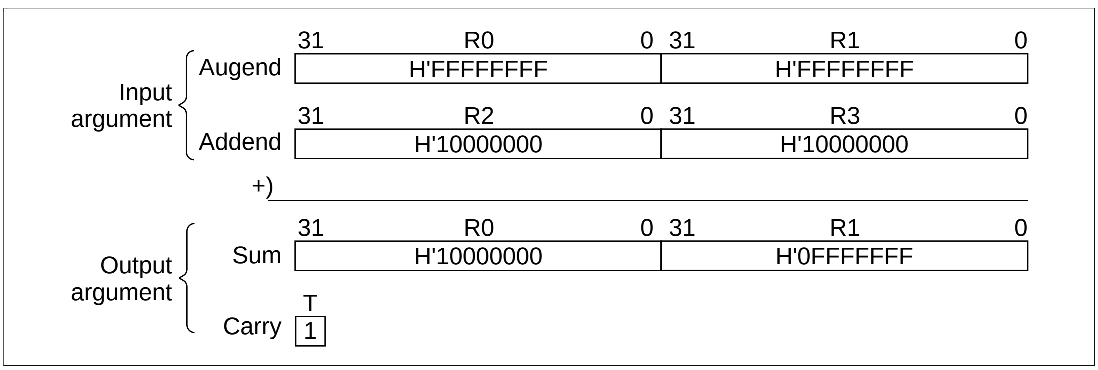


Figure 3.25 ADDU64 Execution Example

3.7.2 Precautions for ADDU64 Use

Since the sum is set in R1 and R2, which contain the augend settings, the augend data is destroyed. Be sure, therefore, to save this augend data beforehand if it is also needed after ADDU64 execution.

3.7.3 ADDU64 RAM Use

RAM is not used with ADDU64.

3.7.4 Example of ADDU64 Use

Make a subroutine call to ADDU64 after setting the augend and addend in the input argument.

```
        MOV.L  DATA1, R0      Set the augend (upper 32 bits) in the input argument
        MOV.L  DATA2, R1      Set the augend (lower 32 bits) in the input argument
        MOV.L  DATA3, R2      Set the addend (upper 32 bits) in the input argument
        BSR    ADDU64         Set the addend (upper 32 bits) in the input argument
        MOV.L  DATA4, R3      Subroutine call ADDU64
        BT     ERROR          Branch to error-processing subroutine when a carry
                              occurs
               ↓
        .align 4
DATA1   .data.l   H'FFFFFFFF
DATA2   .data.l   H'FFFFFFFF
DATA3   .data.l   H'10000000
DATA4   .data.1   H'10000000
```

3.7.5 ADDU64 Operation

Repeats additions in 32-bit units from the LSB by the add with carry instruction (ADDC).

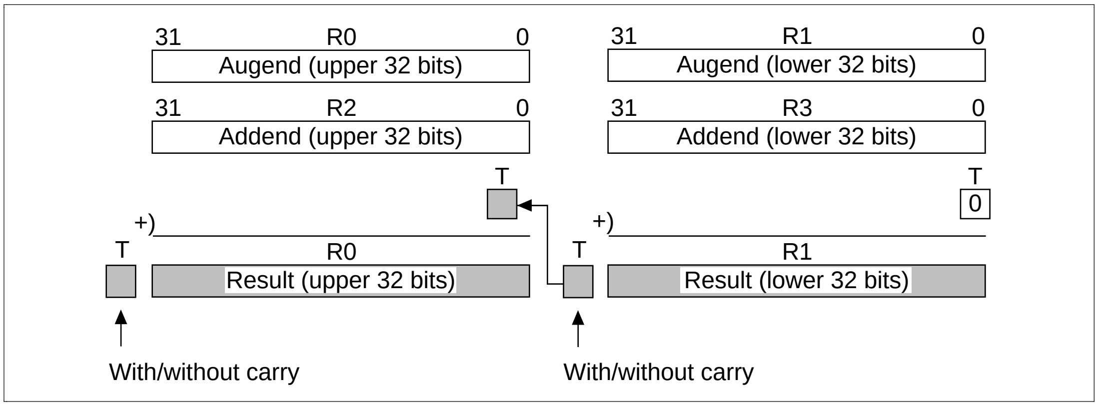


Figure 3.26 Unsigned Addition (ADDU64)

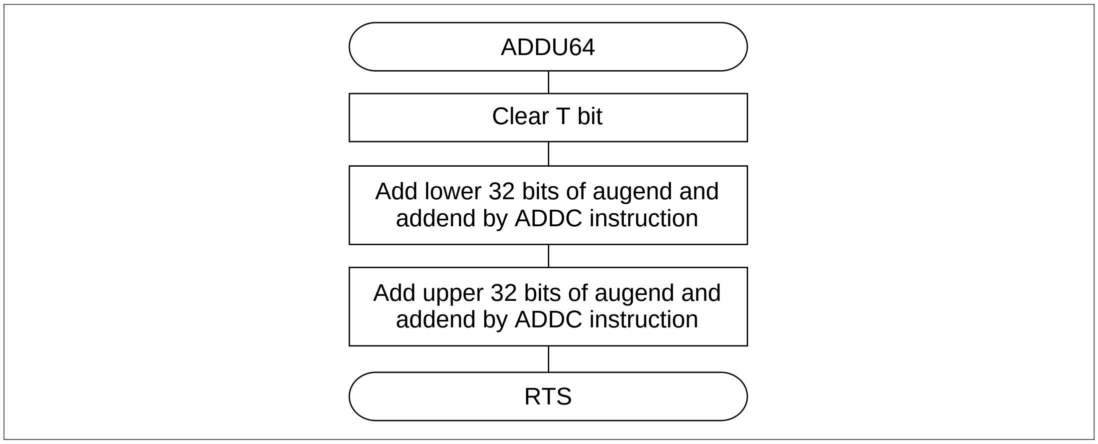


Figure 3.27 ADDU64 Flowchart

3.7.6 ADDU64 Program Listing

```
NAME:    64 BIT UNSIGNED BINARY ADDITION (ADDU64)
ENTRY:   R0 (UPPER 32 BIT AUGEND)
         R1 (LOWER 32 BIT AUGEND)
         R2 (UPPER 32 BIT ADDEND)
         R3 (LOWER 32 BIT ADDEND)
RETURNS: R0 (UPPER 32 BIT SUM)
         R1 (LOWER 32 BIT SUM)
         T BIT (CARRY -> TRUE; T=1, FALSE; T=0)
1                     1                        ;
2                     2                        ;
3                     3                        ;
4                     4                        ;
5                     5                        ;
6                     6                        ;
7                     7                        ;
8                     8                        ;
9                     9                        ;
10                    10                       ;
11                    11                       ;
12                    12                       ;
13                    13                       ;
14                    14                       ;
15                    15                       ;
16  00001000          16    .SECTION  A,CODE,LOCATE=H'1000
17  00001000          17  ADDU64.EQU  $        ;Entry point
18  00001000   0008   18    CLRT               ;Clear T bit
19  00001002   313E   19    ADDC R3,R1         ;Lower 32 bit augend
                                                + Lower 32 bit addend
20  00001004   000B   20    RTS                ;
21  00001006   302E   21    ADDC R2,R0         ;Upper 32 bit augend
                                                + Upper 32 bit addend
22                    22    .END
*****TOTAL  ERRORS 0
*****TOTAL  WARNINGS 0
```

3.8 ADDS64: 64 Bit + 64 Bit = 64 Bit (Signed)

- Instruction: ADDV
- Function: Adds the augend (signed 64 bits) and addend (signed 64 bits), and determines the sum (signed 64 bits). The presence/absence of an overflow or underflow is set in the T bit. Overflow and underflow discrimination is not carried out.

Table 3.19 ADDS64 Arguments

	Contents	Storage Location	Data Length (Bytes)
Input	Upper 32 bits of augend (signed 64 bits)	R0	4
	Lower 32 bits of augend (signed 64 bits)	R1	4
	Upper 32 bits of addend (signed 64 bits)	R2	4
	Lower 32 bits of addend (signed 64 bits)	R3	4
Output	Upper 32 bits of sum (signed 64 bits)	R0	4
	Lower 32 bits of sum (signed 64 bits)	R1	4
	With/without overflow (with: T = 1, without: T = 0)	T bit (SR)	4

	(Pre-execution) → (Post-execution)
R0	Upper 32 bits of augend → Upper 32 bits of sum
R1	Lower 32 bits of augend → Lower 32 bits of sum
R2	Upper 32 bits of addend → No change
R3	Lower 32 bits of addend → No change
R4	Work
R5	Work
R6	
R7	
R8	
R9	
R10	
R11	
R12	
R13	
R14	
R15	(SP)

T bit: change

Figure 3.28 ADDS64 Internal Register Change and Flag Change

Table 3.20 ADDS64 Programming Specifications

Item	Value/State
Program memory (bytes)	28
Data memory (bytes)	0
Stack (bytes)	8
Number of states	15
Reentrant	Enabled
Relocation	
Intermediate interrupt	

3.8.1 ADDS64 Arguments

- R0: Holds the upper 32 bits of the augend (signed 64 bits) as the input argument.
  Holds the upper 32 bits of the sum (signed 64 bits) as the output argument.

- R1: Holds the lower 32 bits of the augend (signed 64 bits) as the input argument.
  Holds the lower 32 bits of the sum (signed 64 bits) as the output argument.

- R2: Holds the upper 32 bits of the addend (signed 64 bits) as the input argument.
- R3: Holds the lower 32 bits of the addend (signed 64 bits) as the input argument.
- T bit (SR): Indicates presence/absence of an overflow/underflow after execution of ADDS64.
- T bit = 1: Overflow/underflow
- T bit = 0: No overflow/underflow

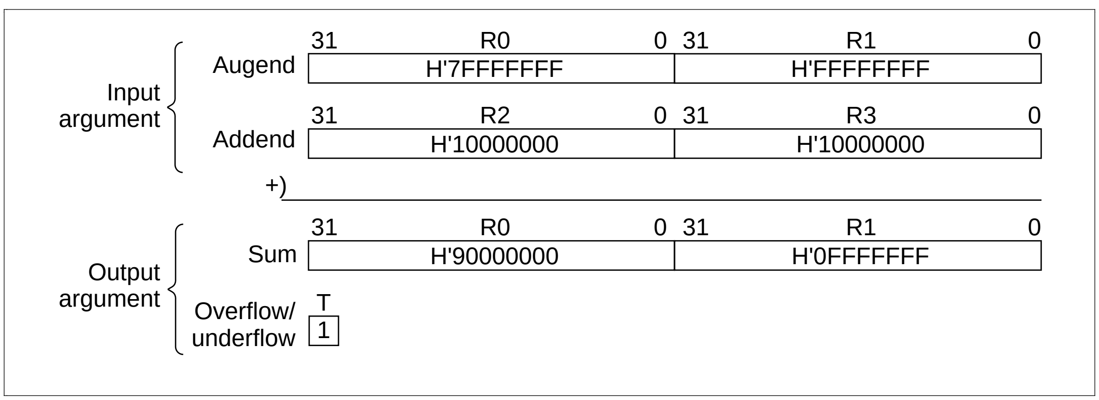


Figure 3.29 ADDS64 Execution Example

3.8.2 Precautions for ADDS64 Use

Since the sum is set in R1 and R2, which contain the augend settings, the augend data is destroyed due to ADDS64 execution. Be sure, therefore, to save this data beforehand if it is also needed after ADDS64 execution.

3.8.3 ADDS64 RAM Use

RAM is not used with ADDS64.

3.8.4 Example of ADDS64 Use

Make a subroutine call to ADDS64 after setting the augend and addend in the input argument.

```
        MOV.L  DATA1,R0        Set augend (upper 32 bits) in the input argument
        MOV.L  DATA2,R1        Set the augend (lower 32 bits) in the input argument
        MOV.L  DATA3,R2        Set the addend (upper 32 bits) in the input argument
        BSR    ADDS64          Subroutine call ADDS64
        MOV.L  DATA4,R3        Set the addend (lower 32 bits) in the input argument
        BT     ERROR           Branch to error-processing subroutine when an
                               overflow/underflow occurs
               ↓
        .align 4
DATA1   .data.l   H'7FFFFFFF
DATA2   .data.l   H'FFFFFFFF
DATA3   .data.l   H'10000000
DATA4   .data.1   H'10000000
```

3.8.5 ADDS64 Operation

1. ADDV adds the lower 32 bits of the augend and addend by the add with carry instruction (ADDC). If a carry occurs after addition, it is indicated in the T bit (figure 3.30 (1)).
2. ADDV adds the carry to the upper 32 bits of the augend. Stores the T bit contents of step 1 in R4, and adds it to the upper 32 bits of the augend with the binary addition with overflow check instruction . If a carry occurs, 1 is added to the upper 32 bits of the augend. If there is no carry, 0 is added to the upper 32 bits of the augend. The overflow/underflow is indicated in the T bit after addition (figure 3.30 (2)).
3. ADDV adds the sum of step 2 to the upper 32 bits of the addend by binary addition with overflow check instruction (ADDV). The overflow/underflow after addition is indicated in the T bit (figure 3.30 (3)).
4. The contents of the T bits from steps 2 and 3 are logically ORed and the result is stored in the T bit. T = 1 indicates an overflow/underflow, and T = 0 indicates no overflow/underflow.

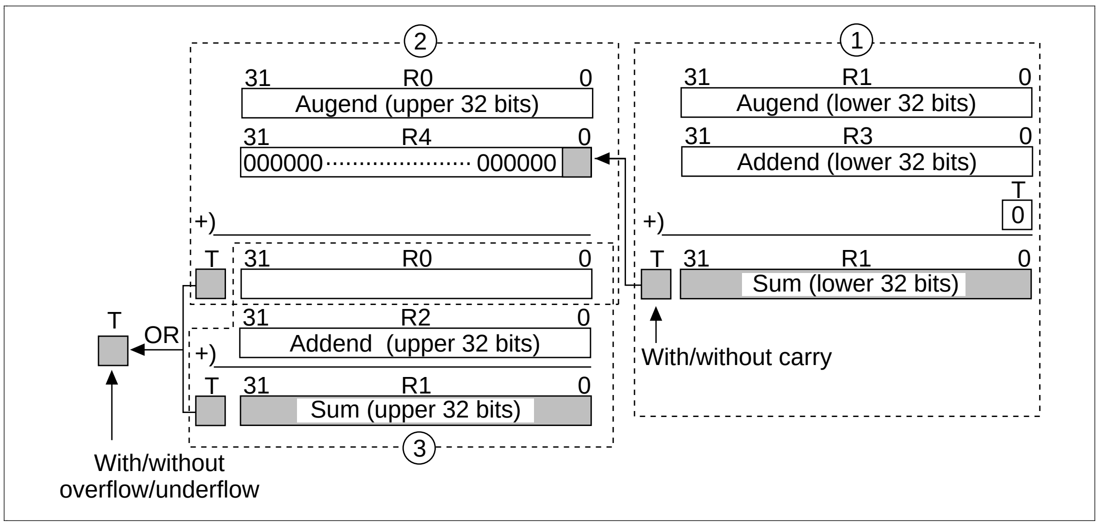


Figure 3.30 Signed Addition (ADDS64)

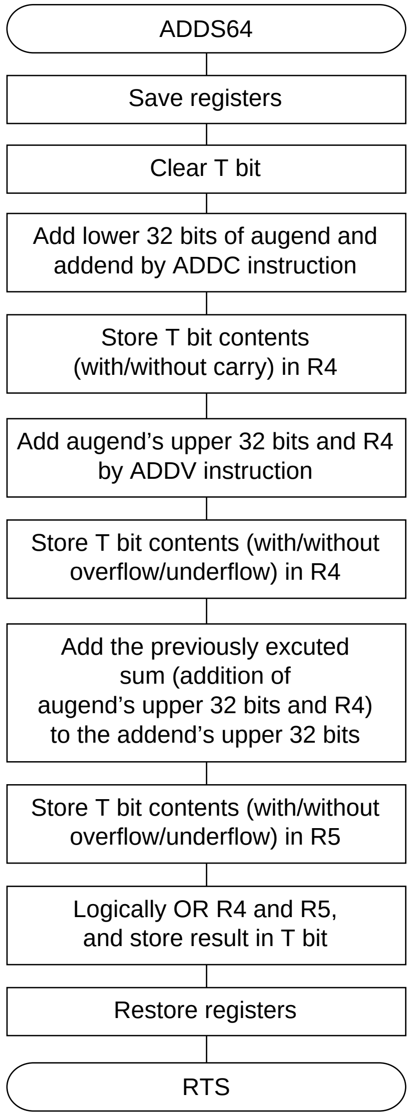


Figure 3.31 ADDS64 Flowchart

3.8.6 ADDS64 Program Listing

```
NAME:    64 BIT SIGNED BINARY ADDITION (ADDS64)
ENTRY:   R0 (UPPER 32 BIT AUGEND)
         R1 (LOWER 32 BIT AUGEND)
         R2 (UPPER 32 BIT ADDEND)
         R3 (LOWER 32 BIT ADDEND)
RETURNS: R0 (UPPER 32 BIT SUM)
         R1 (LOWER 32 BIT SUM)
         T BIT (OVERFLOW/UNDERFLOW → TRUE;T=1,FALSE;T=0)
1                      1                        ;
2                      2                        ;
3                      3                        ;
4                      4                        ;
5                      5                        ;
6                      6                        ;
7                      7                        ;
8                      8                        ;
9                      9                        ;
10                     10                       ;
11                     11                       ;
12                     12                       ;
13                     13                       ;
14                     14                       ;
15                     15                       ;
16  00001000           16    .SECTION  A,CODE,LOCATE=H'1000
17  00001000           17  ADDS64   .EQU $      ;Entry point
18  00001000   2F46    18    MOV.L  R4,@-R15    ;Escape register
19  00001002   2F56    19    MOV.L  R5,@-R15    ;
20  00001004   0008    20    CLRT               ;Clear T bit
21  00001006   313E    21  ADDC     R3,R1       ;Lower 32 bit augend
                                                 + Lower 32 bit addend
22  00001008   0429    22    MOVT   R4          ;R4 ← Carry
23  0000100A   304F    23    ADDV   R4,R0       ;Upper 32 bit augend
                                                 + Carry
24  0000100C   0429    24    MOVT   R4       ;R4 ← Overflow / Underflow
25  0000100E   302F    25    ADDV   R2,R0       ;Upper 32 bit augend
                                                 + Upper 32 bit addend
```

```
26  00001010   0529   26    MOVT   R5        ;R5 ← Overflow / Underflow
27  00001012   245B   27    OR     R5,R4        ;R4 ← R5 or R4
28  00001014   4401   28    SHLR   R4       ;T bit ← Overflow/Underflow
29  00001016   65F6   29    MOV.L  @R15+,R5    ;Return register
30  00001018   000B   30    RTS                 ;
31  0000101A   64F6   31    MOV.L  @R15+,R4    ;
32                    32    .END
*****TOTAL  ERRORS 0
*****TOTAL  WARNINGS 0
```

3.9 MULU32: 32 Bit × 32 Bit = 64 Bit (Unsigned)

- Instructions: MULU, SWAP
- Function: Multiplies the multiplicand (unsigned 32 bits) by the multiplier (unsigned 32 bits) and determines the product (unsigned 64 bits).

Table 3.21MULU32 Arguments

	Contents	Storage Location	Data Length (Bytes)
Input	Multiplicand (unsigned 32 bits)	R0	4
	Multiplier (unsigned 32 bits)	R1	4
Output	Upper 32 bits of product (unsigned 64 bits)	R2	4
	Lower 32 bits of product (unsigned 64 bits)	R3	4

(Pre-execution) → (Post-execution)

R0	Multiplicand (unsigned 32 bits) → No change
R1	Multiplier (unsigned 32 bits) → No change
R2	Undefined → Product (upper 32 bits)
R3	Undefined → Product (lower 32 bits)
R4	Work
R5	Work
R6	Work
R7	
R8	
R9	
R10	
R11	
R12	
R13	
R14	
R15	(SP)

T bit: change

Figure 3.32 MULU32 Internal Register Change and Flag Change

Table 3.22MULU32 Programming Specifications

Item	Value/State
Program memory (bytes)	76
Data memory (bytes)	0
Stack (bytes)	24
Number of states	35
Reentrant	Enabled
Relocation	
Intermediate interrupt	

- Precaution: The number of program states is the value when H'FFFFFFFF × H'FFFFFFFF is calculated.

3.9.1 MULU32 Arguments

- R0: Set the multiplicand (unsigned 32 bits) as the input argument.
- R1: Set the multiplier (unsigned 32 bits) as the input argument.
- R2: The upper 32 bits of the product (unsigned 64 bits) is set as the output argument.
- R3: The lower 32 bits of the product (unsigned 64 bits) is set as the output argument.

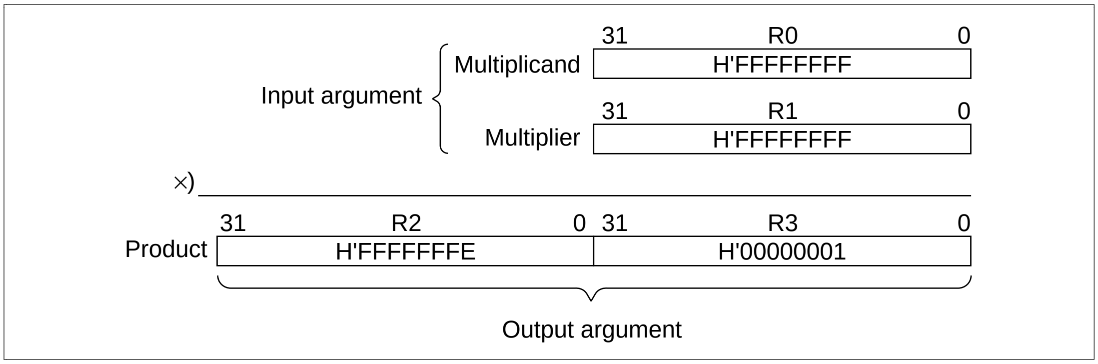


Figure 3.33 MULU32 Execution Example

3.9.2 Precautions for MULU32 Use

There are no particular precautions for MULU32.

3.9.3 MULU32 RAM Use

RAM is not used with MULU32.

3.9.4 Example of MULU32 Use

Make a subroutine call to MULU32 after setting the multiplicand and multiplier.

```
        MOV.L     DATA1, R0          Sets multiplicand in the input argument (R0)
        BSR       MULU32             Subroutine call of MULU32
        MOV.L     DATA2, R1          Sets multiplier in the input argument (R1)
                  ↓
        .align    4
DATA1   .data.l   H'FFFFFFFF
DATA2   .data.l   H'FFFFFFFF
```

3.9.5 MULU32 Operation

Multiplication is performed in 16 bit units (figure 3.34). Partial products (1–4) are determined, and these are added to get the final product (64 bits). The 16-bit unsigned multiplication instruction (MULU) is used in multiplication of partial products.

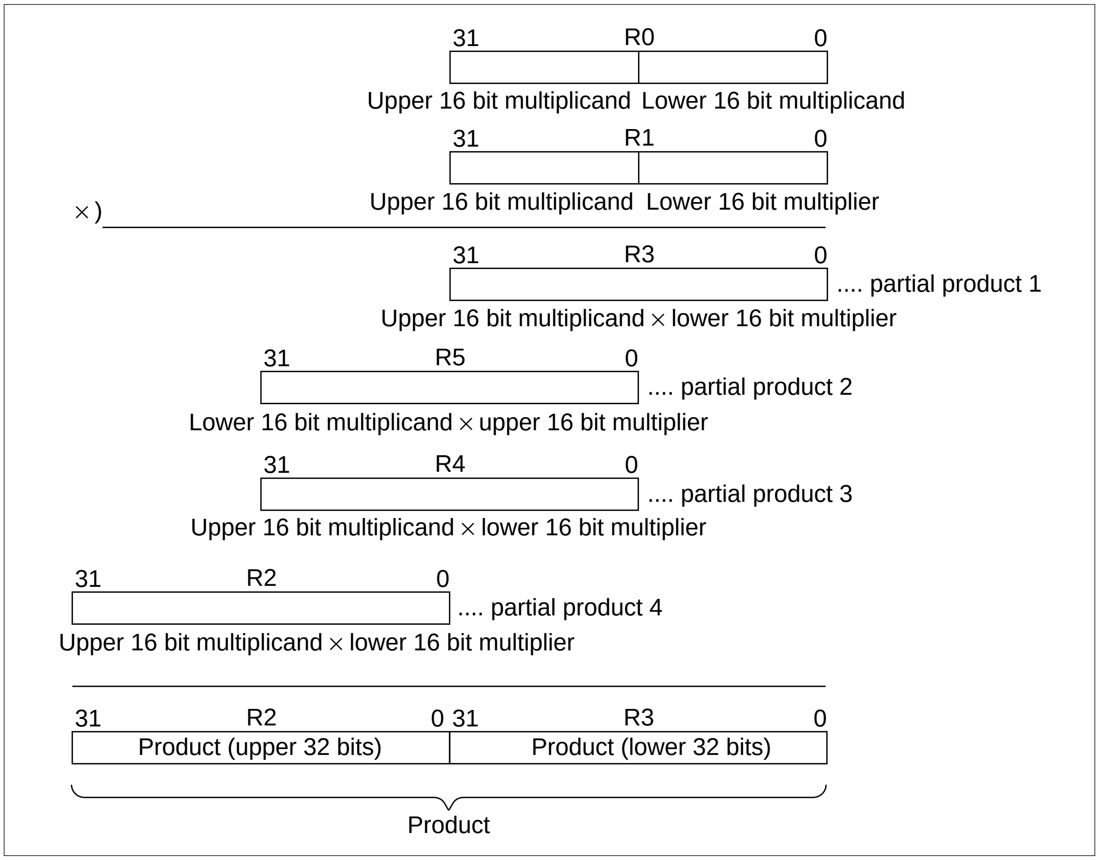


Figure 3.34 Multiplication (MULU32)

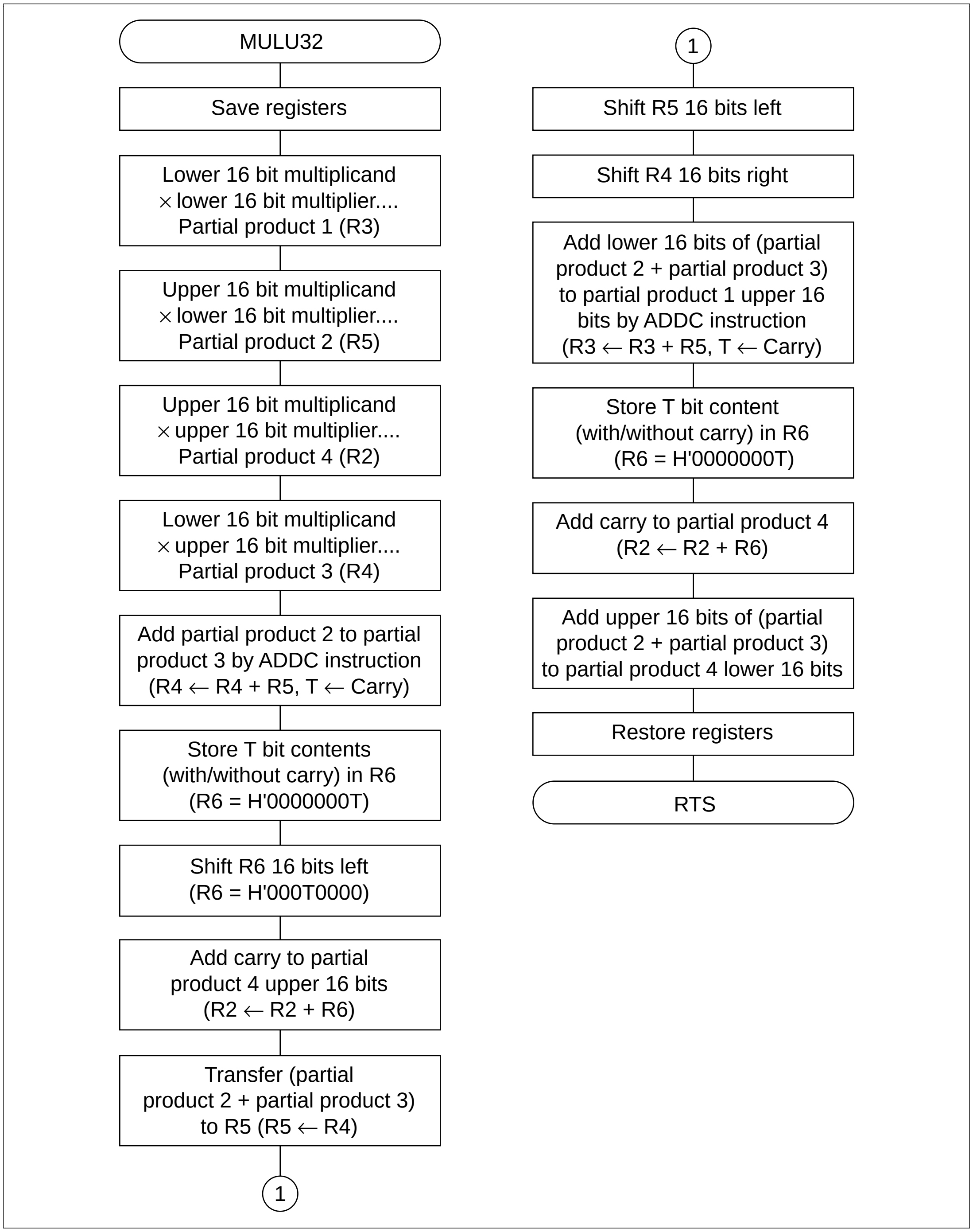


Figure 3.35 MULU32 Flowchart

3.9.6 MULU32 Program Listing

```
NAME:    32 BIT UNSIGNED  MULTIPLICATION  (MULU32)
ENTRY:   R0 (MULTIPLICAND)
         R1 (MULTIPLIER)
RETURNS: R2 (UPPER 32 BIT PRODUCT)
         R3 (LOWER 32 BIT PRODUCT)
                                    ;
                                    ;
                                    ;
                                    ;
                                    ;
                                    ;
                                    ;
                                    ;
                                    ;
                                    ;
                                    ;
                                    ;
00001000              .SECTION  A,CODE,LOCATE=H'1000
00001000            MULU32  .EQU   $    ;Entry point
00001000   4F12       STS.L  MACL,@-R15 ;Escape register
00001002   2F46       MOV.L   R4,@-R15  ;
00001004   2F56       MOV.L   R5,@-R15  ;
00001006   2F66       MOV.L   R6,@-R15  ;
                                    ;
00001008   201E       MULU    R1,R0     ;Lower 16 bit * lower
                                         16 bit → R3
0000100A   6009       SWAP.W  R0,R0     ;
0000100C   031A       STS     MACL,R3   ;
0000100E   201E       MULU    R1,R0     ;Upper 16 bit * lower
                                         16 bit → R5
00001010   6119       SWAP.W  R1,R1     ;
00001012   051A       STS     MACL,R5   ;
00001014   201E       MULU    R1,R0     ;Upper 16 bit * upper
                                         16 bit → R2
00001016   6009       SWAP.W  R0,R0     ;
00001018   021A       STS     MACL,R2   ;
```

```
29  0000101A   201E   29    MULU    R1,R0     ;Lower 16 bit * upper
                                               16 bit → R4
30  0000101C   6119   30    SWAP.W  R1,R1     ;
31  0000101E   041A   31    STS     MACL,R4   ;
32                    32                      ;
33  00001020   0008   33    CLRT              ;
34  00001022   345E   34    ADDC    R5,R4     ;
35  00001024   0629   35    MOVT    R6        ;R6 ← Carry
36  00001026   4628   36    SHLL16  R6        ;
37  00001028   326C   37    ADD  R6,R2        ;Carry = 1 R2 ← R2 +
                                               H'00010000
38                    38                      ;Carry = 0 R2 ← R2 +
                                               H'00000000
39  0000102A   6543   39    MOV     R4,R5     ;
40  0000102C   4528   40    SHLL16  R5        ;
41  0000102E   4429   41    SHLR16  R4        ;
42                    42                      ;
43  00001030   0008   43    CLRT              ;
44  00001032   335E   44    ADDC    R5,R3     ;
45  00001034   0629   45    MOVT    R6        ;R6 ← Carry
46  00001036   326C   46    ADD     R6,R2     ;Carry = 1 R2 ← R2 +
                                               H'00000001
47                    47                      ;Carry = 0  R2 ← R2 +
                                               H'00000000
48  00001038   324C   48    ADD     R4,R2     ;
49                    49                      ;
50  0000103A   66F6   50    MOV.L   @R15+,R6  ; Return register
51  0000103C   65F6   51    MOV.L   @R15+,R5  ;
52  0000103E   64F6   52    MOV.L   @R15+,R4  ;
53  00001040   000B   53    RTS               ;
54  00001042   4F16   54    LDS.L   @R15+,MACL;
55                    55    .END
*****TOTAL  ERRORS 0
*****TOTAL  WARNINGS 0
```

3.10 MULS32: 32 Bit × 32 Bit = 64 Bit (Signed)

- Instructions: MULU, SWAP, NEGC
- Function: Multiplies the multiplicand (signed 32 bits) by the multiplier (signed 32 bits) and determines the product (signed 64 bits).

Table 3.23 MULS32 Arguments

	Contents	Storage Location	Data Length (Bytes)
Input	Multiplicand (signed 32 bits)	R0	4
	Multiplier (signed 32 bits)	R1	4
Output	Upper 32 bits of product (signed 64 bits)	R2	4
	Lower 32 bits of product (signed 64 bits)	R3	4

	(Pre-execution) → (Post-execution)
R0	Multiplicand (unsigned 32 bit) → No change
R1	Multiplier (unsigned 32 bit) → Change
R2	Undefined → Product (upper 32 bit)
R3	Undefined → Product (lower 32 bit)
R4	Work
R5	Work
R6	Work
R7	
R8	
R9	
R10	
R11	
R12	
R13	
R14	
R15	(SP)

T bit: change

Figure 3.36 MULS32 Internal Register Change and Flag Change

Table 3.24MULS32 Programming Specifications

Item	Value/State
Program memory (bytes)	92
Data memory (bytes)	0
Stack (bytes)	16
Number of states	48
Reentrant	Enabled
Relocation	
Intermediate interrupt	

- Precautions: The number of program states is the value when H'7FFFFFFF × H'80000000 is calculated.

3.10.1 MULS32 Arguments

- R0: Holds the multiplicand (signed 32 bits) as the input argument.
- R1: Holds the multiplier (signed 32 bits) as the input argument.
- R2: The upper 32 bits of the product (signed 64 bits) is set as the output argument.
- R3: The lower 32 bits of the product (signed 64 bits) is set as the output argument.

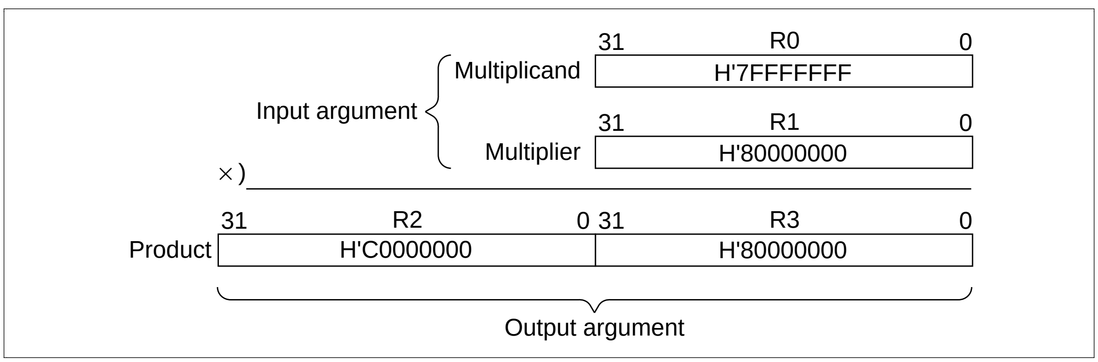


Figure 3.37 MULS32 Execution Example

3.10.2 Precautions for MULS32 Use

The contents of R1, which sets the multiplier, are changed by execution of MULS32. Be sure to save the multiplier data beforehand if it is also needed after MULS32 execution.

3.10.3 MULS32 RAM Use

RAM is not used with MULS32.

3.10.4 Example of MULS32 Use

Make a subroutine call to MULS32 after setting the multiplicand and multiplier.

```
        MOV.L     DATA1, R0          Sets multiplicand in the input argument (R0)
        BSR       MULS32             Subroutine call of MULS32
        MOV.L     DATA2, R1          Sets multiplier in the input argument (R1)
                  ↓
        .align    4
DATA1   .data.l   H'7FFFFFFF
DATA2   .data.l   H'80000000
```

3.10.5 MULS32 Operation

Multiplication is performed in 16 bit units (figure 3.38). Partial products (1–4) are determined, and these are added to get the final product (64 bits). The 16-bit unsigned multiplication instruction (MULU) is used in multiplication of partial products, so if the multiplicand or multiplier are negative, they are converted to positive before multiplication.

The product is calculated as positive, so judge the product as positive or negative using exclusive OR on the multiplicand and multiplier MSB, as shown in table 3.25.

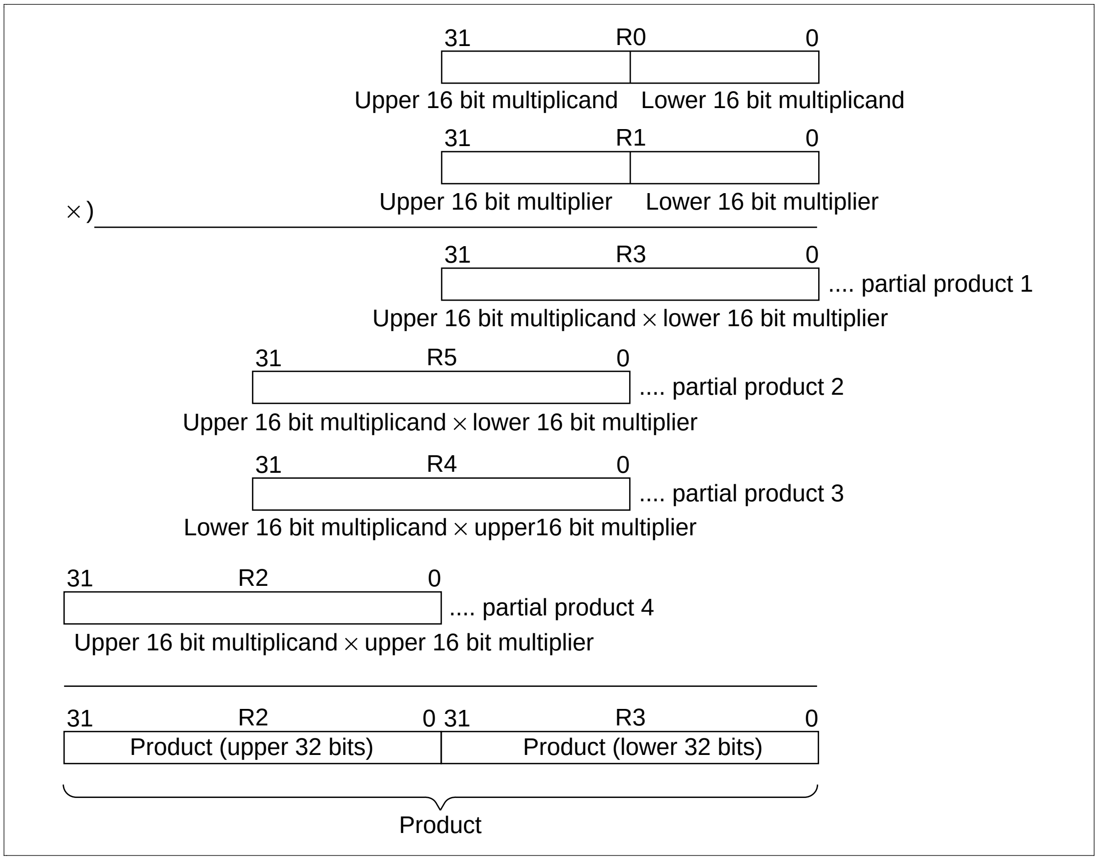


Figure 3.38 Multiplication (MULS32)

Table 3.25 Product Sign Changes

MSB of Multiplicand	MSB of Multiplier	Product Sign Change
Positive	Positive	Positive
Positive	Negative	Negative
Negative	Positive	Negative
Negative	Negative	Positive

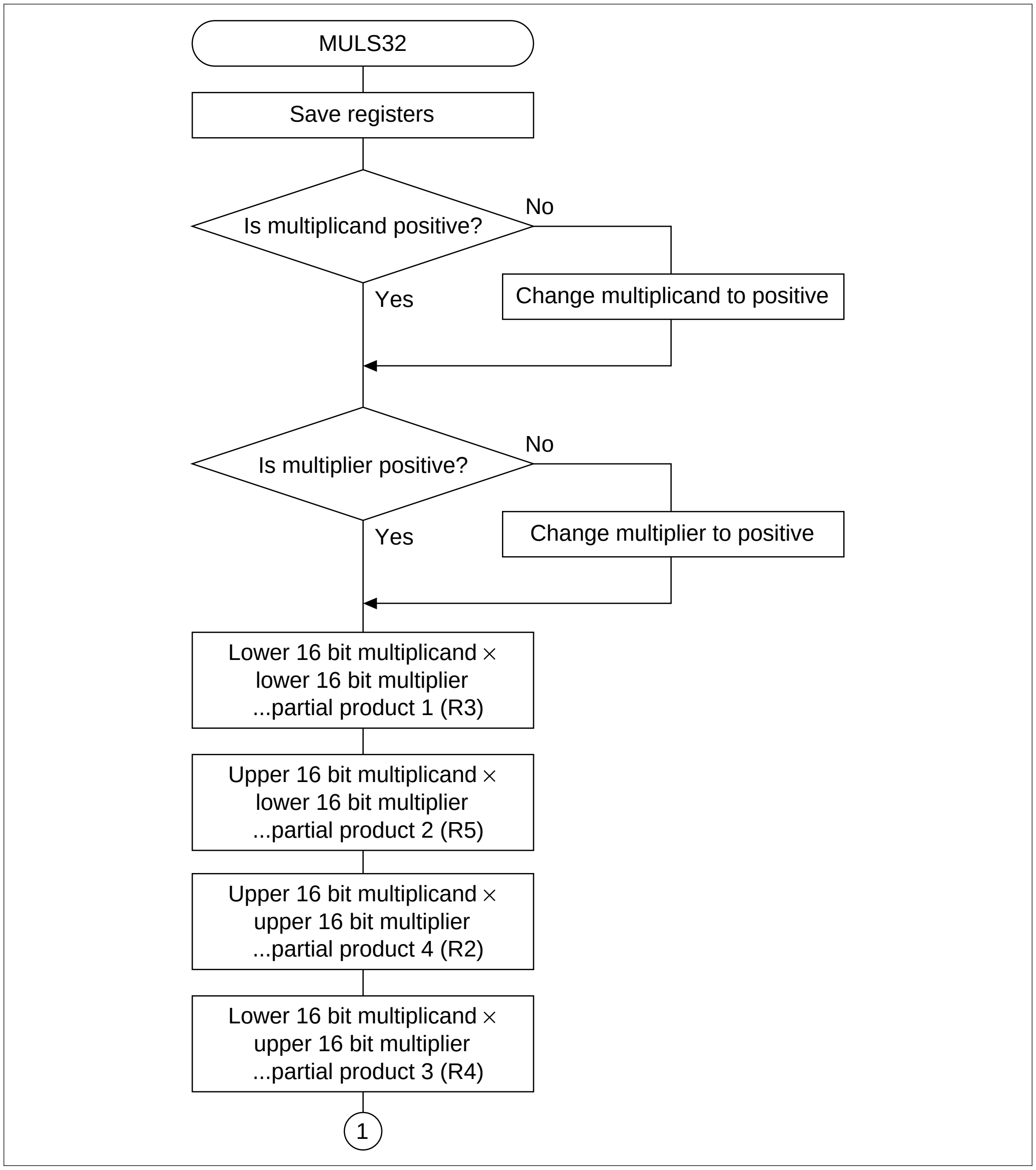


Figure 3.39 MULS32 Flowchart

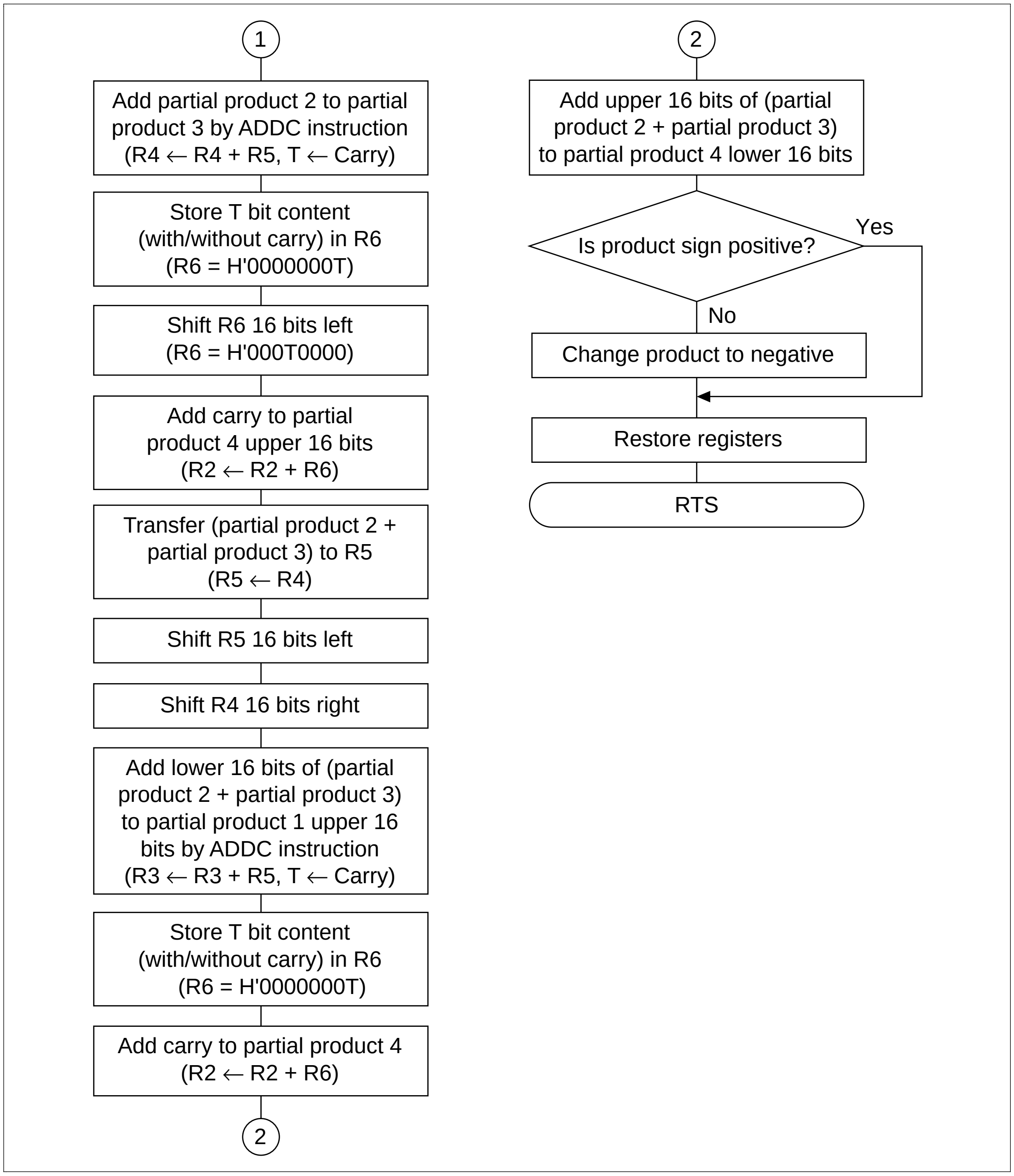


Figure 3.39 MULS32 Flowchart (cont)

3.10.6 MULS32 Program Listing

```
NAME:    32 BIT SIGNED  MULTIPLICATION  (MULS32)
ENTRY:   R0 (MULTIPLICAND)
         R1 (MULTIPLIER)
RETURNS: R2 (UPPER 32 BIT PRODUCT)
         R3 (LOWER 32 BIT PRODUCT)
1                     1                        ;
2                     2                        ;
3                     3                        ;
4                     4                        ;
5                     5                        ;
6                     6                        ;
7                     7                        ;
8                     8                        ;
9                     9                        ;
10                    10                       ;
11                    11                       ;
12                    12                       ;
13  00001000          13    .SECTION  A,CODE,LOCATE=H'1000
14  00001000          14  MULS32  .EQU  $      ;Entry point
15  00001000   4F12   15    STS.L  MACL,@-R15 ;Escape register
16  00001002   2F46   16    MOV.L  R4,@-R15   ;
17  00001004   2F56   17    MOV.L  R5,@-R15   ;
18  00001006   2F66   18    MOV.L  R6,@-R15   ;
19                    19                      ;
20  00001008   4011   20    CMP/PZ  R0        ;Multiplicand ≥ 0?
21  0000100A   8900   21    BT     MULS321    ;Yes
22  0000100C   600B   22    NEG    R0,R0      ;Change plus
23  0000100E          23  MULS321             ;
24  0000100E   4111   24    CMP/PZ  R1        ;Multiplier ≥ 0?
25  00001010   8900   25    BT     MULS322    ;Yes
26  00001012   611B   26    NEG    R1,R1      ;Change plus
27  00001014          27  MULS322             ;
28  00001014   201E   28    MULU   R1,R0      ;Lower 16 bit * lower
                                               16 bit → R3
29  00001016   6009   29    SWAP.W R0,R0      ;
30  00001018   031A   30    STS    MACL,R3    ;
```

```
31  0000101A   201E  31    MULU   R1,R0       ;Upper 16 bit * lower
                                               16 bit → R5
32  0000101C   6119  32    SWAP.W R1,R1       ;
33  0000101E   051A  33    STS  MACL,R5       ;
34  00001020   201E  34    MULU  R1,R0        ;Upper 16 bit * upper
                                               16 bit → R2
35  00001022   6009  35    SWAP.W R0,R0       ;
36  00001024   021A  36    STS    MACL,R2     ;
37  00001026   201E  37    MULU   R1,R0       ;Lower 16 bit * upper
                                               16 bit → R4
38  00001028   6119  38    SWAP.W R1,R1       ;
39  0000102A   041A  39    STS  MACL,R4       ;
40                   40                       ;
41  0000102C   0008  41    CLRT               ;
42  0000102E   345E  42    ADDC R5,R4         ;
43  00001030   0629  43    MOVT R6            ;R6 ← Carry
44  00001032   4628  44    SHLL16  R6         ;
45  00001034   326C  45    ADD     R6,R2      ;Carry = 1 R2 ← R2 +
                                               H'00010000
46                   46                       ;Carry = 0  R2 ← R2 +
                                               H'00000000
47  00001036   6543  47    MOV     R4,R5      ;
48  00001038   4528  48    SHLL16  R5         ;
49  0000103A   4429  49    SHLR16  R4         ;
50                   50                       ;
51  0000103C   0008  51    CLRT               ;
52  0000103E   335E  52    ADDC    R5,R3      ;
53  00001040   0629  53    MOVT    R6         ;R6 ← Carry
54  00001042   326C  54    ADD     R6,R2      ;Carry = 1 R2 ← R2 +
                                               H'00000001
55                   55                       ;Carry = 1 R2 ← R2 +
                                               H'00000000
56  00001044   324C  56    ADD     R4,R2      ;
57                   57                       ;
58  00001046   210A  58    XOR     R0,R1      ;Product < 0 ?
59  00001048   4100  59    SHLL    R1         ;
60                   60                       ;
```

```
61  0000104A   8B02   61    BF      MULS32_END;No
62  0000104C   0008   62    CLRT              ;Change minus
63  0000104E   633A   63    NEGC    R3,R3     ;
64  00001050   622A   64    NEGC    R2,R2     ;
65  00001052          65  MULS32_END          ;
66  00001052   66F6   66    MOV.L   @R15+,R6  ;Return register
67  00001054   65F6   67    MOV.L   @R15+,R5  ;
68  00001056   64F6   68    MOV.L   @R15+,R4  ;
69  00001058   000B   69    RTS               ;
70  0000105A   4F16   70    LDS.L   @R15+,MACL;
71                    71    .END
*****TOTAL  ERRORS 0
*****TOTAL  WARNINGS 0
```

3.11 DIVU32Q: Quotient of 32 Bit ÷ 32 Bit (Unsigned)

- Instructions: DIV0U, DIV1
- Function: Divides the dividend (unsigned 32 bits) by the divisor (unsigned 32 bits), and determines quotient (unsigned 32 bits). Also indicates an error (division by 0) in the T bit.

Table 3.26 DIVU32Q Arguments

	Contents	Storage Location	Data Length (Bytes)
Input	Dividend (unsigned 32 bits)	R1	4
	Divisor (unsigned 32 bits)	R0	4
Output	Quotient (unsigned 32 bits)	R1	4
	Error flag (with: T = 1, without: T = 0)	T bit (SR)	4

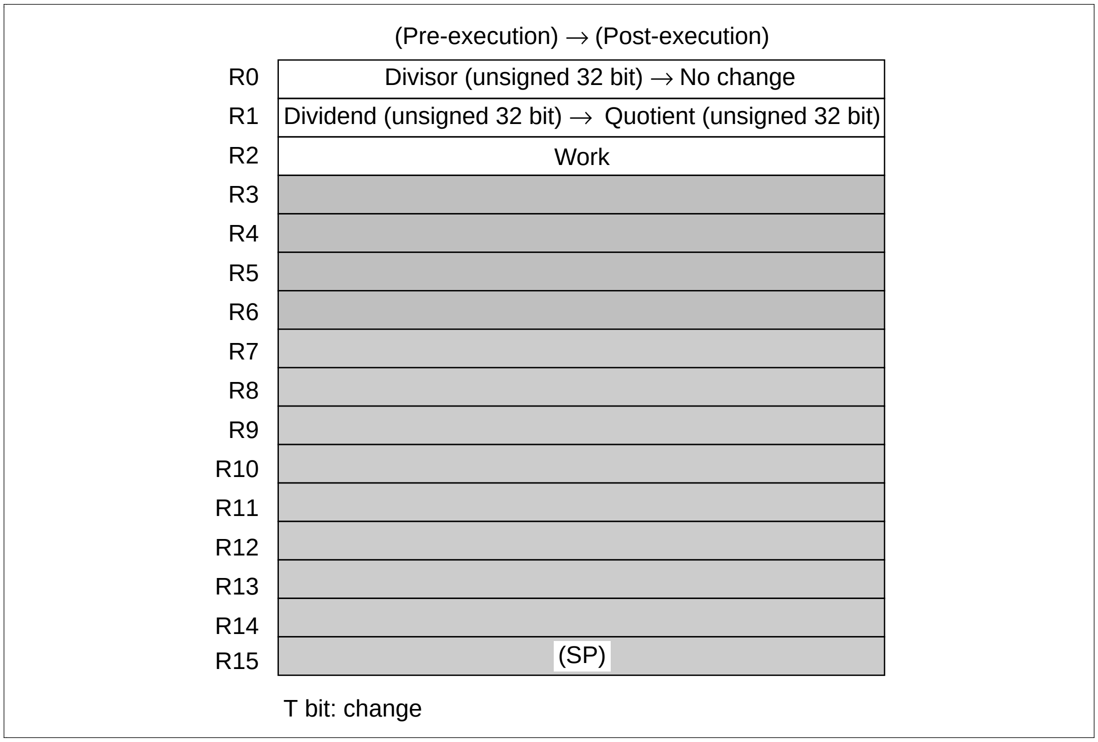


Figure 3.40 DIVU32Q Internal Register Change and Flag Change

Table 3.27DIVU32Q Programming Specifications

Item	Value/State
Program memory (bytes)	152
Data memory (bytes)	0
Stack (bytes)	4
Number of states	74
Reentrant	Enabled
Relocation	
Intermediate interrupt	

- Precautions: The number of program states is the value when H'FFFFFFFF ÷ H'FFFFFFFE is calculated.

3.11.1 DIVU32Q Arguments

- R0: Holds the divisor (unsigned 32 bits) as the input argument.
- R1: Holds the dividend (unsigned 32 bits) as the input argument.
- Holds the quotient (unsigned 32 bits) as the output argument.

- T bit (SR): Indicates whether an error (division by 0) has occurred in division.
- T bit = 1: Indicates an error (division by 0) in the division.
- T bit = 0: Indicates no error in the division.

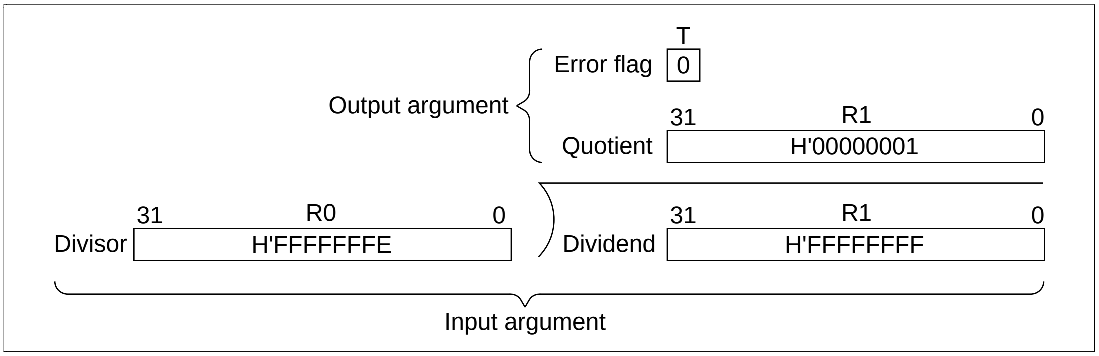


Figure 3.41 DIVU32Q Execution Example

3.11.2 Precautions for DIVU32Q Use

As the quotient is set in R1, which contains the dividend after DIVU32Q execution, the dividend is destroyed. Be sure, therefore, to save the dividend beforehand if it is also needed after DIVU32Q execution.

3.11.3 DIVU32Q RAM Use

RAM is not used by DIVU32Q.

3.11.4 Example of DIVU32Q Use

Make a subroutine call to DIVU32Q after setting the dividend and divisor in the input arguments.

```
        MOV.L     DATA1, R1     Sets dividend (unsigned 32 bits) in the input
                                argument (R1)
        BSR       DIVU32Q       Subroutine call of DIVU32Q
        MOV.L     DATA2, R0     Sets divisor (unsigned 32 bits) in the input argument
                                (R0)
        BT        ERROR         Branch to error processing subroutine when error
                                (division by 0) occurs
                  ↓
        .align    4
DATA1   .data.l   H'FFFFFFFF
DATA2   .data.l   H'FFFFFFFE
```

3.11.5 DIVU32Q Operation

Carries out the following initial settings before division:

- Takes R2's upper 32 bits and zero-extends the dividend to 64 bits (figure 3.42 (1)).
- Sets the M, Q, and T bits used in one-step division to unsigned division values (figure 3.42 (2)).

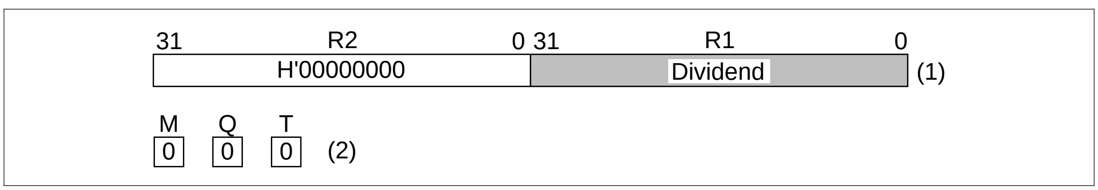


Figure 3.42 Initial Setting (DIVU32Q)

ROCTL and DIV1 repeat the division operation through the number of divisor bits (32 times) (figure 3.43).

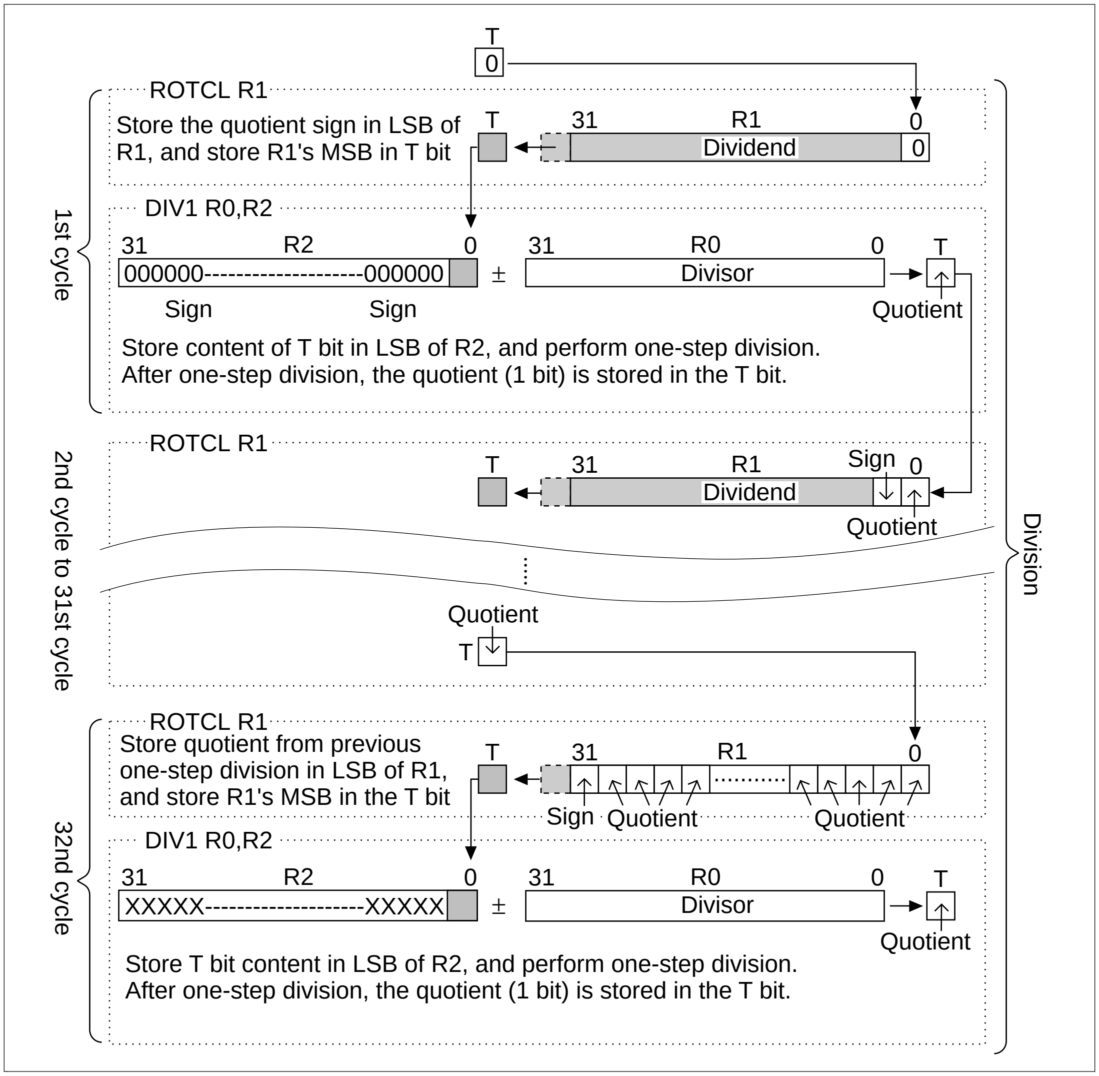


Figure 3.43 Division (DIVU32Q)

At the end of division, the 32nd quotient of one-step division is stored in the T bit. The T bit contents are stored in the LSB of R1, and the R1 contents become the quotient (figure 3.44).

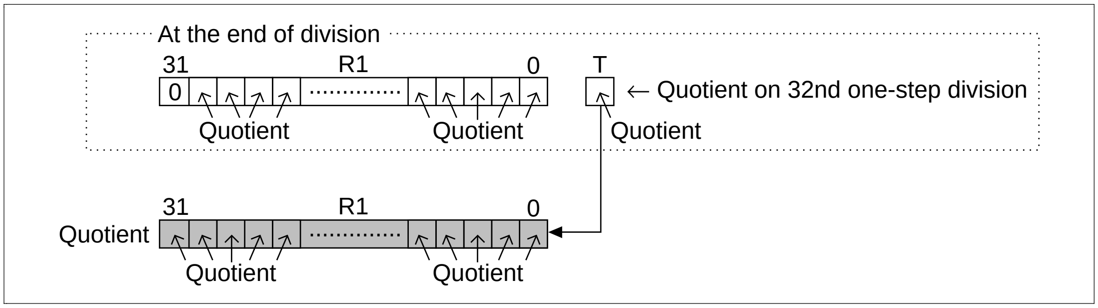


Figure 3.44 Quotient (DIVU32Q)

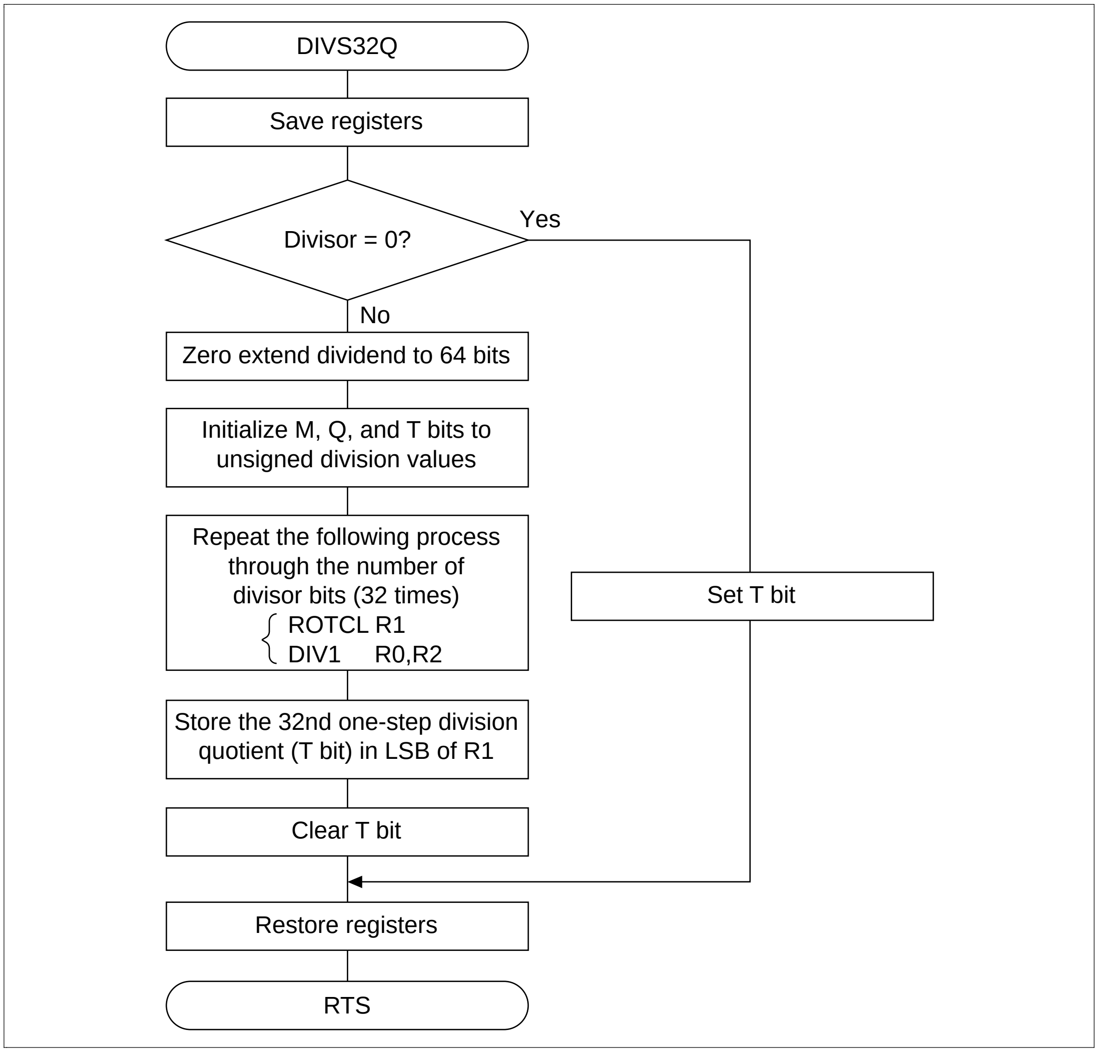


Figure 3.45 DIVU32Q Flowchart

3.11.6 DIVU32Q Program Listing

```
NAME:     QUOTIENT OF 32 BIT UNSIGNED DIVISION (DIVU32Q)
ENTRY:    R1(DIVIDEND)
          R0(DIVISOR)
RETURNS:  R1(QUOTIENT)
          T BIT (ERROR → TRUE;T=1,FALSE;T=0)
                                      ;
                                      ;
                                      ;
                                      ;
                                      ;
                                      ;
                                      ;
                                      ;
                                      ;
                                      ;
                                      ;
                                      ;
00001000           .SECTION A,CODE,LOCATE=H'1000
00001000         DIVU32Q  .EQU  $     ;Entry point
00001000   2F26    MOV.L  R2,@-R15    ;Escape register
00001002   2008    TST    R0,R0       ;Divisor = 0?
00001004   8945    BT     DIVU32Q1    ;Yes
00001006   222A    XOR    R2,R2       ;R2 ← H'00000000
00001008   0019    DIV0U              ;Divide as unsigned
                                      ;
0000100A   4124    ROTCL  R1          ;Divide 1 step
0000100C   3204    DIV1   R0,R2       ;
0000100E   4124    ROTCL  R1          ;
00001010   3204    DIV1   R0,R2       ;
00001012   4124    ROTCL  R1          ;
00001014   3204    DIV1   R0,R2       ;
00001016   4124    ROTCL  R1          ;
00001018   3204    DIV1   R0,R2       ;
0000101A   4124    ROTCL  R1          ;
0000101C   3204    DIV1   R0,R2       ;
0000101E   4124    ROTCL  R1          ;
```

```
00001020   3204   32    DIV1   R0,R2      ;
00001022   4124   33    ROTCL  R1         ;
00001024   3204   34    DIV1   R0,R2      ;
00001026   4124   35    ROTCL  R1         ;
00001028   3204   36    DIV1   R0,R2      ;
                  37                      ;
0000102A   4124   38    ROTCL  R1         ;
0000102C   3204   39    DIV1   R0,R2      ;
0000102E   4124   40    ROTCL  R1         ;
00001030   3204   41    DIV1   R0,R2      ;
00001032   4124   42    ROTCL  R1         ;
00001034   3204   43    DIV1   R0,R2      ;
00001036   4124   44    ROTCL  R1         ;
00001038   3204   45    DIV1   R0,R2      ;
0000103A   4124   46    ROTCL  R1         ;
0000103C   3204   47    DIV1   R0,R2      ;
0000103E   4124   48    ROTCL  R1         ;
00001040   3204   49    DIV1   R0,R2      ;
00001042   4124   50    ROTCL  R1         ;
00001044   3204   51    DIV1   R0,R2      ;
00001046   4124   52    ROTCL  R1         ;
00001048   3204   53    DIV1   R0,R2      ;
                  54                      ;
0000104A   4124   55    ROTCL  R1         ;
0000104C   3204   56    DIV1   R0,R2      ;
0000104E   4124   57    ROTCL  R1         ;
00001050   3204   58    DIV1   R0,R2      ;
00001052   4124   59    ROTCL  R1         ;
00001054   3204   60    DIV1   R0,R2      ;
00001056   4124   61    ROTCL  R1         ;
00001058   3204   62    DIV1   R0,R2      ;
0000105A   4124   63    ROTCL  R1         ;
0000105C   3204   64    DIV1   R0,R2      ;
0000105E   4124   65    ROTCL  R1         ;
00001060   3204   66    DIV1   R0,R2      ;
00001062   4124   67    ROTCL  R1         ;
00001064   3204   68    DIV1   R0,R2      ;
```

```
69  00001066   4124   69    ROTCL  R1          ;
70  00001068   3204   70    DIV1   R0,R2       ;
71                    71                       ;
72  0000106A   4124   72    ROTCL  R1          ;
73  0000106C   3204   73    DIV1   R0,R2       ;
74  0000106E   4124   74    ROTCL  R1          ;
75  00001070   3204   75    DIV1   R0,R2       ;
76  00001072   4124   76    ROTCL  R1          ;
77  00001074   3204   77    DIV1   R0,R2       ;
78  00001076   4124   78    ROTCL  R1          ;
79  00001078   3204   79    DIV1   R0,R2       ;
80  0000107A   4124   80    ROTCL  R1          ;
81  0000107C   3204   81    DIV1   R0,R2       ;
82  0000107E   4124   82    ROTCL  R1          ;
83  00001080   3204   83    DIV1   R0,R2       ;
84  00001082   4124   84    ROTCL  R1          ;
85  00001084   3204   85    DIV1   R0,R2       ;
86  00001086   4124   86    ROTCL  R1          ;
87  00001088   3204   87    DIV1   R0,R2       ;
88                    88                       ;
89  0000108A   4124   89    ROTCL  R1          ;
90  0000108C   0008   90    CLRT               ;T bit ← No error
91  0000108E   000B   91    RTS                ;
92  00001090   62F6   92    MOV.L  @R15+,R2    ;Return register
93  00001092          93  DIVU32Q1             ;
94  00001092   0018   94    SETT               ;T bit ← Error
95  00001094   000B   95    RTS                ;
96  00001096   62F6   96    MOV.L  @R15+,R2    ;Return register
97                    97    .END
*****TOTAL  ERRORS 0
*****TOTAL  WARNINGS 0
```

3.12 DIVU32R: Remainder of 32 Bit ÷ 32 Bit (Unsigned)

- Instructions: DIV0U, DIV1
- Function: Divides the dividend (unsigned 32 bits) by the divisor (unsigned 32 bits), and determines remainder (unsigned 32 bits). Also indicates an error (division by 0) in the T bit.

Table 3.28 DIVU32R Arguments

	Contents	Storage Location	Data Length (Bytes)
Input	Dividend (unsigned 32 bits)	R1	4
	Divisor (unsigned 32 bits)	R0	4
Output	Remainder (unsigned 32 bits)	R2	4
	Error flag (with: T = 1, without: T = 0)	T bit (SR)	4

	(Pre-execution) → (Post-execution)
R0	Divisor (unsigned 32 bit) → No change
R1	Dividend (unsigned 32 bit) → Change
R2	Undefined → Remainder (unsigned 32 bit)
R3	
R4	
R5	
R6	
R7	
R8	
R9	
R10	
R11	
R12	
R13	
R14	
R15	(SP)

T bit: change

Figure 3.46 DIVU32R Internal Register Change and Flag Change

Table 3.29DIVU32R Programming Specifications

Item	Value/State
Program memory (bytes)	148
Data memory (bytes)	0
Stack (bytes)	0
Number of states	74
Reentrant	Enabled
Relocation	
Intermediate interrupt	

- Precautions: The number of program states is the value when H'FFFFFFFF ÷ H'FFFFFFFE is calculated.

3.12.1 DIVU32R Arguments

- R0: Holds the divisor (unsigned 32 bits) as the input argument.
- R1: Holds the dividend (unsigned 32 bits) as the input argument.
- R2: Holds the remainder (unsigned 32 bits) as the output argument.
- T bit (SR): Indicates whether an error (division by 0) has occurred in division.
- T bit = 1: Indicates an error (division by 0) in the division.
- T bit = 0: Indicates no error in the division.

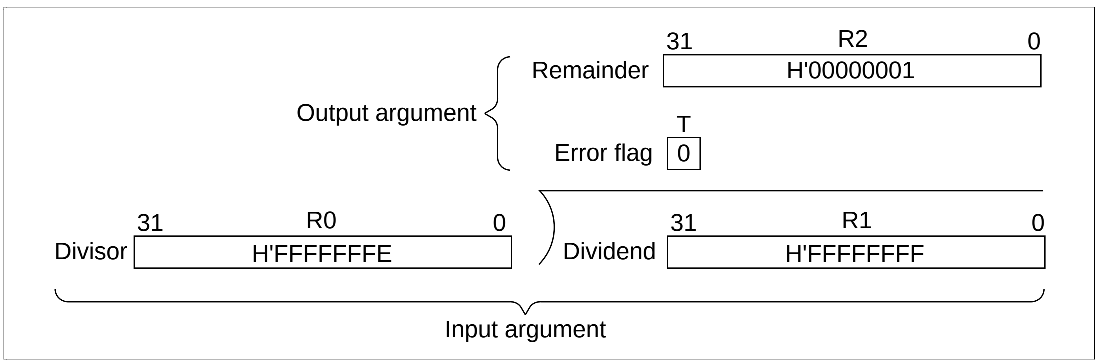


Figure 3.47 DIVU32R Execution Example

3.12.2 Precautions for DIVU32R Use

R1, which contains the dividend, has its contents changed due to DIVU32R execution. Be sure, therefore, to save the dividend beforehand if it is needed also after DIVU32R execution.

3.12.3 DIVU32R RAM Use

RAM is not used by DIVU32R.

3.12.4 Example of DIVU32R Use

Make a subroutine call to DIVU32R after setting the dividend and divisor in the input arguments.

```
        MOV.L    DATA1,R1     Sets dividend (unsigned 32 bits) in the input
                              argument (R1)
        BSR      DIVU32R      Subroutine call of DIVU32R
        MOV.L    DATA2,R0     Sets divisor (unsigned 32 bits) in the input
                              argument (R0)
        BT       ERROR        Branch to error processing subroutine when error
                              (division by 0) occurs
                 ↓
        .align   4
DATA1   .data.l  H'FFFFFFFF
DATA2   .data.l  H'FFFFFFFE
```

3.12.5 DIVU32R Operation

DIVU 32R carries out the following initial settings before division:

- Takes R2’s upper 32 bits and zero-extends the dividend to 64 bits (figure 3.48 (1)).
- Sets the M, Q, and T bits used in one-step division to unsigned division values. (figure 3.48 (2)).

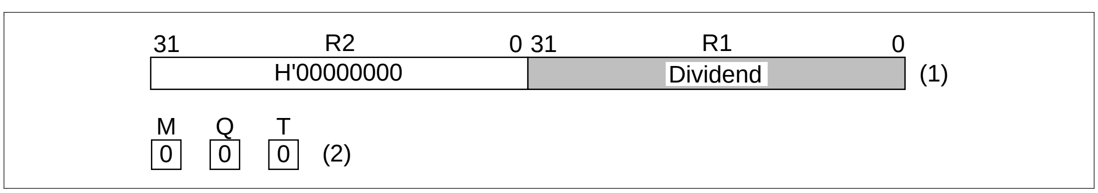


Figure 3.48 Initial Setting (DIVU32R)

ROTCL and DIV1 repeat the division operation through the number of divisor bits (32 times) (figure 3.49).

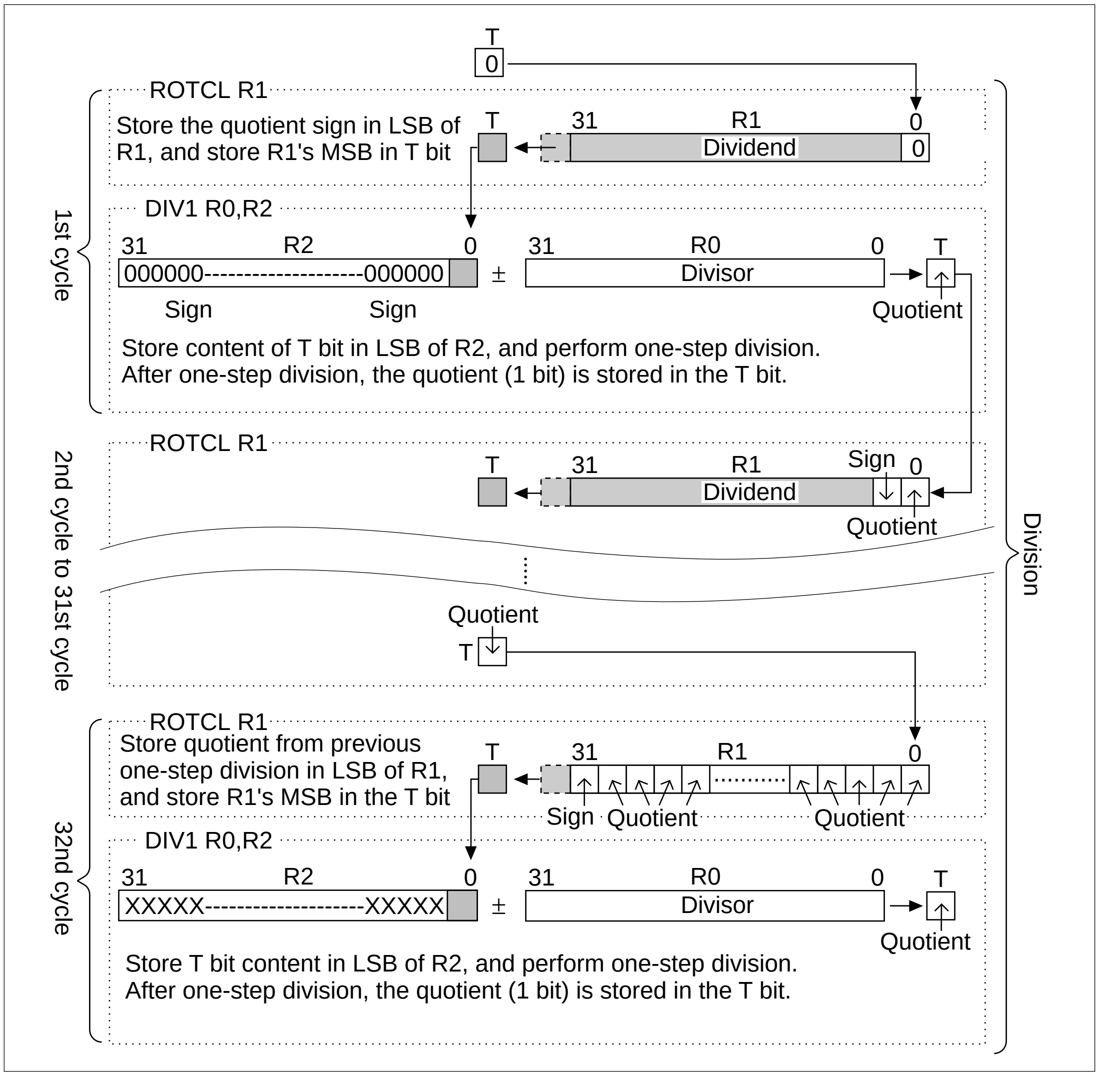


Figure 3.49 Division (DIVU32R)

At the end of division, the remainder is determined as follows from the 32nd quotient of one-step division stored in the T bit (figure 3.50):

- When T bit (quotient of 32nd one-step division) = 1:
  The contents of R2 at the end of the division become the remainder.
- When T bit (quotient of 32nd one-step division) = 0:
  Because the remainder value is subtracted one time too many due to the 32nd one-step division internal processing of R2, the remainder is the value of R2 at the end of division plus the divisor.

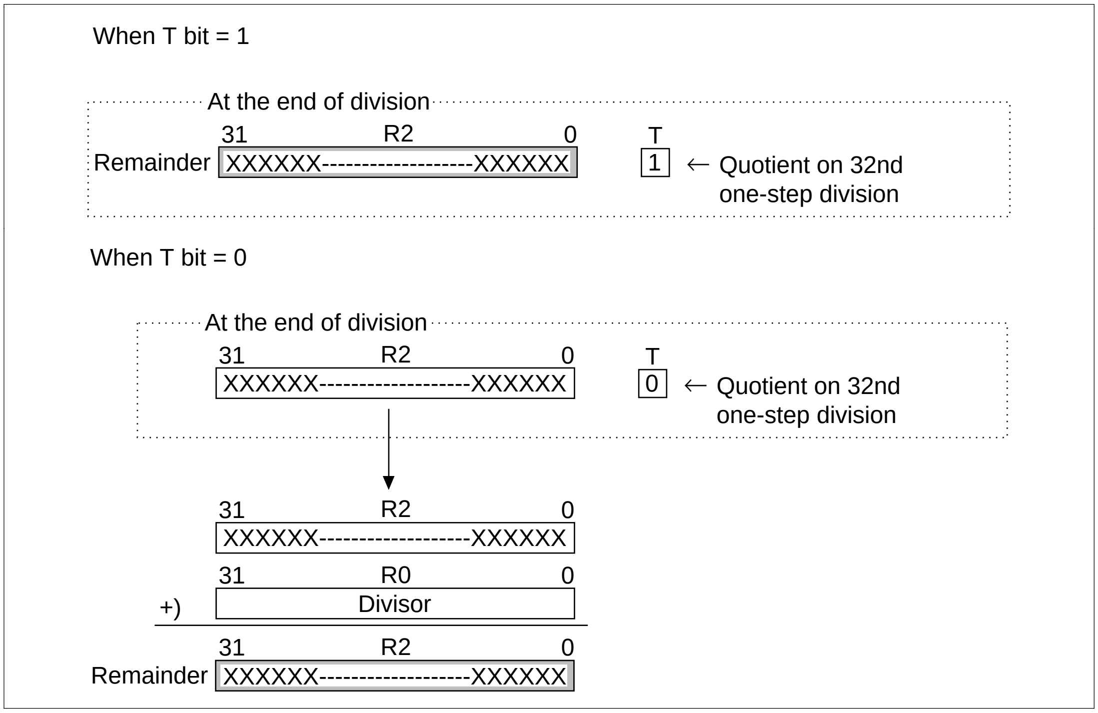


Figure 3.50 Remainder (DIVU32R)

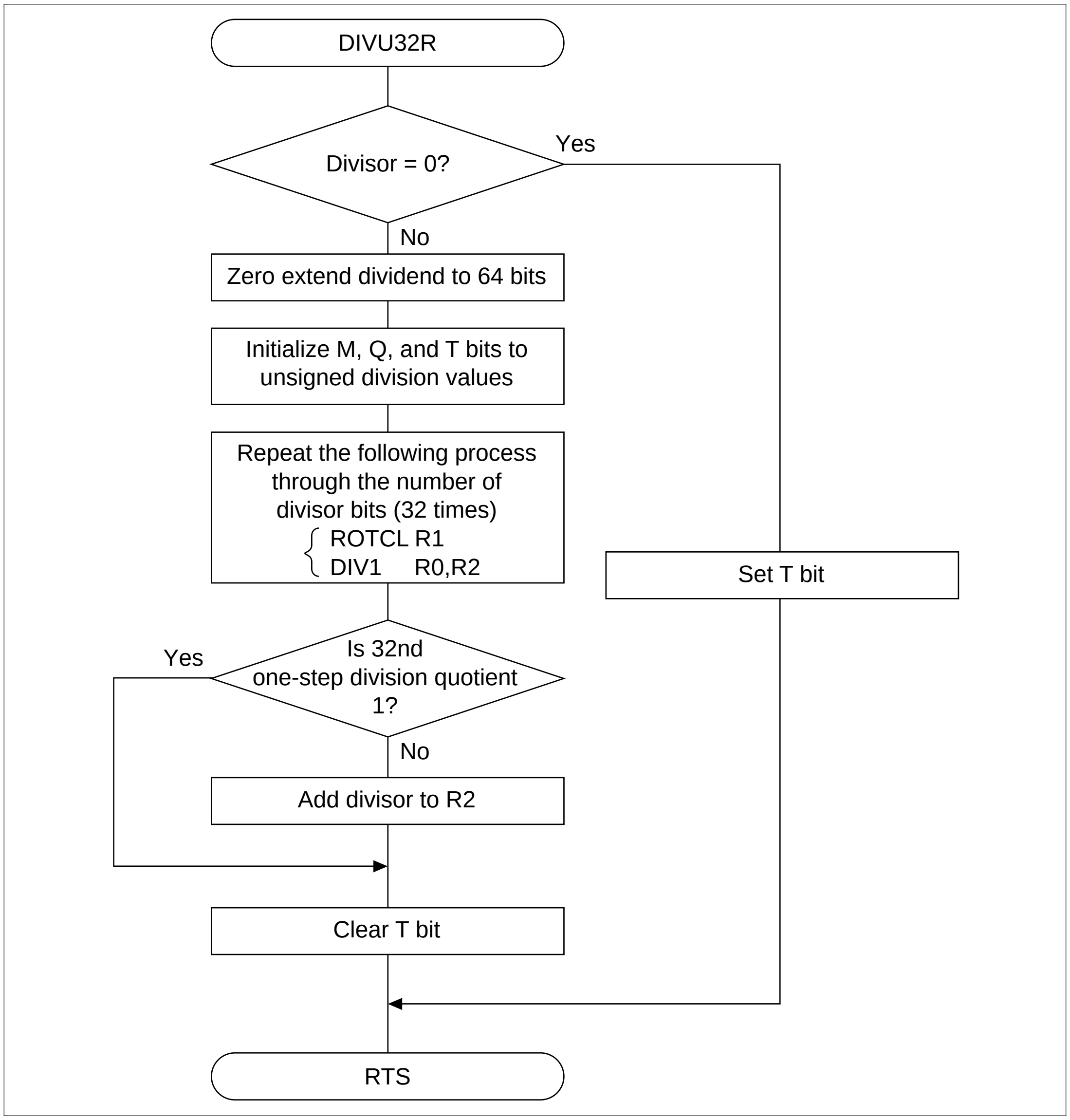


Figure 3.51 DIVU32R Flowchart

3.12.6 DIVU32R Program Listing

```
NAME:    RESIDUAL  OF 32 BIT UNSIGNED  DIVISION (DIVU32R)
ENTRY:   R1 (DIVIDEND)
         R0 (DIVISOR)
RETURNS: R2 (RESIDUAL)
         T BIT (ERROR → TRUE;T=1,FALSE;T=0)
                                             ;
                                             ;
                                             ;
                                             ;
                                             ;
                                             ;
                                             ;
                                             ;
                                             ;
                                             ;
                                             ;
                                             ;
00001000               .SECTION  A,CODE,LOCATE=H'1000
00001000             DIVU32R  .EQU  $    ;Entry point
00001000   2008        TST     R0,R0     ;Divisor = 0?
00001002   8945        BT      DIVU32R2  ;Yes
00001004   222A        XOR     R2,R2     ;R2 ← H'00000000
00001006   0019        DIV0U             ;Divide as unsigned
                                         ;
00001008   4124        ROTCL   R1        ;Divide 1 step
0000100A   3204        DIV1    R0,R2     ;
0000100C   4124        ROTCL   R1        ;
0000100E   3204        DIV1    R0,R2     ;
00001010   4124        ROTCL   R1        ;
00001012   3204        DIV1    R0,R2     ;
00001014   4124        ROTCL   R1        ;
00001016   3204        DIV1    R0,R2     ;
00001018   4124        ROTCL   R1        ;
0000101A   3204        DIV1    R0,R2     ;
0000101C   4124        ROTCL   R1        ;
0000101E   3204        DIV1    R0,R2     ;
```

```
00001020   4124        ROTCL  R1          ;
00001022   3204        DIV1  R0,R2        ;
00001024   4124        ROTCL  R1          ;
00001026   3204        DIV1  R0,R2        ;
                                          ;
00001028   4124        ROTCL  R1          ;
0000102A   3204        DIV1   R0,R2       ;
0000102C   4124        ROTCL  R1          ;
0000102E   3204        DIV1   R0,R2       ;
00001030   4124        ROTCL  R1          ;
00001032   3204        DIV1   R0,R2       ;
00001034   4124        ROTCL  R1          ;
00001036   3204        DIV1   R0,R2       ;
00001038   4124        ROTCL  R1          ;
0000103A   3204        DIV1   R0,R2       ;
0000103C   4124        ROTCL  R1          ;
0000103E   3204        DIV1   R0,R2       ;
00001040   4124        ROTCL  R1          ;
00001042   3204        DIV1   R0,R2       ;
00001044   4124        ROTCL  R1          ;
00001046   3204        DIV1   R0,R2       ;
                                          ;
00001048   4124        ROTCL  R1          ;
0000104A   3204        DIV1   R0,R2       ;
0000104C   4124        ROTCL  R1          ;
0000104E   3204        DIV1   R0,R2       ;
00001050   4124        ROTCL  R1          ;
00001052   3204        DIV1   R0,R2       ;
00001054   4124        ROTCL  R1          ;
00001056   3204        DIV1   R0,R2       ;
00001058   4124        ROTCL  R1          ;
0000105A   3204        DIV1   R0,R2       ;
0000105C   4124        ROTCL  R1          ;
0000105E   3204        DIV1   R0,R2       ;
00001060   4124        ROTCL  R1          ;
00001062   3204        DIV1   R0,R2       ;
00001064   4124        ROTCL  R1          ;
```

```
69  00001066   3204   69    DIV1   R0,R2      ;
70                    70                      ;
71  00001068   4124   71    ROTCL  R1         ;
72  0000106A   3204   72    DIV1   R0,R2      ;
73  0000106C   4124   73    ROTCL  R1         ;
74  0000106E   3204   74    DIV1   R0,R2      ;
75  00001070   4124   75    ROTCL  R1         ;
76  00001072   3204   76    DIV1   R0,R2      ;
77  00001074   4124   77    ROTCL  R1         ;
78  00001076   3204   78    DIV1   R0,R2      ;
79  00001078   4124   79    ROTCL  R1         ;
80  0000107A   3204   80    DIV1   R0,R2      ;
81  0000107C   4124   81    ROTCL  R1         ;
82  0000107E   3204   82    DIV1   R0,R2      ;
83  00001080   4124   83    ROTCL  R1         ;
84  00001082   3204   84    DIV1   R0,R2      ;
85  00001084   4124   85    ROTCL  R1         ;
86  00001086   3204   86    DIV1   R0,R2      ;
87                    87                      ;
88  00001088   8900   88    BT     DIVU32R1   ;T bit = 1?
89  0000108A   320C   89    ADD    R0,R2      ;Clear oversub
90  0000108C          90  DIVU32R1            ;
91  0000108C   000B   91    RTS               ;
92  0000108E   0008   92    CLRT              ;T bit ← No error
93  00001090          93  DIVU32R2            ;
94  00001090   000B   94    RTS               ;
95  00001092   0018   95    SETT              ;T bit ← Error
96                    96    .END
*****TOTAL  ERRORS 0
*****TOTAL  WARNINGS 0
```

3.13 DIVS32Q: Quotient of 32 Bit ÷ 32 Bit (Signed)

- Instructions: DIV0S, DIV1
- Function: Divides the dividend (signed 32 bits) by the divisor (signed 32 bits), and determines the quotient (signed 32 bits). Also indicates an error (division by 0) in the T bit.

Table 3.30 DIVS32Q Arguments

	Contents	Storage Location	Data Length (Bytes)
Input	Dividend (signed 32 bits)	R1	4
	Divisor (signed 32 bits)	R0	4
Output	Quotient (signed 32 bits)	R1	4
	Error flag (with: T = 1, without: T = 0)	T bit (SR)	4

	(Pre-execution) → (Post-execution)
R0	Divisor (signed 32 bit) → No change
R1	Dividend (signed 32 bit) → Quotient (signed 32 bit)
R2	Work
R3	Work
R4	
R5	
R6	
R7	
R8	
R9	
R10	
R11	
R12	
R13	
R14	
R15	(SP)

T bit: change

Figure 3.52 DIVS32Q Internal Register Change and Flag Change

Table 3.31DIVS32Q Programming Specifications

Item	Value/State
Program memory (bytes)	166
Data memory (bytes)	0
Stack (bytes)	8
Number of states	80
Reentrant	Enabled
Relocation	
Intermediate interrupt	

- Precautions: The number of program states is the value when H'80000000 ÷ H'7FFFFFFF is calculated.

3.13.1 DIVS32Q Arguments

- R0: Holds the divisor (signed 32 bits) as the input argument.
- R1: Holds the dividend (signed 32 bits) as the input argument.
  Holds the quotient (signed 32 bits) as the output argument.
- T bit (SR): Indicates whether an error (division by 0) has occurred in division.
- T bit = 1: Error (division by 0).
- T bit = 0: No error.

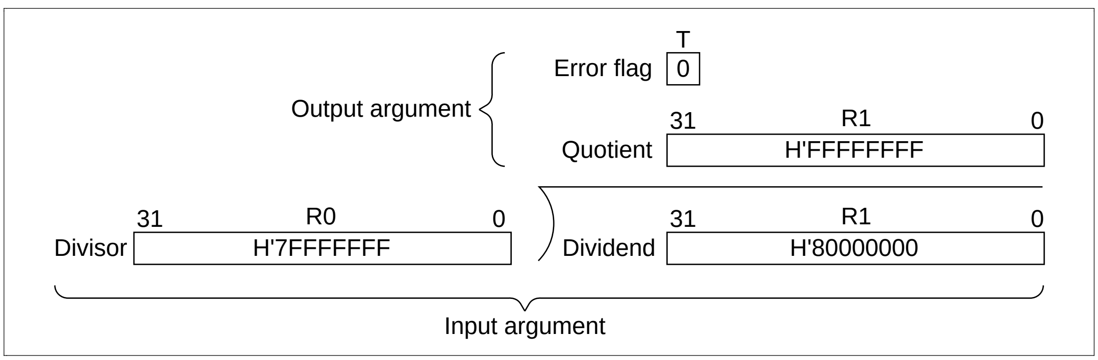


Figure 3.53 DIVS32Q Execution Example

3.13.2 Precautions for DIVS32Q Use

As the quotient is set in R1, which contains the dividend after DIVS32Q execution, the dividend is destroyed. Be sure, therefore, to save the dividend beforehand if it is needed also after DIVS32Q execution.

Also, although H'80000000 ÷ H'FFFFFFFF is subject to overflow, this overflow is not detected with DIVS32Q.

3.13.3 DIVS32Q RAM Use

RAM is not used by DIVS32Q.

3.13.4 Example of DIVS32Q Use

Make a subroutine call to DIVS32Q after setting the dividend and divisor.

```
        MOV.L     DATA1,R1        Sets dividend (signed 32 bits) in input argument R1)
        BSR       DIVS32Q         Subroutine call of DIVS32Q
        MOV.L     DATA2,R0        Sets divisor (signed 32 bits) in input argument (R0)
        BT        ERROR           Branch to error processing subroutine when
                                  error(division by 0) occurs
                  ↓
        .align    4
DATA1   .data.l   H'80000000
DATA2   .data.l   H'7FFFFFFF
```

3.13.5 DIVS32Q Operation

Carries out the following initial settings before division:

1. Takes R2’s upper 32 bits and zero-extends the dividend to 64 bits (figure 3.54 (1)).
2. If the dividend is negative, converts it to 1’s complement for handling by the one-step division instruction (figure 3.54 (2)).
3. Sets the M, Q and T bits used in one-step division to signed division values (M = divisor sign, Q = dividend sign, T = quotient sign) (figure 3.54 (3)).

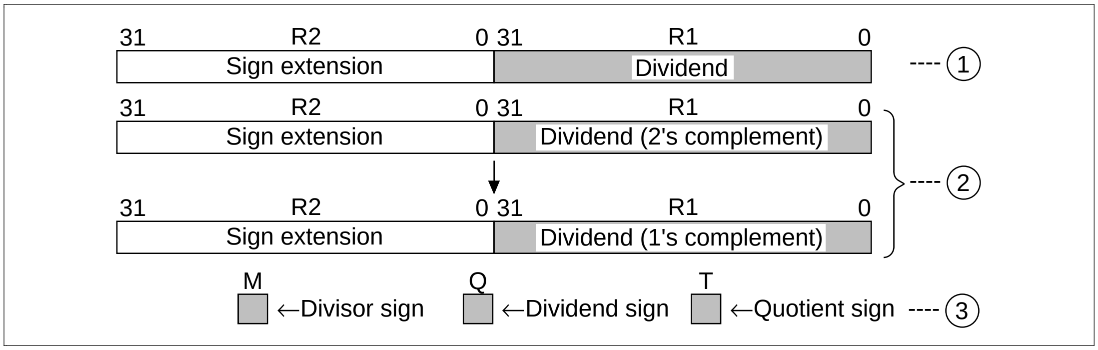


Figure 3.54 Initial Setting (DIVS32Q)

ROTCL and DIV1 repeat the division operation through the number of divisor bits (32 times) (figure 3.55).

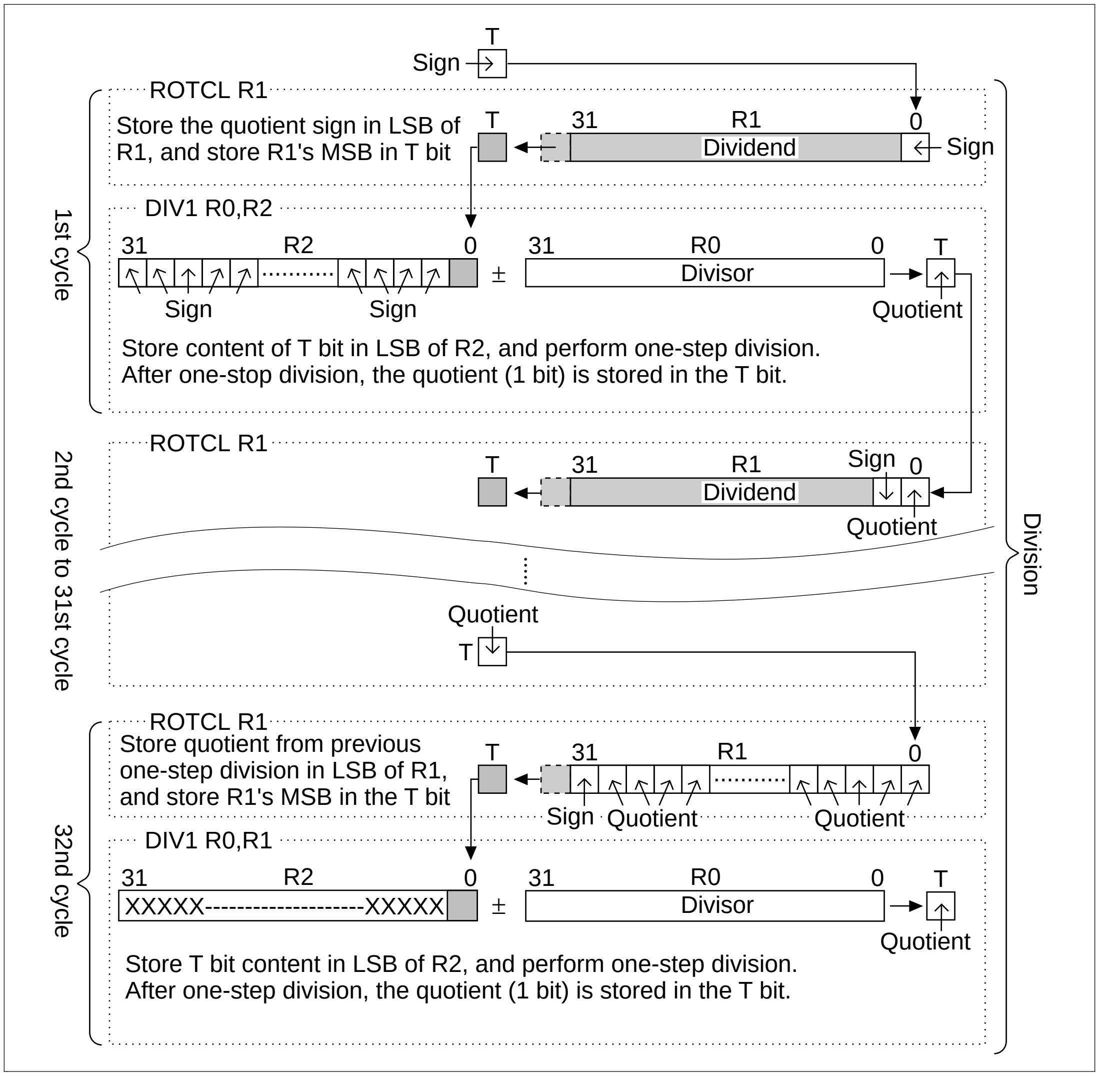


Figure 3.55 Operation Example (DIVS32Q)

At the end of division, the 32nd quotient of one-step division is stored in the T bit, and the quotient sign in the MSB of R1. If the quotient is positive, it becomes the contents of R1, which stores the T bit (32nd quotient of one-step division) in the LSB. If the quotient is negative, it becomes the 1’s complement of the T bit (32nd quotient of one-step division) stored in R1’s LSB, which in turn is converted to 2’s complement (figure 3.56).

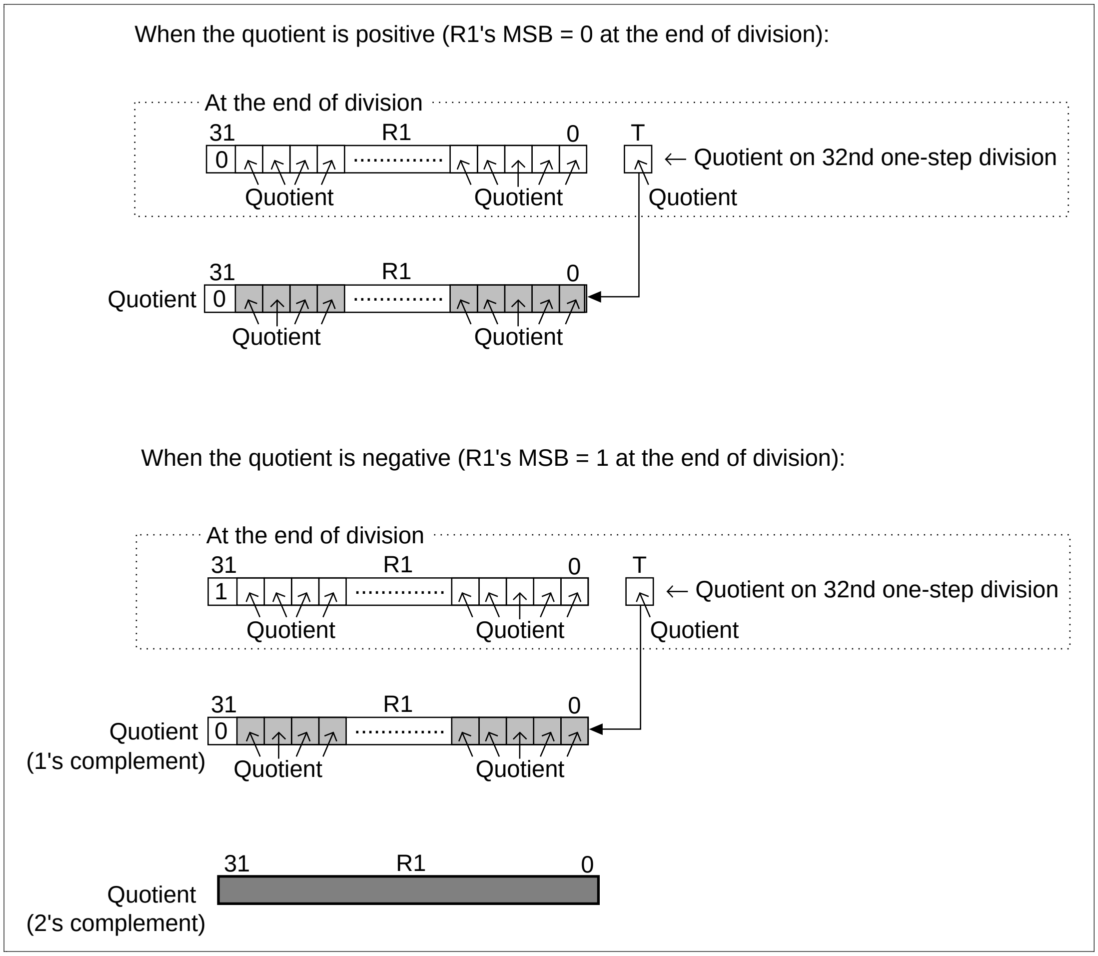


Figure 3.56 Quotient (DIVS32Q)

Carry out the preceding processing as follows with DIVS32Q. Note that R3 stores H'00000000.

`ROTCL`	`R1`	←	Stores quotient sign in T bit, and saves T bit quotient to R1's LSB
`ADDC`	`R3, R1`	←	If quotient is positive, T bit = 0, so there is no change of value. If quotient is negative, T bit = 1, so 1 is added to make it 2's complement.

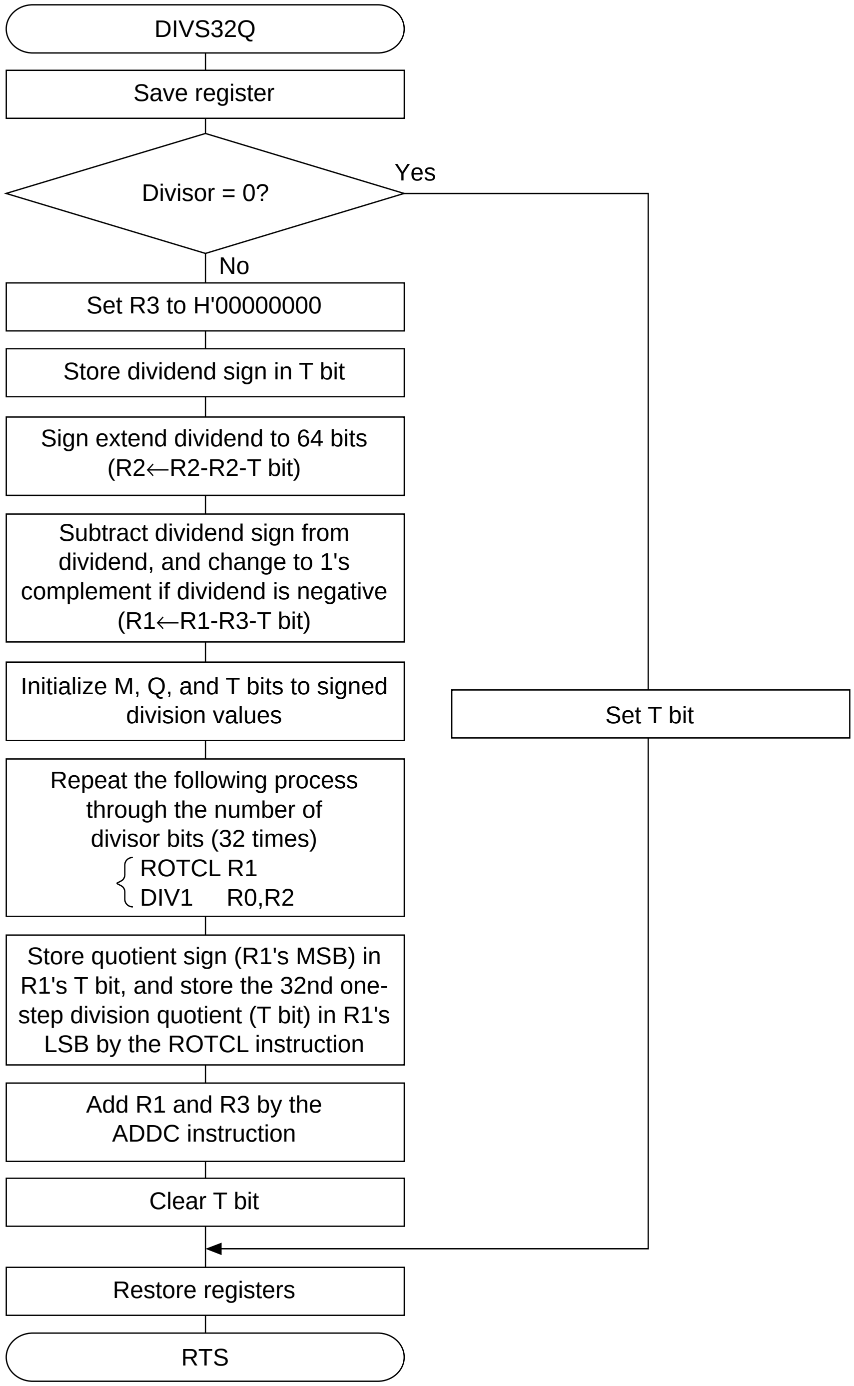


Figure 3.57 DIVS32Q Flowchart

3.13.6 DIVS32Q Program Listing

```
NAME:    QUOTIENT  OF 32 BIT SIGNED DIVISION (DIVS32Q)
ENTRY:   R1 (DIVIDEND)
         R0 (DIVISOR)
RETURNS: R1 (QUOTIENT)
         T BIT (ERROR → TRUE;T=1,FALSE;T=0)
                                     ;
                                     ;
                                     ;
                                     ;
                                     ;
                                     ;
                                     ;
                                     ;
                                     ;
                                     ;
                                     ;
                                     ;
00001000            .SECTION  A,CODE,LOCATE=H'1000
00001000          DIVS32Q  .EQU  $   ;Entry point
00001000   2F26     MOV.L  R2,@-R15  ;Escape register
00001002   2F36     MOV.L  R3,@-R15  ;
00001004   2008     TST    R0,R0     ;Divisor= 0?
00001006   894A     BT     DIVS32Q1  ;Yes
00001008   233A     XOR    R3,R3     ;R3 ← H'00000000
0000100A   2137     DIV0S  R3,R1     ;T bit←Sign of dividend
0000100C   322A     SUBC   R2,R2     ;R2 sign extend
0000100E   313A     SUBC   R3,R1     ;
00001010   2207     DIV0S  R0,R2     ;Divide as signed
                                     ;
00001012   4124     ROTCL  R1        ;Divide 1 step
00001014   3204     DIV1   R0,R2     ;
00001016   4124     ROTCL  R1        ;
00001018   3204     DIV1   R0,R2     ;
0000101A   4124     ROTCL  R1        ;
0000101C   3204     DIV1   R0,R2     ;
```

```
31  0000101E   4124   31    ROTCL  R1          ;
32  00001020   3204   32    DIV1   R0,R2       ;
33  00001022   4124   33    ROTCL  R1          ;
34  00001024   3204   34    DIV1   R0,R2       ;
35  00001026   4124   35    ROTCL  R1          ;
36  00001028   3204   36    DIV1   R0,R2       ;
37  0000102A   4124   37    ROTCL  R1          ;
38  0000102C   3204   38    DIV1   R0,R2       ;
39  0000102E   4124   39    ROTCL  R1          ;
40  00001030   3204   40    DIV1   R0,R2       ;
41                    41                       ;
42  00001032   4124   42    ROTCL  R1          ;
43  00001034   3204   43    DIV1   R0,R2       ;
44  00001036   4124   44    ROTCL  R1          ;
45  00001038   3204   45    DIV1   R0,R2       ;
46  0000103A   4124   46    ROTCL  R1          ;
47  0000103C   3204   47    DIV1   R0,R2       ;
48  0000103E   4124   48    ROTCL  R1          ;
49  00001040   3204   49    DIV1   R0,R2       ;
50  00001042   4124   50    ROTCL  R1          ;
51  00001044   3204   51    DIV1   R0,R2       ;
52  00001046   4124   52    ROTCL  R1          ;
53  00001048   3204   53    DIV1   R0,R2       ;
54  0000104A   4124   54    ROTCL  R1          ;
55  0000104C   3204   55    DIV1   R0,R2       ;
56  0000104E   4124   56    ROTCL  R1          ;
57  00001050   3204   57    DIV1   R0,R2       ;
58                    58                       ;
59  00001052   4124   59    ROTCL  R1          ;
60  00001054   3204   60    DIV1   R0,R2       ;
61  00001056   4124   61    ROTCL  R1          ;
62  00001058   3204   62    DIV1   R0,R2       ;
63  0000105A   4124   63    ROTCL  R1          ;
64  0000105C   3204   64    DIV1   R0,R2       ;
65  0000105E   4124   65    ROTCL  R1          ;
66  00001060   3204   66    DIV1   R0,R2       ;
67  00001062   4124   67    ROTCL  R1          ;
```

```
68  00001064   3204  68    DIV1   R0,R2      ;
69  00001066   4124  69    ROTCL  R1         ;
70  00001068   3204  70    DIV1   R0,R2      ;
71  0000106A   4124  71    ROTCL  R1         ;
72  0000106C   3204  72    DIV1   R0,R2      ;
73  0000106E   4124  73    ROTCL  R1         ;
74  00001070   3204  74    DIV1   R0,R2      ;
75                   75                      ;
76  00001072   4124  76    ROTCL  R1         ;
77  00001074   3204  77    DIV1   R0,R2      ;
78  00001076   4124  78    ROTCL  R1         ;
79  00001078   3204  79    DIV1   R0,R2      ;
80  0000107A   4124  80    ROTCL  R1         ;
81  0000107C   3204  81    DIV1   R0,R2      ;
82  0000107E   4124  82    ROTCL  R1         ;
83  00001080   3204  83    DIV1   R0,R2      ;
84  00001082   4124  84    ROTCL  R1         ;
85  00001084   3204  85    DIV1   R0,R2      ;
86  00001086   4124  86    ROTCL  R1         ;
87  00001088   3204  87    DIV1   R0,R2      ;
88  0000108A   4124  88    ROTCL  R1         ;
89  0000108C   3204  89    DIV1   R0,R2      ;
90  0000108E   4124  90    ROTCL  R1         ;
91  00001090   3204  91    DIV1   R0,R2      ;
92                   92                      ;
93  00001092   4124  93    ROTCL  R1         ;
94  00001094   313E  94    ADDC   R3,R1      ;
95  00001096   0008  95    CLRT              ;T bit ← No error
96  00001098   63F6  96    MOV.L  @R15+,R3   ;Return register
97  0000109A   000B  97    RTS               ;
98  0000109C   62F6  98    MOV.L  @R15+,R2   ;
99  0000109E         99  DIVS32Q1            ;
100 0000109E   0018  100    SETT             ;T bit ← Error
101 000010A0   63F6  101    MOV.L  @R15+,R3  ;Return register
102 000010A2   000B  102    RTS              ;
103 000010A4   62F6  103    MOV.L  @R15+,R2  ;
104                  104    .END
```

```
*****TOTAL  ERRORS 0
*****TOTAL  WARNINGS 0
```

3.14 DIVS32R: Remainder of 32 Bit ÷ 32 Bit (Signed)

- Instructions: DIV0S, DIV1
- Function: Divides the dividend (signed 32 bits) by the divisor (signed 32 bits), and determines remainder (signed 32 bits). Also indicates an error (division by 0) in the T bit.

Table 3.32 DIVS32R Arguments

	Contents	Storage Location	Data Length (Bytes)
Input	Dividend (signed 32 bits)	R1	4
	Divisor (signed 32 bits)	R0	4
Output	Remainder (signed 32 bits)	R2	4
	Error flag (error: T = 1, no error: T = 0)	T bit (SR)	4

	(Pre-execution)→(Post-execution)
R0	Divisor (signed 32 bit)→No change
R1	Dividend (signed 32 bit)→Change
R2	Undefined→Remainder (signed 32 bit)
R3	Work
R4	Work
R5	
R6	
R7	
R8	
R9	
R10	
R11	
R12	
R13	
R14	
R15	(SP)

T bit: change

Figure 3.58 DIVS32R Internal Register Change and Flag Change

Table 3.33DIVS32R Programming Specifications

Item	Value/State
Program memory (bytes)	182
Data memory (bytes)	0
Stack (bytes)	8
Number of states	87
Reentrant	Enabled
Relocation	
Intermediate interrupt	

- Precautions: The number of program states is the value when H'80000000 ÷ H'7FFFFFFF is calculated.

3.14.1 DIVS32R Arguments

- R0: Holds the divisor (signed 32 bits) as the input argument.
- R1: Holds the dividend (signed 32 bits) as the input argument.
- R2: Holds the remainder (signed 32 bits) as the output argument.
- T bit (SR): Indicates whether an error (division by 0) has occurred in division.
- T bit = 1: Error (division by 0).
- T bit = 0: No error.

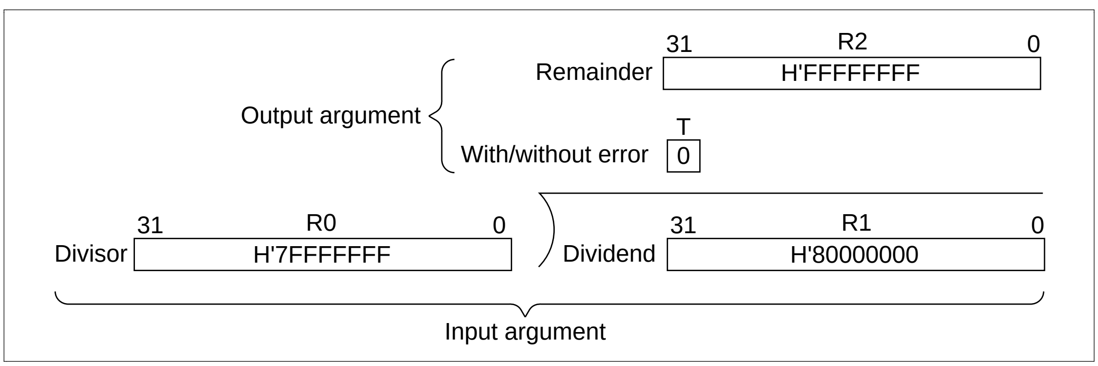


Figure 3.59 DIVS32R Execution Example

3.14.2 Precautions for DIVS32R Use

R1, which contains the dividend, has its contents changed due to DIVS32R execution. Be sure, therefore, to save the dividend beforehand if it is needed also after DIVS32R execution.

3.14.3 DIVS32R RAM Use

RAM is not used by DIVS32R.

3.14.4 Example of DIVS32R Use

```
        MOV.L    DATA1, R1        Sets dividend (signed 32 bits) in input argument (R1)
        BSR      DIVS32R          Subroutine call of DIVS32R
        MOV.L    DATA2, R0        Sets divisor (signed 32 bits) in input argument (R0)
        BT       ERROR            Branch to error processing subroutine when error
                                  (division by 0) occurs
                 ↓
        .align   4
DATA1   .data.l  H'80000000
DATA2   .data.2  H'7FFFFFFF
```

3.14.5 DIVS32R Operation

Carries out the following initial settings before division:

1. Takes R2’s upper 32 bits and zero-extends the dividend to 64 bits (figure 3.60 (1)).
2. If the dividend is negative, converts it to 1’s complement for handling by the one-step division instruction (figure 3.60 (2)).
3. Sets the M, Q, and T bits used in one-step division to signed division values (M = divisor sign, Q = dividend sign, T = quotient sign) (figure 3.60 (3)).

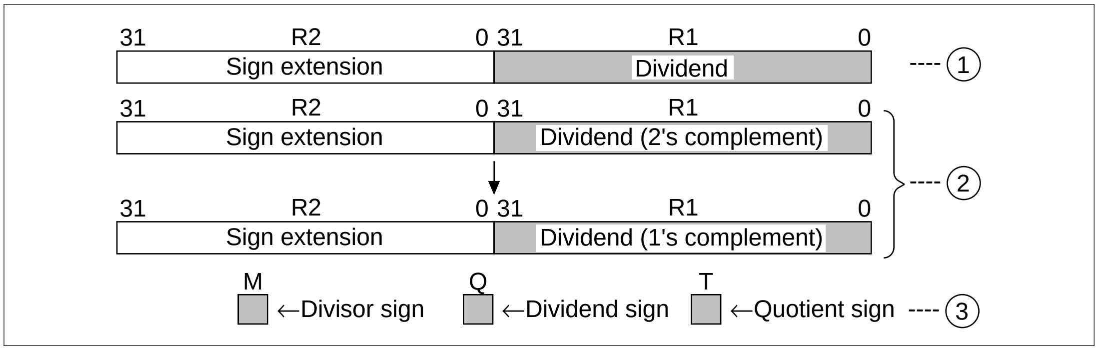


Figure 3.60 Initial Setting (DIVS32R)

ROTCL and DIV1 repeat the division operation through the number of divisor bits (32 times). See figure 3.61.

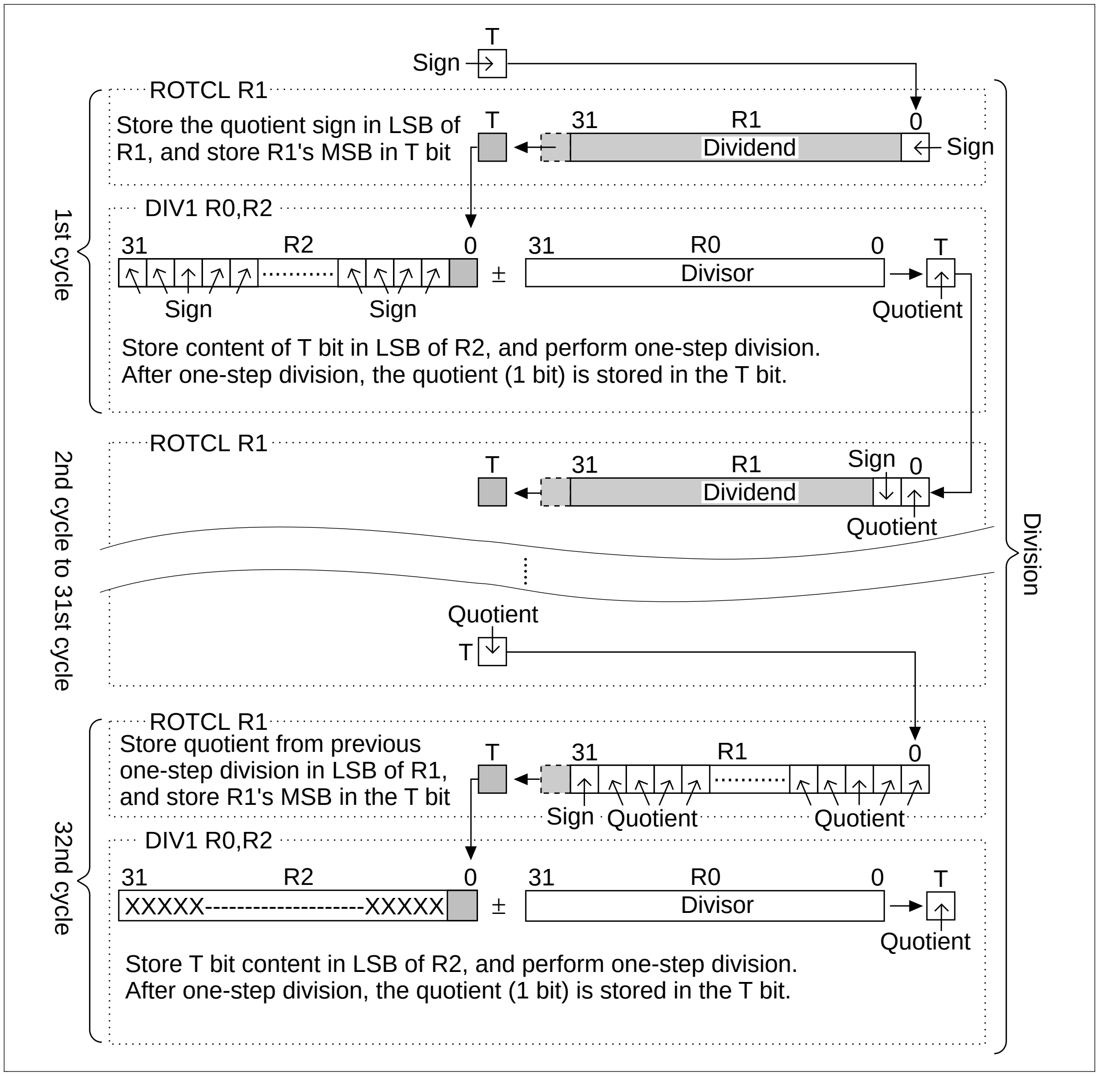


Figure 3.61 Operation Example (DIVS32R)

The method of determining the remainder is different depending on the contents of the dividend sign and the T bit (32nd one-step division quotient).

When the dividend is positive and T = 0, the content of R2 at the end of division becomes the remainder (figure 3.62).

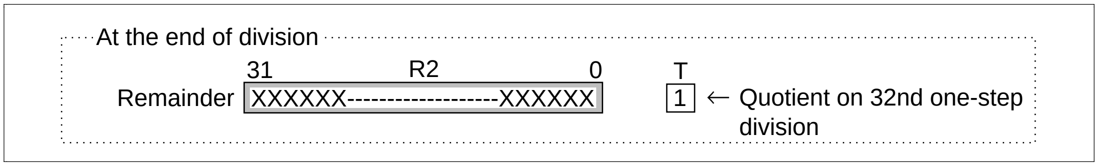


Figure 3.62 Remainder (Dividend Positive, T = 0)

When the dividend is positive and T = 1, the content of R2 at the end of division, due to internal processing of the 32nd one-step division, is a one-time oversubtraction of the divisor from the remainder. Therefore, the divisor is added to R2 (figure 3.63).

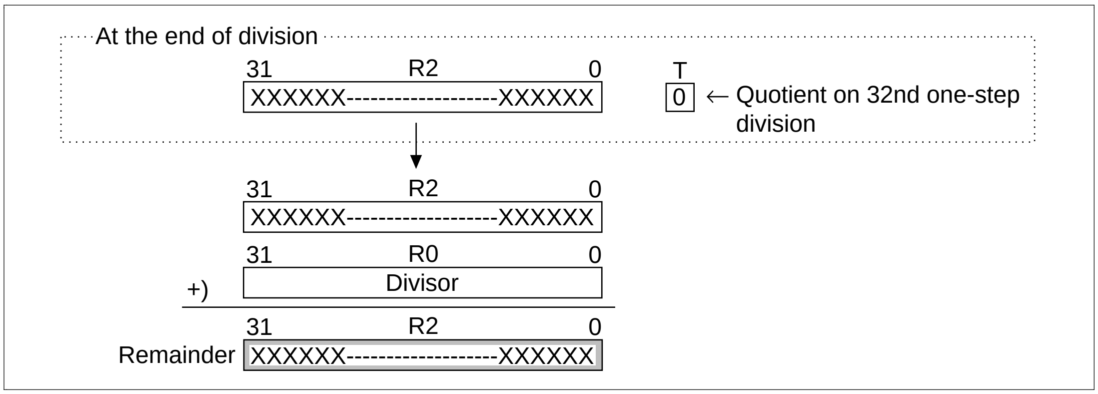


Figure 3.63 Remainder (Dividend Positive, T = 1)

When the dividend is negative and T = 1, the content of R2 at the end of division, due to internal processing of the 32nd one-step division, is the remainder plus one divisor over. Therefore, the divisor is subtracted from R2. Since the content of R2 becomes 1's complement, it must then be changed to 2's complement (figure 3.64).

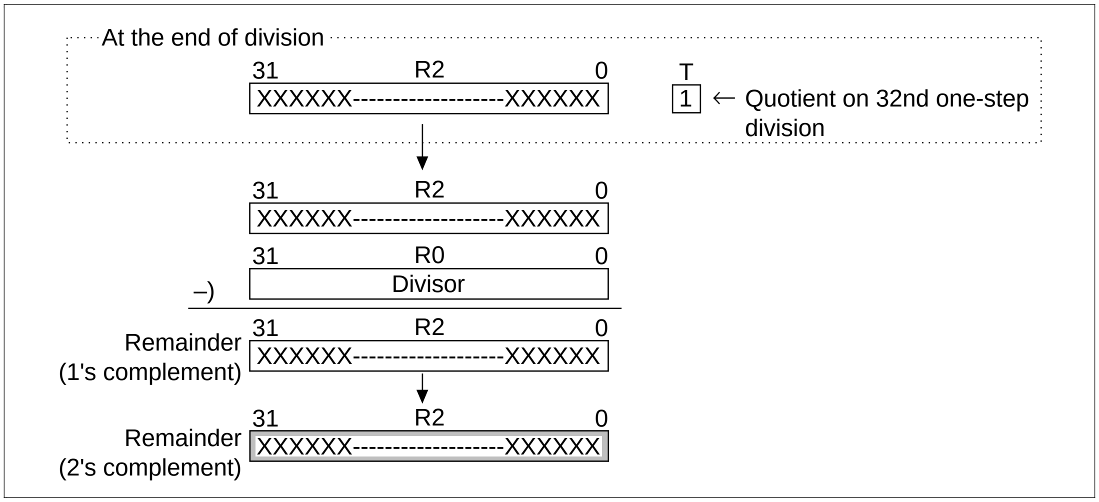


Figure 3.64 Remainder (Dividend Negative, T = 1)

When the dividend is negative and T = 0, the content of R2 at the end of division, since it is a 1's complement remainder, is changed to 2's complement.

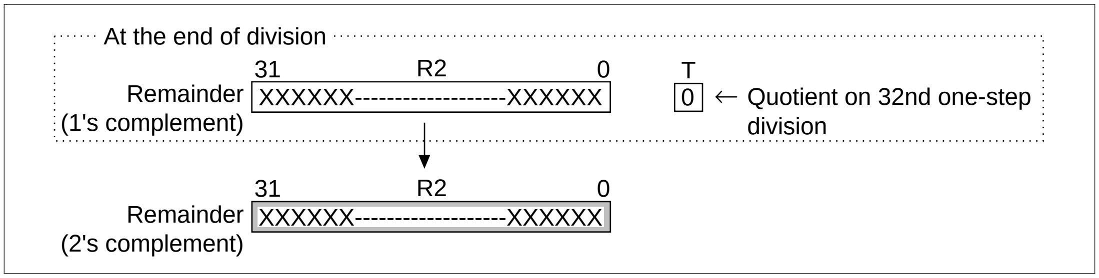


Figure 3.65 Remainder (Dividend Negative, T = 0)

Carry out the preceding processing as follows with DIVS32R. Note that R3 stores H'00000000, and the sign bit of the dividend is stored in the LSB of R4.

Use the initialization instruction (DIV0S) of unsigned division to store the remainder's sign bit in the T bit and store the remainder sign bit in the T bit to R3.

```
DIV0S   R3 R2             Remainder sign bit → R3
MOVT    R3
```

Since the remainder sign bit and dividend sign bit become the same sign (table 3.34), the dividend sign (R4) and remainder sign (R3) are exclusively ORed, and a check is made as to whether the two signs match. If the signs are different, the remainder is the value with the divisor added or subtracted one time too many.

Table 3.34 Dividend/Remainder Signs

Dividend Sign	Remainder Sign
Positive	Positive
Negative	Negative

```
XOR    R4, R3          Dividend sign = remainder sign?
ROTCR  R3
BF     DIVS32R1
```

The DIV1 instruction is used in correction for oversubtraction or overaddition.

```
DIV0S  R0, R2    ←   Initialization of signed division
SHAR   R2        ←   The remainder (R2) is halved due to execution of the next DIV1 instruction,
                     so R2 is doubled
DIV1   R0, R2    ←   When oversubtraction occurs due to DIV1 internal processing, the
                     remainder is added, and the remainder is subtracted when overaddition
                     occurs
```

R4, containing the remainder and dividend sign bit in the LSB is added to the remainder (R2). The remainder and dividend sign bits are the same sign, so if the remainder is positive, 0 is added and the remainder does not change. If the remainder is negative, 1 is added and it becomes 2's complement.

```
ADD  R4, R2
```

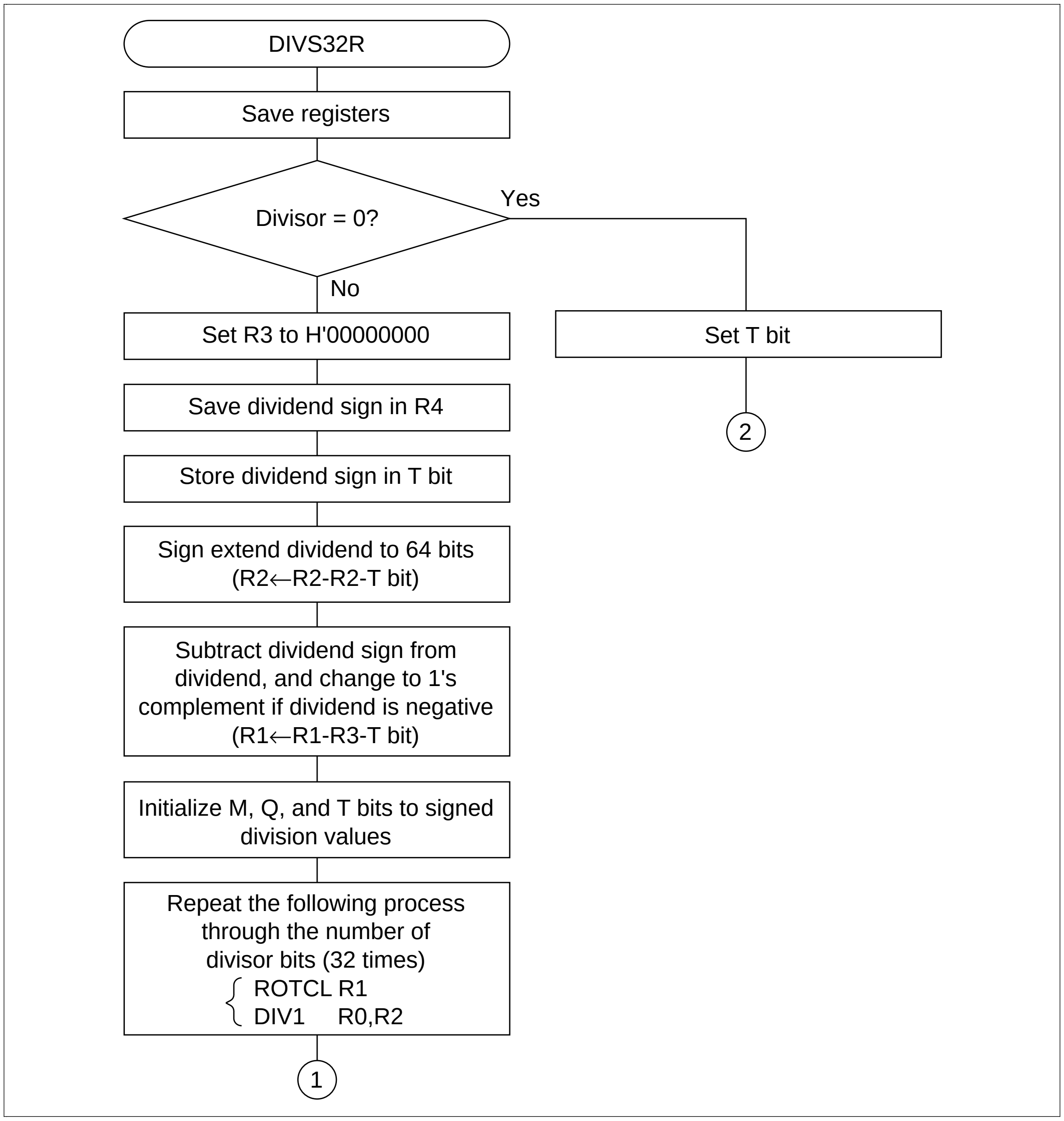


Figure 3.66 DIVS32R Flowchart

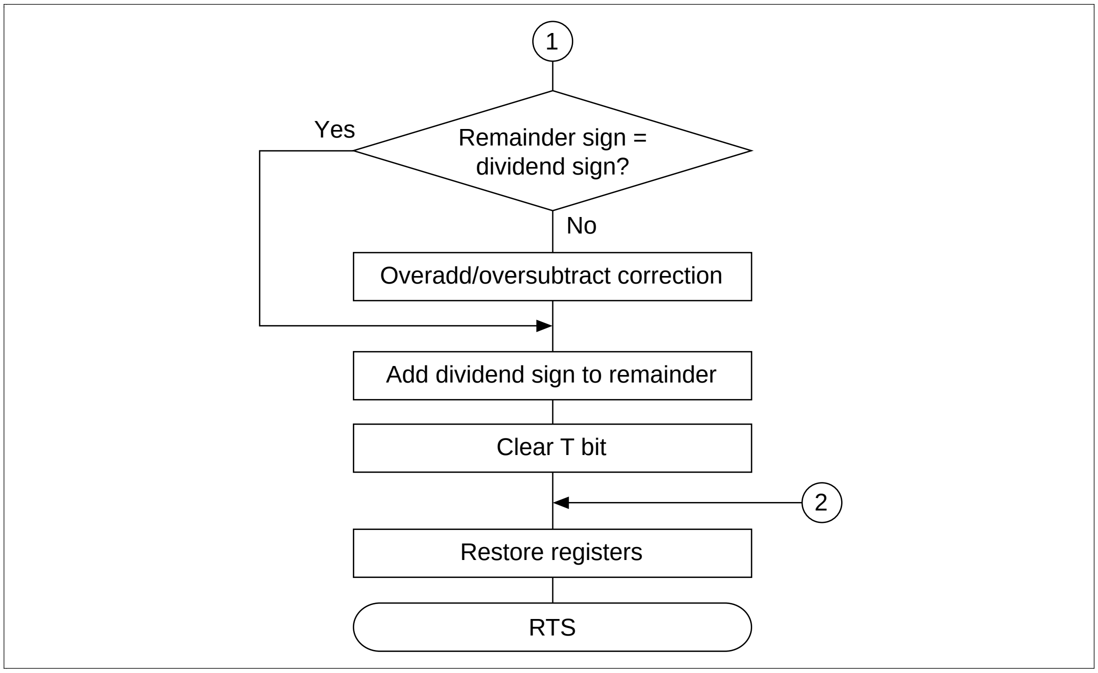


Figure 3.66 DIVS32R Flowchart (cont)

3.14.6 DIVS32R Program Listing

```
NAME:    RESIDUAL OF 32 BIT SIGNEDDIVISION (DIVS32R)
ENTRY:   R1 (DIVIDEND)
         R0 (DIVISOR)
RETURNS: R2 (RESIDUAL)
         T BIT (ERROR → TRUE ;T=1,FALSE ;T=0)
1                    1                        ;
2                    2                        ;
3                    3                        ;
4                    4                        ;
5                    5                        ;
6                    6                        ;
7                    7                        ;
8                    8                        ;
9                    9                        ;
10                   10                       ;
11                   11                       ;
12                   12                       ;
13  00001000         13    .SECTION  A,CODE,LOCATE=H'1000
14  00001000         14  DIVS32R  .EQU  $     ; Entry point
15  00001000   2F36  15    MOV.L  R3,@-R15    ; Escape register
16  00001002   2F46  16    MOV.L  R4,@-R15    ;
17  00001004   2008  17    TST    R0,R0       ; Divisor = 0?
18  00001006   8952  18    BT     DIVS32R2    ; Yes
19  00001008   233A  19    XOR    R3,R3       ;R3 ← H'00000000
20  0000100A   2137  20    DIV0S  R3,R1       ;T bit←Sign of Dividend
21  0000100C   0429  21    MOVT   R4          ;R4 ← T bit
22  0000100E   322A  22    SUBC   R2,R2       ;R2 sign extend
23  00001010   313A  23    SUBC   R3,R1       ;
24  00001012   2207  24    DIV0S  R0,R2       ;Divide as signed
25                   25                       ;
26  00001014   4124  26    ROTCL  R1          ;Divide 1 step
27  00001016   3204  27    DIV1   R0,R2       ;
28  00001018   4124  28    ROTCL  R1          ;
29  0000101A   3204  29    DIV1   R0,R2       ;
30  0000101C   4124  30    ROTCL  R1          ;
31  0000101E   3204  31    DIV1   R0,R2       ;
```

```
00001020   4124   32    ROTCL   R1          ;
00001022   3204   33    DIV1    R0,R2       ;
00001024   4124   34    ROTCL   R1          ;
00001026   3204   35    DIV1    R0,R2       ;
00001028   4124   36    ROTCL   R1          ;
0000102A   3204   37    DIV1    R0,R2       ;
0000102C   4124   38    ROTCL   R1          ;
0000102E   3204   39    DIV1    R0,R2       ;
00001030   4124   40    ROTCL   R1          ;
00001032   3204   41    DIV1    R0,R2       ;
                  42                        ;
00001034   4124   43    ROTCL   R1          ;
00001036   3204   44    DIV1    R0,R2       ;
00001038   4124   45    ROTCL   R1          ;
0000103A   3204   46    DIV1    R0,R2       ;
0000103C   4124   47    ROTCL   R1          ;
0000103E   3204   48    DIV1    R0,R2       ;
00001040   4124   49    ROTCL   R1          ;
00001042   3204   50    DIV1    R0,R2       ;
00001044   4124   51    ROTCL   R1          ;
00001046   3204   52    DIV1    R0,R2       ;
00001048   4124   53    ROTCL   R1          ;
0000104A   3204   54    DIV1    R0,R2       ;
0000104C   4124   55    ROTCL   R1          ;
0000104E   3204   56    DIV1    R0,R2       ;
00001050   4124   57    ROTCL   R1          ;
00001052   3204   58    DIV1    R0,R2       ;
                  59                        ;
00001054   4124   60    ROTCL   R1          ;
00001056   3204   61    DIV1    R0,R2       ;
00001058   4124   62    ROTCL   R1          ;
0000105A   3204   63    DIV1    R0,R2       ;
0000105C   4124   64    ROTCL   R1          ;
0000105E   3204   65    DIV1    R0,R2       ;
00001060   4124   66    ROTCL   R1          ;
00001062   3204   67    DIV1    R0,R2       ;
00001064   4124   68    ROTCL   R1          ;
```

```
69  00001066   3204   69    DIV1   R0,R2       ;
70  00001068   4124   70    ROTCL  R1          ;
71  0000106A   3204   71    DIV1   R0,R2       ;
72  0000106C   4124   72    ROTCL  R1          ;
73  0000106E   3204   73    DIV1   R0,R2       ;
74  00001070   4124   74    ROTCL  R1          ;
75  00001072   3204   75    DIV1   R0,R2       ;
76                    76                       ;
77  00001074   4124   77    ROTCL  R1          ;
78  00001076   3204   78    DIV1   R0,R2       ;
79  00001078   4124   79    ROTCL  R1          ;
80  0000107A   3204   80    DIV1   R0,R2       ;
81  0000107C   4124   81    ROTCL  R1          ;
82  0000107E   3204   82    DIV1   R0,R2       ;
83  00001080   4124   83    ROTCL  R1          ;
84  00001082   3204   84    DIV1   R0,R2       ;
85  00001084   4124   85    ROTCL  R1          ;
86  00001086   3204   86    DIV1   R0,R2       ;
87  00001088   4124   87    ROTCL  R1          ;
88  0000108A   3204   88    DIV1   R0,R2       ;
89  0000108C   4124   89    ROTCL  R1          ;
90  0000108E   3204   90    DIV1   R0,R2       ;
91  00001090   4124   91    ROTCL  R1          ;
92  00001092   3204   92    DIV1   R0,R2       ;
93                    93                       ;
94  00001094   2237   94    DIV0S  R3,R2       ;R2: keep sign
95  00001096   0329   95    MOVT   R3          ;
96  00001098   234A   96    XOR    R4,R3       ;(R4 or R3) = 1 ? →
                                                oversub or overadd
97  0000109A   4325   97    ROTCRR3            ;
98  0000109C   8B02   98    BF    DIVS32R1    ;Tbit = 0 ?
99  0000109E   2207   99    DIV0S R0,R2       ;Clear oversub or overadd
100 000010A0   4221   100   SHAR  R2          ;
101 000010A2   3204   101   DIV1  R0,R2       ;
102 000010A4          102  DIVS32R1           ;
103 000010A4   324C   103    ADD  R4,R2       ;
104 000010A6   0008   104    CLRT             ;T bit ← No error
```

```
105 000010A8   64F6   105    MOV.L  @R15+,R4 ;Return register
106 000010AA   000B   106    RTS             ;
107 000010AC   63F6   107    MOV.L  @R15+,R3 ;
108 000010AE          108  DIVS32R2          ;
109 000010AE   0018   109    SETT            ;T bit ← Error
110 000010B0   64F6   110    MOV.L  @R15+,R4 ;Return register
111 000010B2   000B   111    RTS             ;
112 000010B4   63F6   112    MOV.L  @R15+,R3 ;
113                   113    .END
*****TOTAL   ERRORS  0
*****TOTAL   WARNINGS  0
```

3.15 AFIN: Affine Transform

- Instruction: MAC.W
- Addressing mode: Post-increment register indirect
- Function: Performs matrix operations of the affine transform. The data table shown in figure 3.64 is used.

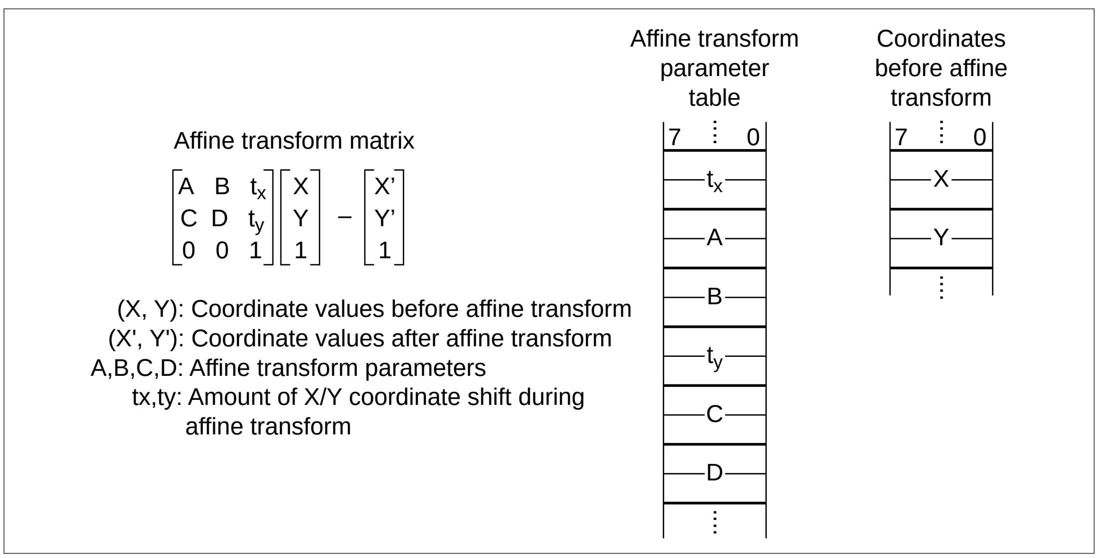


Figure 3.67 Affine Transform Matrix Data

Table 3.35 AFIN Arguments

	Contents	Storage Location	Data Length (Bytes)
Input	Start address of affine transform parameter table	R0	4
	Coordinates storage address before affine transform	R1	4
	Coordinates storage address after affine transform	R2	4

(Pre-execution) → (Post-execution)

R0	Start address of affine transform parameter table → Undefined
R1	Storage address of coordinates before affine transform → Undefined
R2	Storage address of coordinates after affine transform → Undefined
R3	Work
R4	
R5	
R6	
R7	
R8	
R9	
R10	
R11	
R12	
R13	
R14	
R15	(SP)

T bit: no change

Figure 3.68 AFIN Internal Register Change and Flag Change

Table 3.36 AFIN Programming Specifications

Item	Value/State
Program memory (bytes)	34
Data memory (bytes)	0
Stack (bytes)	4
Number of states	22
Reentrant	Enabled
Relocation	
Intermediate interrupt	

3.15.1 AFIN Arguments

- R0: Holds the affine transform parameter table start address as the input argument.
- R1: Holds the coordinates storage address before affine transform as the input argument.
- R2: Holds the coordinates storage address after affine transform as the input argument.

Figure 3.69 shows an AFIN execution example. In memory, affine transform parameters are allocated in advance from address H'10000000 through t_x, A, B, t_y, C, and D, in that order. Coordinates before affine transform are allocated from address H'10001000 in the order X, Y. The affine transform parameter table start address, coordinates storage address before affine transform, and coordinates storage address after affine transform are transferred to the software AFIN as the input argument. Transform matrix operations are performed in AFIN software and coordinates after affine transform are allocated as specified in the input argument from address H'10001100 in the order X', Y'.

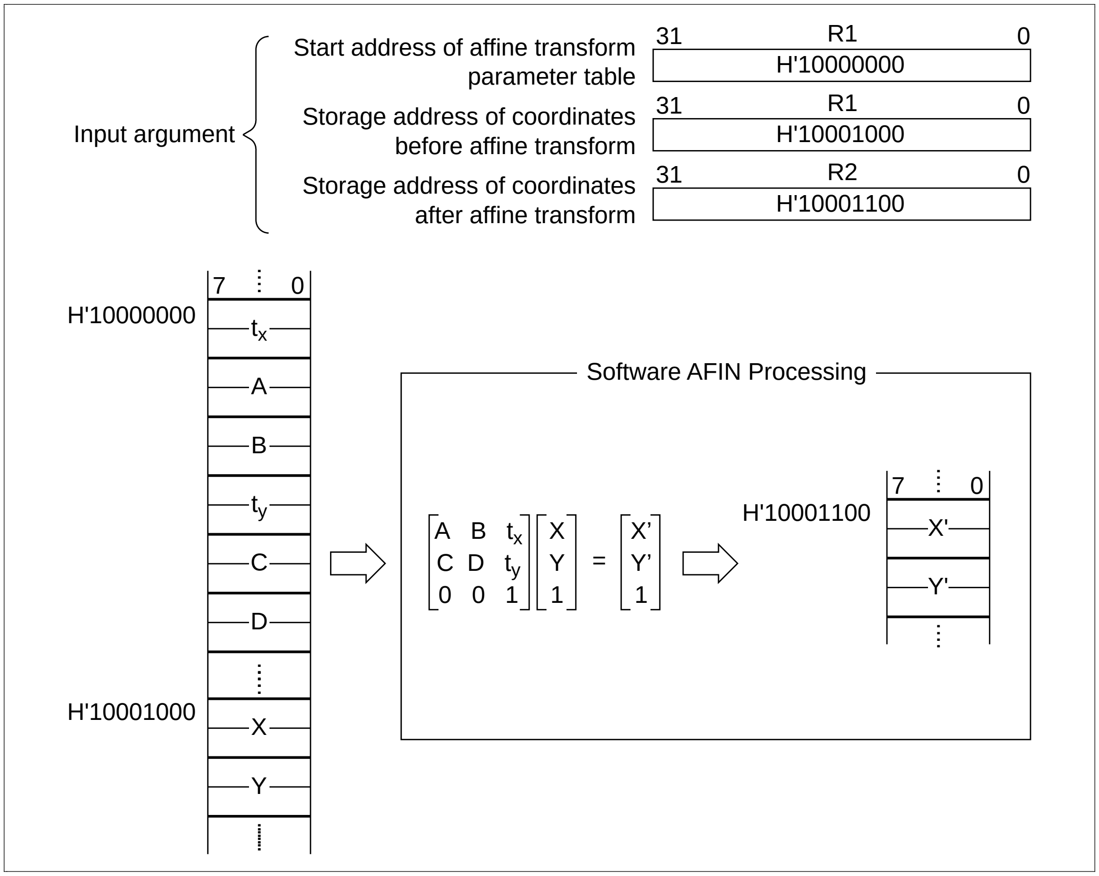


Figure 3.69 AFIN Execution Example

3.15.2 Precautions for AFIN Use

Be sure to allocate affine transform parameters and coordinates before affine transform as shown in figure 3.66.

3.15.3 AFIN RAM Use

RAM is not used with AFIN.

3.15.4 Example of AFIN Use

Affine transform parameter table start address, coordinates storage address before affine transform and coordinates storage address after affine transform are set in the input argument, then subroutine call AFIN software.

```
        MOV.L     DATA1, R0        Sets affine transform parameter table start address
                                   in the input argument
        MOV.L     DATA2, R1        Sets the coordinates storage address before affine
                                   transform in the input argument
        BSR       AFIN             Subroutine call of AFIN
        MOV.L     DATA3, R2        Sets the coordinates storage address after affine
                                   transform in the input argument
                  ↓
        .align    4
DATA1   .data.l   H'10000000
DATA2   .data.l   H'10001000
DATA3   .data.l   H'10001100
```

3.15.5 AFIN Operation

Expanding the affine transform matrix gives the following kind of formula:

$$X' = AX + BY + t_x$$

$$Y' = CX + DY + t_y$$

$AX + BY + t_x$ and $CX + DY + t_y$ are determined using the multiply and accumulate instruction (MAC) (figure 3.70).

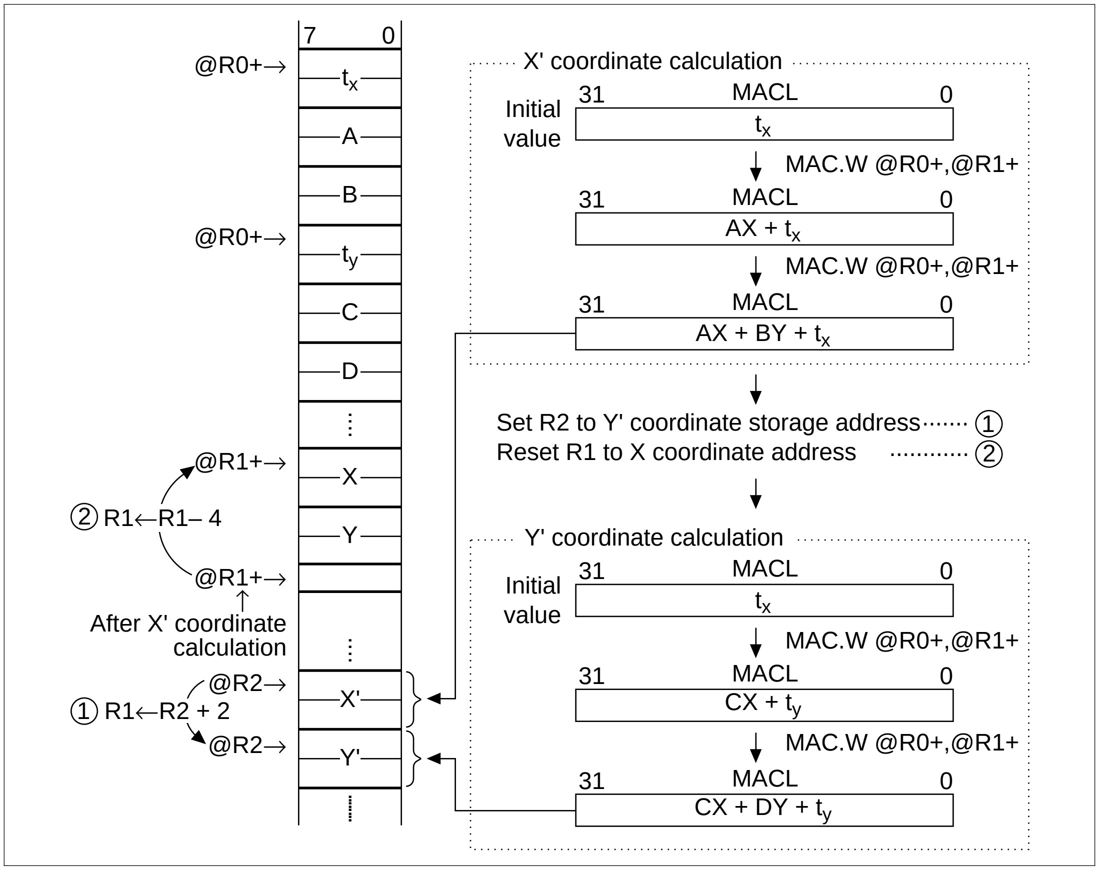


Figure 3.70 Calculation of X' and Y' Coordinates

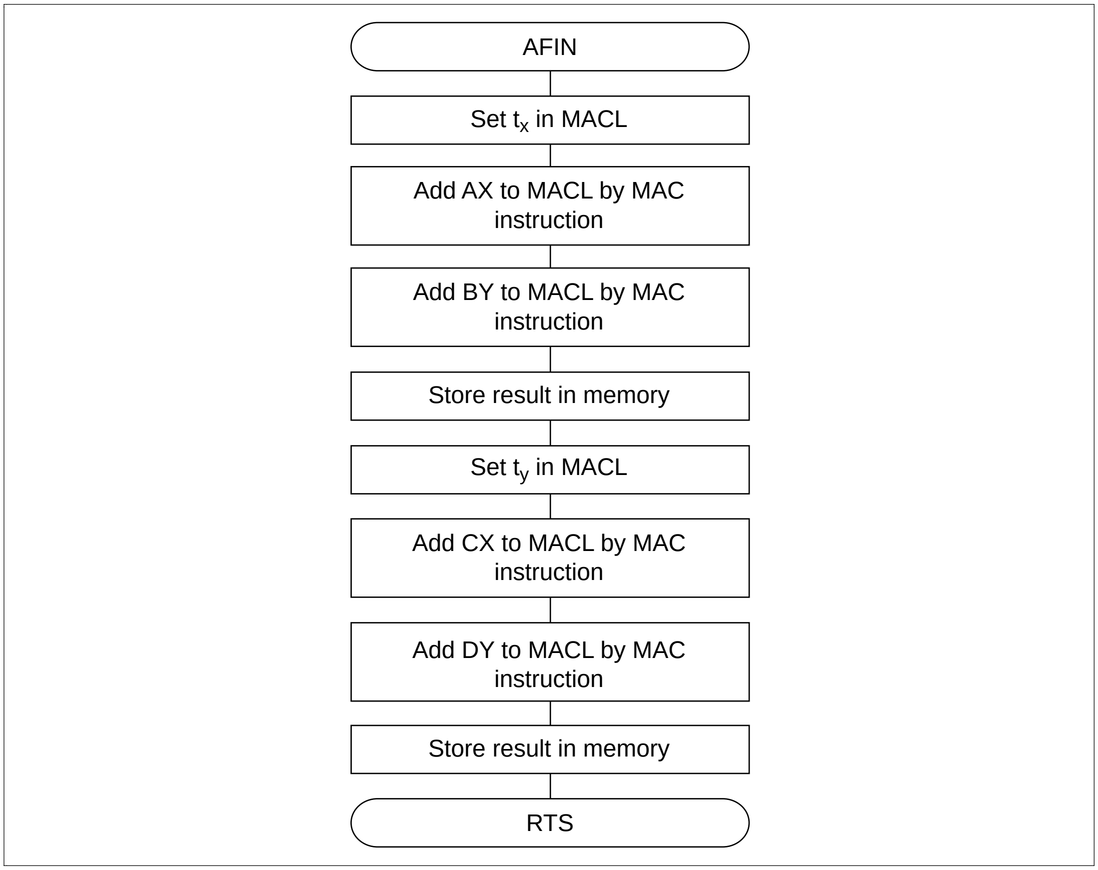


Figure 3.71 AFIN Flowchart

3.15.6 AFIN Program Listing

```
NAME:  AFIN  CONVERSION  (AFIN)
ENTRY: R0(TOP  ADDRESS  OF PARAMETER)
       R1(STORED  ADDRESS  OF BEFORE  AFIN CONVERSION)
       R2(STOR  ADDRESS  OF AFTER AFIN CONVERSION)
 1                     1                     ;
 2                     2                     ;
 3                     3                     ;
 4                     4                     ;
 5                     5                     ;
 6                     6                     ;
 7                     7                     ;
 8                     8                     ;
 9                     9                     ;
10                    10                     ;
11                    11                     ;
12  00001000          12       .SECTION A,CODE,LOCATE=H'1000
13  00001000          13  AFIN .EQU  $        ;Entry point
14  00001000   2F36   14    MOV.L  R3,@-R15   ;Escape register
15  00001002   6305   15    MOV.W  @R0+,R3    ;tx → MACL
16  00001004   431A   16    LDS    R3,MACL    ;
17  00001006   410F   17    MAC.W  @R0+,@R1+  ;A * X + MACL → MACL
                                               (=AX + Tx)
18  00001008   410F   18    MAC.W  @R0+,@R1+  ;B * Y + MACL → MACL
                                               (=AX + BY + Tx)
19  0000100A   031A   19    STS    MACL,R3    ;MACL → X'
20  0000100C   2231   20    MOV.W  R3,@R2     ;
21  0000100E   7202   21    ADD    #2,R2      ;
22  00001010   6305   22    MOV.W  @R0+,R3    ;Ty → MACL
23  00001012   431A   23    LDS    R3,MACL    ;
24  00001014   71FC   24    ADD    #-4,R1     ;
25  00001016   410F   25    MAC.W  @R0+,@R1+  ;C * X + MACL → MACL
                                               (=CX + Ty)
26  00001018   410F   26    MAC.W  @R0+,@R1+  ;D * Y + MACL → MACL
                                               (=CX + DY + Ty)
27  0000101A   031A   27    STS    MACL,R3    ;MACL → Y'
28  0000101C   2231   28    MOV.W  R3,@R2     ;
```

```
29  0000101E   000B   29    RTS               ;
30  00001020   63F6   30    MOV.L  @R15+,R3    ;Return register
31                    31    .END
*****TOTAL  ERRORS 0
*****TOTAL  WARNINGS 0
```

Appendix A Instruction Set

Tables A1 through A6 describe instructions, instruction codes, operations, and execution cycles in order of classification by the following symbols:

Instruction: Represented by mnemonics:

- OP: Operation code
- Sz: Size
- SRC: Source
- DEST: Destination
- Rm: Source register
- Rn: Destination register
- imm: Immediate data
- disp: Displacement

Instruction Code: Represented in MSB × LSB order:

- mmmm: Source register
- nnnn: Destination register
- 0000: R0
- 0001: R1

↓

- 1111: R15
- iiii: Immediate data
- dddd: Displacement

Operation: Represents outline of operation:

- →, ←: Direction of transfer
- (xx): Memory operand
- M/Q/T: Flag bits in SR
- &: Logical AND every bit
- |: Logical OR every bit
- ^: Exclusive OR every bit
- ~: Logical NOT
- <<n: Shift n bits left
- >>n: Shift n bits right

Execution Cycles: The values shown are those during a “no wait” state. The execution cycles shown are minimum values. In practice the number of execution cycles increases when there is contention between instruction fetch and data access or when the destination register of the load instruction (memory → register) and the register used by the instruction immediately after are the same.

T Bit: Gives the T bit value after instruction execution ("—" indicates "no change").

Note : Scaling (×1, ×2, × 4) is performed according to the instruction operand size. See “SH7000/SH7600 Series Programming Manual” for details.

Table A.1 Data Transfer Instructions

Instruction		Instruction Code	Operation	Execution Cycles	T Bit
MOV	#imm, Rn	1110nnnniiiiiiii	imm → sign extension → Rn	1	—
MOV.W	@(disp, PC), Rn	1001nnnndddddddd	(disp × 2 + PC) → sign extension → Rn	1	—
MOV.L	@(disp, PC)Rn	1101nnnndddddddd	(disp × 4 + PC) → Rn	1	—
MOV	Rm, Rn	0110nnnnmmmm0011	Rm → Rn	1	—
MOV.B	Rm, @Rn	0010nnnnmmmm0000	Rm → (Rn)	1	—
MOV.W	Rm, @Rn	0010nnnnmmmm0001	Rm → (Rn)	1	—
MOV.L	Rm, @Rn	0010nnnnmmmm0010	Rm → (Rn)	1	—
MOV.B	@Rm, Rn	0110nnnnmmmm0000	(Rm) → sign extension → Rn	1	—
MOV.W	@Rm, Rn	0110nnnnmmmm0001	(Rm) → sign extension → Rn	1	—
MOV.L	@Rm, Rn	0110nnnnmmmm0010	(Rm) → Rn	1	—
MOV.B	Rm, @–Rn	0010nnnnmmmm0100	Rn–1 → Rn, Rm → (Rn)	1	—
MOV.W	Rm, @–Rn	0010nnnnmmmm0101	Rn–2 → Rn, Rm → (Rn)	1	—
MOV.L	Rm, @–Rn	0010nnnnmmmm0110	Rn–4 → Rn, Rm → (Rn)	1	—
MOV.B	@Rm + , Rn	0110nnnnmmmm0100	(Rm) → sign extension → Rn, Rm + 1 → Rm	1	—
MOV.W	@Rm + , Rn	0110nnnnmmmm0101	(Rm) → sign extension → Rn, Rm + 2 → Rm	1	—
MOV.L	@Rm + , Rn	0110nnnnmmmm0110	(Rm) → Rn, Rm + 4 → Rm	1	—
MOV.B	R0, @(disp, Rn)	10000000nnnndddd	R0 → (disp + Rn)	1	—
MOV.W	R0, @(disp, Rn)	10000001nnnndddd	R0 → (disp × 2 + Rn)	1	—
MOV.L	Rm, @(disp, Rn)	0001nnnnmmmmdddd	Rm → (disp × 4 + Rn)	1	—
MOV.B	@(disp, Rm), R0	10000100mmmmdddd	(disp + Rm) → sign extension → R0	1	—

Table A.1 Data Transfer Instructions (cont)

Instruction		Instruction Code	Operation	Execution Cycles	T Bit
MOV.W	@(disp, Rm), R0	10000101mmmmdddd	(disp × 2 + Rm) → sign extension → R0	1	—
MOV.L	@(disp, Rm), Rn	0101nnnnmmmmdddd	(disp × 4 + Rm) → Rn	1	—
MOV.B	Rm, @(R0, Rn)	0000nnnnmmmm0100	Rm → (R0 + Rn)	1	—
MOV.W	Rm, @(R0, Rn)	0000nnnnmmmm0101	Rm → (R0 + Rn)	1	—
MOV.L	Rm, @(R0, Rn)	0000nnnnmmmm0110	Rm → (R0 + Rn)	1	—
MOV.B	@(R0, Rm), Rn	0000nnnnmmmm1100	(R0 + Rm) → sign extension → Rn	1	—
MOV.W	@(R0, Rm), Rn	0000nnnnmmmm1101	(R0 + Rm) → sign extension → Rn	1	—
MOV.L	@(R0, Rm), Rn	0000nnnnmmmm1110	(R0 + Rm) → Rn	1	—
MOV.B	R0, @(disp, GBR)	11000000dddddddd	R0 → (disp + GBR)	1	—
MOV, W	R0, @(disp, GBR)	11000001dddddddd	R0 → (disp∞2 + GBR)	1	—
MOV.L	R0, @(disp, GBR)	11000010dddddddd	R0 → (disp∞4 + GBR)	1	—
MOV, B	@(disp, GBR), R0	11000100dddddddd	(disp + GBR) → sign extension → R0	1	—
MOV.W	@(disp, GBR), R0	11000101dddddddd	(disp × 2 + GBR) → sign extension → R0	1	—
MOV, L	@(disp, GBR), R0	11000110dddddddd	(disp × 4 + GBR) → R0	1	—
MOVA	@(disp, PC), R0	11000111dddddddd	disp × 4 + PC → R0	1	—
MOVT	Rn	0000nnnn00101001	T → Rn	1	—
SWAP.B	Rm, Rn	0110nnnnmmmm1000	Rm → exchange upper/lower for the lower 2 bytes (swap lower 16 bits) → Rn	1	—
SWAP.W	Rm, Rn	0110nnnnmmmm1001	Rm → exchange upper/lower words → Rn	1	—
XTRCT	Rm, Rn	0010nnnnmmmm1101	Rm: Rn's middle 32 bits → Rn	1	—

Table A.2 Arithmetic Operation Instructions

Instruction		Instruction Code	Operation	Execution Cycles	T Bit
ADD	Rm, Rn	0011nnnnmmmm1100	Rn + Rm → Rn	1	—
ADD	#imm, Rn	0111nnnniiiiiiii	Rn + imm → Rn	1	—
ADDC	Rm, Rn	0011nnnnmmmm1110	Rn + Rm + T→ Rn, carry → T	1	Carry
ADDV	Rm, Rn	0011nnnnmmmm1111	Rn + Rm → Rn, overflow → T	1	Over-flow
CMP/EQ	#imm, R0	10001000iiiiiiii	1 → T when R0 = imm	1	Comparison result
CMP/EQ	Rm, Rn	0011nnnnmmmm0000	1 → T when Rn + Rm	1	Comparison result
CMP/HS	Rm, Rn	0011nnnnmmmm0010	1 → T when Rn ≥ Rm unsigned	1	Comparison result
CMP/GE	Rm, Rn	0011nnnnmmmm0011	1 → T when Rn ≥ Rm signed	1	Comparison result
CMP/HI	Rm, Rn	0011nnnnmmmm0110	1 → T when Rn > Rm unsigned	1	Comparison result
CMP/GT	Rm, Rn	0011nnnnmmmm0111	1 → T when Rn > Rm signed	1	Comparison result
CMP/PZ	Rn	0100nnnn00010001	1 → T when Rn ≥ 0	1	Comparison result
CMP/PL	Rn	0100nnnn00010101	1 → T when Rn > 0	1	Comparison result
CMP/STR	Rm, Rn	0010nnnnmmmm1100	1 → T when both bytes are equal	1	Comparison result
DIV1	Rm, Rn	0011nnnnmmmm0100	1-step division (Rn ÷ Rm)	1	Calculation result
DIV0S	Rm, Rn	0010nnnnmmmm0111	Rn's MSB → Q, Rm's MSB → M, M^Q → T	1	Calculation result
DIV0U		0000000000011001	0 → M/Q/T	1	0

Table A.2 Arithmetic Operation Instructions(cont)

Instruction		Instruction Code	Operation	Execution Cycles	T Bit
EXTS . B	Rm, Rn	0110nnnnmmmm1110	Sign extension from Rm's byte → Rn	1	—
EXTS.W	Rm, Rn	0110nnnnmmmm1111	Sign extension for Rm's word → Rn	1	—
EXTU.B	Rm, Rn	0110nnnnmmmm1100	Zero extension from Rm's byte → Rn	1	—
EXTU.W	Rm, Rn	0110nnnnmmmm1101	Zero extension from Rm's word → Rn	1	—
MAC.W	@Rm+, @Rn+	0100nnnnmmmm1111	(Rn) × (Rm) + MAC as signed → MAC	3/(2) *1	—
MULS	Rm, Rn	0010nnnnmmmm1111	Rn × Rm as signed → MAC	1(–3) *2	—
MULU	Rm, Rn	0010nnnnmmmm1110	Rn × Rm as unsigned → MAC	1(–3) *2	—
NEG	Rm, Rn	0110nnnnmmmm1011	0 – Rm→Rn	1	—
NEGC	Rm, Rn	0110nnnnmmmm1010	0 – Rm – T → Rn, borrow → T	1	Borrow
SUB	Rm, Rn	0011nnnnmmmm1000	Rn – Rm → Rn	1	—
SUBC	Rm, Rn	0011nnnnmmmm1010	Rn – Rm – T → Rn, borrow → T	1	Borrow
SUBV	Rm Rn	0011nnnnmmmm1011	Rn – Rm → Rn, underflow → T	1	Underflow

Notes: 1. Normally 3 clock cycles, but 2 clock cycles is also possible due to the following instruction.

2. Normally, the minimum number of clock cycles is 1, but this becomes 3 clock cycles when the operation result is read from the MAC register immediately following a MUL instruction.

Table A.3 Logic Operation Instructions

Instruction		Instruction Code	Operation	Execu-tion Cycles	T Bit
AND	Rm,Rn	0010nnnnmmmm1001	Rn & Rm → Rn	1	—
AND	#imm,R0	11001001iiiiiiii	R0 & imm → R0	1	—
AND.B	#imm, @(R0,GBR)	11001101iiiiiiii	(R0 + GBR) 7 imm → (R0+GBR)	3	—
NOT	Rm,Rn	0110nnnnmmmm0111	– Rm → Rn	1	—
OR	Rm,Rn	0010nnnnmmmm1011	Rn | Rm → Rn	1	—
OR	#imm, R0	11001011iiiiiiii	R0 | imm → R0	1	—
OR.B	#imm, @(R0,GBR)	11001111iiiiiiii	(R0 + GBR) | imm → (R0+GBR)	3	—
TAS.B	@Rn	0100nnnn00011011	1 → T when (Rn) is 0, 1 → MSB of (Rn)	4	Test result
TST	Rm,Rn	0010nnnnmmmm1000	Rn & Rm, 1 → T when result is 0	1	Test result
TST	#imm.R0	11001000iiiiiiii	R0 & imm, 1 → T when result is 0	1	Test result
TST.B	#imm, @(R0,GBR)	11001100iiiiiiii	(R0 + GBR) & imm, 1 → T when result is 0	3	Test result
XOR	Rm,Rn	0010nnnnmmmm1010	Rn ^ Rm → Rn	1	—
XOR	#imm,R0	11001010iiiiiiii	R0 ^ imm → R0	1	—
XOR.B	#imm, @(R0,GBR)	11001110iiiiiiii	(R0 + GBR) ^ imm → (R0 + GBR)	3	—

Table A.4 Shift Instructions

Instruction		Instruction Code	Operation	Execution Cycles	T Bit
ROTL	Rn	0100nnnn00000100	T ← Rn ← MSB	1	MSB
ROTR	Rn	0100nnnn00000101	LSB → Rn → T	1	LSB
ROTCL	Rn	0100nnnn00100100	T ← Rn ← T	1	MSB
ROTCR	Rn	0100nnnn00100101	T → Rn → T	1	LSB
SHAL	Rn	0100nnnn00100000	T ← Rn ← 0	1	MSB
SHAR	Rn	0100nnnn00100001	MSB → RN → T	1	LSB
SHIL	Rn	0100nnnn00000000	T ← Rn ← 0	1	MSB
SHLR	Rn	0100nnnn00000001	0 → Rn → T	1	LSB
SHLL2	Rn	0100nnnn00001000	Rn << 2 → Rn	1	—
SHLR2	Rn	0100nnnn00001001	Rn >> 2 → Rn	1	—
SHLL8	Rn	0100nnnn00011000	Rn << 8 → Rn	1	—
SHLR8	Rn	0100nnnn00011001	Rn >> 8 → Rn	1	—
SHLL16	Rn	0100nnnn00101000	Rn << 16 → Rn	1	—
SHLR16	Rn	0100nnnn00101001	Rn >> 16 → Rn	1	—

Table A.5 Branch Instructions

Instruction		Instruction Code	Operation	Execution Cycles	T Bit
BF	label	10001011dddddddd	disp × 2 + PC → PC when T = 0, nop when T = 1	3/1*	—
BT	label	10001001dddddddd	disp × 2 + PC → PC when T = 1, nop when T = 0	3/1*	—
BRA	label	1010dddddddddddd	delayed branch, disp × 2 + PC → PC	2	—
BSR	label	1011dddddddddddd	delayed branch, PC → PR, disp × 2 + PC → PC	2	—
JMP	@Rm	0100mmmm00101011	delayed branch, Rm → PC	2	—
JSR	@Rm	0100mmmm00001011	delayed branch, PC → PR, Rm → PC	2	—
RTS		0000000000001011	delayed branch, PR → PC	2	—

Note: 3 cycles when branching, 1 cycle when there is no branching.

Table A.6 System Control Instructions

Instruction		Instruction Code	Operation	Execu-tion Cycles	T Bit
CLRT		0000000000001000	0 → T	1	0
CLRMAC		0000000000101000	0 → MACH,MACL	1	—
LDC	Rm, SR	0100mmmm00001110	Rm → SR	1	LSB
LDC	Rm, GBR	0100mmmm00011110	Rm → GBR	1	—
LDC	Rm,VBR	0100mmmm00101110	Rm→VBR	1	—
LDC.L	@Rm+, SR	0100mmmm00000111	(Rm) → SR, Rm + 4 → Rm	3	LSB
LDC.L	@Rm+, GBR	0100mmmm00010111	(Rm) → GBR, Rm + 4 → Rm	3	—
LDC.L	@Rm+, VBR	0100mmmm00100111	(Rm) → VBR, Rm + 4 → Rm	3	—
LDS	Rm, MACH	0100mmmm00001010	Rm → MACH	1	—
LDS	Rm,MACL	0100mmmm00011010	Rm → MACL	1	—
L0S	Rm, PR	0100mmmm00101010	Rm → PR	1	—
LDS.L	@Rm+, MACH	0100mmmm00000110	(Rm) → MACH, Rm + 4 → Rm	1	—
LDS.L	@Rm+, MACL	0100mmmm00010110	(Rm) → MACL, Rm + 4 → Rm	1	—
LDS.L	@Rm+, PR	0I00mmmm00100110	(Rm) → PR, Rm + 4 → Rm	1	—
NOP		0000000000001001	Non-operation	1	—
RTE		0000000000101011	Delayed branch, stack area → PC/SR	4	LSB
SETT		0000000000011000	1 → T	1	1
SLEEP		0000000000011011	Sleep	3*	—
STC	SR,Rn	0000nnnn00000010	SR → Rn	1	—
STC	GBR, Rn	0000nnnn00010010	GBR → Rn	1	—
STC	VBR, Rn	0000nnnn00100010	VBR → Rn	1	—
STC.L	SR, @–Rn	0100nnnn00000011	Rn-4 → Rn, SR → Rn)	2	—
STC.L	GBR, @–Rn	0100nnnn00010011	Rn-4 → Rn, GBR → Rn)	2	—
STC.L	VBR, @–Rn	0100nnnn00100011	Rn-4 → Rn, VBR → (Rn)	2	—

Table A.6 System Control Instructions (cont)

Instruction		Instruction Code	Operation	Execution Cycles	T Bit
STS	MACH, Rn	0000nnnn00001010	MACH → Rn	1	—
STS	MACL, Rn	0000nnnn00011010	MACL → Rn	1	—
STS	PR, Rn	0000nnnn00101010	PReRn	1	—
STS.L	MACH, @–Rn	0100nnnn00000010	Rn-4 → Rn, MACH → (Rn)	1	—
STS.L	MACL, @ –Rn	0100nnnn00010010	Rn-4 → Rn, MACL → (Rn)	1	—
STS.L	PR, @–Rn	0100nnnn00100010	Rn-4 → Rn, PR → (Rn)	1	—
TRAPA	#imm	11000011iiiiiiii	PC/SR → stack area, (imm × 4 + VBR) → PC	8	—

Appendix B Assembler Control Instruction Functions

B.1 .SECTION

The .SECTION instruction gives the section declaration.

B.1.1 Format

The .SECTION format is:

.SECTION Δ *section name* [, *section attributes* [, { LOCATE = *start address* / ALIGN = *boundary adjustment number* }]]

B.1.2 Elements of the Statement

- Label: Cannot be specified.
- Operation: Uses the mnemonic .SECTION.
- Operand:
  - — First operand: section name. The section name is a kind of symbol. See How to Apply Section Names, Language Manual 1.3.2, How to Apply Symbols.
  - — Second operand: section attribute. The specification contents determine the type of section application. The CODE section is selected if the specification is omitted (table B.1).

Table B.1 Section Application Selection

Specification	Type of Section
CODE	Code section
DATA	Data section
STACK	Stack section
COMMON	Common section
DUMMY	Dummy section

— Third operand: start address or boundary adjustment number. Whether the section type becomes absolute address section or relative address section is determined by the specification contents (table B.2). ALIGN = relative address section 4 is selected when the specification is omitted.

Table B.2 Third Operand Type

Specification	Type of Section
LOCATE = start address	Absolute address section
ALIGN = boundary adjustment number	Relative address section

B.1.3 Explanation

- .SECTION is the assembler control instruction which declares the section. A section is a unit of linkage processing, representing one section of a program.

Figure B.1 shows a simple example of section declaration. In this example, it is assumed that . SECTION should not appear in source statement groups 1–3.

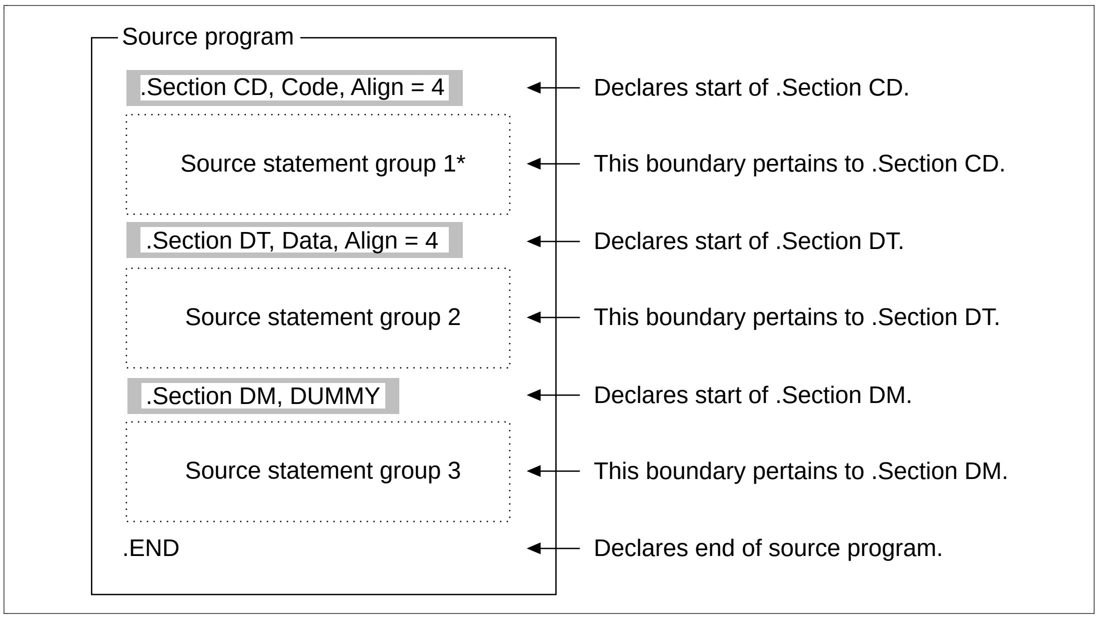


Figure B.1 Example of Section Declaration

- The already declared section can be redeclared in the same file and reopened. The example in figure B.2 illustrates section reopening. In this example, it is assumed that .SECTION should not appear in source statement groups 1–3.

Omit the second and third operands when reopening the section. (The specification at the time that section was reopened is effective as it is.)

```
            .ALIGN     4                    This boundary pertains to default
            .DATA.L    H'11111111           .Section P, which is a code section and
              ~                             a relative address section of ALIGN = 4.
.Section    CD, Code, Align = 4
            MOV        R0, R1               This boundary pertains to .Section CD,
            MOV        R0, R2               which is a code section and  a relative
              ~                             address section of ALIGN = 4.
.Section    DT, DATA, LOCATE = H'00001000
X1:         .DATA.L    H'22222222           This boundary pertains to .Section DT,
            .DATA.L    H'33333333           which is a data section and an absolute
              ~                             address section of start address = H'00001000.
            .END
```

Figure B.2 Example of Section Reopening

- When opening an absolute address section, specify "LOCATE = start address" for the third operand. The start address is the absolute address where that section begins.

  Specify the start address as follows:

  — Specify the absolute value.

  — Specify without use of the forward access symbol.

  Allowable values for start address are H'00000000 – H'FFFFFFFF.

  (decimal notation: –2,147,483,648 to 4,294,967,295).
- When starting the absolute address section, specify "ALIGN = boundary adjustment number" for the third operand. The linkage editor adjusts so that start of the section aligns with an absolute address which is a multiple of the boundary adjustment number.

  Specify the boundary adjustment number as follows:

  — Specify the absolute value.

  — Specify without use of the forward access symbol.
- Allowable values for the boundary adjustment number are powers of 2 (2^0, 2^1, 2^2, ...2^{31})
- The assembler prepares a default section in any of the following cases:

  — Execution instruction description given while section is undeclared

  — Data securing assembler control instruction description given while section is undeclared

  — .ALIGN assembler control instruction description given while section is undeclared

  — .ORG assembler control instruction description given while section is undeclared

  — Location counter accessed while section is undeclared

— Label-only line description given while section is undeclared

- The following kinds of section are the default sections:
  — Section name: P
  — Section type: CODE section
  Absolute address section (boundary adjustment number = 4)

B.2 .ALIGN

.ALIGN performs the correction of location counter value.

B.2.1 Format

The .ALIGN format is:

.ALIGN Δ *boundary adjustment number*

B.2.2 Elements of Statement

- Label: Cannot be specified.
- Operation: Uses mnemonic .ALIGN.
- Operand: Specifies the value you want to set as the boundary adjustment number (reference for adjustment of location counter value).

B.2.3 Explanation

- .ALIGN is the assembler control instruction that corrects the location counter value to a multiple of the boundary adjustment number.
  .ALIGN places the execution instruction or data at a specific boundary (address delimiter).
- The location counter value is set as follows:
  — Specify the absolute address.
  — Specify without use of the forward access symbol.
- Allowable values for the boundary adjustment number are powers of 2 (2^0, 2^1, 2^2,....., 2^{31}).
- When .ALIGN is specified in the CODE section, DATA section or DUMMY section, the assembler embeds the NOP instruction object code in memory, and corrects the location counter value. (This kind of object code does not appear in the assembly list.) Even bytes are filled with H'0 and odd bytes are filled with H'09.
  When .ALIGN is specified in the DUMMY or STACK section, the assembler simply corrects the location counter value, and does not embed object code in memory.

B.2.4 Coding Example

```
          ↓
.DATA.B     H'11
.DATA.B     H'22
.DATA.B     H'33
.ALIGN      2                  Corrects to a multiple of 2 of the location counter value
.DATA.W     H'4444
.ALIGN      4                  Corrects to a multiple of 4 of the location counter value
.DATA.L     H'55555555
          ↓
```

The byte size integer data H'11 is assumed to be originally located in a 4-byte boundary. The assembler embeds the object code and makes boundary adjustment as in figure B.3.

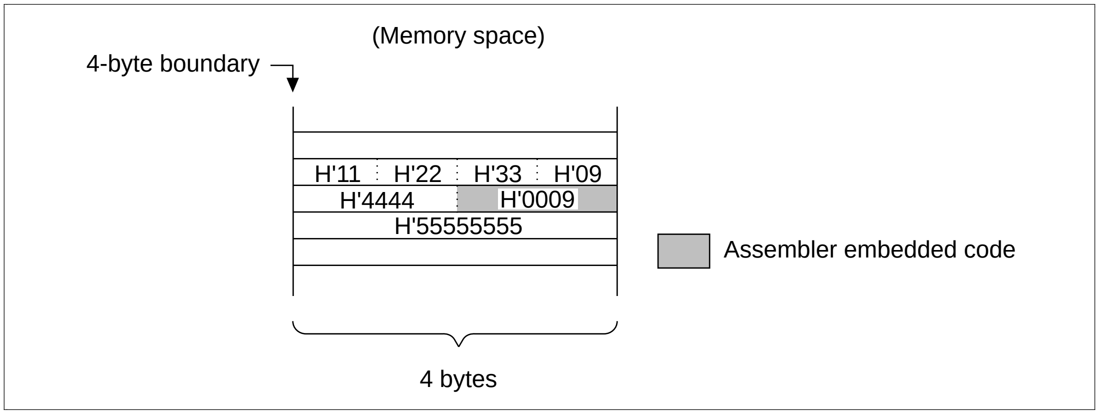


Figure B.3 .ALIGN Coding Example

B.3 .EQU

.EQU sets a value for symbols (not resettable).

B.3.1 Format

The .EQU format is:

Symbol [:] Δ .EQU Δ *symbol value*

B.3.2 Elements of the Statement

- Label: Specifies the symbol you want to set the value for.
- Operation: Uses the mnemonic .EQU.
- Operand: Specifies the value you want to set for the symbol.

B.3.3 Explanation

- .EQU is the assembler control instruction for setting a value for a symbol.

Symbols defined by .EQU cannot be redefined.

- Symbol values are set as follows:
  — Specify the absolute value or address.
  — Specify without use of the forward access symbol.
  Allowable symbol values are H'00000000 to H'FFFFFFFF. (In decimal notation, –2,147,483,648 to 4,294,967,295.)

B.3.4 Coding Example

```
              ↓
X1:    .EQU      10               X1 value becomes 10
X2:    .EQU      20               X2 value becomes 20
        CMP/EQ   #X1,R0           Same as CMP/EQ #10,R0
        BT       LABEL1
        CMP/EQ   #X2,R0           Same as CMP/EQ #20,R0
        BT       LABEL2
              ↓
```

B.4 .DATA

.DATA secures integer data.

B.4.1 Format

The .DATA format is:

[*Symbol* [:]] Δ.DATA [.operation size] Δ *integer data* [.*integer data*...]

B.4.2 Elements of Statement

- Label: When required, describes the marker symbol.
- Operation: Mnemonic is .DATA. The data length is determined by specification, as listed in table B.3. Data length becomes the longword when specification is omitted.

Table B.3 Data Length Determination

Specification	Data Length
B	Byte
W	Word (2 bytes)
L	Longword (4 bytes)

Operand: Describes the value you want to secure as data.

B.4.3 Explanation

- .DATA is the assembler control instruction that secures integer data in memory.
- An arbitrary value can be specified for integer data, including relative values and forward reference symbols.
- The range of integer data which can be specified varies with the operation size, as shown in table B.4.

Table B.4 Integer Data Range and Operation Size

Operation Size	Integer Data Range
B	H'00000000–H'000000FF H'FFFFFF80–H'FFFFFFFF (–128 to 255)
W	H'00000000–H'0000FFFF H'FFFF8000–H'FFFFFFFF (–32, 768 to 65, 535)
L	H'00000000–H'FFFFFFFF (–2, 147, 483, 648 to 4, 294, 967, 295)

Note: Figures in parentheses are decimal values.

B.4.4 Coding Example

```
            ↓
    .ALIGN4   4                        ;  (Corrects location counter value).
X:  .DATA.L   H'11111111               ;
    .DATA.W   H'2222, H'3333           ;  Secures integer data.
    .DATA.B   H'44, H'55               ;
            ↓
```

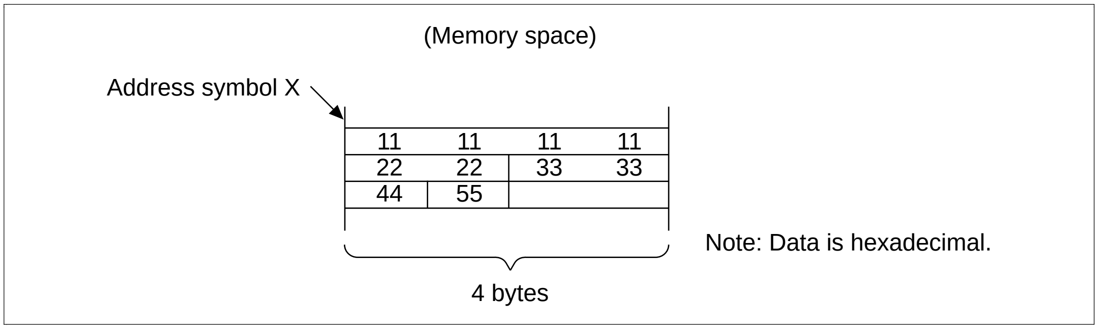


Figure B.4 .DATA Coding Example

B.5 .END

Declares source program end and execution start address.

B.5.1 Format

The .END format is:

.END [Δ *execution start address*]

B.5.2 Elements of Statement

- Label: Cannot be used.
- Operation: Uses mnemonic .END.
- Operand: Execution start address; specifies the address where you want to start simulation.

B.5.3 Explanation

- .END is the assembler control instruction which declares the end of a source program. The assembler stops assembly when an .END appears.

- When specifying the execution start address with .END, the simulator-debugger starts simulation from the specified address.
- The execution start address is specified with absolute value or address value.
- Specify the CODE section address in the execution start address.

B.5.4 Coding Example

```
    .SECTION CD, CODE, ALIGN = 4
START:
             ↓
    .END          START              ;  Declares source program end.
                                     ;  Simulator-debugger begins simulation from the
                                        address indicated by the START symbol.
```