Hitachi SuperH™ RISC engine

SH-3/SH-3E/SH3-DSP

Programming Manual

HITACHI

ADE-602-096B
Rev.3.0
9/25/00
Hitachi, Ltd

Hitachi semiconductor

Cautions

1. Hitachi neither warrants nor grants licenses of any rights of Hitachi’s or any third party’s patent, copyright, trademark, or other intellectual property rights for information contained in this document. Hitachi bears no responsibility for problems that may arise with third party’s rights, including intellectual property rights, in connection with use of the information contained in this document.
2. Products and product specifications may be subject to change without notice. Confirm that you have received the latest product standards or specifications before final design, purchase or use.
3. Hitachi makes every attempt to ensure that its products are of high quality and reliability. However, contact Hitachi’s sales office before using the product in an application that demands especially high quality and reliability or where its failure or malfunction may directly threaten human life or cause risk of bodily injury, such as aerospace, aeronautics, nuclear power, combustion control, transportation, traffic, safety equipment or medical equipment for life support.
4. Design your application so that the product is used within the ranges guaranteed by Hitachi particularly for maximum rating, operating supply voltage range, heat radiation characteristics, installation conditions and other characteristics. Hitachi bears no responsibility for failure or damage when used beyond the guaranteed ranges. Even within the guaranteed ranges, consider normally foreseeable failure rates or failure modes in semiconductor devices and employ systemic measures such as fail-safes, so that the equipment incorporating Hitachi product does not cause bodily injury, fire or other consequential damage due to operation of the Hitachi product.
5. This product is not designed to be radiation resistant.
6. No one is permitted to reproduce or duplicate, in any form, the whole or part of this document without written approval from Hitachi.
7. Contact Hitachi’s sales office for any questions regarding this document or Hitachi semiconductor products.

Introduction

The SH-3/SH-3E/SH3-DSP is a new generation of RISC microcomputers that integrate a RISC-type CPU and the peripheral functions required for system configuration onto a single chip to achieve high-performance operation. It can operate in a power-down state, which is an essential feature for portable equipment.

These CPUs have a RISC-type instruction set. Basic instructions can be executed in one clock cycle, improving instruction execution speed. In addition, the CPU has a 32-bit internal architecture for enhanced data-processing ability.

In addition, the SH-3E supports single-precision floating point calculations as well as entirely PCAPI compatible emulation of double-precision floating point calculations. The SH-3E instructions are a subset of the floating point calculations conforming to the IEEE754 standard.

This programming manual describes in detail the instructions for the SH-3/SH-3E/SH3-DSP and is intended as a reference on instruction operation and architecture. It also covers the pipeline operation, which is a feature of the SH-3/SH-3E/SH3-DSP. For information on the hardware, please refer to the hardware manual for the product in question.

Please contact a Hitachi sales office for information on development environment systems.

Organization of This Manual

Table 1 describes how this manual is organized. Table 2 show the relationships between the items listed and lists the sections within this manual that cover those items.

Table 1 Manual Organization

Category	Section Title	Contents
Introduction	1. Features	CPU features
Architecture (1)	2. Programming model	Types and structure of general registers, control registers and system registers
	3. Data Formats	Data formats for registers and memory
	4. Floating Point Processor Unit	FPU register configuration, FPU exceptions
	5. DSP Operations and Data Transfer	Fixed-point operations, integer operations, logic operations, multiplication, shift operations, overview of DSP operations such as saturation operations, repeat control
Introduction to instructions	6. Instruction Features	Instruction features, addressing modes, and instruction formats
	7. Instruction Sets	Summary of instructions by category and list in alphabetic order
Detailed information on instructions	8. Description of Each Instruction	Operation of each instruction in alphabetical order
Architecture (2)	9. Processing States	Power-down and other processing states
	10. Pipeline Operation	Pipeline operation

Table 2 Subjects and Corresponding Sections

Category	Topic	Section Title
Introduction and features	CPU features	1. Features
	Instruction features	6.1 RISC-Type Instruction Set
	Pipelines	10.1 Basic Configuration of Pipelines 10.2 Slot and Pipeline Flow
Architecture	Organization of registers	2. Programming model
	Data formats	3. Data Formats
	Floating point processor unit	4. Floating Point Processor Unit
	DSP	5. DSP Operations and Data Transfer
	Processing states, reset state, exception processing state, bus release state, program execution state, power-down state, sleep mode and standby mode	9. Processing States
	Pipeline operation	10. Pipeline Operation
Introduction to instructions	Instruction features	6. Instruction Features
	Addressing modes	6.2 Addressing Modes
	Instruction formats	6.3 Instruction Formats
List of instructions	Instruction sets	7.1 Instruction Set by Classification 7.2 Instruction Set in Alphabetical Order
Detailed information on instructions	Detailed information of instruction operation	8. Instruction Description 10.7 Instruction Pipelines
	Number of instruction execution states	10.3 Number of Instruction Execution Cycles

Contents

Section 1 Features

1.1 SH-3 CPU Features

The SH-3/SH-3E/SH3-DSP has RISC-type instruction sets. Basic instructions are executed in one clock cycle, which dramatically improves instruction execution speed. The CPU also has an internal
32-bit architecture for enhanced data processing ability. Table 1-1 lists the SH-3/SH-3E/SH3-DSP CPU features.

Table 1-1 SH-3/SH-3E/SH3-DSP CPU Features

Feature	Description
Architecture	• Hitachi original architecture • 32-bit internal data bus
General-register machine	• Sixteen 32-bit general registers (eight banked registers) • Five 32-bit control registers • Four 32-bit system registers (SH-3) • Six 32-bit system registers (SH-3E)
Instruction set	• Instruction length: 16-bit fixed length for improved code efficiency • Load-store architecture (basic arithmetic and logic operations are executed between registers) • Delayed branch system used for reduced pipeline disruption • Instruction set optimized for C language
Instruction execution time	• One instruction/cycle for basic instructions
Address space	• Architecture makes 4 Gbytes available
On-chip multiplier	• Multiplication operations (32 bits $\times$ 32 bits $\rightarrow$ 64 bits) executed in 2 to 5 cycles, and multiplication/accumulation operations (32 bits $\times$ 32 bits + 64 bits $\rightarrow$ 64 bits) executed in 2 to 5 cycles
Pipeline	• Five-stage pipeline
Processing states	• Reset state • Exception processing state • Program execution state • Power-down state • Bus release state
Power-down states	• Sleep mode • Standby mode • Hardware standby mode

Table 1-1 SH-3/SH-3E/SH3-DSP CPU Features (cont)

Feature	Description
FPU (SH-3E only)	• Single-precision floating point format • Subset of IEEE754 standard data types • Invalid calculation exception and divide-by-zero exception (in compliance with IEEE754 standard) • Rounding to zero (in compliance with IEEE754 standard) • General purpose register file, 16 32-bit floating point registers • Execution pitch for basic instructions: 1 cycle/latency or 2 cycles (FADD, FSUB, FMUL) • FMAC (floating point multiply accumulate) Execution pitch: 1 cycle/latency or 2 cycles • Support for FDIV and FSQRT • Support for FLDI0 and FLDI1 (load constant 0/1)

1.2 SH3-DSP Features

The SH3 CPU only has 16-bit instructions. The SH3-DSP basically has the same 16-bit instructions, but it also has additional 32-bit DSP instructions that it uses for parallel processing of DSP type instructions. The SH3 CPU use a standard Neumann architecture, but the SH3-DSP has the DSP data paths of the expanded Harvard architecture. Table 1-2 lists the added features of SH3-DSP.

Table 1-2 Features of SH3-DSP Series Microprocessor CPUs

Feature	Description
DSP unit	• Multiplier • Arithmetic logic unit (ALU) • Barrel shifter • DSP registers • MSB detection
Multiplier	• 16 bits $\times$ 16 bits $\rightarrow$ 32 bits (fixed decimal point) • 1 cycle multiplier
DSP registers	• Two 40-bit data registers • Six 32-bit data registers • Modulo register (MOD, 32 bits) added to control registers • Repeat counter (RC) added to status registers (SR) • Repeat start register (RS, 32-bit) and repeat end register (RE, 32-bit) added to control registers
DSP data bus	• Expanded Harvard architecture • Simultaneous access of two data bus and one instruction bus
On-chip memory	• 16-kbyte RAM
Parallel processing	• Maximum of four parallel processes (ALU operation, multiplication, and two loads or stores)
Address operator	• Two address operators • Address operations for accessing two memories
DSP data addressing modes	• Increment decrement and index • Increment decrement and index can have modulo addressing or not
Repeat control	• Zero-overhead repeat control (loop)
Instruction set	• 16 or 32 bits — 16 bits (for load or store only) — 32 bits (including for ALU operations and multiplication) • SuperH microprocessor instructions added for accessing DSP registers.
Pipeline	• Five-stage pipeline • Fifth stage is the DSP stage

Section 2 Programming Model

2.1 Organization of Registers

2.1.1 Privileged Mode and Banks

Processing Modes: The SH-3/SH-3E/SH3-DSP has two operating modes: user mode and privileged mode. The SH-3/SH-3E/SH3-DSP operates in user mode under normal conditions and enters privileged mode in response to an exception or interrupt. There are three types of registers: general, system, and control. All of these registers are 32 bits. Which registers can be accessed through software depends on the processing mode.

General-Purpose Registers: There are 16 general-purpose registers, numbered R0 through R15. General-purpose registers R0 to R7 are banked registers that are switched by the processor mode.

In privileged mode, the register bank (RB) bit in the status register (SR) defines which banked registers can be accessed as general-purpose registers and which cannot. Inaccessible registers can be accessed through the load control register (LDC) and store control register (STC) instructions.

When the RB bit is one (BANK1 is selected), BANK1 general-purpose registers R0_BANK1 through R7_BANK1 and non-banked general-purpose registers R8 through R15 (a total of 16 registers) can be accessed as general-purpose registers R0 through R15 and BANK0 general-purpose registers R0_BANK0 through R7_BANK0 (eight registers) are accessed by the LDC and STC instructions. When the RB bit is a zero (BANK0 is selected), BANK0 general-purpose registers R0_BANK0 through R7_BANK0 and nonbanked general-purpose registers R8 through R15 (16 registers) can be accessed as general-purpose registers R0 through R15 and BANK1 general-purpose registers R0_BANK1 through R7_BANK1 (eight registers) are accessed by the LDC and STC instructions.

In user mode, BANK0 general-purpose registers R0_BANK0 through R7_BANK0 and nonbanked general-purpose registers R8 through R15 can be accessed as general-purpose registers R0 through R15 (a total of 16 registers) and BANK1 general-purpose registers R0_BANK1 through R7_BANK1 (eight registers) cannot be accessed.

When the DSP extended features of the SH3-DSP are enabled, DSP instructions use X and Y data memory and L bus data memory (single data) addressing for eight of the 16 general-purpose registers.

To access X memory, R4 and R5 are used as the X address register [Ax] and R8 is used as the X index register [Ix]. To access the Y memory, R6 and R7 are used as the Y address register [Ay] and R9 is used as the Y index register [Iy]. To access single data using the L bus, R2, R3, R4, and R5 are used as the single data address register and R8 as the single data index register [Is].

DSP type instructions can simultaneously access X and Y memory. There are two groups of address pointers for specifying the X and Y data memory addresses.

Control Registers: The control registers include registers that can be accessed in either mode (the global base register (GBR) and status register (SR)) and registers that can only be accessed in privileged mode (the saved status register (SSR), saved program counter (SPC), and vector base register (VBR)). Some bits in the status register (for example, the RB bit) can only be accessed in privileged mode.

System Registers: There are four system registers that can be accessed in either processing mode:

Multiply and accumulate registers
  Multiply and accumulate high (MACH)
  Multiply and accumulate low (MACL)
Procedure register (PR)
Program counter (PC)

The register configurations are shown in figure 2-1 by processing mode. Switch between user and privileged modes using the processing operation mode bit in the status register.

Floating Point Registers and System Registers Used by the FPU (SH-3E Only): There are 16 floating point registers: FR0 to FR15. These are used as source and destination registers for single-precision floating point operations.

The system registers used by the FPU are the floating point communication register (FPUL) and the floating point status/control register (FPSCR). These are used for communication between the FPU and CPU as well as exception handling settings.

The register configurations for the different processing modes are illustrated in Figure 2-1 and Figure 2-2. Refer to 4. Floating Point Unit.

31	0
R0_BANK0[*1, *2]	
R1_BANK0[*2]	
R2_BANK0[*2]	
R3_BANK0[*2]	
R4_BANK0[*2]	
R5_BANK0[*2]	
R6_BANK0[*2]	
R7_BANK0[*2]	
R8	
R9	
R10	
R11	
R12	
R13	
R14	
R15	

31	0
FR0[*3]	
FR1[*3]	
FR2[*3]	
:	
FR15[*3]	

SR
FPSCR[*3]

GBR
MACH
MACL
FPUL[*3]
PR

PC

Notes 1. Register R0 is used as an index register in the indexed register-indirect addressing mode and indexed GBR-indirect addressing mode. There are some instructions for which only R0 can be used as the source or destination register.

2. R0 to R7 are banked registers, and BANK0 is used in the user mode.

3. These registers only exist on the SH-3E. They are used for floating point operations. Refer to 4. Floating Point Unit for details on FR0 to FR15, FPSCR, and FPUL.

Figure 2-1 User Mode Programming Model

(b) User Mode Programming Model (RB= 1) 31 … 0	(c) User Mode Programming Model (RB= 0) 31 … 0
R0_BANK1*1, *2	R0_BANK0*1, *2
R1_BANK1*2	R1_BANK0*2
R2_BANK1*2	R2_BANK0*2
R3_BANK1*2	R3_BANK0*2
R4_BANK1*2	R4_BANK0*2
R5_BANK1*2	R5_BANK0*2
R6_BANK1*2	R6_BANK0*2
R7_BANK1*2	R7_BANK0*2
R8	R8
R9	R9
R10	R10
R11	R11
R12	R12
R13	R13
R14	R14
R15	R15
FR0*4	FR0*4
FR1*4	FR1*4
FR2*4	FR2*4
⋮	⋮
FR15*4	FR15
SR	SR
SSR	SSR
FPSCR*4	FPSCR*4
GBR	GBR
MACH	MACH
MACL	MACL
FPUL*4	FPUL*4
PR	PR
VBR	VBR
PC	PC
SPC	SPC
R0_BANK0*1, *3	R0_BANK1*1, *3
R1_BANK0*3	R1_BANK1*3
R2_BANK0*3	R2_BANK1*3
R3_BANK0*3	R3_BANK1*3
R4_BANK0*3	R4_BANK1*3
R5_BANK0*3	R5_BANK1*3
R6_BANK0*3	R6_BANK1*3
R7_BANK0*3	R7_BANK1*3

(b) User Mode Programming Model (RB= 1) (c) User Mode Programming Model (RB= 0)

Notes 1. Register R0 is used as an index register in the indexed register-indirect addressing mode and indexed GBR-indirect addressing mode.

2. R0 to R7 are banked registers. In privileged mode, the RB bit of register SR determines which bank is accessed:
 BANK0 if the RB bit is set to 0
 BANK1 if the RB bit is set to 1.

3. These banks are accessed by the LDC and STC instructions only. the RB bit of register SR determines which bank is accessed:
 BANK0 if the RB bit is set to 0
 BANK1 if the RB bit is set to 1.

4. These registers only exist on the SH-3E. They are used for floating point operations. Refer to 4. Floating Point Unit for details on FR0 to FR15, FPSCR, and FPUL.

Figure 2-2 Structure of Registers in Privileged Mode

DSP Registers and Registers Used by the DSP (SH3-DSP Only)

The DSP unit has nine DSP registers, divided into eight data registers and one control register.

The DSP data registers include two 40-bit registers (A0 and A1) and six 32-bit registers (M0, M1, X0, X1, Y0, and Y1). The A1 and A0 registers each has eight guard bits, A0G and A1G.

The DSP data registers are used in transferring and processing DSP data as the operand for the DSP instruction. There are three types of instructions that access the DSP data registers: DSP data processing, X data processing, and Y data processing.

The 32-bit DSP status register (DSR) is the control register, which indicates the results of operations. The DSR register has bits to display the results of the operation, which include a signed greater than bit (GT), a zero value bit (Z), a negative value bit (N), an overflow bit (V), a DSP condition bit (DC), and condition select bits, which control the DC bit settings (CS).

The DC bit is one of the status flags; it is very similar to the SuperH microcomputer CPU core's T bit. In the case of conditional DSP type instructions, the execution of DSP data processing is controlled in accordance with the DC bit. This control is related to DSP unit execution only, and only the DSP registers are updated. It is not related to the execution instructions of the SuperH microprocessor's CPU core, such as address calculation and load/store instructions. The control bits CS (bits 0 to 2) specify the condition that the DC bits set.

DSP instructions include both unconditional DSP instructions and conditioned DSP instructions. Data processing of unconditional DSP instructions updates the condition bits and DC bits, except for the PMULS, PWAD, PWSB, MOVX, MOVY, and MOVS instructions. Conditional DSP type instructions are executed in accordance with the status of the DC bit. DSR registers are not updated, regardless of whether these instructions are executed or not.

Figure 2-1 shows the DSP registers. Table 2-1 lists the DSR register bit functions.

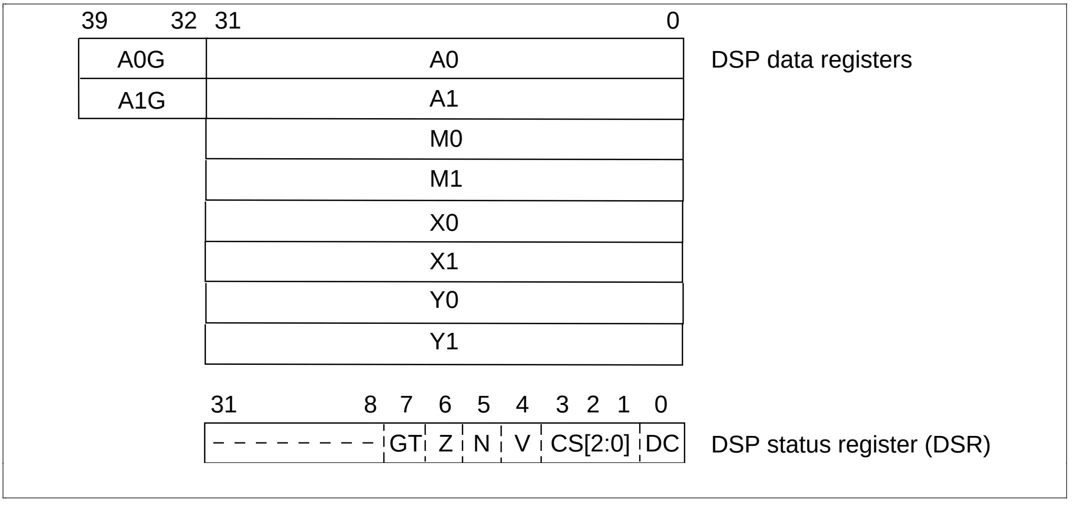


Figure 2-3 Organization of the DSP Registers

Table 2-1 DSR Register Bits

Bits	Name	Function
31–8	Reserved	0: Always reads 0. Always write 0.
7	Signed greater than bit (GT)	Indicates whether the operation result is positive (and nonzero) or whether operand 1 is larger than operand 2. 1: Operation result is positive or operand 1 is larger.
6	Zero value bit (Z)	Indicates whether the operation result is zero or whether of operands 1 and 2 are the same. 1: Operation result is zero or operands 1 and 2 are the same.
5	Negative value bit (N)	Indicates whether the operation result is negative or whether operand 1 is smaller than operand 2. 1: Operation result is negative or operand 1 is smaller.
4	Overflow bit (V)	Indicates that the operation result overflowed. 1: Operation result overflowed.
3–1	Condition select bits (CS)	Specifies the mode for selecting the status of the operation result set in the DC bit. Do not specify 110 or 111. 000: Carry/borrow mode 001: Negative value mode 010: Zero value mode 011: Overflow mode 100: Signed greater than mode 101: Signed equal or greater than mode
0	DSP condition bit (DC)	Sets the operation result status in the mode specified by the CS bits. 0: Specified mode status not achieved 1: Specified mode status achieved.

CPU core instructions use the DSR register as a system register. Data transfer to the DSR register include the following load store instructions:

```
STS    DSR, Rm;
STS.L  DSR, @-Rn;
LDS    Rn, DSR;
LDS.L  @Rn+, DSR;
```

CPU core instructions also use the A0, A1, X0, X1, Y0, and Y1 registers as system registers. There are three DSP control registers: the repeat start (RS) register, the repeat end (RE) register, and the modulo (MOD) register.

The RS and RE registers are used to control program repetition (loops). The number of iterations is specified in the SR register's repeat counter (RC), the repeat start address is specified in the RS register, and the repeat end address is specified in the RE register. The address values stored in the RS and RE registers are not always the same as the physical starting address and ending address of the repeat.

The MOD register uses modulo addressing to buffer the repeat data. Modulo addressing is specified by DMX or DMY in the SR register, the modulo end address (ME) is specified in the top 16 bits of the MOD register, and the modulo start address (MS) is specified in the bottom 16 bits. The DMX and DMY bits cannot simultaneously specify modulo addressing. Modulo addressing can be used for X and Y data transfers (MOVX and MOVY). It cannot be used in single data transfers (MOVS).

Figure 2-5 shows the control registers.

2.2 General-Purpose Registers

Figure 2-4 shows the structure of the general-purpose registers.

31	0
R0*1, *2	
R1*2	
R2*2 [As]*4	
R3*2 [As]*4	
R4*2 [As, Ax]*4	
R5*2 [As, Ax]*4	
R6*2 [Ay]*4	
R7*2 [Ay]*4	
R8 [Ix, Is]*4	
R9 [Iy]*4	
R10	
R11	
R12	
R13	
R14	
R15	

General-purpose registers

Undefined after reset

31	0
FR0*3	
FR1*3	
FR2*3	
FR3*3	
FR4*3	
FR5*3	
FR6*3	
FR7*3	
FR8*3	
FR9*3	
FR10*3	
FR11*3	
FR12*3	
FR13*3	
FR14*3	
FR15*3	

Floating point data register

The FMAC instruction uses FR0 to set the multiplication value.

Notes:
1. R0 functions as an index register in the indexed register-indirect addressing mode and indexed GBR-indirect addressing mode. In some instructions, only R0 can be used as the source or destination register.
2. R0 to R7 are banked registers. In privileged mode, the RB bit of register SR determines which banks (R0_BANK0 to R7_BANK0 or R0_BANK1 to R7_BANK1) are accessed as general-purpose registers.
3. These registers only exist on the SH-3E. They are used for floating point operations. Refer to 4. Floating Point Unit for details on FR0 to FR15.
4. When the DSP instruction extended features of the SH3-DSP are enabled, DSP instructions use these registers as memory address registers and index registers.

Figure 2-4 Structure of the General-Purpose Registers

The symbols R2–R9 are used by the assembler. To change a name to something that indicates the role of the register for DSP instructions, use an alias. The assembler writes as follows:

Ix: .REG (R8)

The name Ix becomes the alias R8. Aliases are also assigned as follows:

```
Ax0:   .REG   (R4)
Ax1:   .REG   (R5)
Ix:    .REG   (R8)
Ay0:   .REG   (R6)
Ay1:   .REG   (R7)
Iy:    .REG   (R9)
As0:   .REG   (R4); defined when an alias is needed for a single data transfer.
As1:   .REG   (R5); defined when an alias is needed for a single data transfer.
As2:   .REG   (R2); defined when an alias is needed for a single data transfer.
As3:   .REG   (R3); defined when an alias is needed for a single data transfer.
Is:    .REG   (R8); defined when an alias is needed for a single data transfer.
```

2.3 Control Registers

Figure 2-5 shows the organization of the control registers.

31	0
SSR	

Saved Status Register (SSR)
Stores current SR value at time of exception to indicate processor status in the return to instruction stream from exception handler. Undefined after reset.

31	0
SPC	

Saved Program Counter (SPC)
Stores current PC value at time of exception to indicate return address at completion of exception processing. Undefined after reset.

31	0
GBR	

Global Base Register (GBR)
Stores the base address of the GBR-indirect addressing mode. The GBR-indirect addressing mode is used to transfer data to the register areas of the resident peripheral modules, and for logic operations. The GBR can be accessed in user mode. Undefined after reset.

31	0
VBR	

Vector Base Register (VBR)
Stores the base address of the exception processing vector area. Initialized to H'00000000 after reset.

31	0
RS	

Repeat Start Register (RS)

31	0
RE	

Repeat End Register (RE)

31 16	15 0
ME	MS

Modulo Register (MOD)
ME: Modulo End Address
MS: Modulo Start Address

Status register (SR)

31	30	29	28	27–16	15–13	12	11	10	9	8	7–4	3	2	1	0
0	MD	RB	BL	RC*	0 — 0	DSP*	DMY*	DMX*	M	Q	I3 I2 I1 I0	RF1*	RF0*	S	T

MD: Processor operation mode bit: Indicates the processor operation mode as follows: 1 = Privileged mode; 0 = User mode. Becomes 1 when an exception or interrupt occurs. Initialized to 1 reset.

RB: Register bank bit: Defines the general-purpose register used as bank in privileged mode. A logic 1 designates R0_BANK1–R7_BANK1 and R8–R15 are accessed as general–purpose registers, and R0_BANK0–R7_BANK0 are only accessed by LDC and STC instructions; a logic zero designates R0_BANK0–R7_BANK0 and R8–R15 are accessed as general-purpose registers, and R0_BANK1–R7_BANK1 are only accessed by LDC and STC instructions. Becomes 1 when an exception or interrupt occurs. Initialized to 1 reset.

BL: Block bit: Masks exceptions and interrupts when 1. For details, see section 5, Exception Processing. When 0, accepts exceptions and interrupts. Becomes 1 when an exception or interrupt occurs. Initialized to 1 at reset.

DSP bit: DSP operation mode. DSP instructions are enabled when set to 1.

M and Q bits: Used by the DIVOS/DIVOU and DIV1 instructions.

RC: Repeat counter. Specifies the number of repeats for repeat (loop) control (2 to 4,095).

DMY: Modulo addressing specification for pointer Y. 1: Modulo addressing mode enabled for Y memory address pointer and Ay (R6 and R7).

DMX: Modulo addressing specification for pointer X. 1: Modulo addressing mode enabled for memory address pointer and Ax (R4 and R5).

I3–I0: Interrupt mask bits: A 4-bit field indicating the interrupt request mask level. The level of interrupt acceptance does not change when an interrupt occurs. Initialized to B'1111 at reset.

S bit: Used by the MAC instruction.

RF1, RF0: Repeat flags. Used for zero-overhead repeat (loop) control.
00: 1-step repeat
01: 2-step repeat
11: 3-step repeat
10: 4-step (or more) repeat

T bit: The MOVT, CMP/cond, TAS, TST, BT, BF, SETT, CLRT, and DT instructions use the T bit to indicate true (logic one) or false (logic zero). The ADDV/ADDC, SUBV/SUBC, DIVOU/DIVOS, DIV1, NEGC, SHAR/SHAL, SHLR/SHRL, ROTA/ROTL, and ROTCR/ROTCL instructions also use the T bit to indicate a carry, borrow, overflow or underflow.

0 bits: Always read as 0, and should always be written as 0.

Notes: Only the M, Q, S, and T can be set or cleared by special instructions from user mode. Undefined after reset. All other bits are read or written from privileged mode.
* 0 for versions other than the SH3-DSP.

Figure 2-5 Control Registers Configuration

2.4 System Registers

The system registers are accessed by the LDS and STS instructions.

Figure 2-3 shows the system register configuration.

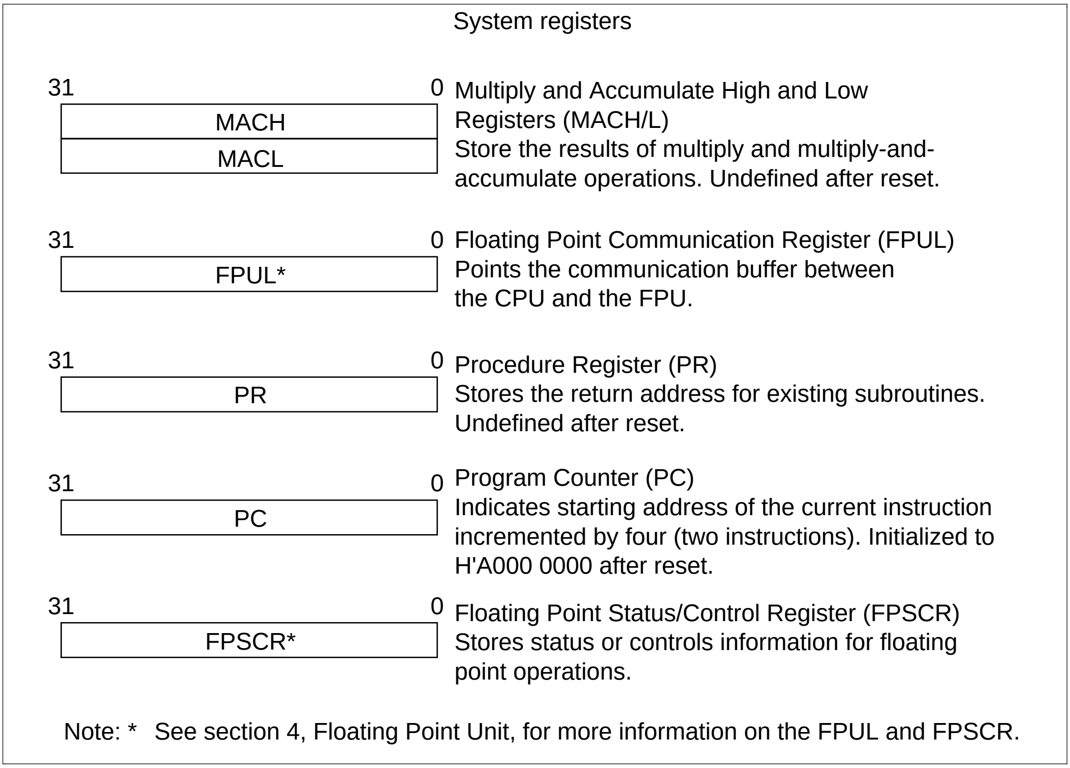


Figure 2-6 System Register Configuration

2.5 Initial Register Value

Table 2-1 shows the register values after a reset.

Table 2-1 Initial Register Values

Register Type	Register	Initial Value*[1]
General purpose	R0–R15	Undefined
	FR0–FR15*[2]	Undefined
Control	SR	MD bit is 1, RB bit is 1, BL bit is 1, bits I3–I0 are 1111 (H'F), bits RC, DMY, and DMX are 0 (SH3-DSP only), reserved bits are 0, and all others are undefined
	GBR, SSR, SPC	Undefined
	VBR	H'00000000
	RS*[2], RE*[2]	Undefined
	MOD*[2]	Undefined
System	MACH, MACL, PR, FPSCR*[1], FPUL*[1]	Undefined
	PC	H'A0000000
DSP	A0, A0G, A1, A1G, M0, M1, X0, X1, Y0, Y1	
	DSR	H'00000000

Notes: 1. These registers only exist on the SH-3E. They are used for floating point operations. Refer to 4. Floating Point Unit for details on FR0 to FR15, FPSCR, and FPUL.

2. These registers only exist on the SH-3E.

Section 3 Data Formats

3.1 Data Format in Registers

Register operands are always longwords (32 bits) (figure 3-1). When the memory operand is only a byte (8 bits) or a word (16 bits), it is sign-extended into a longword when loaded into a register.

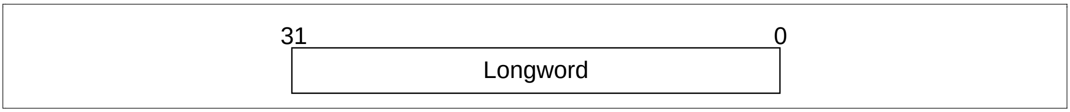


Figure 3-1 Longword Operand

3.2 Data Format in Memory

Memory data formats are classified into bytes, words, and longwords. Memory can be accessed in bytes (8 bits), words (16 bits), or longwords (32 bits). Memory operands that do not fill out 32 bits are sign-extended and stored in a register.

Access word operands from word boundaries (even addresses two bytes apart: 2n addresses) and longword operands from longword boundaries (even addresses four bytes apart: 4n addresses). Other accesses cause address errors. Byte operands can be accessed from any address.

Data formats can use either big endian or little endian byte order. Use the external pin (MD5) to set the endian at power-on reset. When MD5 is low, the processor operates in big endian; when MD5 is high, the processor operates in little endian. Endians cannot be changed dynamically. Numbers are always assigned to bit positions, from most significant to least significant and from left to right. For example, in a longword (32 bits), the leftmost bit (31) is the most significant and the rightmost bit (0) is the least significant.

Figure 2-6 shows the data format in memory. When little endian is used, data written in bytes (8 bits) should be read in bytes. Data written in words (16 bits) should be read in words.

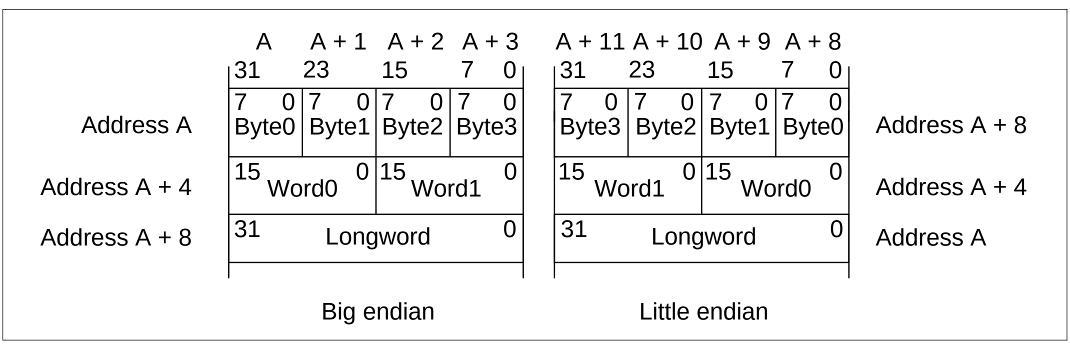


Figure 3-2 Data Formats in Memory

3.3 Data Format for Immediate Data

Immediate data bytes are arranged inside instruction codes.

For the MOV, ADD, and CMP/EQ instructions, immediate data is sign-extended and then processed as registers and longwords. In contrast, for the TST, AND, OR, and XOR instructions, immediate data is zero-extended and then processed as longwords. Consequently, if immediate data is used with the AND instruction, the upper 24 bits of the destination register will always be cleared.

Word and longword immediate data is not arranged inside instruction codes. Instead, it is stored in memory table. Memory tables can be accessed using the immediate data transfer instruction (MOV) in the PC relative addressing mode with displacement.

For specific examples, see 6.1.8 Immediate Data in section 6. Instruction Features.

3.4 DSP Type Data Formats (SH3-DSP Only)

The SH-DSP uses three different data formats for instructions: the fixed decimal point data format, the integer data format, and the logical data format.

The DSP type of fixed decimal point data format places a binary decimal point between bits 31 and 30. This data format can have guard bits, no guard bits, or be multiplication input. The valid bit lengths and values displayed vary for each.

DSP type integer data formats place a binary decimal point between bits 16 and 15. This data format can have guard bits, no guard bits, or be a shift amount. The valid bit lengths and values displayed vary for each. The shift amount for arithmetic shift (PSHA) is a seven-bit area between –64 and +63, although only values between –32 and +32 are valid. The shift amount for logical shifts is a six bit area, although, in the same fashion, only values between –16 and +16 are valid.

The DSP type logical data format has no decimal point. The data format and valid data length vary with the instruction and DSP register.

Figure 3-3 shows the three DSP data formats and the position of the two binary decimal points, as well as the SuperH data format (as reference).

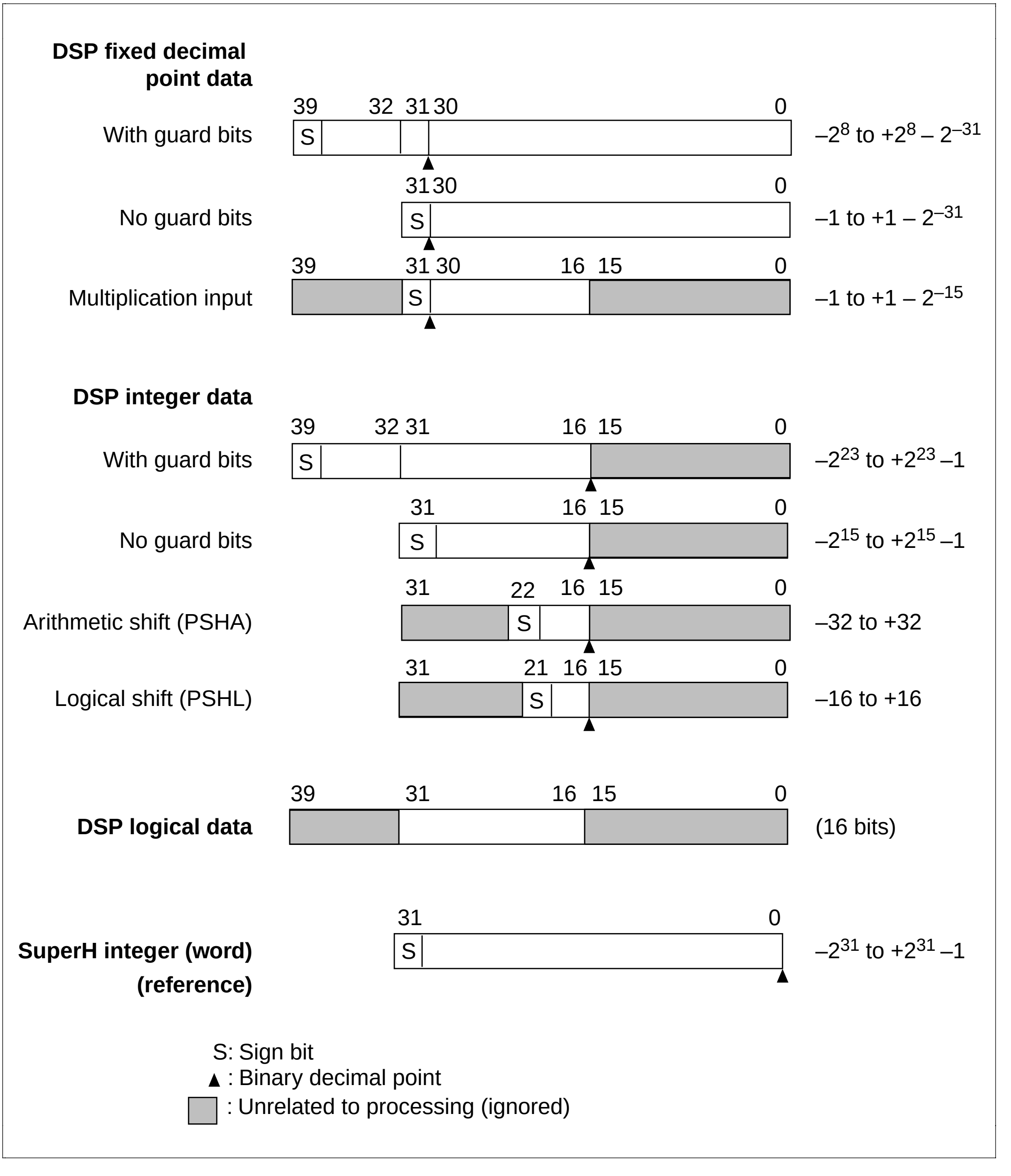


Figure 3-3 DSP Data Formats

Section 4 Floating Point Unit (SH-3E only)

4.1 Introduction

The SH-3E has a built-in floating point operations unit (FPU). Figure 4-1 shows the FPU registers.

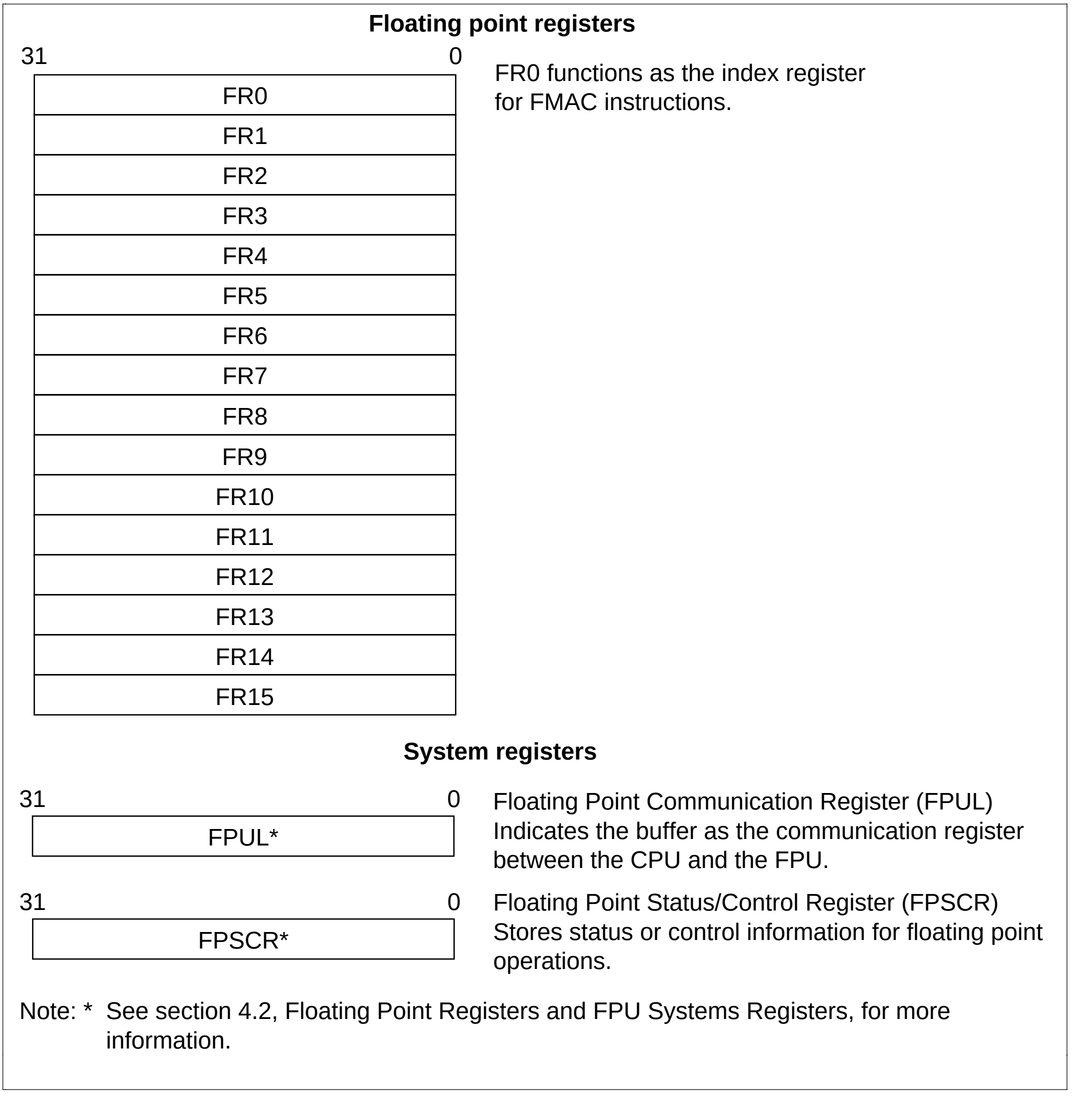


Figure 4-1 Register Set Overview: Floating Point Registers and System Registers Used by the FPU

4.2 Floating Point Registers and System Registers for FPU

4.2.1 Floating Point Register File

The SH-3E provides sixteen 32-bit single-precision floating point registers. Register designators are always 4-bits. In assembly language, the floating point registers are designated as FR0, FR1, FR2, etc. FR0 functions as the index for FMAC instructions.

4.2.2 Floating Point Communication Register (FPUL)

Information is transferred between the FPU and the CPU through a communication register, FPUL, which is analogous to the MACL and MACH registers of the integer unit. The SH-3E provides this communication register because of the differences between integer format and FPU format. FPUL is a 32-bit system register, accessed on the CPU side by LDS and STS instructions.

4.2.3 Floating Point Status/Control Register (FPSCR)

The SH-3E implements a floating point status and control register, FPSCR, as a system register accessed through the LDS and STS instructions (figure 4-2). FPSCR is available for modification by user programs. The FPSCR is part of the process context. It must be saved across context switches and may need to be saved across procedure calls.

The FPSCR is a 32-bit register that controls FPU rounding, handling of denormalized values, and captures details about floating point exceptions.

In the SH-3E, only the following modes are supported for these functions.

- Rounding mode: Rounding toward 0.
- Handling of denormalized values: When denormalized values are in the source or destination operand, they are always treated as 0.
- FPU exceptions: Divide by zero (Z) and invalid (V).

31–19	18	17	16	15	14–12	11	10	9–7	6	5	4–2	1	0
			Cause			Enable			Flag				
0 -------------------- 0	1	0	CV	CZ	0 0 0	EV	EZ	0 0 0	FV	FZ	0 0 0	0	1

CV: Invalid-operation cause bit
- 1: Invalid-operation exception occurred during execution of the current instruction
- 0: Invalid-operation exception did not occur

CZ: Divide-by-zero cause bit
- 1: Divide-by-zero exception occurred during the execution of the current instruction
- 0: Divide-by-zero exception did not occur

EV: Invalid-operation exception enable bit
- 1: Enable invalid-operation exception
- 0: Disable invalid-operation exception and return qNaN as a result

EZ: Divide-by-zero exception enable bit
- 1: Enable divide-by-zero exception
- 0: Disable divide-by-zero exception and return correctly signed infinity

FV: Invalid-operation exception flag bit
- 1: Invalid-operation exception occurred during execution of the current instruction
- 0: Invalid-operation exception did not occur

FZ: Divide-by-zero exception flag bit
- 1: Divide-by-zero exception occurred during the execution of the current instruction
- 0: Divide-by-zero exception did not occur

Note: With the exception of the above bits, all bits are reserved as shown in the figures and cannot be modified even by LDS instruction.

Figure 4-2 Floating Point Status/Control Register

The bits in the cause field indicate the cause of exception for the executing of the current instruction. The cause bits are modified by execution of a floating point instruction. These bits are set to 0 or 1, depending on occurrence or non-occurrence of exception conditions during the execution of a single instruction.

The bits in the enable field indicate the specific types of exceptions that are enabled to cause an exception, that is, change of flow to an exception handling procedure. An exception occurs if the enable bit and the corresponding cause bit are set by the execution of the current instruction.

The bits in the flag field are used to capture the cumulative effect of all exceptions during the execution of a sequence of instructions. These bits, once set by an instruction, can not be reset by following instructions. The bits in this field can only be reset by an explicit store operation on FPSCR.

See section 4.4, Floating Point Exceptions Model, for more information on handling of floating point exceptions.

4.3 Floating Point Format

4.3.1 Floating Point Format

The SH-3E supports single-precision floating point operations. It also conforms fully to the IEEE754 standard.

Floating point numbers are composed of three fields:

Sign field : s

Exponent field : e

Mantissa field : f

The exponent is biased. In other words:

$$e = E + bias$$

The range of unbiased exponents E is E_{min}–1 to E_{max}+1. The two values (E_{min}–1 and E_{max}+1) are distinguished as follows. E_{min}–1 represents zero (sign is both positive and negative) and a denormalized number while E_{max}+1 represents positive and negative infinity and a not-a-number (NaN). In single-precision operations, the bias value is 127, E_{min} is –126, and E_{max} is 127.

31	30 … 23	22 … 0
s	e	f

Figure 4-3 Floating Point Format

The value v of the floating point number is determined as follows:

If E== E_{max}+1 and f!=0, then v is not a number (NaN) regardless of sign s

If E== E_{max}+1 and f==0, then $v=(-1)^s$ (infinity) [positive or negative infinity]

If E_{min}<=E<= E_{max}, then $v =(-1)^s 2^E$ (1.f) [normalized number]

If E== E_{min}–1 and f!=0, then $v =(-1)^s 2^{Emin}$ (0.f) [denormalized number]

If E== E_{min}–1 and f==0, then $v =(-1)^s$ 0 [positive or negative zero]

4.3.2 Not a Number (NaN)

In not-a-number (NaN) expressions in single-precision operations, at least one of the bits 22–0 is set. Set bit 22 for a signaling NaN (sNaN). When bit 22 is reset, the value is then the quiet NaN (qNaN).

The following figure shows the bit pattern of the not-a-number (NaN). Bit N in the figure is set for sNaN and reset for qNaN. An x indicates a don’t-care bit. At least one of bits 22-0 must be set.

In a not-a-number (NaN), the sign bit is a don’t-care bit.

31	30 23	22 0
x	11111111	Nxxxxxxxxxxxxxxxxxxxxxx

N = 1: sNaN
N = 0: qNaN

Figure 4-4 NaN Bit Pattern

When a not-a-number (sNaN) is entered in the operation that generates the floating point value:

When the EV bit is reset in the FPSCR, the operation result (output) is qNaN.

When the EV bit is set in the FPSCR, an invalid operation exception occurs. In such cases, the contents of the register at the destination side of the operation do not change.

When qNaN is input to the operation that generates the floating point value and sNaN is not input to the operation, the output will always be qNaN regardless of how the EV bit is set in the FPSCR. No exception will occur.

4.3.3 Denormalized Values

Denormalized floating point values are expressed by a biased exponent of 0, a nonzero mantissa, and a hidden bit of 0. In the SH-3E’s floating point unit, denormalized values (operand source or operation result) are uniformly flushed with 0 in floating point operations (other than copy) that generate values.

4.3.4 Other Special Values

Other special values are as stipulated by standard IEEE754. Table 4-1 shows the seven different types of special values in floating point value expressions.

Table 4-1 Special Value Expressions in Single-Precision Stipulated in IEEE754

Value	Expression
+0.0	0x00000000
-0.0	0x80000000
Denormalized number	See section 4.3.3, Denormalized Values
+INF	0x7F800000
–INF	0xFF800000
qNaN (quiet NaN)	See section 4.3.2, Not a Number (NaN)
sNaN (signaling NaN)	See section 4.3.2, Not a Number (NaN)

4.4 Floating Point Exception Model

4.4.1 Enabled Exception

Invalid-operation and divide-by-zero exceptions are enabled by setting the enable bit for the relevant exception (the EV or EZ bit) in FPSCR. All exceptions caused by the FPU are mapped as FPU exception events. The meaning of an individual exception is determined by software by reading the FPSCR system register and analyzing the information held there.

4.4.2 Disabled Exception

If enable bit EV is not set in FPSCR, the result of an invalid operation will be qNaN (with the exception of FCMP and FTRC). If enable bit EZ is not set, division by zero will return infinity with the sign of the current expression (+ or -).

The other floating-point exceptions specified in the IEEE754 standard—inexact, overflow, and underflow—are not supported by the SH-3E. In these cases, the SH-3E operates as described below.

- An overflow will produce the number whose absolute value is the largest representable finite number in the format with the correct sign bit. An underflow will produce a correctly signed zero. If the result of an operation is inexact, the destination register will hold the inexact result.

4.4.3 Exception Event and Code for FPU

All FPU exceptions are mapped onto the single general exception event at address H'0x120. Loads and stores of system registers FPUL and FPSCR cause the normal memory management general exceptions.

4.4.4 Alignment of Floating Point Data in Memory

Single precision floating point data is aligned on modulus-4 boundaries, that is, in the same fashion as SH-3E long integers.

4.4.5 Arithmetic with Special Operands

All arithmetic with special operands (qNaN, sNaN, +INF, –INF, +0, –0) follows IEEE754 rules.

4.5 Synchronization Issues

Synchronization with CPU: Floating-point and CPU instructions are issued serially in program order, but may complete out-of-order due to execution cycle differences. A floating point operation that accesses only FPU resources does not require synchronization with the CPU, and subsequent CPU operations can complete before the completion of the floating point operation. Therefore an optimized program can hide the execution cycle of a long-execution-cycle floating point operation such as Divide. A floating point operation such as Compare that accesses CPU resources, however, requires synchronization to ensure program order.

Floating Point Instructions Requiring Synchronization: Loads, stores, compares/tests, and instructions accessing FPUL access CPU resources and therefore require synchronization. Loads and Stores refer to general registers. Post-increment loads and pre-decrement stores modify general registers. Compares/tests modify the T bit. Instructions accessing FPUL refer to or modify FPUL. These references and modifications must be synchronized with the CPU.

Maintaining Program Order on Exceptions: Floating point instructions are never completed until subsequent CPU instructions are completed. If an FPU exception is detected before subsequent CPU instructions finish and an FPU exception occurs, subsequent CPU instructions are canceled.

During a floating point instruction execution, if a subsequent instruction causes an exception, the floating point instruction is left executing and FPU resources cannot be accessed by other instructions. The other instructions must await the completion of the floating point operation before they can access. This ensures program order.

Section 5 DSP Operation Functions and Data Transfers (SH3-DSP Only)

DSP operations and data transfers are listed below:

ALU Fixed Decimal Point Operations: These are fixed decimal point operations with either 40-bit (with guard bits) or 32-bit (with no guard bits) fixed decimal point data. These include addition, subtraction, and comparison instructions.

ALU Integer Operations: These are integer arithmetic operations with either 24-bit (with guard bits) or 16-bit (with no guard bits) integer data. They include increment and decrement instructions.

ALU Logical Operations: These are logical operations with 16-bit logical data. They include AND, OR, and exclusive OR.

Fixed Decimal Point Multiplication: This is fixed decimal point multiplication (arithmetic operation) of the top 16 bits of fixed decimal point data. Condition bits such as the DC bit are not updated.

Shift Operations: These are arithmetic and logical shift operations. Arithmetic shift operations are arithmetic shifts of 40 bits (with guard bits) or 32 bits (with no guard bits) of fixed decimal point data. Logical shift operations are logical operations on 16 bits of logical data. The amount of the arithmetic shift operation is –32 to +32 (negative for right shifts, positive for left shifts); for logical shifts, the amount is –16 to +16.

MSB Detection Instruction: This operation finds the amount of the shift to normalize the data. It finds the position of the MSB bit in either 40-bit (with guard bits) or 32-bit (with no guard bits) fixed decimal point data as either 24 bits (with guard bits) or 16 bits (with no guard bits) integer data.

Rounding Operation: Rounds 40-bit fixed decimal point data (with guard bits) to 24 bits or 32-bit (with no guard bits) fixed decimal point data to 16 bits.

Data Transfers: Data transfers consist of X and Y data transfers, which load or store 16-bit data to and from X and Y memory, and single data transfers, which load and store 16- or 32-bit data from all memories. Two X and Y data transfers can be processed in parallel. Condition bits such as the DC bit are not updated.

The operation instructions include both conditional operation instructions and instructions that are conditionally executed depending on the DC bit. Condition bits such as the DC bit are not updated by conditional instructions. Their settings vary for arithmetic operations, logical operations, arithmetic shifts, and logical shifts. or MSB detection instructions and rounding instructions, set the condition bits like for arithmetic operations.

Arithmetic operations include overflow preventing instructions (saturation operations). When saturation operation is specified with the S bit in the SR register, the maximum (positive) or minimum (negative) value is stored when the result of operation overflows.

5.1 ALU Fixed Decimal Point Operations

5.1.1 Function

ALU fixed decimal point operations basically work with a 32-bit unit to which 8 guard bits are added for a total of 40 bits. When the source operand is a register without guard bits, the register's sign bit is extended and copied to the guard bits. When the destination operand is a register without guard bits, the lower 32 bits of the operation result are stored in the destination register.

ALU fixed decimal point operations are performed between registers. The source and destination operands are selected independently from the DSP register. When there are guard bits in the selected register, the operation is also executed on the guard bits. These operations are executed in the DSP stage (the last stage) of the pipeline.

Whenever an ALU arithmetic operation is executed, the DSR register's DC, N, Z, V, and GT bits are updated by the operation result. For conditional instructions, however, condition bits are not updated even when the specified condition is achieved. For unconditional instructions, the bits are updated according to the operation result.

The condition reflected in the DC bit is selected with the CS[2:0] bits. The DC bits of the PADDC and PSUB instructions, however, are updated regardless of the CS bit settings. In the PADDC instruction, it is updated as a carry flag; in the PSUB instruction, it is updated as a borrow flag.

Figure 5-1 shows the ALU fixed decimal point operation flowchart.

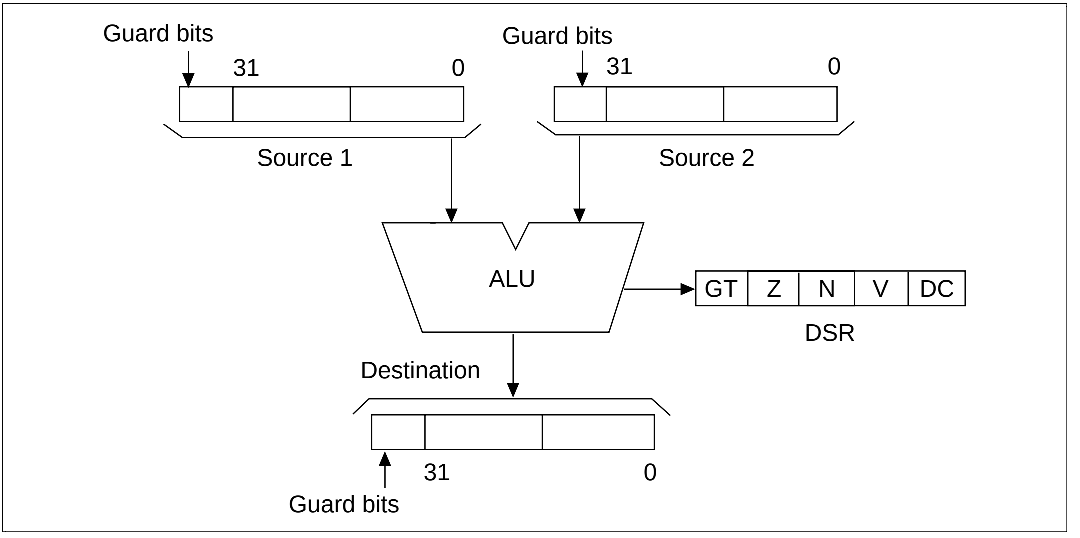


Figure 5-1 ALU Fixed Decimal Point Operation Flowchart

When the memory read destination operand is the same as the ALU operation source operand and the data transfer instruction program is written on the same line as the ALU operation, data loaded from memory in the memory access stage (MA) cannot be used as the source operand of the ALU operation instruction. When this occurs, the result of the instruction executed first is used as the source operand of the ALU operation and is updated as the destination operand of the data load instruction thereafter. Figure 5-2 is a flowchart of the operation.

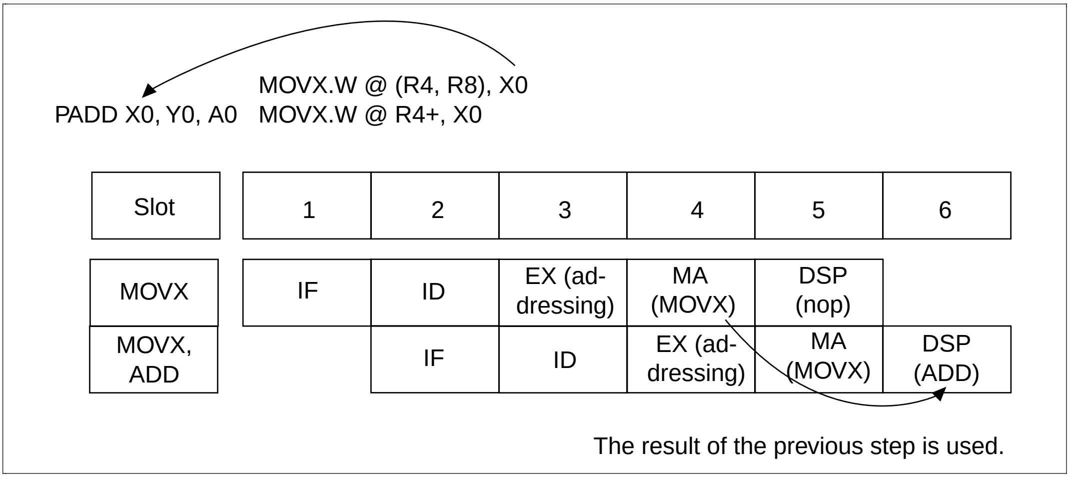


Figure 5-2 Sample Processing Flowchart

5.1.2 Instructions and Operands

Table 5-1 shows the types of ALU fixed decimal point arithmetic operations. Table 5-2 shows the correspondence between the operands and registers.

Table 5-1 Types of ALU Fixed Decimal Point Arithmetic Operations

Mnemonic	Function	Source 1	Source 2	Destination
PADD	Addition	Sx	Sy	Dz (Du)
PSUB	Subtraction	Sx	Sy	Dz (Du)
PADDC	Addition with carry	Sx	Sy	Dz
PSUBC	Subtraction with borrow	Sx	Sy	Dz
PCMP	Compare	Sx	Sy	—
PCOPY	Copy data	Sx	—	Dz
		—	Sy	Dz
PABS	Absolute value	Sx	—	Dz
		—	Sy	Dz
PNEG	Invert sign	Sx	—	Dz
		—	Sy	Dz
PCLR	Zero clear	—	—	Dz

Table 5-2 Correspondence between Operands and Registers for ALU Fixed Decimal Point Arithmetic Operations

Operand	X0	X1	Y0	Y1	M0	M1	A0	A1
Sx	Yes*[1]	Yes					Yes	Yes
Sy			Yes	Yes	Yes	Yes		
Dz	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Du*[2]	Yes		Yes				Yes	Yes

Notes: 1. Yes: Register can be used with operand.

2. Du: Operand when used in combination with multiplication.

5.1.3 DC Bit

The DC bit is set as follows depending on the specification of the CS0-CS2 bits (condition select bits) of the DSR register.

Carry/Borrow Mode: CS2–CS0 = 000: The DC bit indicates whether a carry or borrow has occurred from the MSB of the operation result. The guard bits have no affect on this. This mode is the default. Figure 5-3 shows examples when carries and borrows occur.

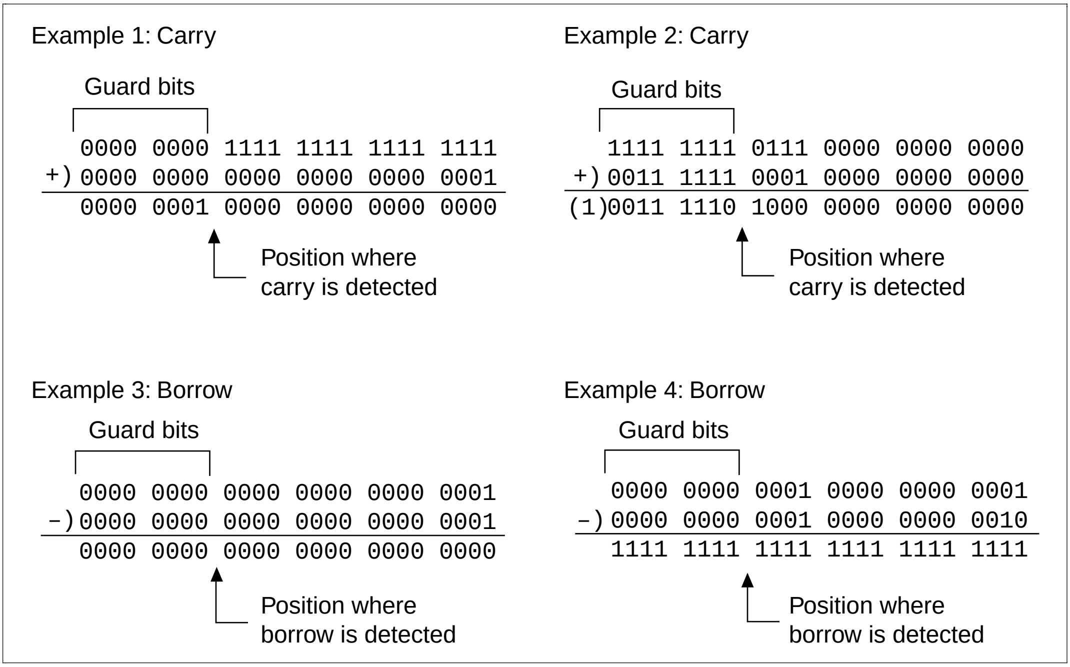


Figure 5-3 Examples of Carries and Borrows

Negative Mode: CS2–CS0 = 001: In this mode, the DC bit is the same as the MSB of the operation result. When a result is negative, the DC bit is 1. When the result is positive, the DC bit is 0. ALU arithmetic operations are always done in 40 bits. The sign bit indicating positive or negative is thus the MSB included in the guard bits of the operation result rather than the MSB of the destination operand. Figure 5-4 shows an example of distinguishing negative from positive. In this mode, the DC bit has the same value as the condition bit N.

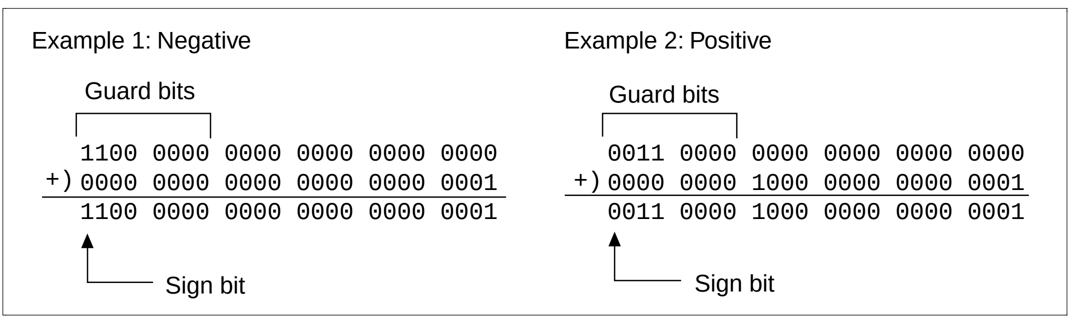


Figure 5-4 Distinguishing Negative and Positive

Zero Mode: CS2–CS0 = 010: The DC bit indicates whether the operation result is zero. When it is, the DC bit is 1. When the operation result is nonzero, the DC bit is 0. In this mode, the DC bit has the same value as the condition bit Z.

Overflow Mode: CS2–CS0 = 011: The DC bit indicates whether the operation result has caused an overflow. When the operation result without the guard bits has exceeded the bounds of the destination register, the DC bit is set to 1. The DC bit considers there to be no guard bits, which makes it an overflow even when there are guard bits. This means that the DC bit is always set to 1 when large numbers use guard bits. In this mode, the DC bit has the same value as the condition bit V. Figure 5-5 shows an example of distinguishing overflows.

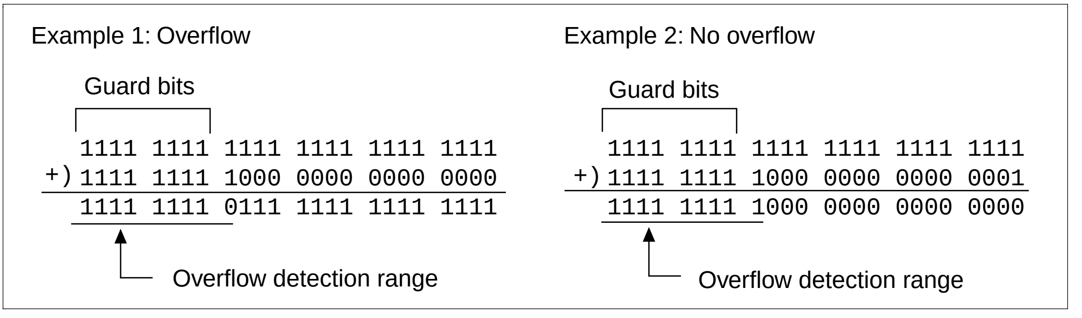


Figure 5-5 Distinguishing Overflows

Signed Greater Than Mode: CS2–CS0 = 100: The DC bit indicates whether the source 1 data (signed) is greater than the source 2 data (signed) in the result of a comparison instruction PCMP. For that reason, the PCMP instruction is executed before checking the DC bit in this mode. When the source 1 data is larger than the source 2 data, the result of the comparison is positive, so this mode becomes similar to the negative mode. When the source 1 data is larger than the source 2 data and the bounds of the destination operand are exceeded, however, the sign of the result of the comparison becomes negative. The DC bit is updated. In this mode, the DC bit has the same value as the condition bit GT. The equation shown below defines the DC bit in this mode. However, VR becomes a positive value when the result including the guard bit area exceeds the display range of the destination operand.

DC bit = ~ {(N bit ^ VR)|Z bit}

When the PCMP instruction is executed in this mode, the DC bit becomes the same value as the T bit that indicates the result of the SH core's CMP/GT instruction. In this mode, the DC bit is updated according to the above definition for instructions other than the PCMP instruction as well.

Signed Greater Than or Equal to Mode: CS2–CS0 = 101: The DC bit indicates whether or not the source 1 data (signed) is greater than or equal to the source 2 data (signed) in the result of the execution of a comparison instruction PCMP. For that reason, the PCMP instruction is executed before checking the DC bit in this mode. This mode is similar to the Signed Greater Than mode except for checking if the operands are the same. The equation shown below defines the DC bit in this mode. However, VR becomes a positive value when the result, including the guard bit area, exceeds the display range of the destination operand.

DC bit = ~ (N bit ^ VR)

When the PCMP instruction is executed in this mode, the DC bit becomes the same value as the T bit that indicates the result of the SuperH core's CMP/GE instruction. In this mode, the DC bit is updated according to the above definition for instructions other than the PCMP instruction as well.

5.1.4 Condition Bits

The condition bits are set as follows:

- The N (negative) bit has the same value as the DC bit when the CS bits specify negative mode. When the operation result is negative, the N bit is 1. When the operation result is positive, the N bit is 0.
- The Z (zero) bit has the same value as the DC bit when the CS bits specify zero mode. When the operation result is zero, the Z bit is 1. When the operation result is nonzero, the Z bit is 0.
- The V (overflow) bit has the same value as the DC bit when the CS bits specify overflow mode. When the operation result exceeds the bounds of the destination register without the guard bits, the V bit is 1. Otherwise, the V bit is 0.
- The GT (greater than) bit has the same value as the DC bit when the CS bits specify Signed Greater Than mode. When the comparison result indicates the source 1 data is greater than the source 2 data, the GT bit is 1. Otherwise, the GT bit is 0.

5.1.5 Overflow Prevention Function (Saturation Operation)

When the S bit of the SR register is set to 1, the overflow prevention function is engaged for the ALU fixed decimal point arithmetic operation executed by the DSP unit. When the operation result overflows, the maximum (positive) or minimum (negative) value is stored.

5.2 ALU Integer Operations

ALU integer operations are basically 24-bit operations on the top word (the top 16 bits, or bits 16 through 31) and 8 guard bits. In ALU integer operations, the bottom word of the source operand (the bottom 16 bits, or bits 0–15) is ignored and the bottom word of the destination operand is cleared with zeros. When the source operand has no guard bits, the sign bit is extended to fill the guard bits. When the destination operand has no guard bits, the top word of the operation result (not including the guard bits) are stored in the top word of the destination register.

Integer operations are basically the same as ALU fixed decimal point arithmetic operations. There are only two types of integer operation instructions, increment and decrement, which change the second operand by +1 or –1. 16 bits of integer data (word data) is loaded to the DSP register and stored in the top word. The operation is performed using the top word in the DSP register. When there are guard bits, they are valid as well. These operations are executed in the DSP stage (the last stage) of the pipeline.

Whenever an ALU integer arithmetic operation is executed, the DSR register's DC, N, Z, V, and GT bits are basically updated by the operation result. This is the same as for ALU fixed decimal point operations.

For conditional instructions, condition bits and flags are not updated even when the specified condition is achieved and the instruction executed. For unconditional instructions, the bits are always updated according to the operation result. Figure 5-6 shows the ALU integer operation flowchart.

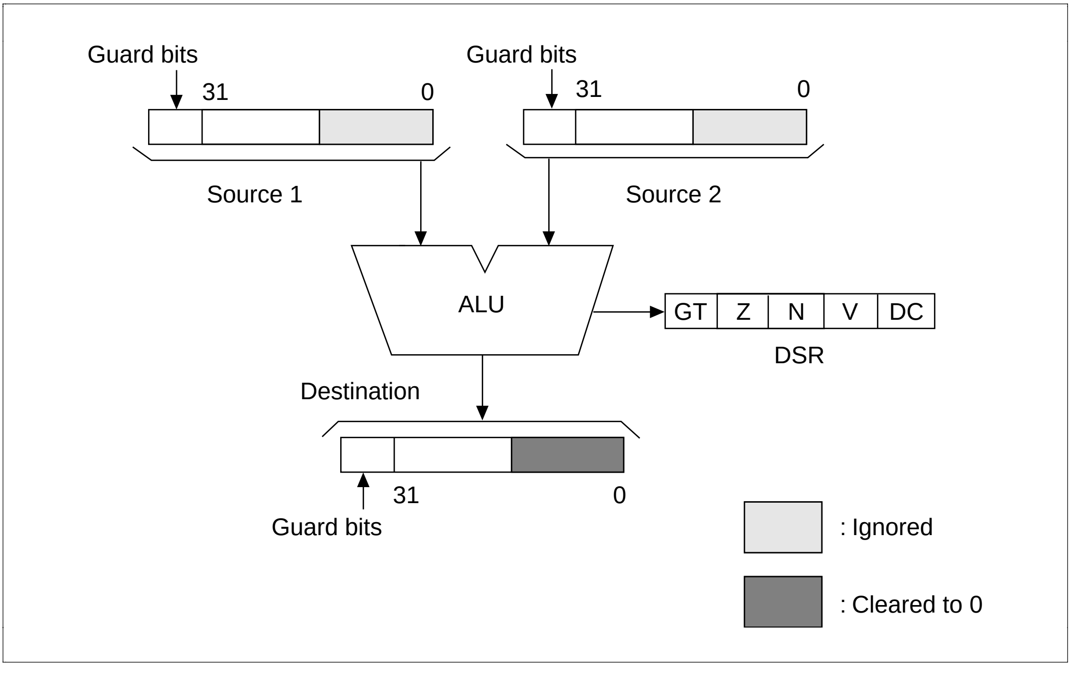


Figure 5-6 ALU Integer Operation Flowchart

Table 5-3 lists the types of ALU integer operations. Table 5-4 shows the correspondence between the operands and registers.

Table 5-3 Types of ALU Integer Operations

Mnemonic	Function	Source 1	Source 2	Destination
PINC	Increment by 1	Sx	(+1)	Dz
		(+1)	Sy	Dz
PDEC	Decrement by 1	Sx	(–1)	Dz
		(–1)	Sy	Dz

Table 5-4 Correspondence between Operands and Registers for ALU Integer Operations

Operand	X0	X1	Y0	Y1	M0	M1	A0	A1
Sx	Yes	Yes					Yes	Yes
Sy			Yes	Yes	Yes	Yes		
Dz	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes

Note: Yes: Register can be used with operand.

When the S bit of the SR register is set to 1, the overflow prevention function (saturation operation) is engaged. The overflow prevention function can be specified for ALU integer arithmetic operations executed by the DSP unit. When the operation result overflows, the maximum (positive) or minimum (negative) value is stored.

5.3 ALU Logical Operations

5.3.1 Function

ALU logical operations are performed between registers. The source and destination operands are selected independently from the DSP register. These operations use only the top word of the respective operands. The bottom word of the source operand and the guard bits are ignored and the bottom word of the destination operand and guard bits are cleared with zeros. These operations are executed in the DSP stage (the last stage) of the pipeline.

Whenever an ALU arithmetic operation is executed, the DSR register’s DC, N, Z, V, and GT bits are basically updated by the operation result. For conditional instructions, condition bits and flags are not updated even when the specified condition is achieved and the instruction executed. For unconditional instructions, the bits are always updated according to the operation result. The DC bit is updated as specified in the CS bits. Figure 5-7 shows the ALU logical operation flowchart.

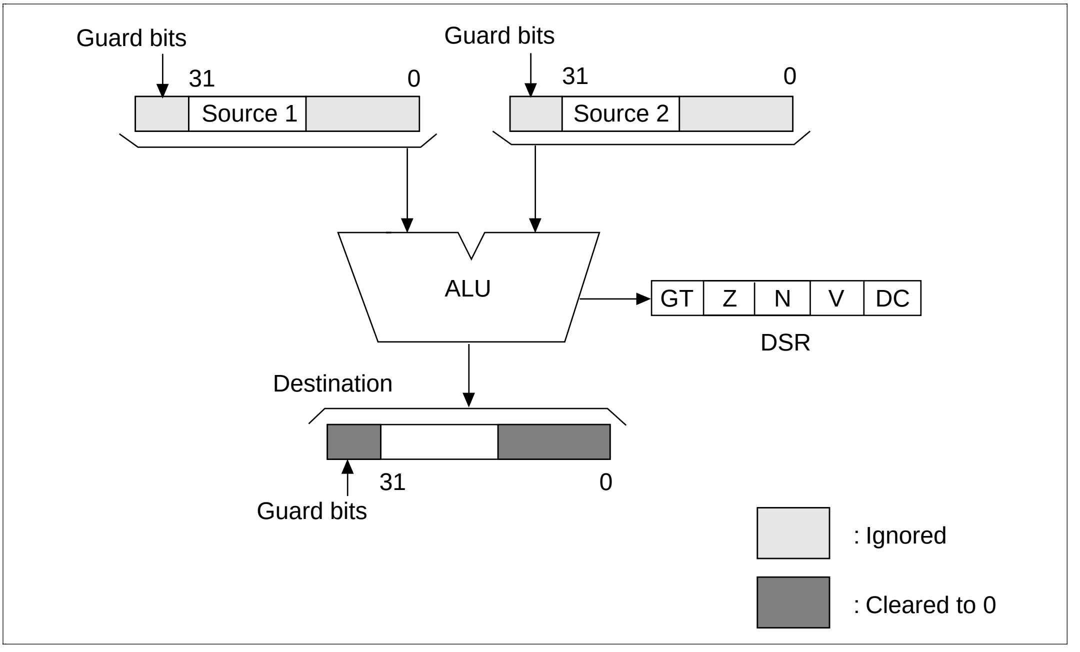


Figure 5-7 ALU Logical Operation Flowchart

5.3.2 Instructions and Operands

Table 5-5 lists the types of ALU logical arithmetic operations. Table 5-6 shows the correspondence between the operands and registers, which is the same as for ALU fixed decimal point operations.

Table 5-5 Types of ALU Logical Arithmetic Operations

Mnemonic	Function	Source 1	Source 2	Destination
PAND	AND	Sx	Sy	Dz
POR	OR	Sx	Sy	Dz
PXOR	Exclusive OR	Sx	Sy	Dz

Table 5-6 Correspondence between Operands and Registers for ALU Logical Arithmetic Operations

Operand	X0	X1	Y0	Y1	M0	M1	A0	A1
Sx	Yes	Yes					Yes	Yes
Sy			Yes	Yes	Yes	Yes		
Dz	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes

Note: Yes: Register can be used with operand.

5.3.3 DC Bit

The DC bit is set in logical operations as follows:

Carry/Borrow Mode: CS2–CS0 = 000: The DC bit is always 0.

Negative Mode: CS2–CS0 = 001: In this mode, the DC bit is the same as the bit 31 of the operation result. In this mode, the DC bit has the same value as bit N.

Zero Mode: CS2–CS0 = 010: The DC bit is 1 when the operation result is zero; otherwise, the DC bit is 0. In this mode, the DC bit has the same value as bit Z.

Overflow Mode: CS2–CS0 = 011: The DC bit is always 0. In this mode, the DC bit has the same value as bit V.

Signed Greater Than Mode: CS2–CS0 = 100: The DC bit is always 0. In this mode, the DC bit has the same value as bit GT.

Signed Greater Than or Equal to Mode: CS2–CS0 = 101: The DC bit is always 0.

5.3.4 Condition Bits

The condition bits are set as follows.

- The N bit is the value of bit 31 of the operation result.
- The Z bit is 1 when the operation result is zero; otherwise, the Z bit is 0.
- The V bit is always 0.
- The GT bit is always 0.

5.4 Fixed Decimal Point Multiplication

Multiplication in the DSP unit is between signed single-length operands. It is processed in one cycle. When double-length multiplication is needed, use the SuperH RISC engine’s double-length multiplication.

Basically, the operation result for multiplication is 32 bits. When a register that has guard bits is specified as the destination operand, it is sign-extended.

In the DSP unit, multiplication is a fixed decimal point arithmetic operation, not an integer operation. This means the top words of the constant and multiplicand are entered into the MAC operator. In SuperH RISC engine multiplication, the bottom words of the two operands are entered into the MAC operator. The operation result thus is different from the SuperH RISC engine. The SuperH RISC engine operation result is matched to the LSB of the destination, while the fixed decimal point multiplication operation result is matched to the MSB. The LSB of the operation result in fixed decimal point multiplication is thus always 0.

Figure 5-8 shows a flowchart of fixed decimal point multiplication.

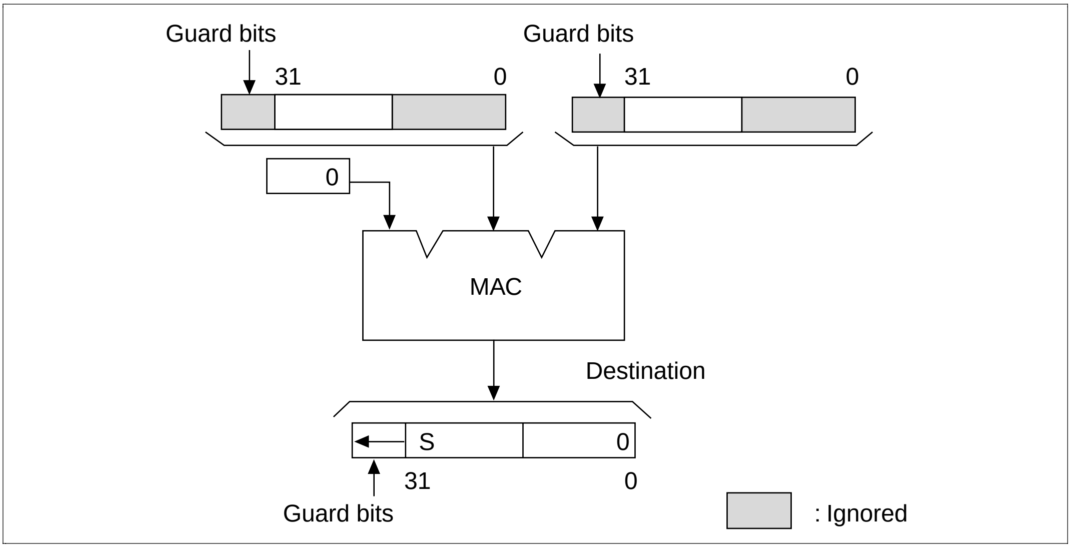


Figure 5-8 Fixed Decimal Point Multiplication Flowchart

Table 5-7 shows the fixed decimal point multiplication instruction. Table 5-8 shows the correspondence between the operands and registers.

Table 5-7 Fixed Decimal Point Multiplication

Mnemonic	Function	Source 1	Source 2	Destination
PMULS	Signed multiplication	Se	Sf	Dg

Table 5-8 Correspondence between Operands and Registers for Fixed Decimal Point Multiplication

Operand	X0	X1	Y0	Y1	M0	M1	A0	A1
Sx	Yes	Yes					Yes	Yes
Sy			Yes	Yes	Yes	Yes		
Dz	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes

Note: Yes: Register can be used with operand.

DSP unit fixed decimal point multiplication completes a single-length 16 bit × 16 bit operation in one cycle. Other multiplication is the same as in the SuperH RISC engines.

Multiplication instructions do not update the DC, N, Z, V, GT, or any condition bit of the DSR register.

The overflow prevention function is valid for DSP unit multiplication. Specify it by setting the S bit of the SR register is set to 1. When an overflow or underflow occurs, the operation result value is the maximum or minimum value respectively. In DSP unit fixed decimal point multiplication, overflows only occur for H'8000 × H'8000 ((–1.0) × (–1.0)). When the S bit is 0, the operation result is H'80000000, which means –1.0 rather than the correct answer of +1.0. When the S bit is 1, the overflow prevention function is engaged and the result is H'007FFFFFFF.

5.5 Shift Operations

The amount of shift in shift operations is specified either through a register or using a direct immediate value. Other source operands and destination operands are registers. There are two types of shift operations: arithmetic and logical. Table 5-9 shows the operation types. The correspondence between operands and registers is the same as for ALU fixed decimal point operations, except for immediate operands. The correspondence is shown in table 5-10.

Table 5-9 Types of Shift Operations

Mnemonic	Function	Source 1	Source 2	Destination
PSHA Sx, Sy, Dz	Arithmetic shift	Sx	Sy	Dz
PSHL Sx, Sy, Dz	Logical shift	Sx	Sy	Dz
PSHA #Imm, Dz	Arithmetic shift with immediate data	Dz	Imm1	Dz
PSHL #Imm, Dz	Logical shift with immediate data	Dz	Imm1	Dz

–32 ≤ Imm1 ≤ +32, –16 ≤ Imm2 ≤ +16

Table 5-10 Correspondence between Operands and Registers for Shift Operations

Operand	X0	X1	Y0	Y1	M0	M1	A0	A1
Sx	Yes	Yes					Yes	Yes
Sy			Yes	Yes	Yes	Yes		
Dz	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes

Note: Yes: Register can be used with operand.

5.5.1 Arithmetic Shift Operations

Function: ALU arithmetic shift operations basically work with a 32-bit unit to which 8 guard bits are added for a total of 40 bits. ALU fixed decimal point operations are basically performed between registers. When the source operand has no guard bits, the register's sign bit is copied to the guard bits. When the destination operand has no guard bits, the lower 32 bits of the operation result are stored in the destination register.

In arithmetic shifts, all bits of the source 1 operand and destination operand are valid. The source 2 operand, which specifies the shift amount, is integer data. The source 2 operand is specified as a register or immediate operand. The valid amount of shift is –32 to +32. Negative values are shifts to the right; positive values are shifts to the left. Between –64 and +63 can be specified for the source 2 operand, but only –32 to +32 is valid. When an invalid number is specified, the results cannot be guaranteed. When an immediate value is specified for the shift amount, the source 1 operand must be the same as the destination operand. The action of the operation is the same as for fixed decimal point operations and is executed in the DSP stage (the last stage) of the pipeline.

Whenever an arithmetic shift operation is executed, the DSR register's DC, N, Z, V, and GT bits are basically updated by the operation result. This is the same as for ALU fixed decimal point operations. For conditional instructions, condition bits are not updated even when the specified condition is achieved and the instruction executed. For unconditional instructions, the bits are always updated according to the operation result.

Figure 5-9 shows the arithmetic shift operation flowchart.

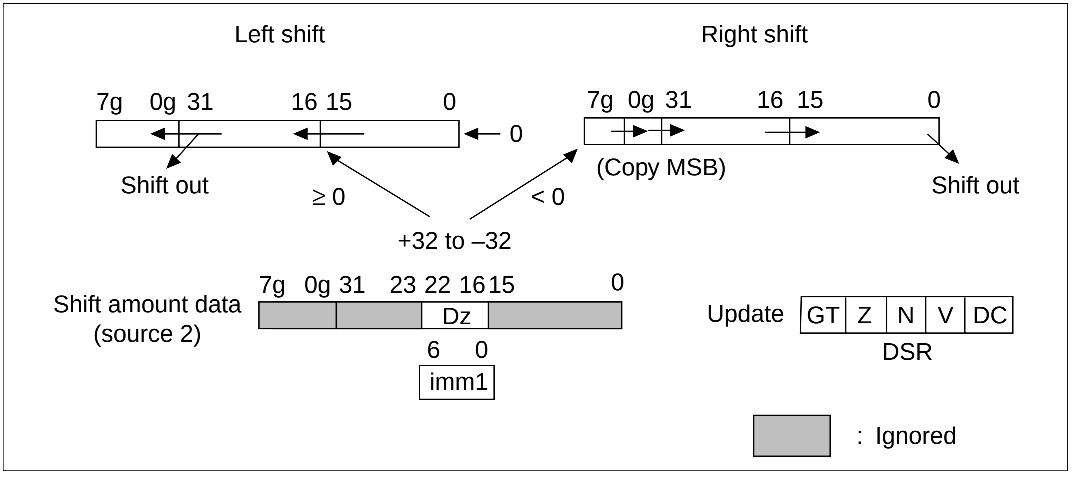


Figure 5-9 Arithmetic Shift Operation Flowchart

DC Bit: The DC bit is set as follows depending on the mode specified by the CS bits:

- Carry/Borrow Mode: CS2–CS0 = 000: The DC bit is the operation result, the value of the bit pushed out by the last shift.
- Negative Mode: CS2–CS0 = 001: Set to 1 for a negative operation result and 0 for a positive operation result. In this mode, the DC bit has the same value as bit N.

- Zero Mode: CS2–CS0 = 010: The DC bit is 1 when the operation result is zero; otherwise, the DC bit is 0. In this mode, the DC bit has the same value as bit Z.
- Overflow Mode: CS2–CS0 = 011: The DC bit is set to 1 by an overflow. In this mode, the DC bit has the same value as bit V.
- Signed Greater Than Mode: CS2–CS0 = 100: The DC bit is always 0. In this mode, the DC bit has the same value as bit GT.
- Signed Greater Than or Equal To Mode: CS2–CS0 = 101: The DC bit is always 0.

Condition Bits: The condition bits are set as follows:

- The N bit is the same as the result of the ALU fixed decimal point arithmetic operation. It is set to 1 for a negative operation result and 0 for a positive operation result.
- The Z bit is the same as the result of the ALU fixed decimal point arithmetic operation. It is set to 1 when the operation result is zero; otherwise, the Z bit is 0.
- The V bit is the same as the result of the ALU fixed decimal point arithmetic operation. It is set to 1 for an overflow.
- The GT bit is always 0.

Overflow Prevention Function (Saturation Operation): When the S bit of the SR register is set to 1, the overflow prevention function is engaged for the ALU fixed decimal point arithmetic operation executed by the DSP unit. When the operation result overflows, the maximum (positive) or minimum (negative) value is stored.

5.5.2 Logical Shift Operations

Function: Logical shift operations use the top words of the source 1 operand and the destination operand. As in ALU logical operations, the guard bits and bottom word of the operands are ignored. The source 2 operand, which specifies the shift amount, is integer data. The source 2 operand is specified as a register or immediate operand. The valid amount of shift is –16 to +16. Negative values are shifts to the right; positive values are shifts to the left. Between –32 and +31 can be specified for the source 2 operand, but only –16 to +16 is valid. When an invalid number is specified, the results cannot be guaranteed. When an immediate value is specified for the shift amount, the source 1 operand must be the same as the destination operand. The action of the operation is the same as for fixed decimal point operations and is executed in the DSP stage (the last stage) of the pipeline.

Whenever a logical shift operation is executed, the DSR register's DC, N, Z, V, and GT bits are basically updated by the operation result. This is the same as for ALU logical operations. For conditional instructions, condition bits are not updated even when the specified condition is achieved and the instruction executed. For unconditional instructions, the bits are always updated according to the operation result.

Figure 5-10 shows the logical shift operation flowchart.

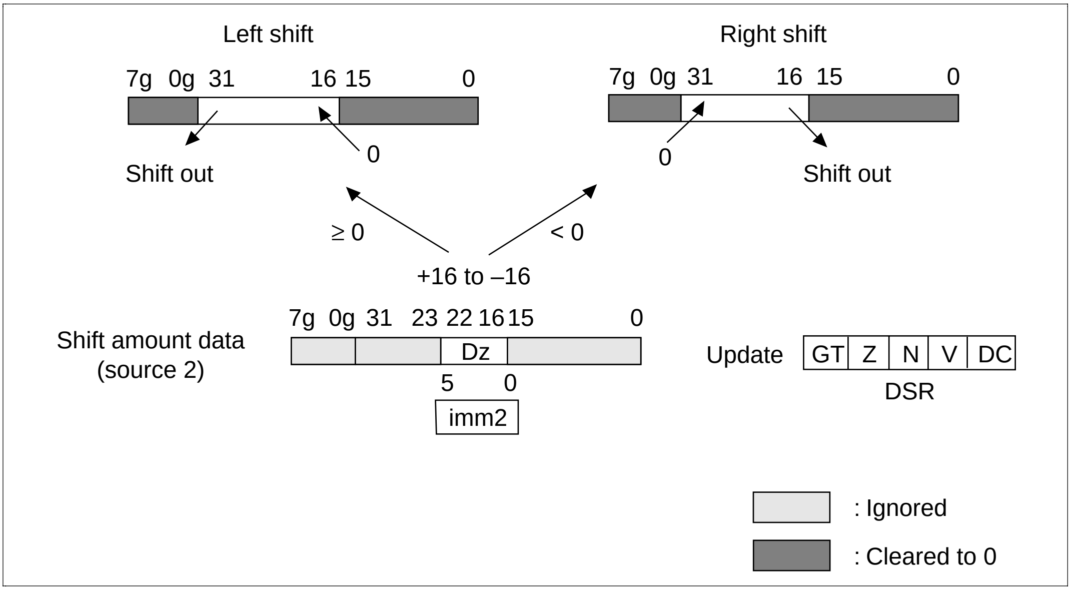


Figure 5-10 Logical Shift Operation Flowchart

DC Bit: The DC bit is set as follows depending on the mode specified by the CS bits.

- Carry/borrow mode: CS2–CS0 = 000: The DC bit is the operation result, the value of the bit pushed out by the last shift.
- Negative Mode: CS2–CS0 = 001: In this mode, the DC bit is the same as the bit 31 of the operation result. In this mode, the DC bit has the same value as bit N.
- Zero Mode: CS2–CS0 = 010: The DC bit is 1 when the operation result is all zeros; otherwise, the DC bit is 0. In this mode, the DC bit has the same value as bit Z.
- Overflow Mode: CS2–CS0 = 011: The DC bit is always 0. In this mode, the DC bit has the same value as bit V.
- Signed Greater Than Mode: CS2–CS0 = 100: The DC bit is always 0. In this mode, the DC bit has the same value as bit GT.
- Signed Greater Than Or Equal To Mode: CS2–CS0 = 101: The DC bit is always 0.

Condition Bits: The condition bits are set as follows.

- The N bit is the same as the result of the ALU logical operation. It is set to the value of bit 31 of the operation result.
- The Z bit is the same as the result of the ALU logical operation. It is set to 1 when the operation result is all zeros; otherwise, the Z bit is 0.
- The V bit is always 0.
- The GT bit is always 0.

5.6 The MSB Detection Instruction

5.6.1 Function

The MSB detection instruction (PDMSB: most significant bit detection) finds the amount of shift for normalizing the data.

The operation result is the same as for ALU integer operations. Basically, the top 16 bits and 8 guard bits are valid for a total 24 bits. When the destination operand is a register that has no guard bits, it is stored in the top 16 bits of the destination register.

The MSB detection instruction works on all bits of the source operand, but gets its operation result in integer data. This is because the shift amount for normalization must be integer data for the arithmetic shift operation. The action of the operation is the same as for fixed decimal point operations and is executed in the DSP stage (the last stage) of the pipeline.

Whenever a PDMSB instruction is executed, the DSR register's DC, N, Z, V, and GT bits are basically updated by the operation result. For conditional instructions, condition bits are not updated even when the specified condition is achieved and the instruction executed. For unconditional instructions, the bits are always updated according to the operation result.

Figure 5-11 shows the MSB detection instruction flowchart. Table 5-11 shows the relationship between source data and destination data.

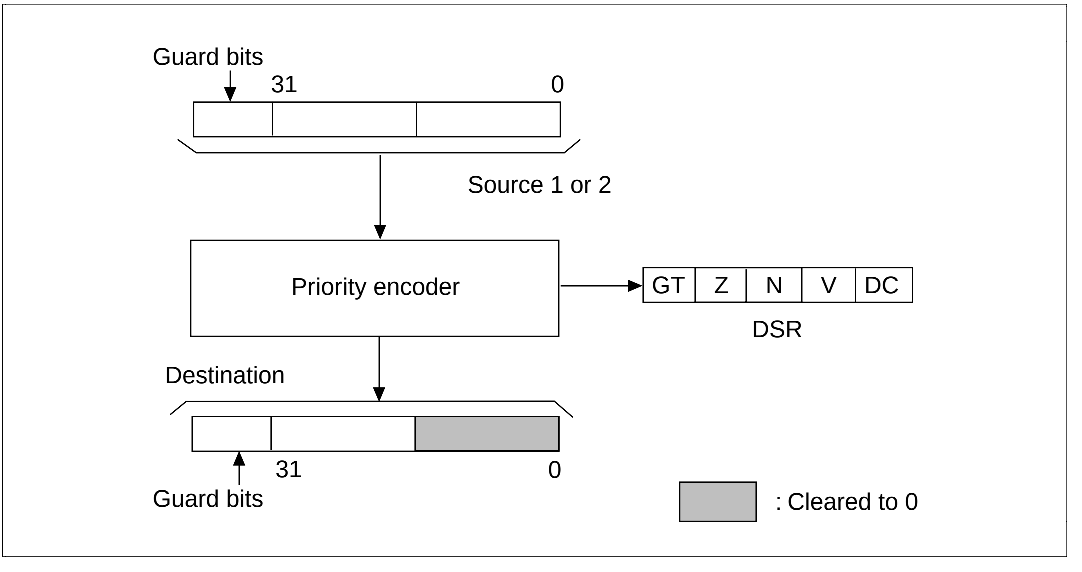


Figure 5-11 MSB Detection Flowchart

Table 5-11 Relationship between Source Data and Destination Data

Source Data

Guard Bits					Top Word					Bottom Word				
7g	**6g**	**5g–2g**	**1g**	**0g**	**31**	**30**	**29**	**28**	**27–4**	**27–4**	**3**	**2**	**1**	**0**
0	0	—	0	0	0	0	0	0	—	—	0	0	0	0
0	0	—	0	0	0	0	0	0	—	—	0	0	0	1
0	0	—	0	0	0	0	0	0	—	—	0	0	1	*
0	0	—	0	0	0	0	0	0	—	—	0	1	*	*
		↓					↓					↓		
0	0	—	0	0	0	0	0	1	—	—	*	*	*	*
0	0	—	0	0	0	0	1	*	—	—	*	*	*	*
0	0	—	0	0	0	1	*	*	—	—	*	*	*	*
0	0	—	0	0	1	*	*	*	—	—	*	*	*	*
0	0	—	0	1	*	*	*	*	—	—	*	*	*	*
		↓					↓					↓		
0	1	—	*	*	*	*	*	*	—	—	*	*	*	*
1	0	—	*	*	*	*	*	*	—	—	*	*	*	*
		↓					↓					↓		
1	1	—	1	0	*	*	*	*	—	—	*	*	*	*
1	1	—	1	1	0	*	*	*	—	—	*	*	*	*
1	1	—	1	1	1	0	*	*	—	—	*	*	*	*
1	1	—	1	1	1	1	0	*	—	—	*	*	*	*
1	1	—	1	1	1	1	1	0	—	—	*	*	*	*
		↓					↓					↓		
1	1	—	1	1	1	1	1	1	—	—	1	0	*	*
1	1	—	1	1	1	1	1	1	—	—	1	1	0	*
1	1	—	1	1	1	1	1	1	—	—	1	1	1	0
1	1	—	1	1	1	1	1	1	—	—	1	1	1	1

Table 5-11 Relationship between Source Data and Destination Data (cont)

		Destination Result						
Guard Bits	**Top word**							
7g–0g	**31–22**	**21**	**20**	**19**	**18**	**17**	**16**	**10 Hexadecimal**
all 0	all 0	0	1	1	1	1	1	+31
		0	1	1	1	1	0	+30
		0	1	1	1	0	1	+29
		0	1	1	1	0	0	+28
↓	↓				↓			↓
all 0	all 0	0	0	0	0	1	0	+2
		0	0	0	0	0	1	+1
		0	0	0	0	0	0	0
all 1	all 1	1	1	1	1	1	1	–1
		1	1	1	1	1	0	–2
↓	↓				↓			↓
all 1	all 1	1	1	1	0	0	0	–8
		1	1	1	0	0	0	–8
↓	↓				↓			↓
all 1	all 1	1	1	1	1	1	0	–2
		1	1	1	1	1	1	–1
all 0	all 0	0	0	0	0	0	0	0
		0	0	0	0	0	1	+1
		0	0	0	0	1	0	+2
↓	↓				↓			↓
all 0	all 0	0	1	1	1	0	0	+28
		0	1	1	1	0	1	+29
		0	1	1	1	1	0	+30
		0	1	1	1	1	1	+31

Note: Don’t care bits have no effect.

5.6.2 Instructions and Operands

Table 5-12 shows the MSB detection instruction. The correspondence between the operands and registers is the same as for ALU fixed decimal point operations. It is shown in table 5-13.

Table 5-12 MSB Detection Instruction

Mnemonic	Function	Source 1	Source 2	Destination
PDMSB	MSB detection	Sx	—	Dz
		—	Sy	Dz

Table 5-13 Correspondence between Operands and Registers for MSB Detection Instructions

Operand	X0	X1	Y0	Y1	M0	M1	A0	A1
Sx	Yes	Yes					Yes	Yes
Sy			Yes	Yes	Yes	Yes		
Dz	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes

Note: Yes: Register can be used with operand.

5.6.3 DC Bit

The DC bit is set as follows depending on the mode specified by the CS bits:

Carry/Borrow Mode: CS2–CS0 = 000: The DC bit is always 0.

Mode: CS2–CS0 = 001: Set to 1 for a negative operation result and 0 for a positive operation result. In this mode, the DC bit has the same value as bit N.

Zero Mode: CS2–CS0 = 010: The DC bit is 1 when the operation result is zero; otherwise, the DC bit is 0. In this mode, the DC bit has the same value as bit Z.

Overflow Mode: CS2–CS0 = 011: The DC bit is always 0. In this mode, the DC bit has the same value as bit V.

Signed Greater Than Mode: CS2–CS0 = 100: Set to 1 for a positive operation result and 0 for a negative operation result. In this mode, the DC bit has the same value as bit GT.

Signed Greater Than or Equal To Mode: CS2–CS0 = 101: Set to 1 for a positive or zero operation result and 0 for a negative operation result.

5.6.4 Condition Bits

The condition bits are set as follows.

- The N bit is the same as the result of the ALU integer operation. It is set to 1 for a negative operation result and 0 for a positive operation result.

- The Z bit is the same as the result of the ALU integer operation. It is set to 1 when the operation result is zero; otherwise, the Z bit is 0.
- The V bit is always 0.
- The GT bit is the same as the result of the ALU integer operation. It is set 1 for a positive operation result and otherwise to 0.

5.7 Rounding

5.7.1 Operation Function

The DSP unit has a function for rounding 32-bit values to 16-bit values. When the value has guard bits, 40 bits are rounded to 24 bits. When the rounding instruction is executed, H'0000 8000 is added to the source operand and the bottom word is then cleared to zeros.

Rounding uses all bits of the source and destination operands. The action of the operation is the same as for fixed decimal point operations and is executed in the DSP stage (the last stage) of the pipeline.

The rounding instruction is unconditional. The DSR register's DC, N, Z, V, and GT bits are thus always updated according to the operation result.

Figure 5-12 shows the rounding flowchart. Figure 5-13 shows the rounding process definitions.

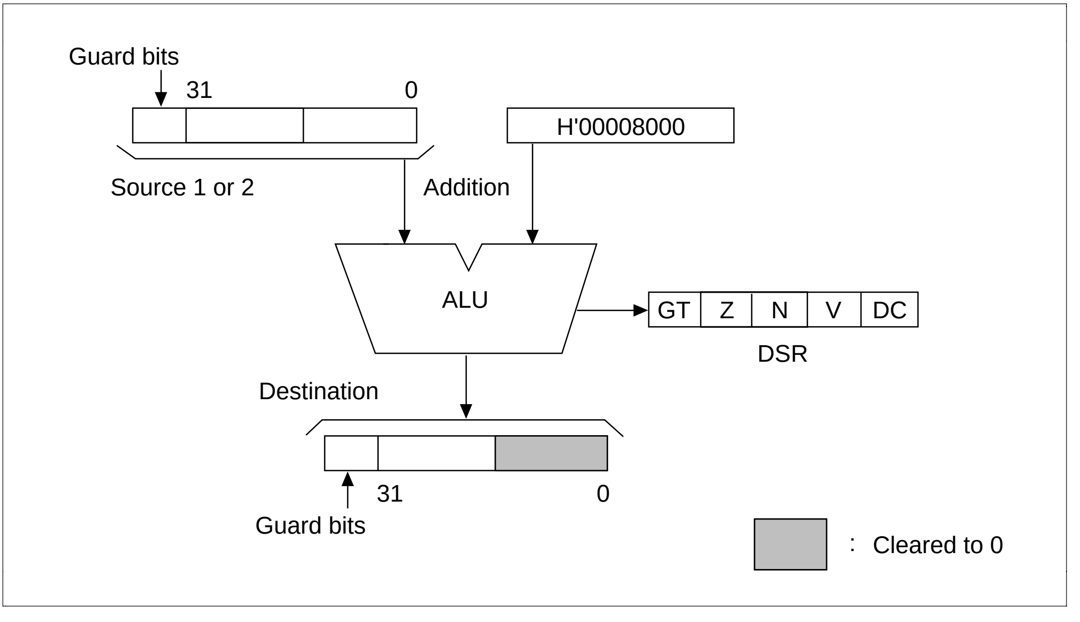


Figure 5-12 Rounding Flowchart

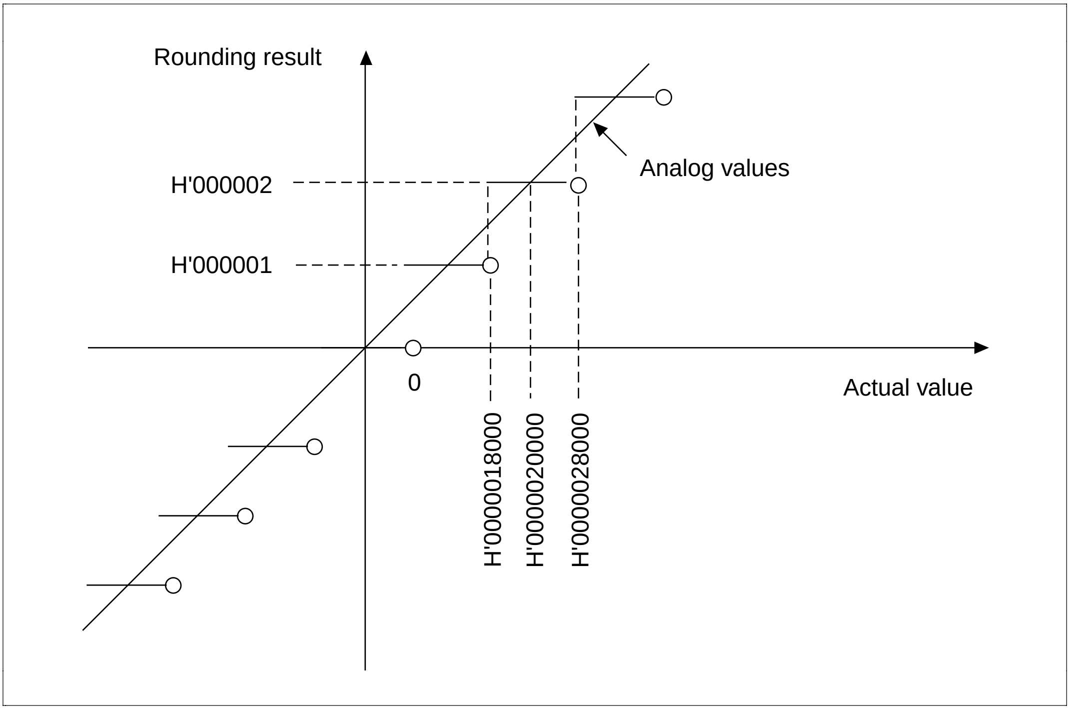


Figure 5-13 Rounding Process Definitions

5.7.2 Instructions and Operands

Table 5-14 shows the instruction. The correspondence between the operands and registers is the same as for ALU fixed decimal point operations. It is shown in table 5-15.

Table 5-14 Rounding Instruction

Mnemonic	Function	Source 1	Source 2	Destination
PRND	Rounding	Sx	—	Dz
		—	Sy	Dz

Table 5-15 Correspondence between Operands and Registers for Rounding Instruction

Operand	X0	X1	Y0	Y1	M0	M1	A0	A1
Sx	Yes	Yes					Yes	Yes
Sy			Yes	Yes	Yes	Yes		
Dz	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes

Note: Yes: Register can be used with operand.

5.7.3 DC Bit

The DC bit is updated as follows depending on the mode specified by the CS bits. Condition bits are updated as for ALU fixed decimal point arithmetic operations.

Carry/Borrow Mode: CS2–CS0 = 000: The DC bit is set to 1 when a carry or borrow from the MSB of the operation result occurs; otherwise, it is set to 0.

Negative Mode: CS2–CS0 = 001: Set to 1 for a negative operation result and 0 for a positive operation result. In this mode, the DC bit has the same value as bit N.

Zero Mode: CS2–CS0 = 010: The DC bit is 1 when the operation result is zero; otherwise, the DC bit is 0. In this mode, the DC bit has the same value as bit Z.

Overflow Mode: CS2–CS0 = 011: The DC bit is set to 1 by an overflow; otherwise, it is set to 0. In this mode, the DC bit has the same value as bit V.

Signed Greater Than Mode: CS2–CS0 = 100: Set to 1 for a positive operation result; otherwise, it is set to 0. In this mode, the DC bit has the same value as bit GT.

Signed Greater Than or Equal To Mode: CS2–CS0 = 101: Set to 1 for a positive or zero operation result; otherwise, it is set to 0..

5.7.4 Condition Bits

The condition bits are set as follows. They are updated as for ALU fixed decimal point arithmetic operations.

- The N bit is the same as the result of the ALU fixed decimal point arithmetic operation. It is set to 1 for a negative operation result and 0 for a positive operation result.
- The Z bit is the same as the result of the ALU fixed decimal point arithmetic operation. It is set to 1 when the operation result is zero; otherwise, the Z bit is 0.
- The V bit is the same as the result of the ALU fixed decimal point arithmetic operation. It is set to 1 for an overflow; otherwise, the V bit is 0.
- The GT bit is the same as the result of the ALU fixed decimal point arithmetic operation and the ALU integer operation. It is set 1 for a positive operation result; otherwise, the GT bit is 0.

5.7.5 Overflow Prevention Function (Saturation Operation)

When the S bit of the SR register is set to 1, the overflow prevention function can be specified for all rounding processing executed by the DSP unit. When the operation result overflows, the maximum (positive) or minimum (negative) value is stored.

5.8 Condition Select Bits (CS) and the DSP Condition Bit (DC)

DSP instructions may be either conditional or unconditional. Unconditional instructions are executed without regard to the DSP condition bit (DC bit), but conditional instructions may reference the DC bit before they are executed. With unconditional instructions, the DSR register's DC bit and condition bits (N, Z, V, and GT) are updated according to the results of the ALU operation or shift operation. The DC bit and condition bits (N, Z, V, and GT) are not updated regardless of whether the conditional instruction is executed. The DC bit is updated according to the specifications of the condition select (CS) bits. Updates differ for arithmetic operations, logical operations, arithmetic shifts and logical shifts. Table 5-16 shows the relationship between the CS bits and the DC bit.

Table 5-16 Condition Select Bits (CS) and DSP Condition Bit (DC)

CS Bits				
2	**1**	**0**	**Condition Mode**	**Description**
0	0	0	Carry/borrow	The DC bit is set to 1 when a carry or borrow occurs in the result of an ALU arithmetic operation. Otherwise, it is cleared to 0. In logical operations, the DC bit is always cleared to 0. For shift operations (the PSHA and PSHL instructions), the bit shifted out last is copied to the DC bit.
0	0	1	Negative	In ALU arithmetic operations or arithmetic shifts (PSHA), the MSB of the result (including the guard bits) is copied to the DC bit. In ALU logical operations and logical shifts (PSHL), the MSB of the result (not including the guard bits) is copied to the DC bit.
0	1	0	Zero	When the result of an ALU or shift operation is all zeros (0), the DC bit is set to 1. Otherwise, it is cleared to 0.
0	1	1	Overflow	In ALU arithmetic operations or arithmetic shifts (PSHA), when the operation result (not including the guard bits) exceeds the destination register's value range, the DC bit is set to 1. Otherwise, it is cleared to 0. In ALU logical operations and logical shifts (PSHL), the DC bit is always cleared to 0.
1	0	0	Signed greater than	This mode is like the Greater Than Or Equal To mode, but the DC bit is cleared to 0 when the operation result is zero (0). When the operation result (including the guard bits) exceeds the expressible limits, the TRUE condition is VR. DC bit = ~{(N bit ^ VR)|Z bit)}; for arithmetic operations DC bit = 0; for logical operations
1	0	1	Greater than or equal to	In ALU arithmetic operations or arithmetic shifts (PSHA), when the result does not overflow, the value is the inversion of the negative mode's DC bit. When the operation result (including the guard bits) exceeds the expressible limits, the value is the same as the negative mode's DC bit. In ALU logical operations and logical shifts (PSHL), the DC bit is always cleared to 0. DC bit = ~(N bit ^ VR)); for arithmetic operations DC bit = 0; for logical operations
1	1	0	Reserved	
1	1	1		

5.9 Overflow Prevention Function (Saturation Operation)

The overflow prevention function (saturation operation) is specified by the S bit of the SR register. This function is valid for arithmetic operations executed by the DSP unit and multiply and accumulate operations executed by the CPU core. An overflow occurs when the operation result exceeds the bounds that can be expressed as a two's complement (not including the guard bits).

Table 5-17 shows the overflow definitions for fixed decimal point arithmetic operations. Table 5-18 shows the overflow definitions for integer arithmetic operations. Multiply/Accumulate calculation instructions (MAC) supported by previous SuperH RISC engines are performed on 64-bit registers (MACH and MACL), so the overflow value differs from the maximum and minimum values. They are defined exactly the same as before.

Table 5-17 Overflow Definitions for Fixed Decimal Point Arithmetic Operations

Sign	Overflow Condition	Maximum/ Minimum	Hexadecimal Display
Positive	Result > $1–2^{–31}$	$1–2^{–31}$	007FFFFFFF
Negative	Result < –1	–1	FF80000000

Table 5-18 Overflow Definitions for Integer Arithmetic Operations

Sign	Overflow Condition	Maximum/ Minimum	Hexadecimal Display
Positive	Result > $2^{–15}$ – 1	$2^{–15}$ – 1	007FFF****
Negative	Result < $–2^{–15}$	$–2^{–15}$	FF8000****

Note: Don't care bits have no effect.

When the overflow prevention function is specified, overflows do not occur. Naturally, the overflow bit (V bit) is not set. When the CS bits specify overflow mode, the DC bit is not set either.

5.10 Data Transfers

The SH3-DSP can perform up to two data transfers in parallel between the DSP register and on-chip memory with the DSP unit. The SH-DSP has the following types of data transfers:

1. X and Y memory data transfers: Data transfer to X and Y memory using the XDB and YDB buses

- Double data transfer: Data transfer only, where transfer in one direction only is permitted
- Parallel data transfers: Data transfer that proceeds in parallel to ALU operation processing

2. Single data transfers: Data transfer to on-chip memory using the LDB bus

Note: Data transfer instructions do not update the DSR register's condition bits.

Table 5-19 shows the various functions.

Table 5-19 Data Transfer Functions

Category	Bus	Length	Parallel Processing with ALU Operation	Parallel Processing with Data Transfer	Instruction Length
X and Y memory data transfer	XDB bus YDB bus	16 bits	None (double)	None (XDB or YDB bus)	16 bits
				Available (XDB and YDB bus)	16 bits
			Available (parallel)	None (XDB or YDB bus)	32 bits
				Available (XDB and YDB bus)	32 bits
Single data transfer	LDB bus	32 bits 16 bits	None	None	16 bits

5.10.1 X and Y Memory Data Transfer

X and Y memory data transfers allow two data transfers to be executed in parallel and allow data transfers to be executed in parallel with DSP data operations. 32-bit instruction code is required for executing DSP data operations and transfers in parallel. This is called a parallel data transfer. When executing an X and Y memory data transfer by itself, 16-bit instruction code is used. This is called a double data transfer.

Data transfers consist of X memory data transfers and Y memory data transfers. X memory data is loaded to either the X0 or X1 register; Y memory data is loaded to the Y0 or Y1 register. The X0, X1, Y0, and Y1 registers become the destination registers. Data can be stored in the X and Y memory if the A0 or A1 register is the source register. All these data transfers involve word data (16 bits). Data is transferred from the top word of the source register. Data is transferred to the top word of the destination register and the bottom word is automatically cleared with zeros.

Specifying a conditional instruction as the operation instruction executed in parallel has no effect on the data transfer instructions.

X and Y memory data transfers access only the X and Y memory; they cannot access other memory areas.

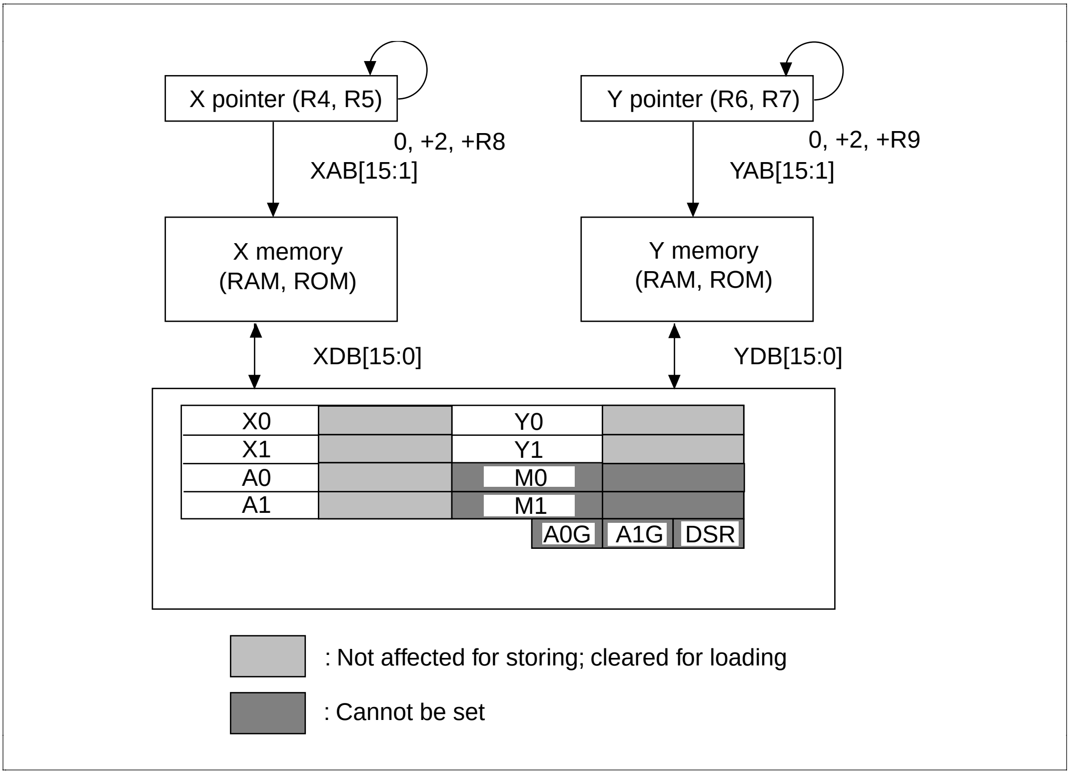


Figure 5-14 Flowchart of X and Y Memory Data Transfers

5.10.2 Single Data Transfers

Single data transfers execute only one data transfer. They use 16-bit instruction code. Single data transfers cannot be processed in parallel with ALU operations. The X pointer, which accesses X memory, and two added pointers are valid; the Y pointer is not valid. As with the SuperH RISC engine, single data transfers can access all memory areas, including external memory. Except for the DSR register, the DSP registers can be specified as source and destination operands. (The DSR register is defined as the system register, so it can transfer data with LDS and STS instructions.) The guard bit registers A0G and A1G can be specified for operands as independent registers. Single data transfers use the LAB and LDB buses in place of the XAB, XDB, YAB, and YDB buses, so contention occurs on the LDB bus between data transfers and instruction fetches.

Single data transfers handle word and longword data. Word data transfers involve only the top word of the register. When data is loaded to a register, it goes to the top word and the bottom word is automatically filled with zeros. If there are guard bits, the sign bit is extended to fill them. When storing from a register, the top word is stored.

When a longword is transferred, 32 bits are valid. When loading a register that has guard bits, the sign bit is extended to fill the guard bits.

When a guard bit register is stored, the top 24 bits become undefined, and the read out is to the LDB bus. When the guard bit registers A0G and A1G load word data as the destination registers of the MOVS.W instruction, the bottom byte is written to the register.

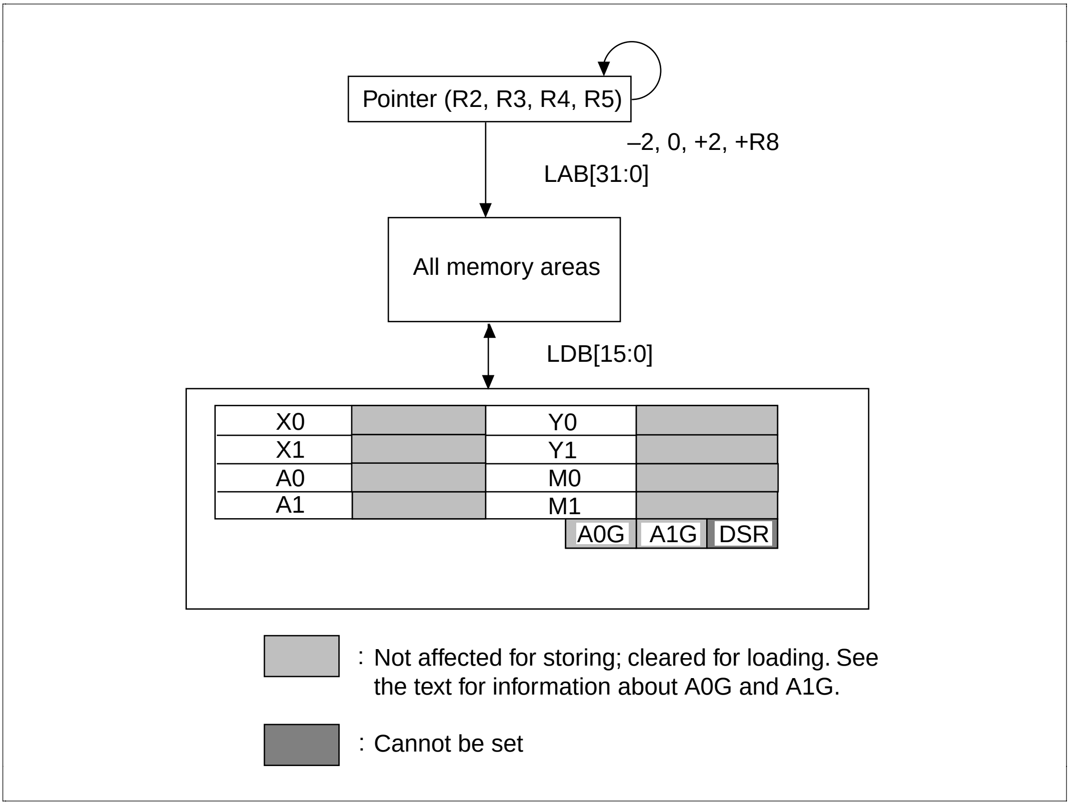


Figure 5-15 Single Data Transfer Flowchart (Word)

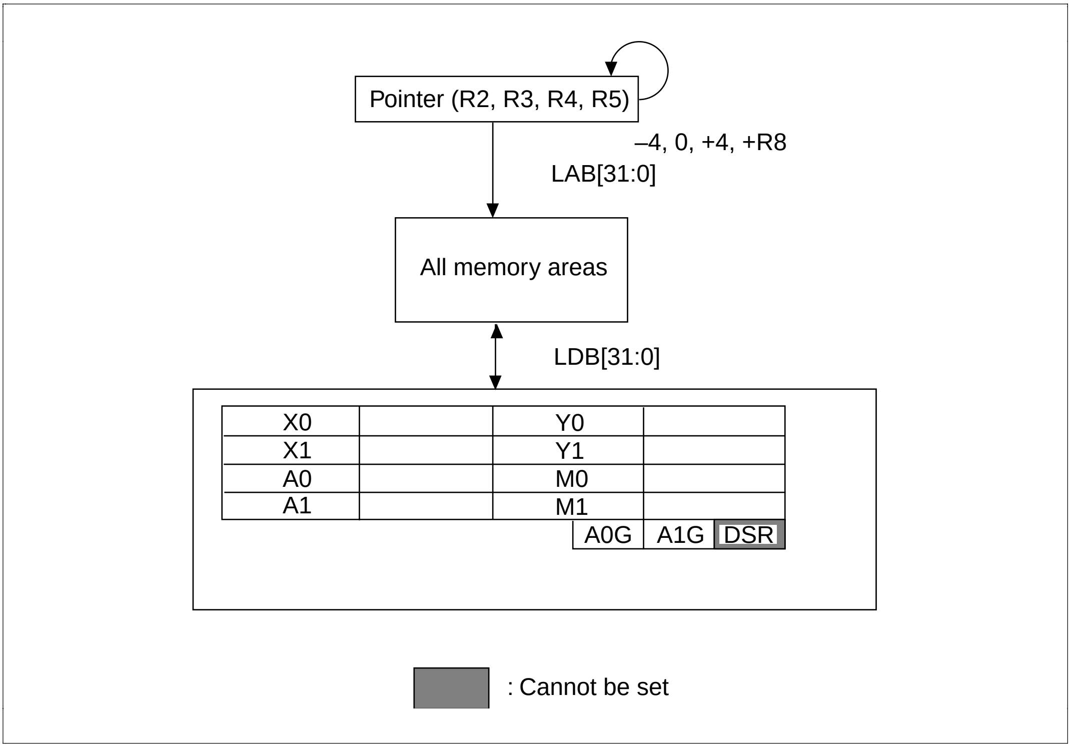


Figure 5-16 Single Data Transfer Flowchart (Longword)

Data transfers are executed in the MA stage of the pipeline while DSP operations are executed in the DSP stage. Since the next data store instruction starts before the data operation instruction has finished, a stall cycle is inserted when the store instruction comes on the instruction line after the data operation instruction. This overhead cycle can be avoided by adding one instruction between the data operation instruction and the data transfer instruction. Figure 5-17 shows an example.

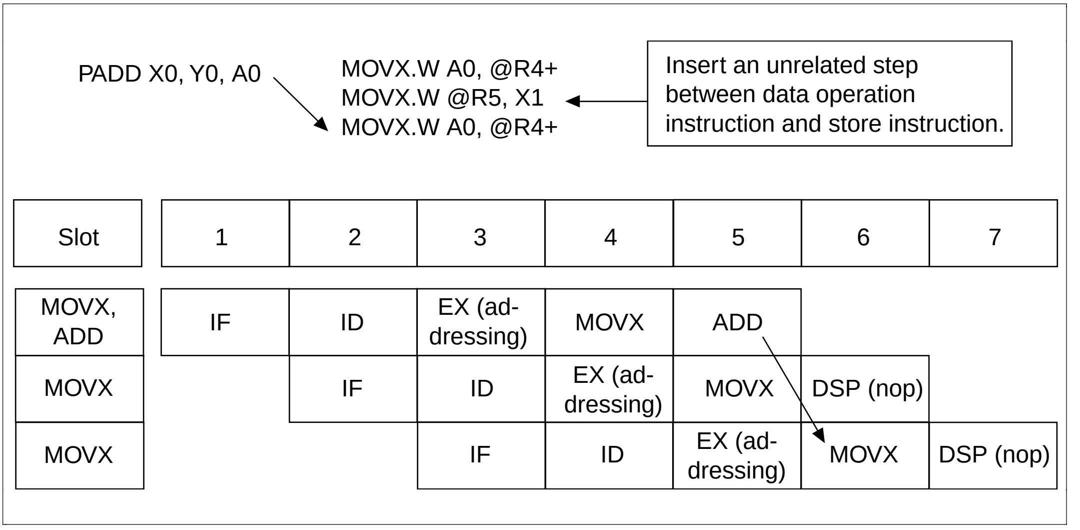


Figure 5-17 Example of the Execution of Operation and Data Store Instructions

5.11 Operand Contention

Data contention occurs when the same register is specified as the destination operand for two or more parallel processing instructions. It occurs in three cases.

1. When the same destination operand is specified for an ALU operation and multiplication (Du, Dg)
2. When the same destination operand is specified for an X memory load and an ALU operation (Dx, Du, Dz)
3. When the same destination operand is specified for a Y memory load and an ALU operation (Dx, Du, Dz)

Results cannot be guaranteed when contention occurs. Table 5-20 shows the operand and register combinations that cause contention.

Some assemblers can detect these types of contention, so pay attention to assembler functions when selecting one.

Table 5-20 Operand and Register Combinations That Create Contention

		DSP Register							
Operation	**Operand**	**X0**	**X1**	**Y0**	**Y1**	**M0**	**M1**	**A0**	**A1**
X memory load	Ax								
	IX								
	Dx	*[2]	*[2]						
Y memory load	Ay								
	Iy								
	Dy			*[3]	*[3]				
6-operand ALU operation	Sx	*[1]	*[1]					*[1]	*[1]
	Sy			*[1]	*[1]	*[1]	*[1]		
	Du	*[2]		*[3]				*[4]	*[4]
3-operand multiplication	Se	*[1]	*[1]	*[1]					*[1]
	Sf	*[1]		*[1]	*[1]				*[1]
	Dg					*[1]	*[1]	*[4]	*[4]
3-operand ALU operation	Sx	*[1]	*[1]					*[1]	*[1]
	Sy			*[1]	*[1]	*[1]	*[1]		
	Dz	*[2]	*[2]	*[3]	*[3]	*[1]	*[1]	*[1]	*[1]

Notes: 1. Register is settable for the operand
2. Dx, Du, and Dz contend
3. Dy, Du, and Dz contend
4. Du and Dg contend

5.12 DSP Repeat (Loop) Control

The SH3-DSP repeat (loop) control function is a special utility for controlling repetition efficiently. The SETRC instruction is executed to hold a repeat count in the repeat counter (RC, 12 bits) and set an execution mode in which the repeat (loop) program is repeated until the RC is 1. Upon completion of the repeat operation, the content of the RC becomes 0.

The repeat start register (RS) holds the start address of the repeated section. The repeat end register (RE) holds the ending address of the repeated section. (There are some exceptions. Refer to Note 1, Actual programming, in this section [below figure 5-18].) The repeat counter (RC) holds the repeat count. The procedure for executing repeat control is shown below:

1. Set the repeat start address in the RS register.
2. Set the repeat end address in the RE register.
3. Set the repeat count in the RC counter.

4. Execute the repeated program (loop).

The following instructions are used for executing 1 and 2:

```
LDRS @(disp,PC);
LDRE @(disp,PC);
```

The SETRC instruction is used to execute 3 and 4. Immediate data or a general register may be used to specify the repeat count as the operand of the SETRC instruction:

```
SETRC #imm;    #imm → Rc, enable repeat control
SETRC Rm;      Rm → Rc, enable repeat control
```

#imm is 8 bits and the RC counter is 12 bits, so to set the RC counter to a value of 256 or greater, use the Rm register. A sample program is shown below.

```
           LDRS   RptStart;
           LDRE   RptEnd;
           SETRC  #imm;      RC=#imm
           instr0;
; instr1~5 executes repeatedly
RptStart:  instrl;
           instr2;
           instr3;
           instr4;
RptEnd:    instr5;
           instr6;
```

There are several restrictions on repeat control:

1. At least one instruction must come between the SETRC instruction and the first instruction of the repeat program (loop).
2. Execute the SETRC instruction after executing the LDRS and LDRE instructions.
3. When there are more than four instructions for the repeat program (loop) and there is no repeat start address (in the above example, it was address instr1) at the long word boundary, one cycle stall (cycle awaiting execution) is required for each repeat.
4. When there are three or fewer instructions in the loop, branch instructions (BRA, BSR, BT, BF, BT/S, BF/S, BSRF, RTS, BRAF, RTE, JSR, JMP), repeat control instructions (SETRC, LDRS, LDRE), SR, RS, and RE load instructions, and TRAPA cannot be used. If such an instruction is used, illegal instruction exception handling starts and the address values shown in Table 5-21 are stored in SPC.

Table 5-21 PC Values Address Stored in SPC (1)

Conditions	Position	Address Stored in SPC
RC>=2	Any	RptStart
RC=1	Any	Program address of illegal instruction

5. If there are four or fewer instructions in the loop, branched instructions (BRA, BSR, BT, BF, BT/S, BF/S, BSRF, RTS, BRAF, RTE, JSR, JMP), repeat control instructions (SETRC, LDRS, LDRE), SR, RS, and RE load instructions, and TRAPA cannot be used for the last three instructions in the repeat program (loop). If such an instruction is used, illegal instruction exception handling starts and the address values shown in Table 5-22 are stored in SPC. In case of repeat control instruction (SETRC, LDRS, LDRE), and SR, RS, and RE load instructions, they cannot be described in positions other than the repeat module. If described, proper operation cannot be guaranteed.

Table 5-22 PC Values Address Stored in SPC (2)

Conditions	Position	Address Stored in SPC
RC>=2	instr3	Program address of illegal instruction
	instr4	RptStart-4
	instr5	RptStart-2
RC=1	Any	Program address of illegal instruction

6. When there are three or fewer instructions in the loop, PC relative instructions (MOVA (disp,PC), R0, or the like) can only be used at the first instruction (instr1).
7. If there are four or more instructions in the loop, PC relative instructions (MOVA (disp,PC), R0, or the like) cannot be used in the final two instructions.
8. The SH3-DSP does not have a repeat valid flag; repeats become invalid when the RC counter becomes 0. When the RC counter is not 0 and the PC counter matches the RE register contents, repeating begins. When the RC counter is set to 0, the repeat program (loop) is invalid but the loop is executed only once and does not return to the starting instruction of the loop as when RC is 1. When the RC counter is set to 1, the repeat module is executed only once. Though it does not return to the repeat program (loop) start instruction, the RC counter becomes zero when the repeat module is executed.
9. If there are four or more instructions in the loop, the branched instructions including the subroutine call back and return instructions cannot be used for the "inst3" through "inst5" instructions as branch destination address. If they are executed, the repeat control does not work correctly. If a repeating portion of a program (a loop) contains three or more instructions and the branching destination is RptStart or an address ahead of it, repeat control does not work properly and the content of RC in the SR register is not updated.

10. While the repeat is being executed, interruption is restricted. Figure 5-18 shows the flow for each stage of EX. The initial EX stage of interruption is usually started immediately after the EX stage of the instruction is completed (indicated by “A”). "B" in the figure below indicates locations where no interruption is accepted.

```
A:  Interruption is accepted.
B:  No interruption is accepted.


When RC>=1

1-step repeat                      2-step repeat                    3-step repeat

                      ← A                              ← A                             ← A
             instr0   ← B                     instr0   ← B                    instr0   ← B
Start(End):  instr1   ← B          Start:     instr1   ← B          Start:    instr1   ← B
             instr2   ← A          End:       instr2   ← B                    instr2   ← B
                                              instr3   ← A          End:      instr3   ← B
                                                                              instr4   ← A

More than 4 steps repeat

                          ← A
          instr0          ← A or B (when returning from instr n)
Start:    instr1          ← A
            :                :
            :             ← A
          instr n-3       ← B
          instr n-2       ← B
          instr n-1       ← B
End:      instr n         ← B
          instr n+1       ← A


When RC=0:  Interruption is accepted.
```

Figure 5-18 Restriction on Acceptance of Interruption by Repeat Module

5.12.1 Usage Notes

Note 1. Actual programming

The repeat start register (RS) and repeat end register (RE) store the repeat start address and repeat end address respectively. Addresses stored in these registers are changed depending on the number of instructions in the repeat program (loop). This rule is shown below.

Repeat_Start: Address of repeat start instruction

Repeat_Start0: Address of instruction one higher than the repeat start instruction

Repeat_Start3: Address of instruction three higher than the repeat end instruction

Table 5-23 RS and RE Setup Rule

	Number of Instructions in Repeat Program (Loop)			
Register	**1**	**2**	**3**	**>=4**
RS	Repeat_start0+8	Repeat_start0+6	Repeat_start0+4	Repeat_Start
RE	Repeat_start0+4	Repeat_start0+4	Repeat_start0+4	Repeat_End3+4

An example of an actual repeat program (loop) assuming various cases based on the above table is given below:

Case 1: One repeat instruction

```
            LDRS    RptStart0+8;
            LDRE    RptStart0+4;
            SETRC   RptCount;
                ----
RptStart0:  instr0;
RtpStart:   instr1; Repeat instruction
            instr2;
```

Case 2: Two repeat instructions

```
            LDRS    RptStart0+6;
            LDRE    RptStart0+4;
            SETRC   RptCount;
                ----
RptStart0:  instr0;
RtpStart:   instr1; Repeat instruction 1
RptEnd: instr2;     Repeat instruction 2
            instr3;
```

Case 3: Three repeat instructions

```
            LDRS    RptStart0+4;
            LDRE    RptStart0+4;
            SETRC   RptCount;
                ----
  RptStart0: instr0;
  RtpStart:  instr1;Repeat instruction 1
             instr2;Repeat instruction 2
  RptEnd:instr3;Repeat instruction 3
             instr4;
```

Case 4: Four or more instructions

```
            LDRS    RptStart;
            LDRE    RptStart3+4;
            SETRC   RptCount;
                ----
  RptStart0: instr0;
  RtpStart:  instr1;Repeat instruction 1
             instr2;Repeat instruction 2
             instr3;Repeat instruction 3
  ----------------------------------------
  RptEnd3:   instrN-3;  Repeat instruction N
             instrN-2;  Repeat instruction N-2
             instrN-1;  Repeat instruction N-1
  RptEnd:instrN;Repeat instruction N
             instrN+1
```

The above example can be used as a template when programming this repeat program (loop) sequence. Extension instruction “REPEAT” can simplify the problems of such complicated labeling and offset. Details are described in Note 2 below.

Note 2. Extension instruction REPEAT

The extension instruction REPEAT can simplify the handling of the labeling and offset described in Table 5-23. Labels used are shown below.

RptStart: RptStart: Address of first instruction of repeat program (loop)

RptEnd: Address of last instruction of repeat program (loop)

PptCount: Repeat count immediate No.

Use this instruction as described below.

Repeat count can be designated as immediate value #Imm or register indirect value Rn.

Case 1: One repeat instruction

```
REPEAT RptStart, RptEnd, RptCount
               ----
           instr0;
RptStart:  instr1;Repeat instruction 1
           instr2;
```

Case 2: Two repeat instructions

```
REPEAT RptStart, RptEnd, RptCount
               ----
           instr0;
RptStart:  instr1;Repeat instruction 1
RptEnd:instr2;Repeat instruction 2
```

Case 3: Three repeat instructions

```
REPEAT RptStart, RptEnd, RptCount
               ----
           instr0;
RptStart:  instr1;Repeat instruction 1
           instr2;Repeat instruction 2

RptEnd:instr3;Repeat instruction 3
```

Case 4: Four or more instructions

```
REPEAT RptStart, RptEnd, RptCount
               ----
           instr0;
RtpStart:  instr1;Repeat instruction 1
           instr2;Repeat instruction 2
           instr3;Repeat instruction 3
------------------------------------------
           instrN-3;  Repeat instruction N
           instrN-2;  Repeat instruction N-2
           instrN-1;  Repeat instruction N-1
RptEnd:instrN;Repeat instruction N
           instrN+1
```

Result of extension of each case corresponds to the case 1 in Note 1.

5.13 Conditional Instructions and Data Transfers

Data operation instructions include both unconditional and conditional instructions. Data transfer instructions that execute both in parallel can be specified, but they will always execute regardless of whether the condition is met without affecting the data transfer instruction.

The following is an example of a conditional instruction and a data transfer:

```
DCT PADD X0, Y0, A0  MOVX.W @R4+, X0  MOVY.W A0, @R6+R9;
```

When condition is true:

```
Before execution:  X0=H'33333333, Y0=H'55555555, A0=H'123456789A,
                   R4=H'00008000, R6=H'00008232, R1=H'00000004
                   (R4)=H'1111, (R6)=H'2222
After execution:   X0=H'11110000, Y0=H'55555555, A0=H'0088888888,
                   R4=H'00008002, R6=H'00008236, R1=H'00000004
                   (R4)=H'1111, (R6)=H'1234
```

When condition is false:

```
Before execution:  X0=H'33333333, Y0=H'55555555, A0=H'123456789A,
                   R4=H'00008000, R6=H'00008232, R1=H'00000004
                   (R4)=H'1111, (R6)=H'2222
After execution:   X0=H'11110000, Y0=H'55555555, A0=H'123456789A,
                   R4=H'00008002, R6=H'00008236, R1=H'00000004
                   (R4)=H'1111, (R6)=H'1234
```

Section 6 Instruction Features

6.1 RISC-Type Instruction Set

All instructions are RISC type. Their features are detailed in this section.

6.1.1 16-Bit Fixed Length

In the SH-3 CPU all instructions have a fixed length of 16 bits. This contributes to increased code efficiency.

Like SH-3, the SH-3DSP has 16-bit instructions, but additional 32-bit DSP instructions are provided to allow parallel processing of DSP instructions. For details on the DSP, see 5. DSP Operations and Data Transfer.

6.1.2 One Instruction/Cycle

Basic instructions can be executed in one cycle using the pipeline system.

6.1.3 Data Length

Longword is the standard data length for all operations. Memory can be accessed in bytes, words, or longwords. Byte or word data accessed from memory is sign-extended and handled as longword data (table 6-1). Immediate data is sign-extended for arithmetic operations or zero-extended for logic operations. It also is handled as longword data.

Table 6-1 Sign Extension of Word Data

SH-3/SH-3E/SH3-DSP CPU		Description	Example for Conventional CPU	
MOV.W	@(disp,PC),R1	Data is sign-extended to 32 bits, and R1 becomes H'00001234. It is next operated upon by an ADD instruction.	ADD.W	#H'1234,R0
ADD	R1,R0			
....				
.DATA.W	H'1234			

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

6.1.4 Load-Store Architecture

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

6.1.5 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 6-2).

Table 6-2 Delayed Branch Instructions

SH-3/SH-3E/SH3-DSP CPU		Description	Example for Conventional CPU	
BRA	TRGET	Executes an ADD before branching to TRGET.	ADD.W	R1,R0
ADD	R1,R0		BRA	TRGET

6.1.6 Multiplication/Accumulation Operation

Multiplication of two 16-bit values to produce a 32-bit result is executed in one to three cycles (one to two cycles for the SH3-DSP), and multiplication of two 32-bit values to produce a 64-bit result is executed in two to five cycles (two to three cycles for the SH3-DSP). Multiplication/accumulation, in which two 32-bit values are multiplied and one 32-bit value is added, is executed in two to five cycles (two to four cycles for the SH3-DSP) when the MAC instruction is used and in one system when the FMAC instruction* is used.

Note: The FMAC instruction is only available on the SH-3E (floating point calculation instruction).

6.1.7 T Bit

The T bit in the status register changes according to the result of the comparison, and in turn is the condition (true/false) that determines if the program will branch (table 6-3). The number of instructions after T bit in the status register is kept to a minimum to improve the processing speed.

Table 6-3 T Bit

SH-3/SH-3E/SH3-DSP CPU		Description	Example for Conventional CPU	
CMP/GE	R1,R0	T bit is set when R0 ≥ R1. The program branches to TRGET0 when R0 ≥ R1 and to TRGET1 when R0 < R1.	CMP.W	R1,R0
BT	TRGET0		BGE	TRGET0
BF	TRGET1		BLT	TRGET1
ADD	#–1,R0	T bit is not changed by ADD. T bit is set when R0 = 0. The program branches if R0 = 0.	SUB.W	#1,R0
CMP/EQ	#0,R0		BEQ	TRGET
BT	TRGET			

6.1.8 Immediate Data

Byte immediate data is located in instruction code. Word or longword immediate data is not input via instruction codes but is stored in a memory table. The memory table is accessed by an immediate data transfer instruction (MOV) using the PC relative addressing mode with displacement (table 6-4).

Table 6-4 Immediate Data Accessing

Classification	SH-3/SH-3E/SH3-DSP CPU	Example for Conventional CPU
8-bit immediate	`MOV #H'12,R0`	`MOV.B #H'12,R0`

16-bit immediate	`MOV.W @(disp,PC),R0`
	....
	`.DATA.W H'1234`
		`MOV.W #H'1234,R0`

32-bit immediate	`MOV.L @(disp,PC),R0`
	....
	`.DATA.L H'12345678`
		`MOV.L #H'12345678,R0`

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

6.1.9 Absolute Address

When data is accessed by absolute address, the value already in the absolute address is placed in the memory table. Loading the immediate data when the instruction is executed transfers that value to the register and the data is accessed in the indirect register addressing mode.

Table 6-5 Absolute Address

Classification	SH-3/SH-3E/SH3-DSP CPU	Example for Conventional CPU

Absolute address	`MOV.L @(disp,PC),R1`
	`MOV. B @R1,R0`
	....
	`.DATA.L H'12345678`
		`MOV.B @H'12345678,R0`

6.1.10 16-Bit/32-Bit Displacement

When data is accessed by 16-bit or 32-bit displacement, the pre-existing displacement value is placed in the memory table. Loading the 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 6-6 16-Bit/32-Bit Displacement

Classification	SH-3/SH-3E/SH3-DSP CPU	Example for Conventional CPU

16-bit displacement	`MOV.W @(disp,PC),R0`
	`MOV.W @(R0,R1),R2`
	....
	`.DATA.W H'1234`
		`MOV.W @(H'1234,R1),R2`

6.1.11 Privileged Instructions

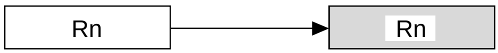
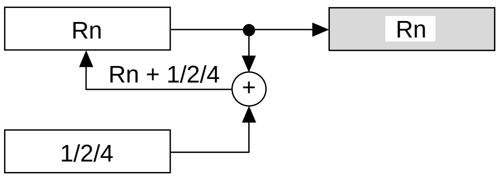
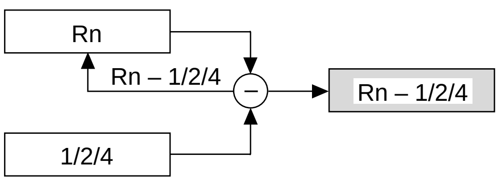
The processor has two operation modes (user/privileged). If these instructions are used in user mode, an illegal instruction exception is detected. Privileged instructions are:

- LDC
- STC
- RTE
- LDTLB
- SLEEP

6.2 CPU Instruction Addressing Modes

Addressing modes and effective address calculation are described in table 6-7.

Table 6-7 Addressing Modes and Effective Addresses

Addressing Mode	Instruction Format	Effective Addresses Calculation	Equation
Direct register addressing	Rn	The effective address is register Rn. (The operand is the contents of register Rn.)	—
Indirect register addressing	@Rn	The effective address is the content of register Rn. 	Rn
Post-increment indirect register addressing	@Rn +	The effective address is the content of register Rn. A constant is added to the content of Rn after the instruction is executed. 1 is added for a byte operation, 2 for a word operation, and 4 for a longword operation. 	Rn (After the instruction is executed) Byte: Rn + 1 → Rn Word: Rn + 2 → Rn Longword: Rn + 4 → Rn
Pre-decrement indirect register addressing	@–Rn	The effective address is the value obtained by subtracting a constant from Rn. 1 is subtracted for a byte operation, 2 for a word operation, and 4 for a longword operation. 	Byte: Rn – 1 → Rn Word: Rn – 2 → Rn Longword: Rn – 4 → Rn (Instruction executed with Rn after calculation)

[shaded box] : Effective address

Table 6-7 Addressing Modes and Effective Addresses (cont)

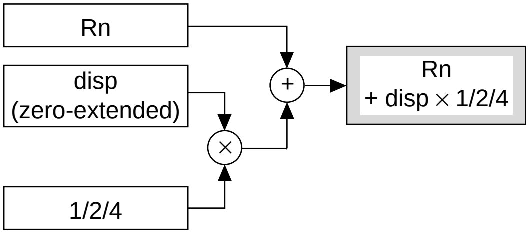
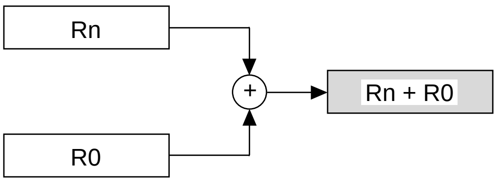
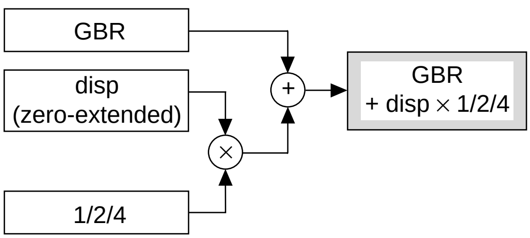
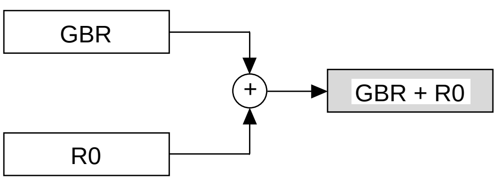
Addressing Mode	Instruction Format	Effective Addresses Calculation	Equation
Indirect register addressing with displace-ment	@(disp:4, Rn)	The effective address is Rn plus a 4-bit displacement (disp). The value of disp is zero-extended, and remains the same for a byte operation, is doubled for a word operation, and is quadrupled for a longword operation. 	Byte: Rn + disp Word: Rn + disp × 2 Longword: Rn + disp × 4
Indirect indexed register addressing	@(R0, Rn)	The effective address is the Rn value plus R0. 	Rn + R0
Indirect GBR addressing with displace-ment	@(disp:8, GBR)	The effective address is the GBR value plus an 8-bit displacement (disp). The value of disp is zero-extended, and remains the same for a byte operation, is doubled for a word operation, and is quadrupled for a longword operation. 	Byte: GBR + disp Word: GBR + disp × 2 Longword: GBR + disp × 4
Indirect indexed GBR addressing	@(R0, GBR)	The effective address is the GBR value plus the R0. 	GBR + R0

Table 6-7 Addressing Modes and Effective Addresses (cont)

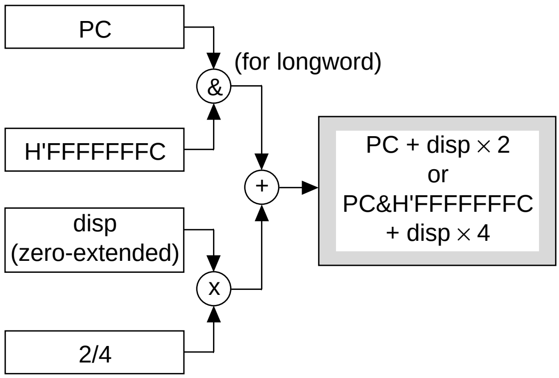
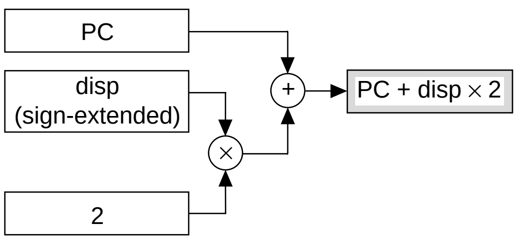
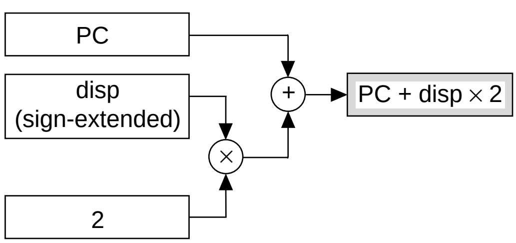
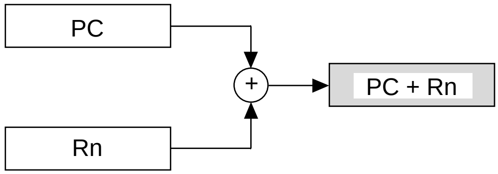
Addressing Mode	Instruction Format	Effective Addresses Calculation	Equation
Indirect PC addressing with displace-ment	@(disp:8, PC)	The effective address is the PC value plus an 8-bit displacement (disp). The value of disp is zero-extended, and remains the same for a byte operation, is doubled for a word operation, and is quadrupled for a longword operation. For a longword operation, the lowest two bits of the PC are masked.  	Word: PC + disp × 2 Longword: PC & H'FFFFFFFC + disp × 4
PC relative addressing	disp:8	The effective address is the PC value sign-extended with an 8-bit displacement (disp), doubled, and added to the PC.  	PC + disp × 2
	disp:12	The effective address is the PC value sign-extended with a 12-bit displacement (disp), doubled, and added to the PC.  	PC + disp × 2

Table 6-7 Addressing Modes and Effective Addresses (cont)

Addressing Mode	Instruction Format	Effective Addresses Calculation	Equation
PC relative addressing (cont)	Rn	The effective address is the register PC plus R0. 	PC + R0
Immediate addressing	#imm:8	The 8-bit immediate data (imm) for the TST, AND, OR, and XOR instructions are zero-extended.	—
	#imm:8	The 8-bit immediate data (imm) for the MOV, ADD, and CMP/EQ instructions are sign-extended.	—
	#imm:8	Immediate data (imm) for the TRAPA instruction is zero-extended and is quadrupled.	—

6.3 DSP Data Addressing (SH3-DSP Only)

The DSP command performs two different types of memory accesses. One uses the X and Y data transfer instructions (MOVX.W and MOVY.W) while the other uses the single data transfer instructions (MOVS.W and MOVS.L). Data addressing for these two types of instructions also differs. Table 6-8 summarizes the data transfer instructions.

Table 6-8 Summary of Data Transfer Instructions

Item	X and Y Data Transfer Processing (MOVX.W and MOVY.W)	Single Data Transfer Processing (MOVS.W and MOVS.L)
Address registers	Ax: R4, R5; Ay: R6, R7	As: R2, R3, R4, R5
Index registers	Ix: R8; Iy: R9	Is: R8
Addressing	Nop/Inc(+2)/Index addition: Post updating	Nop/Inc(+2, +4)/Index addition: Post updating
	—	Dec(–2, –4): Pre updating
Modulo addressing	Available	Not available
Data buses	XDB, YDB	LDB
Data length	16 bits (word)	16 or 32 bits (word or longword)
Bus contention	None	Occurs
Memory	X and Y data memories	All memory spaces
Source registers	Dx, Dy: A0, A1	Ds: A0/A1, M0/M1, X0/X1, Y0/Y1, A0G, A1G
Destination registers	Dx: X0/X1; Dy: Y0/Y1	Ds: A0/A1, M0/M1, X0/X1, Y0/Y1, A0G, A1G

6.3.1 X and Y Data Addressing

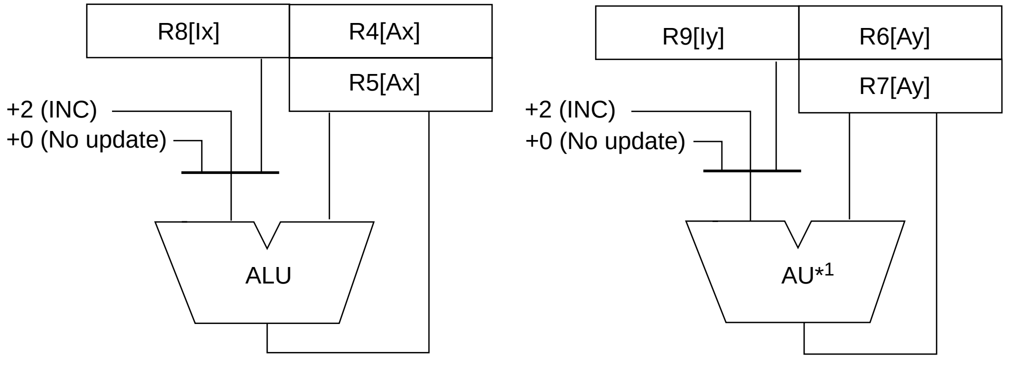
The DSP command allows X and Y data memories to be accessed simultaneously using the MOVX.W and MOVY.W instructions. DSP instructions have two pointers so they can access the X and Y data memories simultaneously. DSP instructions have only pointer addressing; immediate addressing is not available. Address registers are divided in two. The R4 and R5 registers become the X memory address register (Ax) while the R6 and R7 registers become the Y memory address register (Ay). The following three types of addressing may be used with X and Y data transfer instructions.

- Address registers with no update: The Ax and Ay registers are address pointers. They are not updated.
- Addition index register addressing: The Ax and Ay registers are address pointers. The values of the Ix and Iy registers are added to the Ax and Ay registers respectively after data transfer (post updating).
- Increment address register addressing: The Ax and Ay registers are address pointers. +2 is added to them after data transfer (post updating).

Each of the address pointers has an index register. Register R8 becomes the index register (Ix) for the X memory address register (Ax); register R9 becomes the index register (Iy) for the Y memory address register (Ay).

X and Y data transfer instructions are processed in words. X and Y data memory is accessed in 16 bit units. Increment processing for that purpose adds two to the address register. To decrement them, set -2 in the index register and specify addition index register addressing.

Figure 6-1 shows the X and Y data transfer addressing.



Notes: 1. Adder added for DSP processing
2. All three addressing methods (increment, index register addition (Ix, Iy), and no update) are post-updating methods. To decrement the address pointer, set the index register to –2 or –4.

Figure 6-1 X and Y Data Transfer Addressing

6.3.2 Single Data Addressing

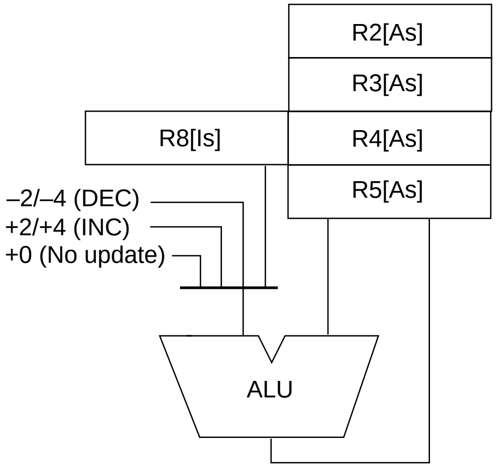
The DSP command has single data transfer instructions (MOVS.W and MOVS.L) that load data to DSP registers and store data from DSP registers. With these instructions, the R2–R5 registers are used as address registers (As) for single data transfers.

There are four types of data addressing for single data transfer instructions.

- Address registers with no update: The As register is the address pointer. It is not updated.
- Addition index register addressing: The As register is the address pointer. The value of the Is register is added to the As register after data transfer (post updating).
- Increment address register addressing: The As register is the address pointer. +2 or +4 is added to it after data transfer (post updating).

- Decrement address register addressing: The As register is the address pointer. –2 or –4 (or +2 or +4) is added to it before data transfer (pre updating).

The address pointer uses the R8 register as its index register (Is). Figure 6-2 shows the single data transfer addressing.



Note: There are four addressing methods (no update, index register addition (Is), increment, and decrement). Index register addition and increment are post-updating methods. Decrement is a pre-updating method.

Figure 6-2 Single Data Transfer Addressing

6.3.3 Modulo Addressing

Like other DSPs, the SH3-DSP has a modulo addressing mode. Address registers are updated in the same way in this mode. When a modulo end address in which the address pointer value is already set is reached, the address pointer becomes the modulo start address.

Modulo addressing is only effective for X and Y data transfer instructions (MOVX.W and MOVY.W). When the DMX bit of the SR register is set, the X address register enters modulo addressing mode; when the DMY bit is set, the Y address register enters modulo addressing mode. Modulo addressing cannot be used on both X and Y address registers at once. Accordingly, do not set DMX and DMY at the same time. Should they both be set at once, only DMY will be valid.

The MOD register is provided for specifying the start and end addresses for the modulo address area. The MOD register stores the MS (modulo start) and ME (modulo end). The following shows how to use the modulo register (MS and ME).

```
            MOV.L ModAddr,Rn;               Rn=ModEnd, ModStart
            LDC Rn,MOD;                     ME=ModEnd, MS=ModStart
ModAddr:    .DATA.WmEnd;             Lower 8bit of ModEnd
            .DATA.W    mStart;              Lower 8bit of ModStart


ModStart:   .DATA
              :
ModEnd:.DATA
```

Set the start and end addresses in MS and ME and then set the DMX or DMY bit to 1. The address register contents are compared to ME. If they match ME, the start address MS is stored in the address register. The bottom 16 bits of the address register are compared to ME. The maximum modulo size is 64 kbytes. This is ample for accessing the X and Y data memory. Figure 6-3 shows a block diagram of modulo addressing.

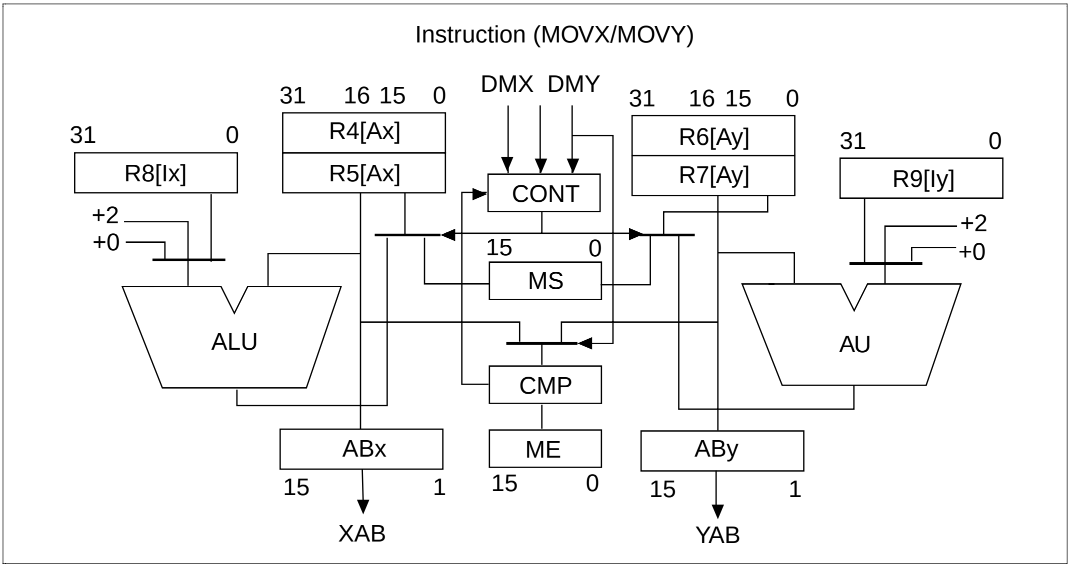


Figure 6-3 Modulo Addressing

The following is an example of modulo addressing.

```
MS=H'08; ME=H'0C; R4=H'C008;
DMX=1; DMY=0; (Sets modulo addressing for address register Ax (R4, R5))
```

The above setting changes the R4 register as shown below.

```
        R4: H'C008
Inc.    R4: H'C00A
Inc.    R4: H'C00C
Inc.    R4: H'C008
```
(Becomes the modulo start address when the modulo end address is reached)

Place data so the top 16 bits of the modulo start and end address are the same, since the modulo start address only swaps the bottom 16 bits of the address register.

Note: When using addition index as the DSP data addressing, the address pointer may exceed this value without matching ME. Should this occur, the address pointer will not return to the modulo start address.

6.3.4 DSP Addressing Operation

The following shows how DSP addressing works in the execution stage (EX) of a pipeline (including modulo addressing).

```
if ( Operation is MOVX.W MOVY.W ) {
    ABx=Ax; ABy=Ay’
    /* memory access cycle uses Abx and Aby. The addresses to be used have
not been updated */

    /* Ax is one of R4,5 */
    if ( DMX==0 || DMX==1 @@ DMY==1 )} Ax=Ax+(+2 or R8[Ix} or +0);
    /* Inc,Index,Not-Update */
    else if (!not-update) Ax=modulo( Ax, (+2 or R8[Ix]) );

    /* Ay is one of R6,7 */
    if ( DMY==0 ) Ay=Ay+(+2 or R9[Iy] or +0; /* Inc,Index,Not-Update */
    else if (! not-update) Ay=modulo( Ay, (+2 or R9[Iy]) );
}
else if ( Operation is MOVS.W or MOVS.L ) {
    if ( Addressing is Nop, Inc, Add-index-reg ) {
        MAB=As;
        /* memory access cycle uses MAB. The address to be used has not been
updated */
        /* As is one of R2–5 */
        As=As+(+2 or +4 or R8[Is] or +0); /* Inc.Index,Not-Update */
    else { /* Decrement, Pre-update */
    /* As is one of R2–5 */
    As=As+(–2 or –4);
```

```
    MAB=As
    /* memory access cycle uses MAB. The address to be used has been updated
*/
}

/* The value to be added to the address register depends on addressing
operations.
For example, (+2 or R8[Ix] or +0) means that
        +2:        if operation is increment
        R8[Ix}: if operation is add-index-reg
        +0:        if operation is not-update
/*

function modulo ( AddrReg, Index ) {
    if ( AdrReg[15:0]==ME ) AdrReg[15:0]==MS;
    else AdrReg=AdrReg+Index
    return AddrReg;
}
```

6.4 Instruction Format of CPU Instructions

The instruction format table, table 6-8, refers to the source operand and the destination operand. The meaning of the operand depends on the instruction code. The symbols are used as follows:

- xxxx: Instruction code
- mmmm: Source register
- nnnn: Destination register
- iiii: Immediate data
- dddd: Displacement

Table 6-9 Instruction Formats

Instruction Formats	Source Operand	Destination Operand	Example
0 format 15 … 0 xxxx xxxx xxxx xxxx	—	—	`NOP`
n format 15 … 0 xxxx | nnnn | xxxx xxxx	—	nnnn: Direct register	`MOVT Rn`
	Control register or system register	nnnn: Direct register	`STS MACH,Rn`
	Control register or system register	nnnn: Indirect pre-decrement register	`STC.L SR,@-Rn`
m format 15 … 0 xxxx | mmmm | xxxx xxxx	mmmm: Direct register	Control register or system register	`LDC Rm,SR`
	mmmm: Indirect post-increment register	Control register or system register	`LDC.L @Rm+,SR`
	mmmm: Direct register	—	`JMP @Rm`
	mmmm: PC relative using Rm	—	`BRAF Rm`

Table 6-9 Instruction Formats (cont)

Instruction Formats	Source Operand	Destination Operand	Example
nm format 15 (bits) 0 xxxx / nnnn / mmmm / xxxx	mmmm: Direct register	nnnn: Direct register	`ADD Rm,Rn`
	mmmm: Direct register	nnnn: Direct register	`MOV.L Rm,@Rn`
	mmmm: Indirect post-increment register (multiply/ accumulate) nnnn: Indirect post-increment register (multiply/ accumulate)*	MACH, MACL	`MAC.W @Rm+,@Rn+`
	mmmm: Indirect post-increment register	nnnn: Direct register	`MOV.L @Rm+,Rn`
	mmmm: Direct register	nnnn: Indirect pre-decrement register	`MOV.L Rm,@-Rn`
	mmmm: Direct register	nnnn: Indirect indexed register	`MOV.L Rm,@(R0,Rn)`
md format 15 (bits) 0 xxxx xxxx / mmmm / dddd	mmmmdddd: indirect register with displacement	R0 (Direct register)	`MOV.B @(disp,Rm),R0`
nd4 format 15 (bits) 0 xxxx xxxx / nnnn / dddd	R0 (Direct register)	nnnndddd: Indirect register with displacement	`MOV.B R0,@(disp,Rn)`

Note: * In multiply/accumulate instructions, nnnn is the source register.

Table 6-9 Instruction Formats (cont)

Instruction Formats	Source Operand	Destination Operand	Example
nmd format 15 … 0 xxxx | nnnn | mmmm | dddd	mmmm: Direct register	nnnndddd: Indirect register with displacement	`MOV.L Rm,@(disp,Rn)`
	mmmmdddd: Indirect register with displacement	nnnn: Direct register	`MOV.L @(disp,Rm),Rn`
d format 15 … 0 xxxx xxxx | dddd dddd	dddddddd: Indirect GBR with displacement	R0 (Direct register)	`MOV.L @(disp,GBR),R0`
	R0(Direct register)	dddddddd: Indirect GBR with displacement	`MOV.L R0,@(disp,GBR)`
	dddddddd: PC relative with displacement	R0 (Direct register)	`MOVA @(disp,PC),R0`
	dddddddd: PC relative	—	`BF label`
d12 format 15 … 0 xxxx | dddd dddd dddd	dddddddddddd: PC relative	—	`BRA label` `(label = disp + PC)`
nd8 format 15 … 0 xxxx | nnnn | dddd dddd	dddddddd: PC relative with displacement	nnnn: Direct register	`MOV.L @(disp,PC),Rn`
i format 15 … 0 xxxx xxxx | i i i i i i i i	iiiiiiii: Immediate	Indirect indexed GBR	`AND.B #imm,@(R0,GBR)`
	iiiiiiii: Immediate	R0 (Direct register)	`AND #imm,R0`
	iiiiiiii: Immediate	—	`TRAPA #imm`
ni format 15 … 0 xxxx | nnnn | i i i i i i i i	iiiiiiii: Immediate	nnnn: Direct register	`ADD #imm,Rn`

6.5 Instruction Formats for DSP Instructions (SH3-DSP Only)

New instructions have been added to the SH3-DSP for use in digital signal processing. The new instructions are divided into two groups.

- Double and single data transfer instructions for memory and DSP registers (16 bits)
- Parallel processing instructions processed by the DSP unit (32 bits)

Figure 6-4 shows their instruction formats.

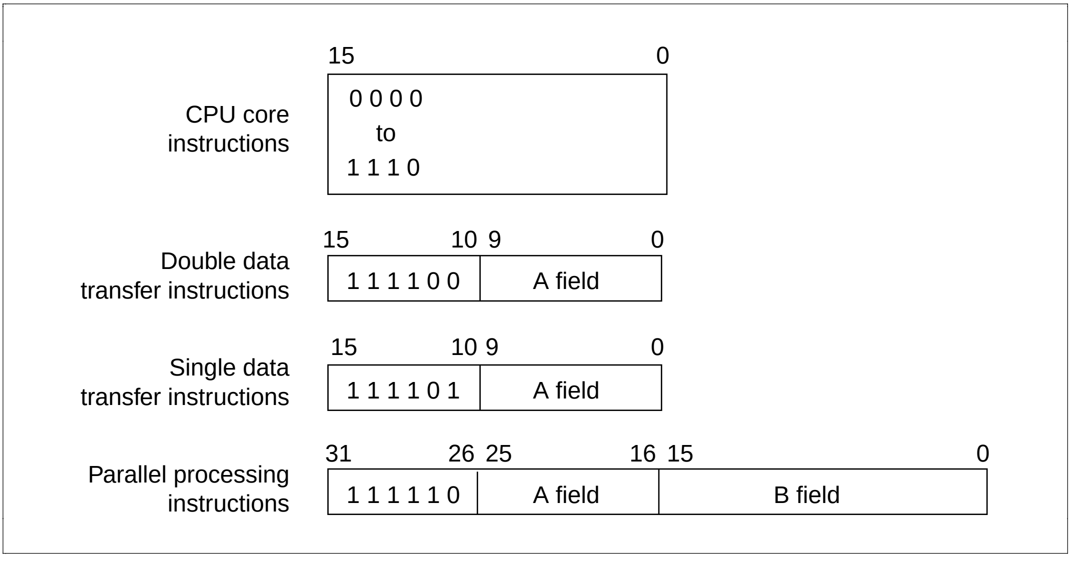


Figure 6-4 Instruction Formats of DSP Instructions

6.5.1 Double and Single Data Transfer Instructions

Table 6-10 shows the instruction formats for double data transfer instructions. Table 6-11 shows the instruction formats for single data transfer instructions

Table 6-10 Instruction Formats for Double Data Transfers

Category	Mnemonic		15	14	13	12	11	10	9	8
X memory data transfers	NOPX		1	1	1	1	0	0	0	
	MOVX.W	@Ax,Dx							Ax	
	MOVX.W	@Ax+,Dx								
	MOVX.W	@Ax+Ix,Dx								
	MOVX.W	Da,@Ax								
	MOVX.W	Da,@Ax+								
	MOVX.W	Da,@Ax+Ix								
Y memory data transfers	NOPY		1	1	1	1	0	0		0
	MOVY.W	@Ay,Dy								Ay
	MOVY.W	@Ay+,Dy								
	MOVY.W	@Ay+Iy,Dy								
	MOVY.W	Da,@Ay								
	MOVY.W	Da,@Ay+								
	MOVY.W	Da,@Ay+Iy								

Table 6-10 Instruction Formats for Double Data Transfers (cont)

Category	Mnemonic		7	6	5	4	3	2	1	0
X memory data transfers	NOPX		0		0		0	0		
	MOVX.W	@Ax,Dx	Dx		0		0	1		
	MOVX.W	@Ax+,Dx					1	0		
	MOVX.W	@Ax+Ix,Dx					1	1		
	MOVX.W	Da,@Ax	Da		1		0	1		
	MOVX.W	Da,@Ax+					1	0		
	MOVX.W	Da,@Ax+Ix					1	1		
Y memory data transfers	NOPY			0		0			0	0
	MOVY.W	@Ay,Dy		Dy		0			0	1
	MOVY.W	@Ay+,Dy							1	0
	MOVY.W	@Ay+Iy,Dy							1	1
	MOVY.W	Da,@Ay		Da		1			0	1
	MOVY.W	Da,@Ay+							1	0
	MOVY.W	Da,@Ay+Iy							1	1

Ax: 0=R4, 1=R5 Ay: 0=R6, 1=R7 Dx: 0=X0, 1=X1 Dy: 0=Y0, 1=Y1 Da: 0=A0, 1=A1

Table 6-11 Instruction Formats for Single Data Transfers

Category	Mnemonic		15	14	13	12	11	10	9	8
Single data transfer	MOVS.W	@–As,Ds	1	1	1	1	0	1		As
	MOVS.W	@As,Ds								0: R4
	MOVS.W	@As+,Ds								1: R5
	MOVS.W	@As+Is,Ds								2: R2
	MOVS.W	Ds,@A–s								3: R3
	MOVS.W	Ds,@As								
	MOVS.W	Ds,@As+								
	MOVS.W	Ds,@As+Is								
	MOVS.L	@–As,Ds								
	MOVS.L	@As,Ds								
	MOVS.L	@As+,Ds								
	MOVS.L	@As+Is,Ds								
	MOVS.L	Ds,@A–s								
	MOVS.L	Ds,@As								
	MOVS.L	Ds,@As+								
	MOVS.L	Ds,@As+Is								

Table 6-11 Instruction Formats for Single Data Transfers (cont)

Category	Mnemonic		7	6	5	4	3	2	1	0
Single data transfer	MOVS.W	@–As,Ds		Ds		0: (*)	0	0	0	0
	MOVS.W	@As,Ds				1: (*)	0	1		
	MOVS.W	@As+,Ds				2: (*)	1	0		
	MOVS.W	@As+Is,Ds				3: (*)	1	1		
	MOVS.W	Ds,@A–s				4: (*)	0	0		1
	MOVS.W	Ds,@As				5: A1	0	1		
	MOVS.W	Ds,@As+				6: (*)	1	0		
	MOVS.W	Ds,@As+Is				7: A0	1	1		
	MOVS.L	@–As,Ds				8: X0	0	0		0
	MOVS.L	@As,Ds				9: X1	0	1		
	MOVS.L	@As+,Ds				A: Y0	1	0		
	MOVS.L	@As+Is,Ds				B: Y1	1	1		
	MOVS.L	Ds,@A–s				C: M0	0	0		1
	MOVS.L	Ds,@As				D: A1G	0	1		
	MOVS.L	Ds,@As+				E:M1	1	0		
	MOVS.L	Ds,@As+Is				F:A0G	1	1		

Note: * System reserved code

6.5.2 Parallel Processing Instructions

Parallel processing instructions are used by the SH3-DSP to increase the execution efficiency of digital signal processing using the DSP unit. They are 32 bits long and four can be processed in parallel (one ALU operation, one multiplication, and two data transfers).

Parallel processing instructions are divided into two fields, A and B. The data transfer instructions are defined in field A and the ALU operation instruction and multiplication instruction are defined in field B. These instructions can be defined independently, processed independently, and can be executed simultaneously in parallel. Table 6-12 lists the field A parallel data transfer instructions, and Table 6-13 shows the field B ALU operation instructions and multiplication instructions. The field A instructions are identical to the double data transfer instructions shown in Table 6-10.

Table 6-12 Field A Parallel Data Transfer Instructions

Category	Mnemonic	31	30	29	28	27	26	25	24	23
X memory data transfers	`NOPX`	1	1	1	1	1	0	0		0

	`MOVX.W @Ax,Dx`
	`MOVX.W @Ax+,Dx`
	`MOVX.W @Ax+Ix,Dx`
								Ax		Dx

	`MOVX.W Da,@Ax`
	`MOVX.W Da,@Ax+`
	`MOVX.W Da,@Ax+Ix`
										Da

Y memory data transfers	`NOPY`								0	

	`MOVY.W @Ay,Dy`
	`MOVY.W @Ay+,Dy`
	`MOVY.W @Ay+Iy,Dy`
	`MOVY.W Da,@Ay`
	`MOVY.W Da,@Ay+`
	`MOVY.W Da,@Ay+Iy`
									Ay	

Table 6-12 Field A Parallel Data Transfer Instructions (cont)

Category	Mnemonic	22	21	20	19	18	17	16	15–0
X memory data transfers	NOPX		0		0	0			Field B
	MOVX.W @Ax,Dx		0		0	1			
	MOVX.W @Ax+,Dx				1	0			
	MOVX.W @Ax+Ix,Dx				1	1			
	MOVX.W Da,@Ax		1		0	1			
	MOVX.W Da,@Ax+				1	0			
	MOVX.W Da,@Ax+Ix				1	1			
Y memory data transfers	NOPY	0		0			0	0	
	MOVY.W @Ay,Dy	Dy		0			0	1	
	MOVY.W @Ay+,Dy						1	0	
	MOVY.W @Ay+Iy,Dy						1	1	
	MOVY.W Da,@Ay	Da		1			0	1	
	MOVY.W Da,@Ay+						1	0	
	MOVY.W Da,@Ay+Iy						1	1	

Ax: 0=R4, 1=R5 Ay: 0=R6, 1=R7 Dx: 0=X0, 1=X1 Dy: 0=Y0, 1=Y1 Da: 0=A0, 1=A1

Table 6-13 Field B ALU Operation Instructions and Multiplication Instructions

Category	Mnemonic	31–27	26	25–16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
imm. shift	PSHL #imm, Dz	1	0	Field A	0	0	0	0	0	–16 ≤ imm ≤ +16							Dz			
	PSHA #imm, Dz				0	0	0	1	0	– 32 ≤ imm ≤ +32							Dz			
	Reserved				0	0	0		1											
					0	0	1													
Six operand parallel instruction	PMULS Se, Sf, Dg				0	1	0	0	Se		Sf		Sx		Sy		Dg		Du	
	Reserved				0	1	0	1	0:X0 1:X1 2:Y0 3:A1		0:Y0 1:Y1 2:X0 3:A1		0:X0 1:X1 2:A0 3:A1		0:Y0 1:Y1 2:M0 3:M1		0:M0 1:M1 2:A0 3:A1		0:X0 1:Y0 2:A0 3:A1	
	PSUB Sx, Sy, Du PMULS Se, Sf, Dg				0	1	1	0												
	PADD Sx, Sy, Du PMULS Se, Sf, Dg				0	1	1	1												
Three operand instructions	Reserved				1	0	0	0	0	0	0	0					Dz			
							0	1												
	PSUBC Sx, Sy, Dz						1	0												
	PADDC Sx, Sy, Dz						1	1												
	PCMP Sx, Sy						0	0	0	1										
	Reserved						0	1												
	PWSB Sx, Sy, Dz						1	0												
	PWAD Sx, Sy, Dz						1	1												
	PABS Sx, Dz						0	0	1	0										
	PRND Sx, Dz						0	1												
	PABS Sy, Dz						1	0												
	PRND Sy, Dz						1	1												
	Reserved						0	0	1	1										
							0	1												
							1	0												
							1	1												

Dz (bits 3–0, three operand instructions):
0: (*1)
1: (*1)
2: (*1)
3: (*1)
4: (*1)
5: A1
6: (*1)
7: A0
8: X0
9: X1
A: Y0
B: Y1
C: M0
D: (*1)
E: M1
F: (*1)

Table 6-13 Field B ALU Operation Instructions and Multiplication Instructions (cont)

Category	Mnemonic	31–27	26	25–16	15 14	13 12	11 10	9 8	7 6	5 4	3 2 1 0
Conditional three operand instructions	(if cc)*1 PSHL Sx, Sy, Dz	1	0	Field A	1 0	0 0	0 0	if cc 01:*2 10:DCT 11:DCF	Sx 0:X0 1:X1 2:Y0 3:Y1	Sy 0:Y0 1:Y1 2:M0 3:M1	Dz 0:(*1) 1:(*1) 2:(*1) 3:(*1) 4:(*1) 5:A1 6:(*1) 7:A0 8:X0 9:X1 A:Y0 B:Y1 C:M0 D:(*1) E:M1 F:(*1)
	(if cc) PSHA Sx, Sy, Dz					0 1					
	(if cc) PSUB Sx, Sy, Dz					1 0					
	(if cc) PADD Sx, Sy, Dz					1 1					
	Reserved					0 0	0 1				
	(if cc) PAND Sx, Sy, Dz					0 1					
	(if cc) PXOR Sx, Sy, Dz					1 0					
	(if cc) POR Sx, Sy, Dz					1 1					
	(if cc) PDEC Sx, Dz					0 0	1 0				
	(if cc) PINC Sx, Dz					0 1					
	(if cc) PDEC Sy, Dz					1 0					
	(if cc) PINC Sy, Dz					1 1					
	(if cc) PCLR Dz					0 0	1 1				
	(if cc) PDMSB Sx, Dz					0 1					
	Reserved					1 0					
	(if cc) PDMSB Sy, Dz					1 1					
	(if cc) PNEG Sx, Dz				1 1	0 0	1 0				
	(if cc) PCOPY Sx, Dz					0 1					
	(if cc) PNEG Sy, Dz					1 0					
	(if cc) PCOPY Sy, Dz					1 1					
	Reserved							0 0			
	(if cc) PSTS MACH, Dz					0 0	1 1	if cc			
	(if cc) PSTS MACL, Dz					0 1					
	(if cc) PLDS Dz, MACH					1 0					
	(if cc) PLDS Dz, MACL					1 1					
	Reserved							0 0			
							0*3				
	Reserved	1		1							

Notes: 1. [if cc]: DCT (DC bit true), DCF (DC bit false), or none (unconditional instruction)
2. Unconditional
3. System reserved code

Section 7 Instruction Set

7.1 Instruction Set by Classification

The SH-3 instruction set includes 68 basic instruction types, and the SH-3E instruction set includes 84 basic instruction types, divided into seven functional classifications, as shown in Table 7-1. Tables 7-3 to 7-9 summarize instruction notation, machine mode, execution time, and function.

Table 7-1 Classification of Instructions

Classification	Types	Operation Code	Function	No. of Instructions
Data transfer	5	MOV	Data transfer Immediate data transfer Peripheral module data transfer Structure data transfer	39
		MOVA	Effective address transfer	
		MOVT	T bit transfer	
		SWAP	Swap of upper and lower bytes	
		XTRCT	Extraction of the middle of registers connected	
		PREF	Prefetching data to cache	
Arithmetic operations	21	ADD	Binary addition	33
		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	
		DMULS	Signed double-length multiplication	
		DMULU	Unsigned double-length multiplication	
		DT	Decrement and test	
		EXTS	Sign extension	
		EXTU	Zero extension	
		MAC	Multiply/accumulate, double-length multiply/accumulate operation	
		MUL	Double-length multiplication (32 × 32 bits)	
		MULS	Signed multiplication (16 × 16 bits)	
		MULU	Unsigned multiplication (16 × 16 bits)	
		NEG	Negation	
		NEGC	Negation with borrow	
		SUB	Binary subtraction	
		SUBC	Binary subtraction with carry	
		SUBV	Binary subtraction with underflow check	

Table 7-1 Classification of Instructions (cont)

Classification	Types	Operation Code	Function	No. of Instructions
Logic operations	6	AND	Logical AND	14
		NOT	Bit inversion	
		OR	Logical OR	
		TAS	Memory test and bit set	
		TST	Logical AND and T bit set	
		XOR	Exclusive OR	
Shift	12	ROTL	One-bit left rotation	16
		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	
		SHAD	Dynamic arithmetic shift	
		SHLD	Dynamic logical shift	
Branch	9	BF	Conditional branch, conditional branch with delay (T = 0)	11
		BT	Conditional branch, conditional branch with delay (T = 1)	
		BRA	Unconditional branch	
		BRAF	Unconditional branch	
		BSR	Branch to subroutine procedure	
		BSRF	Branch to subroutine procedure	
		JMP	Unconditional branch	
		JSR	Branch to subroutine procedure	
		RTS	Return from subroutine procedure	

Table 7-1 Classification of Instructions (cont)

Classification	Types	Operation Code	Function	No. of Instructions
System control	15	CLRT	T bit clear	83 (75)*
		CLRMAC	MAC register clear	
		CLRS	S bit clear	
		LDC	Load to control register	
		LDS	Load to system register	
		LDTLB	Load PTE to TLB	
		NOP	No operation	
		RTE	Return from exception processing	
		SETS	S bit set	
		SETT	T bit set	
		SLEEP	Shift into power-down mode	
		STC	Storing control register data	
		STS	Storing system register data	
		TRAPA	Trap exception handling	
Floating point instructions (SH-3E only)	16	FABS	Floating point absolute value	23
		FADD	Floating point add	
		FCMP	Floating point compare	
		FDIV	Floating point divide	
		FLDI0	Floating point load immediate 0	
		FLDI1	Floating point load immediate 1	
		FLDS	Floating point load to system register FPUL	
		FLOAT	Floating point convert from integer	
		FMAC	Floating point multiply accumulate	
		FMOV	Floating point move	
		FMUL	Floating point multiply	
		FNEG	Floating point negate	
		FSQRT	Floating point square root	
		FSTS	Floating point store from system register FPUL	
		FSUB	Floating point subtract	
		FTRC	Floating point truncate and convert to integer	
Total:	84			219 (188)*

Note: * The LDS and STS instructions include instructions to load/store to the FPU system register. These instructions can only be used with the SH-3E. The figure in parentheses ( ) is the total excluding the SH-3E instructions.

Instruction codes, operation, and execution states are listed as shown in Table 7-2 in order by classification.

Tables 7-3 to 7-8 list the minimum number of clock cycles required for execution. In practice, the number of execution cycles increases when the instruction fetch is in contention with data access or when the destination register of a load instruction (memory → register) is the same as the register used by the next instruction.

Table 7-2 Instruction Code Format

Item	Format	Explanation
Instruction	`OP.Sz SRC,DEST`	OP: Operation code Sz: Size SRC: Source DEST: Destination Rm: Source register Rn: Destination register imm: Immediate data disp: Displacement
Operation	→, ← (xx) M/Q/T & | ^ ~ <<n, >>n	Direction of transfer Memory operand Flag bits in the SR Logical AND of each bit Logical OR of each bit Exclusive OR of each bit Logical NOT of each bit n-bit shift
Code	MSB ↔ LSB	mmmm: Source register nnnn: Destination register 0000: R0 0001: R1 .... 1111: R15 iiii: Immediate data dddd: Displacement
Privilege		Indicates a privileged instruction
Cycles		The execution cycles shown in the table are minimums. The actual number of cycles may be increased: 1. When contention occurs between instruction fetches and data access, or 2. When the destination register of the load instruction (memory → register) and the register used by the next instruction are the same.
T bit		Value of T bit after instruction is executed —: No change

Note: Scaling (×1, ×2, ×4) is performed according to the instruction operand size. See "8. Instruction Descriptions" for details.

7.1.1 Data Transfer Instructions

Table 7-3 Data Transfer Instructions

Instruction		Operation	Code	Privilege	Cycles	T Bit
MOV	#imm,Rn	imm → Sign extension → Rn	1110nnnniiiiiiii	—	1	—
MOV.W	@(disp,PC),Rn	(disp × 2 + PC) → Sign extension → Rn	1001nnnndddddddd	—	1	—
MOV.L	@(disp,PC),Rn	(disp × 4 + PC) → Rn	1101nnnndddddddd	—	1	—
MOV	Rm,Rn	Rm → Rn	0110nnnnmmmm0011	—	1	—
MOV.B	Rm,@Rn	Rm → (Rn)	0010nnnnmmmm0000	—	1	—
MOV.W	Rm,@Rn	Rm → (Rn)	0010nnnnmmmm0001	—	1	—
MOV.L	Rm,@Rn	Rm → (Rn)	0010nnnnmmmm0010	—	1	—
MOV.B	@Rm,Rn	(Rm) → Sign extension → Rn	0110nnnnmmmm0000	—	1	—
MOV.W	@Rm,Rn	(Rm) → Sign extension → Rn	0110nnnnmmmm0001	—	1	—
MOV.L	@Rm,Rn	(Rm) → Rn	0110nnnnmmmm0010	—	1	—
MOV.B	Rm,@–Rn	Rn–1 → Rn, Rm → (Rn)	0010nnnnmmmm0100	—	1	—
MOV.W	Rm,@–Rn	Rn–2 → Rn, Rm → (Rn)	0010nnnnmmmm0101	—	1	—
MOV.L	Rm,@–Rn	Rn–4 → Rn, Rm → (Rn)	0010nnnnmmmm0110	—	1	—
MOV.B	@Rm+,Rn	(Rm) → Sign extension → Rn,Rm + 1 → Rm	0110nnnnmmmm0100	—	1	—
MOV.W	@Rm+,Rn	(Rm) → Sign extension → Rn,Rm + 2 → Rm	0110nnnnmmmm0101	—	1	—
MOV.L	@Rm+,Rn	(Rm) → Rn,Rm + 4 → Rm	0110nnnnmmmm0110	—	1	—
MOV.B	R0,@(disp,Rn)	R0 → (disp + Rn)	10000000nnnndddd	—	1	—
MOV.W	R0,@(disp,Rn)	R0 → (disp × 2 + Rn)	10000001nnnndddd	—	1	—
MOV.L	Rm,@(disp,Rn)	Rm → (disp × 4 + Rn)	0001nnnnmmmmdddd	—	1	—
MOV.B	@(disp,Rm),R0	(disp + Rm) → Sign extension → R0	10000100mmmmdddd	—	1	—
MOV.W	@(disp,Rm),R0	(disp × 2 + Rm) → Sign extension → R0	10000101mmmmdddd	—	1	—
MOV.L	@(disp,Rm),Rn	(disp × 4 + Rm) → Rn	0101nnnnmmmmdddd	—	1	—
MOV.B	Rm,@(R0,Rn)	Rm → (R0 + Rn)	0000nnnnmmmm0100	—	1	—
MOV.W	Rm,@(R0,Rn)	Rm → (R0 + Rn)	0000nnnnmmmm0101	—	1	—

Table 7-3 Data Transfer Instructions (cont)

Instruction		Operation	Code	Privilege	Cycles	T Bit
MOV.L	Rm,@(R0,Rn)	Rm → (R0 + Rn)	0000nnnnmmmm0110	—	1	—
MOV.B	@(R0,Rm),Rn	(R0 + Rm) → Sign extension → Rn	0000nnnnmmmm1100	—	1	—
MOV.W	@(R0,Rm),Rn	(R0 + Rm) → Sign extension → Rn	0000nnnnmmmm1101	—	1	—
MOV.L	@(R0,Rm),Rn	(R0 + Rm) → Rn	0000nnnnmmmm1110	—	1	—
MOV.B	R0,@(disp,GBR)	R0 → (disp + GBR)	11000000dddddddd	—	1	—
MOV.W	R0,@(disp,GBR)	R0 → (disp × 2 + GBR)	11000001dddddddd	—	1	—
MOV.L	R0,@(disp,GBR)	R0 → (disp × 4 + GBR)	11000010dddddddd	—	1	—
MOV.B	@(disp,GBR),R0	(disp + GBR) → Sign extension → R0	11000100dddddddd	—	1	—
MOV.W	@(disp,GBR),R0	(disp × 2 + GBR) → Sign extension → R0	11000101dddddddd	—	1	—
MOV.L	@(disp,GBR),R0	(disp × 4 + GBR) → R0	11000110dddddddd	—	1	—
MOVA	@(disp,PC),R0	disp × 4 + PC → R0	11000111dddddddd	—	1	—
MOVT	Rn	T → Rn	0000nnnn00101001	—	1	—
PREF	@Rn	(Rn) → cache	0000nnnn10000011	—	1/2*	—
SWAP.B	Rm,Rn	Rm → Swap the bottom two bytes → REG	0110nnnnmmmm1000	—	1	—
SWAP.W	Rm,Rn	Rm → Swap two consecutive words → Rn	0110nnnnmmmm1001	—	1	—
XTRCT	Rm,Rn	Rm: Middle 32 bits of Rn → Rn	0010nnnnmmmm1101	—	1	—

Note: * Two cycles on the SH3-DSP.

7.1.2 Arithmetic Instructions

Table 7-4 Arithmetic Instructions

Instruction		Operation	Code	Privilege	Cycles	T Bit
ADD	Rm,Rn	Rn + Rm → Rn	0011nnnnmmmm1100	—	1	—
ADD	#imm,Rn	Rn + imm → Rn	0111nnnniiiiiiii	—	1	—
ADDC	Rm,Rn	Rn + Rm + T → Rn, Carry → T	0011nnnnmmmm1110	—	1	Carry
ADDV	Rm,Rn	Rn + Rm → Rn, Overflow → T	0011nnnnmmmm1111	—	1	Overflow
CMP/EQ	#imm,R0	If R0 = imm, 1 → T	10001000iiiiiiii	—	1	Comparison result
CMP/EQ	Rm,Rn	If Rn = Rm, 1 → T	0011nnnnmmmm0000	—	1	Comparison result
CMP/HS	Rm,Rn	If Rn≥Rm with unsigned data, 1 → T	0011nnnnmmmm0010	—	1	Comparison result
CMP/GE	Rm,Rn	If Rn ≥ Rm with signed data, 1 → T	0011nnnnmmmm0011	—	1	Comparison result
CMP/HI	Rm,Rn	If Rn > Rm with unsigned data, 1 → T	0011nnnnmmmm0110	—	1	Comparison result
CMP/GT	Rm,Rn	If Rn > Rm with signed data, 1 → T	0011nnnnmmmm0111	—	1	Comparison result
CMP/PZ	Rn	If Rn ≥ 0, 1 → T	0100nnnn00010001	—	1	Comparison result
CMP/PL	Rn	If Rn > 0, 1 → T	0100nnnn00010101	—	1	Comparison result
CMP/STR	Rm,Rn	If Rn and Rm have an equivalent byte, 1 → T	0010nnnnmmmm1100	—	1	Comparison result
DIV1	Rm,Rn	Single-step division (Rn/Rm)	0011nnnnmmmm0100	—	1	Calculation result
DIV0S	Rm,Rn	MSB of Rn → Q, MSB of Rm → M, M ^ Q → T	0010nnnnmmmm0111	—	1	Calculation result
DIV0U		0 → M/Q/T	0000000000011001	—	1	0

Table 7-4 Arithmetic Instructions (cont)

Instruction	Operation	Code	Privilege	Cycles	T Bit
`DMULS.L Rm,Rn`	Signed operation of Rn × Rm → MACH, MACL 32 × 32 → 64 bits	`0011nnnnmmmm1101`	—	2 (to 5/4)*1	—
`DMULU.L Rm,Rn`	Unsigned operation of Rn × Rm → MACH, MACL 32 × 32 → 64 bits	`0011nnnnmmmm0101`	—	2 (to 5/4)*1	—
`DT Rn`	Rn – 1 → Rn, if Rn = 0, 1 → T, else 0 → T	`0100nnnn00010000`	—	1	Comparison result
`EXTS.B Rm,Rn`	A byte in Rm is sign-extended → Rn	`0110nnnnmmmm1110`	—	1	—
`EXTS.W Rm,Rn`	A word in Rm is sign-extended → Rn	`0110nnnnmmmm1111`	—	1	—
`EXTU.B Rm,Rn`	A byte in Rm is zero-extended → Rn	`0110nnnnmmmm1100`	—	1	—
`EXTU.W Rm,Rn`	A word in Rm is zero-extended → Rn	`0110nnnnmmmm1101`	—	1	—
`MAC.L @Rm+, @Rn+`	Signed operation of (Rn) × (Rm) + MAC → MAC	`0000nnnnmmmm1111`	—	2 (to 5/4)*1	—
`MAC.W @Rm+, @Rn+`	Signed operation of (Rn) × (Rm) + MAC → MAC 16 × 16 + 64 → 64 bits	`0100nnnnmmmm1111`	—	2 (to 5)*1	—
`MUL.L Rm,Rn`	Rn × Rm → MACL 32 × 32 → 32 bits	`0000nnnnmmmm0111`	—	2 (to 5/4)*1	—
`MULS.W Rm,Rn`	Signed operation of Rn × Rm → MAC 16 × 16 → 32 bits	`0010nnnnmmmm1111`	—	1 (to 3)*2	—
`MULU.W Rm,Rn`	Unsigned operation of Rn × Rm → MAC 16 × 16 → 32 bits	`0010nnnnmmmm1110`	—	1 (to 3)*2	—
`NEG Rm,Rn`	0–Rm → Rn	`0110nnnnmmmm1011`	—	1	—
`NEGC Rm,Rn`	0–Rm–T → Rn, Borrow → T	`0110nnnnmmmm1010`	—	1	Borrow
`SUB Rm,Rn`	Rn–Rm → Rn	`0011nnnnmmmm1000`	—	1	—
`SUBC Rm,Rn`	Rn–Rm–T → Rn, Borrow → T	`0011nnnnmmmm1010`	—	1	Borrow
`SUBV Rm,Rn`	Rn–Rm → Rn, Underflow → T	`0011nnnnmmmm1011`	—	1	Underflow

Notes: 1. The normal minimum number of execution cycles is 2, but 5 cycles (4 cycles on the SH3-DSP) are required when the results of an operation are read from the MAC register immediately after the instruction.

2. The normal minimum number of execution cycles is 1, but 3 cycles are required when the results of an operation are read from the MAC register immediately after a MUL instruction.

7.1.3 Logic Operation Instructions

Table 7-5 Logic Operation Instructions

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

Note: * Four cycles on the SH3-DSP.

7.1.4 Shift Instructions

Table 7-6 Shift Instructions

Instruction		Operation	Code	Privilege	Cycles	T Bit
ROTL	Rn	T ← Rn ← MSB	0100nnnn00000100	—	1	MSB
ROTR	Rn	LSB → Rn → T	0100nnnn00000101	—	1	LSB
ROTCL	Rn	T ← Rn ← T	0100nnnn00100100	—	1	MSB
ROTCR	Rn	T → Rn → T	0100nnnn00100101	—	1	LSB
SHAD	Rm,Rn	Rn ≥ 0; Rn << Rm → Rn Rn < 0; Rn >> Rm → [MSB→Rn]	0100nnnnmmmm1100	—	1	—
SHAL	Rn	T ← Rn ← 0	0100nnnn00100000	—	1	MSB
SHAR	Rn	MSB → Rn → T	0100nnnn00100001	—	1	LSB
SHLD	Rm,Rn	Rn ≥ 0; Rn << Rm → Rn Rn < 0; Rn >> Rm → [0→Rn]	0100nnnnmmmm1101	—	1	—
SHLL	Rn	T ← Rn ← 0	0100nnnn00000000	—	1	MSB
SHLR	Rn	0 → Rn → T	0100nnnn00000001	—	1	LSB
SHLL2	Rn	Rn << 2 → Rn	0100nnnn00001000	—	1	—
SHLR2	Rn	Rn >> 2 → Rn	0100nnnn00001001	—	1	—
SHLL8	Rn	Rn << 8 → Rn	0100nnnn00011000	—	1	—
SHLR8	Rn	Rn >> 8 → Rn	0100nnnn00011001	—	1	—
SHLL16	Rn	Rn << 16 → Rn	0100nnnn00101000	—	1	—
SHLR16	Rn	Rn >> 16 → Rn	0100nnnn00101001	—	1	—

7.1.5 Branch Instructions

Table 7-7 Branch Instructions

Instruction		Operation	Code	Privilege	Cycles	T Bit
BF	label	If T = 0, disp × 2 + PC → PC; if T = 1, nop	10001011dddddddd	—	3/1*	—
BF/S	label	Delayed branch, if T = 0, disp × 2 + PC → PC; if T = 1, nop	10001111dddddddd	—	2/1*	—
BT	label	Delayed branch, if T = 1, disp × 2 + PC → PC; if T = 0, nop	10001001dddddddd	—	3/1*	—
BT/S	label	If T = 1, disp × 2 + PC → PC; if T = 0, nop	10001101dddddddd	—	2/1*	—
BRA	label	Delayed branch, disp × 2 + PC → PC	1010dddddddddddd	—	2	—
BRAF	Rn	Rn + PC → PC	0000nnnn00100011	—	2	—
BSR	label	Delayed branch, PC → PR, disp × 2 + PC → PC	1011dddddddddddd	—	2	—
BSRF	Rn	PC → PR, Rn + PC → PC	0000nnnn00000011	—	2	—
JMP	@Rn	Delayed branch, Rn → PC	0100nnnn00101011	—	2	—
JSR	@Rn	Delayed branch, PC → PR, Rn → PC	0100nnnn00001011	—	2	—
RTS		Delayed branch, PR → PC	0000000000001011	—	2	—

Note: * One state when it does not branch.

7.1.6 System Control Instructions

Table 7-8 System Control Instructions

Instruction	Operation	Code	Privilege	Cycles	T Bit
CLRMAC	0 → MACH, MACL	0000000000101000	—	1	—
CLRS	0 → S	0000000001001000	—	1	—
CLRT	0 → T	0000000000001000	—	1	0
LDC Rm,SR	Rm → SR	0100mmmm00001110	√	5	LSB
LDC Rm,GBR	Rm → GBR	0100mmmm00011110	—	1/3*[1]	—
LDC Rm,VBR	Rm → VBR	0100mmmm00101110	√	1/3*[1]	—
LDC Rm,SSR	Rm → SSR	0100mmmm00111110	√	1/3*[1]	—
LDC Rm,SPC	Rm → SPC	0100mmmm01001110	√	1/3*[1]	—
LDC Rm,R0_BANK	Rm → R0_BANK	0100mmmm10001110	√	1/3*[1]	—
LDC Rm,R1_BANK	Rm → R1_BANK	0100mmmm10011110	√	1/3*[1]	—
LDC Rm,R2_BANK	Rm → R2_BANK	0100mmmm10101110	√	1/3*[1]	—
LDC Rm,R3_BANK	Rm → R3_BANK	0100mmmm10111110	√	1/3*[1]	—
LDC Rm,R4_BANK	Rm → R4_BANK	0100mmmm11001110	√	1/3*[1]	—
LDC Rm,R5_BANK	Rm → R5_BANK	0100mmmm11011110	√	1/3*[1]	—
LDC Rm,R6_BANK	Rm → R6_BANK	0100mmmm11101110	√	1/3*[1]	—
LDC Rm,R7_BANK	Rm → R7_BANK	0100mmmm11111110	√	1/3*[1]	—
LDC.L @Rm+,SR	(Rm) → SR, Rm + 4 → Rm	0100mmmm00000111	√	7	LSB
LDC.L @Rm+,GBR	(Rm) → GBR, Rm + 4 → Rm	0100mmmm00010111	—	1/5*[2]	—
LDC.L @Rm+,VBR	(Rm) → VBR, Rm + 4 → Rm	0100mmmm00100111	√	1/5*[2]	—
LDC.L @Rm+,SSR	(Rm) → SSR, Rm + 4 → Rm	0100mmmm00110111	√	1/5*[2]	—
LDC.L @Rm+,SPC	(Rm) → SPC, Rm + 4 → Rm	0100mmmm01000111	√	1/5*[2]	—
LDC.L @Rm+,R0_BANK	(Rm) → R0_BANK, Rm + 4 → Rm	0100mmmm10000111	√	1/5*[2]	—
LDC.L @Rm+,R1_BANK	(Rm) → R1_BANK, Rm + 4 → Rm	0100mmmm10010111	√	1/5*[2]	—
LDC.L @Rm+,R2_BANK	(Rm) → R2_BANK, Rm + 4 → Rm	0100mmmm10100111	√	1/5*[2]	—
LDC.L @Rm+,R3_BANK	(Rm) → R3_BANK, Rm + 4 → Rm	0100mmmm10110111	√	1/5*[2]	—

Table 7-8 System Control Instructions (cont)

Instruction		Operation	Code	Privilege	Cycles	T Bit
LDC.L	@Rm+,R4_ BANK	(Rm) → R4_BANK, Rm + 4 → Rm	0100mmmm11000111	√	1/5*2	—
LDC.L	@Rm+,R5_ BANK	(Rm) → R5_BANK, Rm + 4 → Rm	0100mmmm11010111	√	1/5*2	—
LDC.L	@Rm+,R6_ BANK	(Rm) → R6_BANK, Rm + 4 → Rm	0100mmmm11100111	√	1/5*2	—
LDC.L	@Rm+,R7_ BANK	(Rm) → R7_BANK, Rm + 4 → Rm	0100mmmm11110111	√	1/5*2	—
LDS	Rm,MACH	Rm → MACH	0100mmmm00001010	—	1	—
LDS	Rm,MACL	Rm → MACL	0100mmmm00011010	—	1	—
LDS	Rm,PR	Rm → PR	0100mmmm00101010	—	1	—
LDS.L	@Rm+,MACH	(Rm) → MACH, Rm + 4 → Rm	0100mmmm00000110	—	1	—
LDS.L	@Rm+,MACL	(Rm) → MACL, Rm + 4 → Rm	0100mmmm00010110	—	1	—
LDS.L	@Rm+,PR	(Rm) → PR, Rm + 4 → Rm	0100mmmm00100110	—	1	—
LDTLB		PTEH/PTEL → TLB	0000000000111000	√	1	—
NOP		No operation	0000000000001001	—	1	—
PREF	@Rn	(Rn) → cache	0000nnnn10000011	—	1	—
RTE		Delayed branch, SSR/SPC → SR/PC	0000000000101011	√	4	—
SETS		1 → S	0000000001011000	—	1	—
SETT		1 → T	0000000000011000	—	1	1
SLEEP		Sleep	0000000000011011	√	4*3	—
STC	SR,Rn	SR → Rn	0000nnnn00000010	√	1	—
STC	GBR,Rn	GBR → Rn	0000nnnn00010010	—	1	—
STC	VBR,Rn	VBR → Rn	0000nnnn00100010	√	1	—
STC	SSR,Rn	SSR → Rn	0000nnnn00110010	√	1	—
STC	SPC,Rn	SPC → Rn	0000nnnn01000010	√	1	—

Table 7-8 System Control Instructions (cont)

Instruction	Operation	Code	Privilege	Cycles	T Bit
STC R0_BANK,Rn	R0_BANK→ Rn	0000nnnn10000010	√	1	—
STC R1_BANK,Rn	R1_BANK→ Rn	0000nnnn10010010	√	1	—
STC R2_BANK,Rn	R2_BANK→ Rn	0000nnnn10100010	√	1	—
STC R3_BANK,Rn	R3_BANK→ Rn	0000nnnn10110010	√	1	—
STC R4_BANK,Rn	R4_BANK→ Rn	0000nnnn11000010	√	1	—
STC R5_BANK,Rn	R5_BANK→ Rn	0000nnnn11010010	√	1	—
STC R6_BANK,Rn	R6_BANK→ Rn	0000nnnn11100010	√	1	—
STC R7_BANK,Rn	R7_BANK→ Rn	0000nnnn11110010	√	1	—
STC.L SR,@–Rn	Rn–4 → Rn, SR → (Rn)	0100nnnn00000011	√	1/2*4	—
STC.L GBR,@–Rn	Rn–4 → Rn, GBR → (Rn)	0100nnnn00010011	—	1/2*4	—
STC.L VBR,@–Rn	Rn–4 → Rn, VBR → (Rn)	0100nnnn00100011	√	1/2*4	—
STC.L SSR,@–Rn	Rn–4 → Rn, SSR → (Rn)	0100nnnn00110011	√	1/2*4	—
STC.L SPC,@–Rn	Rn–4 → Rn, SPC → (Rn)	0100nnnn01000011	√	1/2*4	—
STC.L R0_BANK,@–Rn	Rn–4 → Rn, R0_BANK → (Rn)	0100nnnn10000011	√	2	—
STC.L R1_BANK,@–Rn	Rn–4 → Rn, R1_BANK → (Rn)	0100nnnn10010011	√	2	—
STC.L R2_BANK,@–Rn	Rn–4 → Rn, R2_BANK → (Rn)	0100nnnn10100011	√	2	—
STC.L R3_BANK,@–Rn	Rn–4 → Rn, R3_BANK → (Rn)	0100nnnn10110011	√	2	—
STC.L R4_BANK,@–Rn	Rn–4 → Rn, R4_BANK → (Rn)	0100nnnn11000011	√	2	—
STC.L R5_BANK,@–Rn	Rn–4 → Rn, R5_BANK → (Rn)	0100nnnn11010011	√	2	—
STC.L R6_BANK,@–Rn	Rn–4 → Rn, R6_BANK → (Rn)	0100nnnn11100011	√	2	—
STC.L R7_BANK,@–Rn	Rn–4 → Rn, R7_BANK → (Rn)	0100nnnn11110011	√	2	—
STS MACH,Rn	MACH → Rn	0000nnnn00001010	—	1	—
STS MACL,Rn	MACL → Rn	0000nnnn00011010	—	1	—
STS PR,Rn	PR → Rn	0000nnnn00101010	—	1	—

Table 7-8 System Control Instructions (cont)

Instruction		Operation	Code	Privilege	Cycles	T Bit
STS.L	MACH,@–Rn	Rn–4 → Rn, MACH → (Rn)	0100nnnn00000010	—	1	—
STS.L	MACL,@–Rn	Rn–4 → Rn, MACL → (Rn)	0100nnnn00010010	—	1	—
STS.L	PR,@–Rn	Rn–4 → Rn, PR → (Rn)	0100nnnn00100010	—	1	—
TRAPA	#imm	PC/SR → SPC/SSR, #imm<<2 → TRA, 0x160 → EXPEVT VBR + H'0100 → PC	11000011iiiiiiii	—	6/8*5	—

Notes: The number of execution states before the chip enters the sleep state. This table lists the minimum execution cycles. In practice, the number of execution cycles increases when the instruction fetch is in contention with data access or when the destination register of a load instruction (memory → register) is the same as the register used by the next instruction.

1. Three cycles on the SH3-DSP.
2. Five cycles on the SH3-DSP.
3. Number of cycles before transition to sleep state.
4. Two cycles on the SH3-DSP.
5. Eight cycles on the SH3-DSP.

7.1.7 Floating Point Instructions (SH-3E Only)

Table 7-9 Floating Point Instructions

Instruction		Operation	Code	Privilege	Cycles	T Bit
FABS	FRn	| FRn | → FRn	1111nnnn01011101	—	1	—
FADD	FRm,FRn	FRn + FRm → FRn	1111nnnnmmmm0000	—	1	—
FCMP/EQ	FRm,FRn	FRn == FRm? 1:0 → T	1111nnnnmmmm0100	—	1	Comparison result
FCMP/GT	FRm,FRn	FRn > FRm? 1:0 → T	1111nnnnmmmm0101	—	1	Comparison result
FDIV	FRm,FRn	FRn / FRm → FRn	1111nnnnmmmm0011	—	13	—
FLDI0	FRn	H'00000000 → FRn	1111nnnn10001101	—	1	—
FLDI1	FRn	H'3F800000 → FRn	1111nnnn10011101	—	1	—
FLDS	FRm,FPUL	FRm → FPUL	1111nnnn00011101	—	1	—
FLOAT	FPUL,FRn	(float)FPUL → FRn	1111nnnn00101101	—	1	—
FMAC	FR0,FRm,FRn	FR0 × FRm + FRn → FRn	1111nnnnmmmm1110	—	1	—
FMOV	FRm,FRn	FRm → FRn	1111nnnnmmmm1100	—	1	—
FMOV.S	@(R0,Rm),FRn	(R0 + Rm) → FRn	1111nnnnmmmm0110	—	1	—
FMOV.S	@Rm+,FRn	(Rm) → FRn, Rm+4 → Rm	1111nnnnmmmm1001	—	1	—
FMOV.S	@Rm,FRn	(Rm) → FRn	1111nnnnmmmm1000	—	1	—
FMOV.S	FRm,@(R0,Rn)	FRm → (R0 + Rn)	1111nnnnmmmm0111	—	1	—
FMOV.S	FRm,@-Rn	Rn-4 → Rn, FRm → (Rn)	1111nnnnmmmm1011	—	1	—
FMOV.S	FRm,@Rn	FRm → (Rn)	1111nnnnmmmm1010	—	1	—
FMUL	FRm,FRn	Fm × FRm → FRn	1111nnnnmmmm0010	—	1	—
FNEG	FRn	–FRn → FRn	1111nnnn01001101	—	1	—
FSQRT	FRn	√FRn → FRn	1111nnnn01101101	—	13	—
FSTS	FPUL,FRn	FPUL → FRn	1111nnnn00001101	—	1	—
FSUB	FRm,FRn	FRn - FRm → FRn	1111nnnnmmmm0001	—	1	—
FTRC	FRm,FPUL	(long)FRm → FPUL	1111nnnn00111101	—	1	—

7.1.8 FPU System Register Related CPU Instructions (SH-3E Only)

Table 7-10 FPU Related CPU Instructions

Instruction	Operation	Code	Privilege	Cycles	T Bit
`LDS Rm,FPSCR`	Rm → FPSCR	0100nnnn01101010	—	1	—
`LDS Rm,FPUL`	Rm → FPUL	0100nnnn01011010	—	1	—
`LDS.L @Rm+ ,FPSCR`	@Rm → FPSCR, Rm+4 → Rm	0100nnnn01100110	—	1	—
`LDS.L @Rm+ ,FPUL`	@Rm → FPUL, Rm+4 → Rm	0100nnnn01010110	—	1	—
`STS FPSCR, Rn`	FPSCR → Rn	0000nnnn01101010	—	1	—
`STS FPUL, Rn`	FPUL → Rn	0000nnnn01011010	—	1	—
`STS.L FPSCR,@- Rn`	Rn-4 → Rn, FPSCR → @Rn	0100nnnn01100010	—	1	—
`STS.L FPUL,@-Rn`	Rn-4 → Rn, FPUL → @Rn	0100nnnn01010010	—	1	—

7.1.9 CPU Instructions That Support DSP Functions (SH3-DSP Only)

Several system control instructions have been added to the CPU core instructions to support DSP functions. The RS, RE, and MOD registers (which support modulo addressing) have been added, and an RC counter has been added to the SR register. LDC and STC instructions have been added to access these. LDS and STS instructions have also been added for accessing the DSP registers DSR, A0, X0, X1, Y0, and Y1.

A SETRC instruction has been added for setting the value of the repeat counter (RC) in the SR register (bits 16–27). When the operand of the SETRC instruction is immediate, 8 bits of immediate data are set in bits 16–23 of the SR register and bits 24–27 are cleared. When the operand is a register, the 12 bits 0–11 of the register are set in bits 16–27 of the SR register.

In addition to the new LDC instructions, the LDRE and LDRS instructions have been added for setting the repeat start address and repeat end address in the RS and RE registers.

Table 7-11 shows the added instructions.

Table 7-11 Added CPU Instructions

Instruction	Operation	Code	Cycles	T Bit
LDC Rm,MOD	Rm→MOD	0100mmmm01011110	3	—
LDC Rm,RE	Rm→RE	0100mmmm01111110	3	—
LDC Rm,RS	Rm→RS	0100mmmm01101110	3	—
LDC.L @Rm+,MOD	(Rm)→MOD,Rm+4→Rm	0100mmmm01010111	5	—
LDC.L @Rm+,RE	(Rm)→RE,Rm+4→Rm	0100mmmm01110111	5	—
LDC.L @Rm+,RS	(Rm)→RS,Rm+4→Rm	0100mmmm01100111	5	—
STC MOD,Rn	MOD→Rn	0000nnnn01010010	1	—
STC RE,Rn	RE→Rn	0000nnnn01110010	1	—
STC RS,Rn	RS→Rn	0000nnnn01100010	1	—
STC.L MOD,@-Rn	Rn–4→Rn,MOD→(Rn)	0100nnnn01010011	2	—
STC.L RE,@-Rn	Rn–4→Rn,RE→(Rn)	0100nnnn01110011	2	—
STC.L RS,@-Rn	Rn–4→Rn,RS→(Rn)	0100nnnn01100011	2	—
LDS Rm,DSR	Rm→DSR	0100mmmm01101010	1	—
LDS.L @Rm+,DSR	(Rm)→DSR,Rm+4→Rm	0100mmmm01100110	1	—
LDS Rm,A0	Rm→A0	0100mmmm01110110	1	—
LDS.L @Rm+,A0	(Rm)→A0,Rm+4→Rm	0100mmmm01100110	1	—
LDS Rm,X0	Rm→X0	0100mmmm01110110	1	—
LDS.L @Rm+,X0	(Rm)→X0,Rm+4→Rm	0100mmmm01100110	1	—
LDS Rm,X1	Rm→X1	0100mmmm01110110	1	—
LDS.L @Rm+,X1	(Rm)→X1,Rm+4→Rm	0100mmmm01100110	1	—
LDS Rm,Y0	Rm→Y0	0100mmmm01110110	1	—
LDS.L @Rm+,Y0	(Rm)→Y0,Rm+4→Rm	0100mmmm01100110	1	—
LDS Rm,Y1	Rm→Y1,Rm+4→Rm	0100mmmm01110110	1	—
LDS.L @Rm+,Y1	(Rm)→Y1,Rm+4→Rm	0100mmmm01100110	1	—
STS DSR,Rn	DSR→Rn	0000nnnn01101010	1	—
STS.L DSR,@-Rn	Rn–4→Rn,DSR→(Rn)	0100nnnn01100010	1	—
STS A0,Rn	A0→Rn	0000nnnn01111010	1	—
STS.L A0,@-Rn	Rn–4→Rn,A0→(Rn)	0100nnnn01110010	1	—
STS X0,Rn	X0→Rn	0000nnnn01111010	1	—
STS.L X0,@-Rn	Rn–4→Rn,X0→(Rn)	0100nnnn01110010	1	—
STS X1,Rn	X1→Rn	0000nnnn01111010	1	—
STS.L X1,@-Rn	Rn–4→Rn,X1→(Rn)	0100nnnn01110010	1	—

Table 7-11 Added CPU Instructions (cont)

Instruction	Operation	Code	Cycles	T Bit
STS Y0,Rn	Y0→Rn	0000nnnn10101010	1	—
STS.L Y0,@-Rn	Rn–4→Rn,Y0→(Rn)	0100nnnn10100010	1	—
STS Y1,Rn	Y1→Rn	0000nnnn10111010	1	—
STS.L Y1,@-Rn	Rn–4→Rn,Y1→(Rn)	0100nnnn10110010	1	—
SETRC Rm	Rm[11:0]→RC (SR[27:16])	0100mmmm00010100	3	—
SETRC #imm	imm→RC(SR[23:16]), zeros→SR[27:24]	10000010iiiiiiii	3	—
LDRS @(disp,pc)	disp × 2+PC→RS	10001100dddddddd	3	—
LDRE @(disp,pc)	disp × 2+PC→RE	10001110dddddddd	3	—

7.2 Instruction Set in Alphabetical Order

Table 7-12 alphabetically lists the instruction codes and number of execution cycles for each instruction.

Table 7-12 Instruction Set Listed Alphabetically

Instruction		Operation	Code	Privilege	Cycles	T Bit
ADD	#imm,Rn	Rn + imm → Rn	0111nnnniiiiiiii	—	1	—
ADD	Rm,Rn	Rn + Rm → Rn	0011nnnnmmmm1100	—	1	—
ADDC	Rm,Rn	Rn + Rm + T → Rn, Carry → T	0011nnnnmmmm1110	—	1	Carry
ADDV	Rm,Rn	Rn + Rm → Rn, Overflow → T	0011nnnnmmmm1111	—	1	Overflow
AND	#imm,R0	R0 & imm → R0	11001001iiiiiiii	—	1	—
AND	Rm,Rn	Rn & Rm → Rn	0010nnnnmmmm1001	—	1	—
AND.B	#imm,@(R0,GBR)	(R0 + GBR) & imm → (R0 + GBR)	11001101iiiiiiii	—	3	—
BF	label	If T = 0, disp + PC → PC; if T = 1, nop	10001011dddddddd	—	3/1*2	—
BF/S	label	If T = 0, disp + PC → PC; if T = 1, nop	10001111dddddddd	—	2/1*2	—
BRA	label	Delayed branch, disp + PC → PC	1010dddddddddddd	—	2	—
BRAF	Rn	Delayed branch, Rn + PC → PC	0000nnnn00100011	—	2	—

Table 7-12 Instruction Set Listed Alphabetically (cont)

Instruction		Operation	Code	Privilege	Cycles	T Bit
BSR	label	Delayed branch, PC → PR, disp + PC → PC	1011dddddddddddd	—	2	—
BSRF	Rn	Delayed branch, PC → PR, Rn + PC → PC	0000nnnn00000011	—	2	—
BT	label	If T = 1, disp + PC → PC; if T = 0, nop	10001001dddddddd	—	3/1*2	—
BT/S	label	If T = 1, disp + PC → PC; if T = 0, nop	10001101dddddddd	—	2/1*2	—
CLRMAC		0 → MACH, MACL	0000000000101000	—	1	—
CLRS		0 → S	0000000001001000	—	1	—
CLRT		0 → T	0000000000001000	—	1	0
CMP/EQ	#imm,R0	If R0 = imm, 1 → T	10001000iiiiiiii	—	1	Comparison result
CMP/EQ	Rm,Rn	If Rn = Rm, 1 → T	0011nnnnmmmm0000	—	1	Comparison result
CMP/GE	Rm,Rn	If Rn ≥ Rm with signed data, 1 → T	0011nnnnmmmm0011	—	1	Comparison result
CMP/GT	Rm,Rn	If Rn > Rm with signed data, 1 → T	0011nnnnmmmm0111	—	1	Comparison result
CMP/HI	Rm,Rn	If Rn > Rm with unsigned data,	0011nnnnmmmm0110	—	1	Comparison result
CMP/HS	Rm,Rn	If Rn ≥ Rm with unsigned data, 1 → T	0011nnnnmmmm0010	—	1	Comparison result
CMP/PL	Rn	If Rn>0, 1 → T	0100nnnn00010101	—	1	Comparison result
CMP/PZ	Rn	If Rn ≥ 0, 1 → T	0100nnnn00010001	—	1	Comparison result
CMP/STR	Rm,Rn	If Rn and Rm have an equivalent byte, 1 → T	0010nnnnmmmm1100	—	1	Comparison result
DIV0S	Rm,Rn	MSB of Rn → Q, MSB of Rm → M, M ^ Q → T	0010nnnnmmmm0111	—	1	Calculation result
DIV0U		0 → M/Q/T	0000000000011001	—	1	0
DIV1	Rm,Rn	Single-step division (Rn/Rm)	0011nnnnmmmm0100	—	1	Calculation result
DMULS.L	Rm,Rn	Signed operation of Rn × Rm → MACH, MACL	0011nnnnmmmm1101	—	2 (to 5)*1	—

Table 7-12 Instruction Set Listed Alphabetically (cont)

Instruction		Operation	Code	Privilege	Cycles	T Bit
DMULU.L	Rm,Rn	Unsigned operation of Rn × Rm → MACH, MACL	0011nnnnmmmm0101	—	2 (to 5)*1	—
DT	Rn	Rn - 1 → Rn, when Rn is 0, 1 → T. When Rn is nonzero, 0 → T	0100nnnn00010000	—	1	Comparison result
EXTS.B	Rm,Rn	A byte in Rm is sign-extended → Rn	0110nnnnmmmm1110	—	1	—
EXTS.W	Rm,Rn	A word in Rm is sign-extended → Rn	0110nnnnmmmm1111	—	1	—
EXTU.B	Rm,Rn	A byte in Rm is zero-extended → Rn	0110nnnnmmmm1100	—	1	—
EXTU.W	Rm,Rn	A word in Rm is zero-extended → Rn	0110nnnnmmmm1101	—	1	—
FABS	FRn*3	| FRn | → FRn	1111nnnn01011101	—	1	—
FADD	FRm ,FRn*3	FRn + FRm → FRn	1111nnnnmmmm0000	—	1	—
FCMP/EQ	FRm ,FRn*3	(FRn == FRm)? 1:0 → T	1111nnnnmmmm0100	—	1	Comparison result
FCMP/GT	FRm ,FRn*3	(FRn > FRm) ? 1:0 → T	1111nnnnmmmm0101	—	1	Comparison result
FDIV	FRm ,FRn*3	FRn /FRm → FRn	1111nnnnmmmm0011	—	13	—
FLDI0	FRn*3	H'00000000 → FRn	1111nnnn10001101	—	1	—
FLDI1	FRn*3	H'3F800000 → FRn	1111nnnn10011101	—	1	—
FLDS	FRm ,FPUL*3	FRm → FPUL	1111nnnn00011101	—	1	—
FLOAT	FPUL, FRn*3	(float)FPUL → FRn	1111nnnn00101101	—	1	—
FMAC	FR0,FRm,FRn*3	FR0 × FRm + FRn → FRn	1111nnnnmmmm1110	—	1	—
FMOV	FRm ,FRn*3	FRm → FRn	1111nnnnmmmm1100	—	1	—
FMOV.S	@(R0,Rm),FRn*3	(R0 + Rm) → FRn	1111nnnnmmmm0110	—	1	—
FMOV.S	@Rm+,FRn*3	(Rm) → FRn,Rm + 4 = Rm	1111nnnnmmmm1001	—	1	—
FMOV.S	@Rm,FRn*3	(Rm) → FRn	1111nnnnmmmm1000	—	1	—
FMOV.S	FRm,@(R0,Rn)*3	(FRm) → (R0 + Rn)	1111nnnnmmmm0111	—	1	—
FMOV.S	FRm,@-Rn**3	Rn-4 → Rn, FRm → (Rn)	1111nnnnmmmm1011	—	1	—
FMOV.S	FRm,@Rn*3	FRm → (Rn)	1111nnnnmmmm1010	—	1	—
FMUL	FRm,FRn*3	FRn × FRm → FRn	1111nnnnmmmm0010	—	1	—

Table 7-12 Instruction Set Listed Alphabetically (cont)

Instruction		Operation	Code	Privilege	Cycles	T Bit
FNEG	FRn*3	–FRn → FRn	1111nnnn01001101	—	1	—
FSQRT	FRn*3	√ FRn → FRn	1111nnnn01101101	—	13	—
FSTS	FPUL,FRn*3	FPUL → FRn	1111nnnn00001101	—	1	—
FSUB	FRm,FRn*3	FRn – FRm → FRn	1111nnnnmmmm0001	—	1	—
FTRC	FRm,FPUL*3	(long)FRm → FPUL	1111nnnn00111101	—	1	—
JMP	@Rn	Delayed branch, Rn → PC	0100nnnn00101011	—	2	—
JSR	@Rn	Delayed branch, PC → PR, Rn → PC	0100nnnn00001011	—	2	—
LDC	Rm,GBR	Rm → GBR	0100mmmm00011110	—	1/3*4	—
LDC	Rm,SR	Rm → SR	0100mmmm00001110	√	5	LSB
LDC	Rm,VBR	Rm → VBR	0100mmmm00101110	√	1/3*4	—
LDC	Rm,SSR	Rm → SSR	0100mmmm00111110	√	1/3*4	—
LDC	Rm,SPC	Rm → SPC	0100mmmm01001110	√	1/3*4	—
LDC	Rm,MOD*9	Rm→ MOD	0100mmmm01011110	√	3	—
LDC	Rm,RE*9	Rm→ RE	0100mmmm01101110	√	3	—
LDC	Rm,RS*9	Rm→ RS	0100mmmm01101110	√	3	—
LDC	Rm,R0_BANK	Rm → R0_BANK	0100mmmm10001110	√	1/3*4	—
LDC	Rm,R1_BANK	Rm → R1_BANK	0100mmmm10011110	√	1/3*4	—
LDC	Rm,R2_BANK	Rm → R2_BANK	0100mmmm10101110	√	1/3*4	—
LDC	Rm,R3_BANK	Rm → R3_BANK	0100mmmm10111110	√	1/3*4	—
LDC	Rm,R4_BANK	Rm → R4_BANK	0100mmmm11001110	√	1/3*4	—
LDC	Rm,R5_BANK	Rm → R5_BANK	0100mmmm11011110	√	1/3*4	—
LDC	Rm,R6_BANK	Rm → R6_BANK	0100mmmm11101110	√	1/3*4	—
LDC	Rm,R7_BANK	Rm → R7_BANK	0100mmmm11111110	√	1/3*4	—
LDC.L	@Rm+,GBR	(Rm) → GBR, Rm + 4 → Rm	0100mmmm00010111	—	1/5*5	—
LDC.L	@Rm+,SR	(Rm) → SR, Rm + 4 → Rm	0100mmmm00000111	√	7	LSB
LDC.L	@Rm+,VBR	(Rm) → VBR, Rm + 4 → Rm	0100mmmm00100111	√	1/5*5	—
LDC.L	@Rm+,SSR	(Rm) → SSR, Rm + 4 → Rm	0100mmmm00110111	√	1/5*5	—
LDC.L	@Rm+,SPC	(Rm) → SPC, Rm + 4 → Rm	0100mmmm01000111	√	1/5*5	—
LDC.L	@Rm+,MOD*9	(Rm) → MOD,Rm + 4 → Rm	0100mmmm01010111	√	5	—
LDC.L	@Rm+,RE*9	(Rm) → RE,Rm + 4 → Rm	0100mmmm01110111	√	5	—
LDC.L	@Rm+,RS*9	(Rm) → RS,Rm + 4 → Rm	0100mmmm01100111	√	5	—

Table 7-12 Instruction Set Listed Alphabetically (cont)

Instruction	Operation	Code	Privilege	Cycles	T Bit
`LDC.L @Rm+,R0_BANK`	(Rm) → R0_BANK, Rm + 4 → Rm	0100mmmm10000111	√	1/5*5	—
`LDC.L @Rm+,R1_BANK`	(Rm) → R1_BANK, Rm + 4 → Rm	0100mmmm10010111	√	1/5*5	—
`LDC.L @Rm+,R2_BANK`	(Rm) → R2_BANK, Rm + 4 → Rm	0100mmmm10100111	√	1/5*5	—
`LDC.L @Rm+,R3_BANK`	(Rm) → R3_BANK, Rm + 4 → Rm	0100mmmm10110111	√	1/5*5	—
`LDC.L @Rm+,R4_BANK`	(Rm) → R4_BANK, Rm + 4 → Rm	0100mmmm11000111	√	1/5*5	—
`LDC.L @Rm+,R5_BANK`	(Rm) → R5_BANK, Rm + 4 → Rm	0100mmmm11010111	√	1/5*5	—
`LDC.L @Rm+,R6_BANK`	(Rm) → R6_BANK, Rm + 4 → Rm	0100mmmm11100111	√	1/5*5	—
`LDC.L @Rm+,R7_BANK`	(Rm) → R7_BANK, Rm + 4 → Rm	0100mmmm11110111	√	1/5*5	—
`LDRE @(disp,PC)`*9	disp × 2 + PC → RE	10001110dddddddd	—	3	—
`LDRS @(disp,PC)`*9	disp × 2 + PC → RS	10001100dddddddd	—	3	—
`LDS Rm,FPSCR`*3	Rm → FPSCR	0100nnnn01101010	—	1	—
`LDS Rm,FPUL`*3	Rm → FPUL	0100nnnn01011010	—	1	—
`LDS Rm,MACH`	Rm → MACH	0100mmmm00001010	—	1	—
`LDS Rm,MACL`	Rm → MACL	0100mmmm00011010	—	1	—
`LDS Rm,PR`	Rm → PR	0100mmmm00101010	—	1	—
`LDS Rm,A0`*9	Rm → DSR	0100mmmm01101010	—	1	—
`LDS Rm,DSR`*9	Rm → A0	0100mmmm01111010	—	1	—
`LDS Rm,X0`*9	Rm → X0	0100mmmm10001010	—	1	—
`LDS Rm,X1`*9	Rm → X1	0100mmmm10011010	—	1	—
`LDS Rm,Y0`*9	Rm → Y0	0100mmmm10101010	—	1	—
`LDS Rm,Y1`*9	Rm → Y1	0100mmmm10111010	—	1	—
`LDS.L @Rm+ ,FPSCR`*3	@Rm → FPSCR , Rm+4 → Rn	0100nnnn01100110	—	1	—
`LDS.L @Rm+ ,FPUL`*3	@Rm → FPUL , Rm+4 → Rn	0100nnnn01010110	—	1	—
`LDS.L @Rm+,MACH`	(Rm) → MACH, Rm + 4 → Rm	0100mmmm00000110	—	1	—
`LDS.L @Rm+,MACL`	(Rm) → MACL, Rm + 4 → Rm	0100mmmm00010110	—	1	—

Table 7-12 Instruction Set Listed Alphabetically (cont)

Instruction	Operation	Code	Privilege	Cycles	T Bit
LDS.L @Rm+,PR	(Rm) → PR, Rm + 4 → Rm	0100mmmm00100110	—	1	—
LDS.L @Rm+,DSR*[9]	(Rm) → DSR, Rm+4 → Rm	0100mmmm01100110	—	1	—
LDS.L @Rm+,A0*[9]	(Rm) → A0, Rm+4 → Rm	0100mmmm01110110	—	1	—
LDS.L @Rm+,X0*[9]	(Rm) → X0, Rm+4 → Rm	0100mmmm10000110	—	1	—
LDS.L @Rm+,X1*[9]	(Rm) → X1, Rm+4 → Rm	0100mmmm10010110	—	1	—
LDS.L @Rm+,Y0*[9]	(Rm) → Y0, Rm+4 → Rm	0100mmmm10100110	—	1	—
LDS.L @Rm+,Y1*[9]	(Rm) → Y1, Rm+4 → Rm	0100mmmm10110110	—	1	—
LDTLB	PTEH/PTEL → TLB	0000000000111000	√	1	—
MAC.L @Rm+,@Rn+	Signed operation of (Rn) × (Rm) + MAC → MAC	0000nnnnmmmm1111	—	2 (to 5)*[1]	—
MAC.W @Rm+,@Rn+	Signed operation of (Rn) × (Rm) + MAC → MAC	0100nnnnmmmm1111	—	2 (to 5)*[1]	—
MOV #imm,Rn	#imm → Sign extension → Rn	1110nnnniiiiiiii	—	1	—
MOV Rm,Rn	Rm → Rn	0110nnnnmmmm0011	—	1	—
MOV.B @(disp,GBR),R0	(disp + GBR) → Sign extension → R0	11000100dddddddd	—	1	—
MOV.B @(disp,Rm),R0	(disp + Rm) → Sign extension → R0	10000100mmmmdddd	—	1	—
MOV.B @(R0,Rm),Rn	(R0 + Rm) → Sign extension → Rn	0000nnnnmmmm1100	—	1	—
MOV.B @Rm+,Rn	(Rm) → Sign extension → Rn, Rm + 1 → Rm	0110nnnnmmmm0100	—	1	—
MOV.B @Rm,Rn	(Rm) → Sign extension → Rn	0110nnnnmmmm0000	—	1	—
MOV.B R0,@(disp,GBR)	R0 → (disp + GBR)	11000000dddddddd	—	1	—
MOV.B R0,@(disp,Rn)	R0 → (disp + Rn)	10000000nnnndddd	—	1	—
MOV.B Rm,@(R0,Rn)	Rm → (R0 + Rn)	0000nnnnmmmm0100	—	1	—
MOV.B Rm,@–Rn	Rn–1 → Rn, Rm → (Rn)	0010nnnnmmmm0100	—	1	—
MOV.B Rm,@Rn	Rm → (Rn)	0010nnnnmmmm0000	—	1	—

Table 7-12 Instruction Set Listed Alphabetically (cont)

Instruction		Operation	Code	Privilege	Cycles	T Bit
MOV.L	@(disp,GBR),R0	(disp + GBR) → R0	11000110dddddddd	—	1	—
MOV.L	@(disp,PC),Rn	(disp + PC) → Rn	1101nnnndddddddd	—	1	—
MOV.L	@(disp,Rm),Rn	(disp + Rm) → Rn	0101nnnnmmmmdddd	—	1	—
MOV.L	@(R0,Rm),Rn	(R0 + Rm) → Rn	0000nnnnmmmm1110	—	1	—
MOV.L	@Rm+,Rn	(Rm) → Rn, Rm + 4 → Rm	0110nnnnmmmm0110	—	1	—
MOV.L	@Rm,Rn	(Rm) → Rn	0110nnnnmmmm0010	—	1	—
MOV.L	R0,@(disp,GBR)	R0 → (disp + GBR)	11000010dddddddd	—	1	—
MOV.L	Rm,@(disp,Rn)	Rm → (disp + Rn)	0001nnnnmmmmdddd	—	1	—
MOV.L	Rm,@(R0,Rn)	Rm → (R0 + Rn)	0000nnnnmmmm0110	—	1	—
MOV.L	Rm,@–Rn	Rn–4 → Rn, Rm → (Rn)	0010nnnnmmmm0110	—	1	—
MOV.L	Rm,@Rn	Rm → (Rn)	0010nnnnmmmm0010	—	1	—
MOV.W	@(disp,GBR),R0	(disp + GBR) → Sign extension → R0	11000101dddddddd	—	1	—
MOV.W	@(disp,PC),Rn	(disp + PC) → Sign extension → Rn	1001nnnndddddddd	—	1	—
MOV.W	@(disp,Rm),R0	(disp + Rm) → Sign extension → R0	10000101mmmmdddd	—	1	—
MOV.W	@(R0,Rm),Rn	(R0 + Rm) → Sign extension → Rn	0000nnnnmmmm1101	—	1	—
MOV.W	@Rm+,Rn	(Rm) → Sign extension → Rn, Rm + 2 → Rm	0110nnnnmmmm0101	—	1	—
MOV.W	@Rm,Rn	(Rm) → Sign extension → Rn	0110nnnnmmmm0001	—	1	—
MOV.W	R0,@(disp,GBR)	R0 → (disp + GBR)	11000001dddddddd	—	1	—
MOV.W	R0,@(disp,Rn)	R0 → (disp + Rn)	10000001nnnndddd	—	1	—
MOV.W	Rm,@(R0,Rn)	Rm → (R0 + Rn)	0000nnnnmmmm0101	—	1	—
MOV.W	Rm,@–Rn	Rn–2 → Rn, Rm → (Rn)	0010nnnnmmmm0101	—	1	—
MOV.W	Rm,@Rn	Rm → (Rn)	0010nnnnmmmm0001	—	1	—
MOVA	@(disp,PC),R0	disp + PC → R0	11000111dddddddd	—	1	—
MOVT	Rn	T → Rn	0000nnnn00101001	—	1	—
MUL.L	Rm,Rn	Rn × Rm → MAC	0000nnnnmmmm0111	—	2 (to 5)*[1]	—
MULS.W	Rm,Rn	Signed operation of Rn × Rm → MAC	0010nnnnmmmm1111	—	1 (to 3)*[1]	—

Table 7-12 Instruction Set Listed Alphabetically (cont)

Instruction		Operation	Code	Privilege	Cycles	T Bit
MULU.W	Rm,Rn	Unsigned operation of Rn × Rm → MAC	0010nnnnmmmm1110	—	1 (to 3)*1	—
NEG	Rm,Rn	0–Rm → Rn	0110nnnnmmmm1011	—	1	—
NEGC	Rm,Rn	0–Rm–T → Rn, Borrow → T	0110nnnnmmmm1010	—	1	Borrow
NOP		No operation	0000000000001001	—	1	—
NOT	Rm,Rn	~Rm → Rn	0110nnnnmmmm0111	—	1	—
OR	#imm,R0	R0 | imm → R0	11001011iiiiiiii	—	1	—
OR	Rm,Rn	Rn | Rm → Rn	0010nnnnmmmm1011	—	1	—
OR.B	#imm, @(R0,GBR)	(R0 + GBR) | imm → (R0 + GBR)	11001111iiiiiiii	—	3	—
PREF	@Rn	(Rn) → cache	0000nnnn10000011	—	1/2*6	—
ROTCL	Rn	T ← Rn ← T	0100nnnn00100100	—	1	MSB
ROTCR	Rn	T → Rn → T	0100nnnn00100101	—	1	LSB
ROTL	Rn	T ← Rn ← MSB	0100nnnn00000100	—	1	MSB
ROTR	Rn	LSB → Rn → T	0100nnnn00000101	—	1	LSB
RTE		Delayed branch, SSR/SPC → SR/PC	0000000000101011	√	4	—
RTS		Delayed branch, PR → PC	0000000000001011	—	2	—
SETRC	Rm*9	12 lower bits of Rm → RC (SR bits 27 to 16), repeat control flag → RF1, RF0	0100mmmm00010100	—	3	—
SETRC	#imm*9	imm → RC (SR bits 23 to 16), repeat control flag → RF1, RF0	10000010iiiiiiii	—	3	—
SETS		1 → S	0000000001011000	—	1	—
SETT		1 → T	0000000000011000	—	1	1
SHAD	Rm,Rn	Rn ≥ 0; Rn << Rm → Rn Rn < 0; Rn >> Rm → (MSB→)Rn	0100nnnnmmmm1100	—	1	—
SHAL	Rn	T ← Rn ← 0	0100nnnn00100000	—	1	MSB
SHAR	Rn	MSB → Rn → T	0100nnnn00100001	—	1	LSB
SHLD	Rm,Rn	Rn ≥ 0; Rn << Rm → Rn Rn < 0; Rn >> Rm → (0→)Rn	0100nnnnmmmm1101	—	1	—
SHLL	Rn	T ← Rn ← 0	0100nnnn00000000	—	1	MSB

Table 7-12 Instruction Set Listed Alphabetically (cont)

Instruction		Operation	Code	Privilege	Cycles	T Bit
SHLL2	Rn	Rn << 2 → Rn	0100nnnn00001000	—	1	—
SHLL8	Rn	Rn << 8 → Rn	0100nnnn00011000	—	1	—
SHLL16	Rn	Rn << 16 → Rn	0100nnnn00101000	—	1	—
SHLR	Rn	0 → Rn → T	0100nnnn00000001	—	1	LSB
SHLR2	Rn	Rn>>2 → Rn	0100nnnn00001001	—	1	—
SHLR8	Rn	Rn>>8 → Rn	0100nnnn00011001	—	1	—
SHLR16	Rn	Rn>>16 → Rn	0100nnnn00101001	—	1	—
SLEEP		Sleep	0000000000011011	√	4	—
STC	GBR,Rn	GBR → Rn	0000nnnn00010010	—	1	—
STC	SR,Rn	SR → Rn	0000nnnn00000010	√	1	—
STC	VBR,Rn	VBR → Rn	0000nnnn00100010	√	1	—
STC	SSR,Rn	SSR → Rn	0000nnnn00110010	√	1	—
STC	SPC,Rn	SPC → Rn	0000nnnn01000010	√	1	—
STC	MOD,Rn*9	MOD → Rn	0000nnnn01010010	—	1	—
STC	RE,Rn*9	RE → Rn	0000nnnn01110010	—	1	—
STC	RS,Rn*9	RS → Rn	0000nnnn01100010	—	1	—
STC	R0_BANK,Rn	R0_BANK→ Rn	0000nnnn10000010	√	1	—
STC	R1_BANK,Rn	R1_BANK→ Rn	0000nnnn10010010	√	1	—
STC	R2_BANK,Rn	R2_BANK→ Rn	0000nnnn10100010	√	1	—
STC	R3_BANK,Rn	R3_BANK→ Rn	0000nnnn10110010	√	1	—
STC	R4_BANK,Rn	R4_BANK→ Rn	0000nnnn11000010	√	1	—
STC	R5_BANK,Rn	R5_BANK→ Rn	0000nnnn11010010	√	1	—
STC	R6_BANK,Rn	R6_BANK→ Rn	0000nnnn11100010	√	1	—
STC	R7_BANK,Rn	R7_BANK→ Rn	0000nnnn11110010	√	1	—
STC.L	GBR,@–Rn	Rn–4 → Rn, GBR → (Rn)	0100nnnn00010011	—	1/2*6	—
STC.L	SR,@–Rn	Rn–4 → Rn, SR → (Rn)	0100nnnn00000011	√	1/2*6	—
STC.L	VBR,@–Rn	Rn–4 → Rn, VBR → (Rn)	0100nnnn00100011	√	1/2*6	—
STC.L	SSR,@–Rn	Rn–4 → Rn, SSR → (Rn)	0100nnnn00110011	√	1/2*6	—
STC.L	SPC,@–Rn	Rn–4 → Rn, SPC → (Rn)	0100nnnn01000011	√	1/2*6	—
STC.L	MOD,@-Rn*9	Rn–4 → Rn, MOD → (Rn)	0100nnnn01010011	√	2	—
STC.L	RE,@-Rn*9	Rn–4 → Rn, RE → (Rn)	0100nnnn01110011	√	2	—

Table 7-12 Instruction Set Listed Alphabetically (cont)

Instruction	Operation	Code	Privilege	Cycles	T Bit
STC.L RS,@-Rn*9	Rn–4 → Rn, RS → (Rn)	0100nnnn01100011	√	2	—
STC.L R0_BANK,@–Rn	Rn–4 → Rn, R0_BANK → (Rn)	0100nnnn10000011	√	2	—
STC.L R1_BANK,@–Rn	Rn–4 → Rn, R1_BANK → (Rn)	0100nnnn10010011	√	2	—
STC.L R2_BANK,@–Rn	Rn–4 → Rn, R2_BANK → (Rn)	0100nnnn10100011	√	2	—
STC.L R3_BANK,@–Rn	Rn–4 → Rn, R3_BANK → (Rn)	0100nnnn10110011	√	2	—
STC.L R4_BANK,@–Rn	Rn–4 → Rn, R4_BANK → (Rn)	0100nnnn11000011	√	2	—
STC.L R5_BANK,@–Rn	Rn–4 → Rn, R5_BANK → (Rn)	0100nnnn11010011	√	2	—
STC.L R6_BANK,@–Rn	Rn–4 → Rn, R6_BANK → (Rn)	0100nnnn11100011	√	2	—
STC.L R7_BANK,@–Rn	Rn–4 → Rn, R7_BANK → (Rn)	0100nnnn11110011	√	2	—
STS FPSCR, Rn*3	FPSCR → Rn	0000nnnn01101010	—	1	—
STS FPUL, Rn*3	FPUL → Rn	0000nnnn01011010	—	1	—
STS MACH,Rn	MACH → Rn	0000nnnn00001010	—	1	—
STS MACL,Rn	MACL → Rn	0000nnnn00011010	—	1	—
STS PR,Rn	PR → Rn	0000nnnn00101010	—	1	—
STS DSR,Rn*9	DSR → Rn	0000nnnn01101010	—	1	—
STS A0,Rn*9	A0 → Rn	0000nnnn01111010	—	1	—
STS X0,Rn*9	X0 → Rn	0000nnnn10001010	—	1	—
STS X1,Rn*9	X1 → Rn	0000nnnn10011010	—	1	—
STS Y0,Rn*9	Y0 → Rn	0000nnnn10101010	—	1	—
STS Y1,Rn*9	Y1 → Rn	0000nnnn10111010	—	1	—
STS.L FPSCR,@-Rn*3	Rn-4 → Rn, FPSCR → @Rn	0100nnnn01100010	—	1	—
STS.L FPUL,@-Rn*3	Rn-4 → Rn, FPUL → @Rn	0100nnnn01010010	—	1	—
STS.L MACH,@–Rn	Rn–4 → Rn, MACH → (Rn)	0100nnnn00000010	—	1	—
STS.L MACL,@–Rn	Rn–4 → Rn, MACL → (Rn)	0100nnnn00010010	—	1	—
STS.L PR,@–Rn	Rn–4 → Rn, PR → (Rn)	0100nnnn00100010	—	1	—
STS.L DSR,@-Rn*9	Rn–4 → Rn, DSR → (Rn)	0100nnnn01100010	—	1	—

Table 7-12 Instruction Set Listed Alphabetically (cont)

Instruction	Operation	Code	Privilege	Cycles	T Bit
STS.L A0,@-Rn*9	Rn–4 → Rn, A0 → (Rn)	0100nnnn01110010	—	1	—
STS.L X0,@-Rn*9	Rn–4 → Rn, X0 → (Rn)	0100nnnn10000010	—	1	—
STS.L X1,@-Rn*9	Rn–4 → Rn, X1 → (Rn)	0100nnnn10010010	—	1	—
STS.L Y0,@-Rn*9	Rn–4 → Rn, Y0 → (Rn)	0100nnnn10100010	—	1	—
STS.L Y1,@-Rn*9	Rn–4 → Rn, Y1 → (Rn)	0100nnnn10110010	—	1	—
SUB Rm,Rn	Rn–Rm → Rn	0011nnnnmmmm1000	—	1	—
SUBC Rm,Rn	Rn–Rm–T → Rn, Borrow → T	0011nnnnmmmm1010	—	1	Borrow
SUBV Rm,Rn	Rn–Rm → Rn, Underflow → T	0011nnnnmmmm1011	—	1	Underflow
SWAP.B Rm,Rn	Rm → Swap the two lowest-order bytes → Rn	0110nnnnmmmm1000	—	1	—
SWAP.W Rm,Rn	Rm → Swap two consecutive words → Rn	0110nnnnmmmm1001	—	1	—
TAS.B @Rn	If (Rn) is 0, 1 → T; 1 → MSB of (Rn)	0100nnnn00011011	—	3/4*7	Test result
TRAPA #imm	PC/SR → SPC/SSR, (#imm) <<2 → TRA VBR + H'0100 → PC	11000011iiiiiiii	—	6/8*8	—
TST #imm,R0	R0 & imm; if the result is 0, 1 → T	11001000iiiiiiii	—	1	Test result
TST Rm,Rn	Rn & Rm; if the result is 0, 1 → T	0010nnnnmmmm1000	—	1	Test result
TST.B #imm, @(R0,GBR)	(R0 + GBR) & imm; if the result is 0, 1 → T	11001100iiiiiiii	—	3	Test result
XOR #imm,R0	R0 ^ imm → R0	11001010iiiiiiii	—	1	—
XOR Rm,Rn	Rn ^ Rm → Rn	0010nnnnmmmm1010	—	1	—
XOR.B #imm, @(R0,GBR)	(R0 + GBR) ^ imm → (R0 + GBR)	11001110iiiiiiii	—	3	—
XTRCT Rm,Rn	Rm: Middle 32 bits of Rn → Rn	0010nnnnmmmm1101	—	1	—

Notes:
1. The normal minimum number of execution cycles. The number in parentheses is the number of cycles when there is contention with following instructions.
2. One state when it does not branch.
3. Indicates floating point instructions and FPU related CPU instructions. These instructions can only be used with the SH-3E.
4. Three cycles on the SH3-DSP.
5. Five cycles on the SH3-DSP.
6. Two cycles on the SH3-DSP.
7. Four cycles on the SH3-DSP.
8. Eight cycles on the SH3-DSP.
9. CPU instructions to provide support for DSP functions. These instructions can only be used with the SH3-DSP.

7.3 DSP Data Transfer Instruction Set (SH3-DSP Only)

Table 7-13 shows the DSP data transfer instructions by category.

Table 7-13 DSP Data Transfer Instruction Categories

Category	Instruction Types	Operation Code	Function	No. of Instructions
Double data transfer instructions	4	NOPX	X memory no operation	14
		MOVX	X memory data transfer	
		NOPY	Y memory no operation	
		MOVY	Y memory data transfer	
Single data transfer instructions	1	MOVS	Single data transfer	16
	Total 5			Total 30

The data transfer instructions are divided into two groups, double data transfers and single data transfers. Double data transfers are combined with DSP operation instructions to create DSP parallel processing instructions. Parallel processing instructions are 32 bits long and include a double data transfer instruction in field A. Double data transfers that are not parallel processing instructions and single data transfer instructions are 16 bits long.

In double data transfers, X memory and Y memory can be accessed simultaneously in parallel. One instruction is specified each for the respective X and Y memory data accesses. The Ax pointer is used for accessing X memory; the Ay pointer is used for accessing Y memory. Double data transfers can only access X and Y memory.

Single data transfers can be accessed from any area. In single data transfers, the Ax pointer and two other pointers are used as the As pointer.

7.3.1 Double Data Transfer Instructions (X Memory Data)

Table 7-14 Double Data Transfer Instructions (X Memory Data)

Instruction	Operation	Code	Cycles	T Bit
NOPX	No Operation	1111000*0*0*00**	1	—
MOVX.W @Ax,Dx	(Ax)→MSW of Dx,0→LSW of Dx	111100A*D*0*01**	1	—
MOVX.W @Ax+,Dx	(Ax)→MSW of Dx,0→LSW of Dx,Ax+2→Ax	111100A*D*0*10**	1	—
MOVX.W @Ax+Ix,Dx	(Ax)→MSW of Dx,0→LSW of Dx,Ax+Ix→Ax	111100A*D*0*11**	1	—
MOVX.W Da,@Ax	MSW of Da→(Ax)	111100A*D*1*01**	1	—
MOVX.W Da,@Ax+	MSW of Da→(Ax),Ax+2→Ax	111100A*D*1*10**	1	—
MOVX.W Da,@Ax+Ix	MSW of Da→(Ax),Ax+Ix→Ax	111100A*D*1*11**	1	—

7.3.2 Double Data Transfer Instructions (Y Memory Data)

Table 7-15 Double Data Transfer Instructions (Y Memory Data)

Instruction	Operation	Code	Cycles	T Bit
NOPY	No Operation	111100*0*0*0**00	1	—
MOVY.W @Ay,Dy	(Ay)→MSW of Dy,0→LSW of Dy	111100*A*D*0**01	1	—
MOVY.W @Ay+,Dy	(Ay)→MSW of Dy,0→LSW of Dy, Ay+2→Ay	111100*A*D*0**10	1	—
MOVY.W @Ay+Iy,Dy	(Ay)→MSW of Dy,0→LSW of Dy, Ay+Iy→Ay	111100*A*D*0**11	1	—
MOVY.W Da,@Ay	MSW of Da→(Ay)	111100*A*D*1**01	1	—
MOVY.W Da,@Ay+	MSW of Da→(Ay),Ay+2→Ay	111100*A*D*1**10	1	—
MOVY.W Da,@Ay+Iy	MSW of Da→(Ay),Ay+Iy→Ay	111100*A*D*1**11	1	—

7.3.3 Single Data Transfer Instructions

Table 7-16 Single Data Transfer Instructions

Instruction	Operation	Code	Cycles	T Bit
MOVS.W @-As,Ds	As–2→As,(As)→MSW of Ds,0→LSW of Ds	111101AADDDD0000	1	—
MOVS.W @As,Ds	(As)→MSW of Ds,0→LSW of Ds	111101AADDDD0100	1	—
MOVS.W @As+,Ds	(As)→MSW of Ds,0→LSW of Ds, As+2→As	111101AADDDD1000	1	—
MOVS.W @As+Ix,Ds	(As)→MSW of Ds,0→LSW of Ds, As+Ix→As	111101AADDDD1100	1	—
MOVS.W Ds,@-As	As–2→As,MSW of Ds→(As)*	111101AADDDD0001	1	—
MOVS.W Ds,@As	MSW of Ds→(As)*	111101AADDDD0101	1	—
MOVS.W Ds,@As+	MSW of Ds→(As),As+2→As*	111101AADDDD1001	1	—
MOVS.W Ds,@As+Is	MSW of Ds→(As),As+Is→As*	111101AADDDD1101	1	—
MOVS.L @-As,Ds	As–4→As,(As)→Ds	111101AADDDD0010	1	—
MOVS.L @As,Ds	(As)→Ds	111101AADDDD0110	1	—
MOVS.L @As+,Ds	(As)→Ds,As+4→As	111101AADDDD1010	1	—
MOVS.L @As+Is,Ds	(As)→Ds,As+Is→As	111101AADDDD1110	1	—
MOVS.L Ds, @-As	As–4→As,Ds→(As)	111101AADDDD0011	1	—
MOVS.L Ds,@As	Ds→(As)	111101AADDDD0111	1	—
MOVS.L Ds,@As+	Ds→(As),As+4→As	111101AADDDD1011	1	—
MOVS.L Ds,@As+Is	Ds→(As),As+Is→As	111101AADDDD1111	1	—

Note: * When guard bit registers A0G and A1G are specified for the source operand Ds, data is output to the LDB[7:0] bus and the sign bit is output to the top bits [31:8].

Table 7-17 lists the correspondence between DSP data transfer operands and registers. CPU core registers are used as pointer addresses to indicate memory addresses.

Table 7-17 Correspondence between DSP Data Transfer Operands and Registers

Operand	SuperH (CPU Core) Registers									
	R0	R1	R2 (As2)	R3 (As3)	R4 (Ax0) (As0)	R5 (Ax1) (Ax0)	R6 (Ay0)	R7 (Ay1)	R8 (Ix)	R9 (Iy)
Ax	—	—	—	—	Yes	Yes	—	—	—	—
Ix (Is)	—	—	—	—	—	—	—	—	Yes	—
Dx	—	—	—	—	—	—	—	—	—	—
Ay	—	—	—	—	—	—	Yes	Yes	—	—
Iy	—	—	—	—	—	—	—	—	—	Yes
Dy	—	—	—	—	—	—	—	—	—	—
Da	—	—	—	—	—	—	—	—	—	—
As	—	—	Yes	Yes	Yes	Yes	—	—	—	—
Ds	—	—	—	—	—	—	—	—	—	—

Operand	DSP Registers									
	X0	X1	Y0	Y1	M0	M1	A0	A1	A0G	A1G
Ax	—	—	—	—	—	—	—	—	—	—
Ix (Is)	—	—	—	—	—	—	—	—	—	—
Dx	Yes	Yes	—	—	—	—	—	—	—	—
Ay	—	—	—	—	—	—	—	—	—	—
Iy	—	—	—	—	—	—	—	—	—	—
Dy	—	—	Yes	Yes	—	—	—	—	—	—
Da	—	—	—	—	—	—	Yes	Yes	—	—
As	—	—	—	—	—	—	—	—	—	—
Ds	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes

Note: Yes indicates that the register can be set.

7.4 DSP Operation Instruction Set (SH3-DSP Only)

DSP operation instructions are digital signal processing instructions that are processed by the DSP unit. Their instruction code is 32 bits long. Multiple instructions can be processed in parallel. The instruction code is divided into two fields, A and B. Field A specifies a parallel data transfer instruction and field B specifies a single or double data operation instruction. Instructions can be specified independently, and their execution is independent and in parallel. Parallel data transfer instructions specified in field A are exactly the same as double data transfer instructions.

The data operation instructions of field B are of three types: double data operation instructions, conditional single data operation instructions, and unconditional single data operation instructions. Table 7-18 shows the format of DSP operation instructions. The operands are selected independently from the DSP register. Table 7-19 shows the correspondence of DSP operation instruction operands and registers.

Table 7-18 Instruction Formats for DSP Operation Instructions

Classification		Instruction Forms	Instruction
Double data operation instructions (6 operands)		ALUop. Sx, Sy, Du MLTop. Se, Sf, Dg	PADD PMULS, PSUB PMULS
Conditional single data operation instructions	3 operands	ALUop. Sx, Sy, Dz DCT ALUop. Sx, Sy, Dz DCF ALUop. Sx, Sy, Dz	PADD, PAND, POR, PSHA, PSHL, PSUB, PXOR
	2 operands	ALUop. Sx, Dz DCT ALUop. Sx, Dz DCF ALUop. Sx, Dz ALUop. Sy, Dz DCT ALUop. Sy, Dz DCF ALUop. Sy, Dz	PCOPY, PDEC, PDMSB, PINC, PLDS, PSTS, PNEG
	1 operand	ALUop. Dz DCT ALUop. Dz DCF ALUop. Dz	PCLR, PSHA #imm, PSHL #imm
Unconditional single data operation instructions	3 operands	ALUop. Sx, Sy, Du MLTop. Se, Sf, Dg	PADDC, PSUBC, PWADD, PWSB, PMULS
	2 operands	ALUop. Sx, Dz ALUop. Sy, Dz	PCMP, PABS, PRND

Table 7-19 Correspondence between DSP Operation Instruction Operands and Registers

	ALU and BPU Instructions				Multiplication Instructions		
Register	**Sx**	**Sy**	**Dz**	**Du**	**Se**	**Sf**	**Dg**
A0	Yes	—	Yes	Yes	—	—	Yes
A1	Yes	—	Yes	Yes	Yes	Yes	Yes
M0	—	Yes	Yes	—	—	—	Yes
M1	—	Yes	Yes	—	—	—	Yes
X0	Yes	—	Yes	Yes	Yes	Yes	—
X1	Yes	—	Yes	—	Yes	—	—
Y0	—	Yes	Yes	Yes	Yes	Yes	—
Y1	—	Yes	Yes	—	—	Yes	—

When writing parallel instructions, first write the field B instruction, then the field A instruction. The following is an example of a parallel processing program.

```
PADD A0,M0,A0 PMULSX0,Y0,M0      MOVX.W @R4+,X0        MOVY.W @R6+,Y0[;]
DCF PINC X1,A1                   MOVX.W A0,@R5+R8      MOVY.W@R7+,Y0[;]
PCMP X1,M0                       MOVX.W @R4            [NOPY][;]
```

Text in brackets ([]) can be omitted. The no operation instructions NOPX and NOPY can be omitted. Semicolons (;) are used to demarcate instruction lines, but can be omitted. If semicolons are used, the space after the semicolon can be used for comments.

The individual status codes (DC, N, Z, V, GT) of the DSR register is always updated by unconditional ALU operation instructions and shift operation instructions. Conditional instructions do not update the status codes, even if the conditions have been met. Multiplication instructions also do not update the status codes. DC bit definitions are determined by the specifications of the CS bits in the DSR register.

Table 7-20 shows the DSP operation instructions by category.

Table 7-20 DSP Operation Instruction Categories

Classification		Instruction Types	Operation Code	Function	No. of Instructions
ALU arithmetic operation instructions	ALU fixed decimal point operation instructions	11	PABS	Absolute value operation	28
			PADD	Addition	
			PADD PMULS	Addition and signed multiplication	
			PADDC	Addition with carry	
			PCLR	Clear	
			PCMP	Compare	
			PCOPY	Copy	
			PNEG	Invert sign	
			PSUB	Subtraction	
			PSUB PMULS	Subtraction and signed multiplication	
			PSUBC	Subtraction with borrow	
	ALU integer operation instructions	2	PDEC	Decrement	12
			PINC	Increment	
	MSB detection instruction	1	PDMSB	MSB detection	6
	Rounding operation instruction	1	PRND	Rounding	2
ALU logical operation instructions		3	PAND	Logical AND	9
			POR	Logical OR	
			PXOR	Logical exclusive OR	
Fixed decimal point multiplication instruction		1	PMULS	Signed multiplication	1
Shift	Arithmetic shift operation instruction	1	PSHA	Arithmetic shift	4
	Logical shift operation instruction	1	PSHL	Logical shift	4
System control instructions		2	PLDS	System register load	12
			PSTS	Store from system register	
		Total 23			Total 78

7.4.1 ALU Arithmetic Operation Instructions

ALU Fixed Decimal Point Operation Instructions

Table 7-21 ALU Fixed Decimal Point Operation Instructions

Instruction	Operation	Code	Cycles	DC Bit
PABS Sx,Dz	If Sx≥0,Sx→Dz If Sx<0,0– Sx→Dz	111110********** 10001000xx00zzzz	1	Update
PABS Sy,Dz	If Sy≥0,Sy→Dz If Sy<0,0–Sy→Dz	111110********** 1010100000yyzzzz	1	Update
PADD Sx,Sy,Dz	Sx+Sy→Dz	111110********** 10110001xxyyzzzz	1	Update
DCT PADD Sx,Sy,Dz	if DC=1,Sx+Sy→Dz if 0,nop	111110********** 10110010xxyyzzzz	1	—
DCF PADD Sx,Sy,Dz	if DC=0,Sx+Sy→Dz if 1,nop	111110********** 10110011xxyyzzzz	1	—
PADD Sx,Sy,Du PMULS Se,Sf,Dg	Sx+Sy→Du MSW of Se × MSW of Sf→Dg	111110********** 0111eeffxxyygguu	1	Update
PADDC Sx,Sy,Dz	Sx+Sy+DC→Dz	111110********** 10110000xxyyzzzz	1	Update
PCLR Dz	H'00000000→Dz	111110********** 100011010000zzzz	1	Update
DCT PCLR Dz	if DC=1,H'00000000→Dz if 0,nop	111110********** 100011100000zzzz	1	—
DCF PCLR Dz	if DC=0,H'00000000→Dz if 1,nop	111110********** 100011110000zzzz	1	—
PCMP Sx,Sy	Sx–Sy	111110********** 10000100xxyy0000	1	Update
PCOPY Sx,Dz	Sx→Dz	111110********** 11011001xx00zzzz	1	Update
PCOPY Sy,Dz	Sy→Dz	111110********** 1111100100yyzzzz	1	Update
DCT PCOPY Sx,Dz	if DC=1,Sx→Dz if 0,nop	111110********** 11011010xx00zzzz	1	—

Table 7-21 ALU Fixed Decimal Point Operation Instructions (cont)

Instruction	Operation	Code	Cycles	DC Bit
DCT PCOPY Sy,Dz	if DC=1,Sy→Dz if 0,nop	111110********** 1111101000yyzzzz	1	—
DCF PCOPY Sx,Dz	if DC=0,Sx→Dz if 1,nop	111110********** 11011011xx00zzzz	1	—
DCF PCOPY Sy,Dz	if DC=0,Sy→Dz if 1,nop	111110********** 1111101100yyzzzz	1	—
PNEG Sx,Dz	0–Sx→Dz	111110********** 11001001xx00zzzz	1	Update
PNEG Sy,Dz	0–Sy→Dz	111110********** 1110100100yyzzzz	1	Update
DCT PNEG Sx,Dz	if DC=1,0–Sx→Dz if 0,nop	111110********** 11001010xx00zzzz	1	—
DCT PNEG Sy,Dz	if DC=1,0–Sy→Dz if 0,nop	111110********** 1110101000yyzzzz	1	—
DCF PNEG Sx,Dz	if DC=0,0–Sx→Dz if 1,nop	111110********** 11001011xx00zzzz	1	—
DCF PNEG Sy,Dz	if DC=0,0–Sy→Dz if 1,nop	111110********** 1110101100yyzzzz	1	—
PSUB Sx,Sy,Dz	Sx–Sy→Dz	111110********** 10100001xxyyzzzz	1	Update
DCT PSUB Sx,Sy,Dz	if DC=1,Sx–Sy→Dz if 0,nop	111110********** 10100010xxyyzzzz	1	—
DCF PSUB Sx,Sy,Dz	if DC=0,Sx–Sy→Dz if 1,nop	111110********** 10100011xxyyzzzz	1	—
PSUB Sx,Sy,Du PMULS Se,Sf,Dg	Sx–Sy→Du MSW of Se × MSW of Sf→Dg	111110********** 0110eeffxxyygguu	1	Update
PSUBC Sx,Sy,Dz	Sx–Sy–DC→Dz	111110********** 10100000xxyyzzzz	1	Update

ALU Integer Operation Instructions

Table 7-22 ALU Integer Operation Instructions

Instruction	Operation	Code	Cycles	DC Bit
PDEC Sx,Dz	MSW of Sx – 1 → MSW of Dz, clear LSW of Dz	111110********** 10001001xx00zzzz	1	Update
PDEC Sy,Dz	MSW of Sy – 1 → MSW of Dz, clear LSW of Dz	111110********** 10101001xx00zzzz	1	Update
DCT PDEC Sx,Dz	If DC=1, MSW of Sx – 1 → MSW of Dz, clear LSW of Dz; if 0, nop	111110********** 10001010xx00zzzz	1	—
DCT PDEC Sy,Dz	If DC=1, MSW of Sy – 1 → MSW of Dz, clear LSW of Dz; if 0, nop	111110********** 10101010xx00zzzz	1	—
DCF PDEC Sx,Dz	If DC=0, MSW of Sx – 1 → MSW of Dz, clear LSW of Dz; if 1, nop	111110********** 10001011xx00zzzz	1	—
DCF PDEC Sy,Dz	If DC=0, MSW of Sy – 1 → MSW of Dz, clear LSW of Dz; if 1, nop	111110********** 10101011xx00zzzz	1	—
PINC Sx,Dz	MSW of Sx + 1 → MSW of Dz, clear LSW of Dz	111110********** 10011001xx00zzzz	1	Update
PINC Sy,Dz	MSW of Sy + 1 → MSW of Dz, clear LSW of Dz	111110********** 1011100100yyzzzz	1	Update
DCT PINC Sx,Dz	If DC=1, MSW of Sx + 1 → MSW of Dz, clear LSW of Dz; if 0, nop	111110********** 10011010xx00zzzz	1	—
DCT PINC Sy,Dz	If DC=1, MSW of Sy + 1 → MSW of Dz, clear LSW of Dz; if 0, nop	111110********** 1011101000yyzzzz	1	—
DCF PINC Sx,Dz	If DC=0, MSW of Sx + 1 → MSW of Dz, clear LSW of Dz; if 1, nop	111110********** 10011011xx00zzzz	1	—
DCF PINC Sy,Dz	If DC=0, MSW of Sy + 1 → MSW of Dz, clear LSW of Dz; if 1, nop	111110********** 1011101100yyzzzz	1	—

MSB Detection Instructions

Table 7-23 MSB Detection Instructions

Instruction	Operation	Code	Cycles	DC Bit
PDMSB Sx,Dz	Sx data MSB position → MSW of Dz, clear LSW of Dz	111110********** 10011101xx00zzzz	1	Update
PDMSB Sy,Dz	Sy data MSB position → MSW of Dz, clear LSW of Dz	111110********** 1011110100yyzzzz	1	Update
DCT PDMSB Sx,Dz	If DC=1, Sx data MSB position → MSW of Dz, clear LSW of Dz; if 0, nop	111110********** 10011110xx00zzzz	1	—
DCT PDMSB Sy,Dz	If DC=1, Sy data MSB position → MSW of Dz, clear LSW of Dz; if 0, nop	111110********** 1011111000yyzzzz	1	—
DCF PDMSB Sx,Dz	If DC=0, Sx data MSB position → MSW of Dz, clear LSW of Dz; if 1, nop	111110********** 10011111xx00zzzz	1	—
DCF PDMSB Sy,Dz	If DC=0, Sy data MSB position → MSW of Dz, clear LSW of Dz; if 1, nop	111110********** 1011111100yyzzzz	1	—

Rounding Operation Instructions

Table 7-24 Rounding Operation Instructions

Instruction	Operation	Code	Cycles	DC Bit
PRND Sx,Dz	Sx+H'00008000→Dz clear LSW of Dz	111110********** 10011000xx00zzzz	1	Update
PRND Sy,Dz	Sy+H'00008000→Dz clear LSW of Dz	111110********** 1011100000yyzzzz	1	Update

7.4.2 ALU Logical Operation Instructions

Table 7-25 ALU Logical Operation Instructions

Instruction	Operation	Code	Cycles	DC Bit
PAND Sx,Sy,Dz	Sx & Sy → Dz, clear LSW of Dz	111110********** 10010101xxyyzzzz	1	Update
DCT PAND Sx,Sy,Dz	If DC=1, Sx & Sy → Dz, clear LSW of Dz; if 0, nop	111110********** 10010110xxyyzzzz	1	—
DCF PAND Sx,Sy,Dz	If DC=0, Sx & Sy → Dz, clear LSW of Dz; if 1, nop	111110********** 10010111xxyyzzzz	1	—
POR Sx,Sy,Dz	Sx | Sy → Dz, clear LSW of Dz	111110********** 10110101xxyyzzzz	1	Update
DCT POR Sx,Sy,Dz	If DC=1, Sx | Sy → Dz, clear LSW of Dz; if 0, nop	111110********** 10110110xxyyzzzz	1	—
DCF POR Sx,Sy,Dz	If DC=0, Sx | Sy → Dz, clear LSW of Dz; if 1, nop	111110********** 10110111xxyyzzzz	1	—
PXOR Sx,Sy,Dz	Sx ^ Sy → Dz, clear LSW of Dz	111110********** 10100101xxyyzzzz	1	Update
DCT PXOR Sx,Sy,Dz	If DC=1, Sx ^ Sy → Dz, clear LSW of Dz; if 0, nop	111110********** 10100110xxyyzzzz	1	—
DCF PXOR Sx,Sy,Dz	If DC=0, Sx ^ Sy → Dz, clear LSW of Dz; if 1, nop	111110********** 10100111xxyyzzzz	1	—

7.4.3 Fixed Decimal Point Multiplication Instructions

Table 7-26 Fixed Decimal Point Multiplication Instructions

Instruction	Operation	Code	Cycles	DC Bit
PMULS Se,Sf,Dg	MSW of Se × MSW of Sf→Dg	111110********** 0100eeff0000gg00	1	—

7.4.4 Shift Operation Instructions

Arithmetic Shift Instructions

Table 7-27 Arithmetic Shift Instructions

Instruction	Operation	Code	Cycles	DC Bit
PSHA Sx,Sy,Dz	if Sy≥0,Sx<<Sy→Dz if Sy<0,Sx>>Sy→Dz	111110********** 10010001xxyyzzzz	1	Update
DCT PSHA Sx,Sy,Dz	if DC=1 & Sy≥0,Sx<<Sy→Dz if DC=1 & Sy<0,Sx>>Sy→Dz if DC=0,nop	111110********** 10010010xxyyzzzz	1	—
DCF PSHA Sx,Sy,Dz	if DC=0 & Sy≥0,Sx<<Sy→Dz if DC=0 & Sy<0,Sx>>Sy→Dz if DC=1,nop	111110********** 10010011xxyyzzzz	1	—
PSHA #imm,Dz	if imm≥0,Dz<<imm→Dz if imm<0,Dz>>imm→Dz	111110********** 00000iiiiiiizzzz	1	Update

Logical Shift Operation Instructions

Table 7-28 Logical Shift Operation Instructions

Instruction	Operation	Code	Cycles	DC Bit
`PSHL Sx,Sy,Dz`	if Sy≥0,Sx<<Sy→Dz, clear LSW of Dz if Sy<0,Sx>>Sy→Dz, clear LSW of Dz	`111110**********` `10000001xxyyzzzz`	1	Update
`DCT PSHL Sx,Sy,Dz`	if DC=1 & Sy≥0,Sx<<Sy→Dz, clear LSW of Dz if DC=1 & Sy<0,Sx>>Sy→Dz, clear LSW of Dz if DC=0,nop	`111110**********` `10000010xxyyzzzz`	1	—
`DCF PSHL Sx,Sy,Dz`	if DC=0 & Sy≥0,Sx<<Sy→Dz, clear LSW of Dz if DC=0 & Sy<0,Sx>>Sy→Dz, clear LSW of Dz if DC=1,nop	`111110**********` `10000011xxyyzzzz`	1	—
`PSHL #imm,Dz`	if imm≥0,Dz<<imm→Dz, clear LSW of Dz if imm<0,Dz>>imm→Dz, clear LSW of Dz	`111110**********` `00010iiiiiiizzzz`	1	Update

7.4.5 System Control Instructions

Table 7-29 System Control Instructions

Instruction	Operation	Code	Cycles	DC Bit
PLDS Dz,MACH	Dz→MACH	111110********** 111011010000zzzz	1	—
PLDS Dz,MACL	Dz→MACL	111110********** 111111010000zzzz	1	—
DCT PLDS Dz,MACH	if DC=1,Dz→MACH if 0,nop	111110********** 111011100000zzzz	1	—
DCT PLDS Dz,MACL	if DC=1,Dz→MACL if 0,nop	111110********** 111111100000zzzz	1	—
DCF PLDS Dz,MACH	if DC=0,Dz→MACH if 1,nop	111110********** 111011110000zzzz	1	—
DCF PLDS Dz,MACL	if DC=0,Dz→MACL if 1,nop	111110********** 111111110000zzzz	1	—
PSTS MACH,Dz	MACH→Dz	111110********** 110011010000zzzz	1	—
PSTS MACL,Dz	MACL→Dz	111110********** 110111010000zzzz	1	—
DCT PSTS MACH,Dz	if DC=1,MACH→Dz if 0,nop	111110********** 110011100000zzzz	1	—
DCT PSTS MACL,Dz	if DC=1,MACL→Dz if 0,nop	111110********** 110111100000zzzz	1	—
DCF PSTS MACH,Dz	if DC=0,MACH→Dz if 1,nop	111110********** 110011110000zzzz	1	—
DCF PSTS MACL,Dz	if DC=0,MACL→Dz if 1,nop	111110********** 110111110000zzzz	1	—

7.4.6 NOPX and NOPY Instruction Code

When there is no data transfer instruction to be processed in parallel with the DSP operation instruction, a NOPX or NOPY instruction can be written as the data transfer instruction or the instruction can be omitted. The operation code is the same in either case. Table 7-30 shows the NOPX and NOPY instruction code.

Table 7-30 Sample NOPX and NOPY Instruction Code

Instruction			Code
PADD X0, Y0, A0	MOVX. W @R4+, X0	MOVY.W @R6+R9, Y0	1111100010110000 1000000010100000
PADD X0, Y0, A0	NOPX	MOVY.W @R6+R9, Y0	1111100000110000 1000000010100000
PADD X0, Y0, A0	NOPX	NOPY	1111100000000000 1000000010100000
PADD X0, Y0, A0	NOPX		
PADD X0, Y0, A0			
	MOVX. W @R4+, X0	MOVY.W @R6+R9, Y0	1111000010110000
	MOVX. W @R4+, X0	NOPY	1111000010000000
	MOVS. W @R4+, X0		1111011010000000
	NOPX	MOVY.W @R6+R9, Y0	1111000000110000
		MOVY.W @R6+R9, Y0	
	NOPX	NOPY	1111000000000000
NOP			0000000000001001

Section 8 Instruction Descriptions

This section describes instructions in alphabetical order using the format shown below in section 8.1. The actual descriptions begin at section 8.2.

8.1 Sample Description (Name): Classification

Class: Indicates if the instruction is a delayed branch instruction or interrupt disabled instruction

Format	Abstract	Code	Cycle	T Bit
Assembler input format; imm and disp are numbers, expressions, or symbols	A brief description of operation	Displayed in order MSB ↔ LSB	Number of cycles when there is no wait state	The value of T bit after the instruction is executed

Note: Section 8.2 contains an description of CPU instructions common to the SH-3, SH-3E, and SH3-DSP, section 8.3 covers floating point instructions that can only be used with the SH-3E, and section 8.4 covers DSP data transfer instructions that can only be used with the SH3-DSP.

The number of execution cycles required for floating point instructions is determined by the latency and pitch values. "Latency" refers to the number of cycles required to generate the result value for the operation, and "pitch" indicates the number of wait cycles required before execution of the next instruction can begin. The latency and pitch values are the same for most CPU instructions, indicating that they each require one execution cycle.

Description: Description of operation

Notes: Notes on using the instruction

Operation: Operation written in C language. This part is just a reference to help understanding of an operation. The following resources should be used.

- Reads data of each length from address Addr. An address error will occur if word data is read from an address other than 2n or if longword data is read from an address other than 4n:

```
unsigned char   Read_Byte(unsigned long Addr);
unsigned short  Read_Word(unsigned long Addr);
unsigned long   Read_Long(unsigned long Addr);
```

- Writes data of each length to address Addr. An address error will occur if word data is written to an address other than 2n or if longword data is written to an address other than 4n:

```
unsigned char   Write_Byte(unsigned long Addr, unsigned long Data);
unsigned short  Write_Word(unsigned long Addr, unsigned long Data);
unsigned long   Write_Long(unsigned long Addr, unsigned long Data);
```

- Starts execution from the slot instruction located at an address (Addr – 4). For Delay_Slot (4), execution starts from an instruction at address 0 rather than address 4. The following instructions are detected before execution as having illegal slots (they become illegal slot instructions when used as delay slot instructions):

  BF, BT, BRA, BSR, JMP, JSR, RTS, RTE, TRAPA, BF/S, BT/S, BRAF, BSRF

```
Delay_Slot(unsigned long Addr);
```

- List registers:

```
unsigned long R[16];
unsigned long SR,GBR,VBR;
unsigned long MACH,MACL,PR;
unsigned long PC;
```

- Definition of SR structures:

```
struct SR0 {
    unsigned long   dummy0:4;
    unsigned long      RC0:12;
    unsigned long   dummy1:4;
    unsigned long     DMY0:1;
    unsigned long     DMX0:1;
    unsigned long       M0:1;
    unsigned long       Q0:1;
    unsigned long       I0:4;
    unsigned long     RF10:1;
    unsigned long     RF00:1;
    unsigned long       S0:1;
    unsigned long       T0:1;
};
```

- Definition of bits in SR:

```
#define M ((*(struct SR0 *)(&SR)).M0)
#define Q ((*(struct SR0 *)(&SR)).Q0)
#define S ((*(struct SR0 *)(&SR)).S0)
#define T ((*(struct SR0 *)(&SR)).T0)
#define RF1 ((*(struct SR0 *)(&SR)).RF10)
#define RF0 ((*(struct SR0 *)(&SR)).RF00)
```

- Error display function:

```
Error( char *er );
```

The PC should point to the location four bytes (the second instruction) after the current instruction. Therefore, `PC = 4;` means the instruction starts execution from address 0, not address 4.

Examples: Examples are written in assembler mnemonics and describe status before and after executing the instruction. Characters in italics such as *.align* are assembler control instructions (listed below). For more information, see the *Cross Assembler User Manual*.

.org	Location counter set
.data.w	Securing integer word data
.data.l	Securing integer longword data
.sdata	Securing string data
.align 2	2-byte boundary alignment
.align 4	2-byte boundary alignment
.arepeat 16	16-repeat expansion
.arepeat 32	32-repeat expansion
.aendr	End of repeat expansion of specified number

Notes: The SH series cross assembler version 1.0 does not support the conditional assembler functions.

1. For the following addressing modes involving displacement (disp), the assembler descriptors in this manual indicate values before scaling ((1, (2, (3, (4) to match the operand size. This is done to clarify the operation of the LSI device. Refer to the applicable assembler notation rules for the actual assembler descriptors.

   @(disp: 4, Rn); Register indirect with displacement

   @(disp: 8, GBR); GBR indirect with displacement

   @(disp: 8, PC); PC relative with displacement

   disp: 8, disp: 12; PC relative
2. Of the 16 bits of the instruction code, codes not assigned as instructions or privileged instructions in the user mode (excluding instructions that access GBR) are treated as general invalid instructions and invalid instruction exception processing is performed.

   Example: H'FFFF [general invalid instruction]
3. If the instruction following a delayed branching instruction such as BRA and BT/S is a general invalid instruction or a PC overwrite instruction (branching instruction, etc.) (such instructions are referred to as "slot invalid instructions"), slot invalid instruction exception processing is performed.
4. In the SH3-DSP, if a general invalid instruction, a PC overwrite instruction (branching instruction, etc.), or an instruction (SETRC, LDRS, LDRE, LDC) that overwrites the SR, RS, or RE register is contained within a repeating program (loop) consisting of

three or fewer instructions or within the final three instructions of a repeating program (loop) consisting of four or more instructions, invalid instruction exception processing is performed. For details, refer to 5.12 DSP Repeat (Loop) Control.

8.2 Instruction Description (Listing and Description of Instructions Common to the SH-3, SH-3E and SH3-DSP)

8.2.1 ADD (Add Binary): Arithmetic Instruction

Format		Abstract	Code	Cycle	T Bit
ADD	Rm,Rn	Rm + Rn → Rn	0011nnnnmmmm1100	1	—
ADD	#imm,Rn	Rn + imm → Rn	0111nnnniiiiiiii	1	—

Description: Adds general register Rn data to Rm data, and stores the result in Rn. 8-bit immediate data can be added instead of Rm data. Since the 8-bit immediate data is sign-extended to 32 bits, this instruction can add and subtract immediate data.

Operation:

```
ADD(long m,long n)    /* ADD Rm,Rn */
{
    R[n]+=R[m];
    PC+=2;
}

ADDI(long i,long n)   /* ADD #imm,Rn */
{
    if ((i&0x80)==0) R[n]+=(0x000000FF & (long)i);
    else R[n]+=(0xFFFFFF00 | (long)i);
    PC+=2;
}
```

Examples:

```
ADD    R0,R1      ;Before execution   R0 = H'7FFFFFFF,  R1 = H'00000001
                  ;After execution    R1 = H'80000000

ADD    #H'01,R2   ;Before execution   R2 = H'00000000
                  ;After execution    R2 = H'00000001

ADD    #H'FE,R3   ;Before execution   R3 = H'00000001
                  ;After execution    R3 = H'FFFFFFFF
```

8.2.2 ADDC (Add with Carry): Arithmetic Instruction

Format		Abstract	Code	Cycle	T Bit
ADDC	Rm,Rn	Rn + Rm + T → Rn, carry → T	0011nnnnmmmm1110	1	Carry

Description: Adds general register Rm data and the T bit to Rn data, and stores the result in Rn. The T bit changes according to the result. This instruction can add data that has more than 32 bits.

Operation:

```
ADDC (long m,long n)      /* ADDC Rm,Rn */
{
    unsigned long tmp0,tmp1;

    tmp1=R[n]+R[m];
    tmp0=R[n];
    R[n]=tmp1+T;
    if (tmp0>tmp1) T=1;
    else T=0;
    if (tmp1>R[n]) T=1;
    PC+=2;
}
```

Examples:

CLRT		;R0:R1 (64 bits) + R2:R3 (64 bits) = R0:R1 (64 bits)	
ADDC	R3,R1	;Before execution	T = 0, R1 = H'00000001, R3 = H'FFFFFFFF
		;After execution	T = 1, R1 = H'0000000
ADDC	R2,R0	;Before execution	T = 1, R0 = H'00000000, R2 = H'00000000
		;After execution	T = 0, R0 = H'00000001

8.2.3 ADDV (Add with V Flag Overflow Check): Arithmetic Instruction

Format		Abstract	Code	Cycle	T Bit
ADDV	Rm,Rn	Rn + Rm → Rn, overflow → T	0011nnnnmmmm1111	1	Overflow

Description: Adds general register Rn data to Rm data, and stores the result in Rn. If an overflow occurs, the T bit is set to 1.

Operation:

```
ADDV(long m,long n)      /*ADDV Rm,Rn */
{
    long dest,src,ans;

    if ((long)R[n]>=0) dest=0;
    else dest=1;
    if ((long)R[m]>=0) src=0;
    else src=1;
    src+=dest;
    R[n]+=R[m];
    if ((long)R[n]>=0) ans=0;
    else ans=1;
    ans+=dest;
    if (src==0 || src==2) {
        if (ans==1) T=1;
        else T=0;
    }
    else T=0;
    PC+=2;
}
```

Examples:

ADDV	R0,R1	;Before execution	R0 = H'00000001, R1 = H'7FFFFFFE, T = 0
		;After execution	R1 = H'7FFFFFFF, T = 0
ADDV	R0,R1	;Before execution	R0 = H'00000002, R1 = H'7FFFFFFE, T = 0
		;After execution	R1 = H'80000000, T = 1

8.2.4 AND (AND Logical): Logic Operation Instruction

Format	Abstract	Code	Cycle	T Bit
AND Rm,Rn	Rn & Rm → Rn	0010nnnnmmmm1001	1	—
AND #imm,R0	R0 & imm → R0	11001001iiiiiiii	1	—
AND.B #imm,@(R0,GBR)	(R0 + GBR) & imm → (R0 + GBR)	11001101iiiiiiii	3	—

Description: Logically ANDs the contents of general registers Rn and Rm, and stores the result in Rn. The contents of general register R0 can be ANDed with zero-extended 8-bit immediate data. 8-bit memory data pointed to by GBR relative addressing can be ANDed with 8-bit immediate data.

Note: After AND #imm, R0 is executed and the upper 24 bits of R0 are always cleared to 0.

Operation:

```
AND(long m,long n)    /* AND Rm,Rn */
{
    R[n]&=R[m]
    PC+=2;
}

ANDI(long i)   /* AND #imm,R0 */
{
    R[0]&=(0x000000FF & (long)i);
    PC+=2;
}

ANDM(long i)   /* AND.B #imm,@(R0,GBR) */
{
    long temp;

    temp=(long)Read_Byte(GBR+R[0]);
    temp&=(0x000000FF & (long)i);
    Write_Byte(GBR+R[0],temp);
    PC+=2;
}
```

Examples:

```
AND     R0,R1              ;Before execution   R0 = H'AAAAAAAA, R1 = H'55555555
                           ;After execution    R1 = H'00000000

AND     #H'0F,R0           ;Before execution   R0 = H'FFFFFFFF
                           ;After execution    R0 = H'0000000F

AND.B   #H'80,@(R0,GBR)    ;Before execution   @(R0,GBR) = H'A5
                           ;After execution    @(R0,GBR) = H'80
```

8.2.5 BF (Branch if False): Branch Instruction

Format	Abstract	Code	Cycle	T Bit
`BF label`	When T = 0, disp × 2 + PC → PC; When T = 1, nop	`10001011dddddddd`	3/1	—

Description: Reads the T bit, and conditionally branches. If T = 1, BF executes the next instruction. If T = 0, it branches. The branch destination is an address specified by PC + displacement. The PC points to the starting address of the second instruction after the branch instruction. The 8-bit displacement is sign-extended and doubled. Consequently, the relative interval from the branch destination is –256 to +254 bytes. If the displacement is too short to reach the branch destination, use BF with the BRA instruction or the like.

Note: When branching, three cycles; when not branching, one cycle. If this instruction is located in a delayed slot immediately following a delayed branch instruction, it is acknowledged as an illegal slot instruction.

Operation:

```
BF(long d) /* BF disp */
{
    long disp;

    if ((d&0x80)==0) disp=(0x000000FF & (long)d);
    else disp=(0xFFFFFF00 | (long)d);
    if (T==0) PC=PC+(disp<<1)+4;
    else PC+=2;
}
```

Example:

```
          CLRT                  ;T is always cleared to 0
          BT     TRGET_T        ;Does not branch, because T = 0
          BF     TRGET_F        ;Branches to TRGET_F, because T = 0
          NOP
          NOP                   ;← The PC location is used to calculate the branch destination
                                ;address of the BF instruction
TRGET_F:                        ;← Branch destination of the BF instruction
```

8.2.6 BF/S (Branch if False with Delay Slot): Branch Instruction

Class: Delayed branch instruction

Format	Abstract	Code	Cycle	T Bit
BF label	When T = 0, disp × 2 + PC → PC; When T = 1, nop	10001111dddddddd	2/1	—

Description: Reads the T bit, and if T = 1, BF executes the next instruction. If T = 0, it branches after executing the next instruction. The branch destination is an address specified by PC + displacement. The PC points to the starting address of the second instruction after the branch instruction. The 8-bit displacement is sign-extended and doubled. Consequently, the relative interval from the branch destination is –256 to +254 bytes. If the displacement is too short to reach the branch destination, use BF with the BRA instruction or the like.

Note: The BF/S instruction is a conditional delayed branch instruction:

Taken case: The instruction immediately following is executed before the branch. Between the time this instruction and the instruction immediately following are executed, no interrupts are accepted. When the instruction immediately following is a branch instruction, it is recognized as an illegal slot instruction.

Not taken case: This instruction operates as a nop instruction. Between the time this instruction and the instruction immediately following are executed, interrupts are accepted. When the instruction immediately following is a branch instruction, it is not recognized as an illegal slot instruction.

Operation:

```
BFS(long d)   /* BFS disp */
{
    long disp;
    unsigned long temp;

    temp=PC;
    if ((d&0x80)==0) disp=(0x000000FF & (long)d);
    else disp=(0xFFFFFF00 | (long)d);
    if (T==0) {
        PC=PC+(disp<<1)+4;
        Delay_Slot(temp+2);
    }
    else PC+=2;
}
```

Examples:

```
          SETT                ;T is always 1
          BF/S  TARGET_F      ;Does not branch, because T = 1
          NOP
          BT/S  TARGET_T      ;Branches to TARGET, because T = 1
          ADD   R0,R1         ;Executed before branch.
          NOP                 ;← The PC location is used to calculate the branch destination
                              ;address of the BT/S instruction
TRGET_T:                      ;← Branch destination of the BT/S instruction
```

Note: In delayed branching, the branching operation itself takes place after the slot instruction has been executed. However, execution of instructions (register updating, etc.) should always be done in the sequence of delayed branch instruction followed by delayed slot instruction. For example, even if a delayed slot updates a register in which the branching destination address is stored, the contents of the register before updating will be used as the branching destination address.

8.2.7 BRA (Branch): Branch Instruction

Class: Delayed branch instruction

Format		Abstract	Code	Cycle	T Bit
BRA	label	disp × 2 + PC → PC	1010dddddddddddd	2	—

Description: Branches unconditionally after executing the instruction following this BRA instruction. The branch destination is an address specified by PC + displacement. The PC points to the starting address of the second instruction after this BRA instruction. The 12-bit displacement is sign-extended and doubled. Consequently, the relative interval from the branch destination is –4096 to +4094 bytes. If the displacement is too short to reach the branch destination, this instruction must be changed to the JMP instruction. Here, a MOV instruction must be used to transfer the destination address to a register.

Note: Since this is a delayed branch instruction, the instruction after BRA is executed before branching. No interrupts are accepted between this instruction and the next instruction. If the next instruction is a branch instruction, it is acknowledged as an illegal slot instruction. If this instruction is located in a delayed slot immediately following a delayed branch instruction, it is acknowledged as an illegal slot instruction.

Operation:

```
BRA(long d)    /* BRA disp */
{
    unsigned long temp;
    long disp;

    if ((d&0x800)==0) disp=(0x00000FFF & d);
    else disp=(0xFFFFF000 | d);
    temp=PC;
    PC=PC+(disp<<1)+4;
    Delay_Slot(temp+2);
}
```

Examples:

```
        BRA     TRGET   ;Branches to TRGET
        ADD     R0,R1   ;Executes ADD before branching
        NOP             ;← The PC location is used to calculate the branch destination
                        ;address of the BRA instruction
   TRGET:               ;← Branch destination of the BRA instruction
```

Note: In delayed branching, the branching operation itself takes place after the slot instruction has been executed. However, execution of instructions (register updating, etc.) should always be done in the sequence of delayed branch instruction followed by delayed slot instruction. For example, even if a delayed slot updates a register in which the branching destination address is stored, the contents of the register before updating will be used as the branching destination address.

8.2.8 BRAF (Branch Far): Branch Instruction

Class: Delayed branch instruction

Format		Abstract	Code	Cycle	T Bit
BRAF	Rm	Rm + PC → PC	0000nnnn00100011	2	—

Description: Branches unconditionally. The branch destination is PC + the 32-bit contents of the general register Rn. PC is the start address of the second instruction after this instruction.

Note: Since this is a delayed branch instruction, the instruction after BRAF is executed before branching. No interrupts and address errors are accepted between this instruction and the next instruction. If the next instruction is a branch instruction, it is acknowledged as an illegal slot instruction. If this instruction is located in a delayed slot immediately following a delayed branch instruction, it is acknowledged as an illegal slot instruction.

Operation:

```
BRAF(long m)   /* BRAF Rm */
{
    unsigned long temp;

    temp=PC;
    PC+=R[m];
    Delay_Slot(temp+2);
}
```

Examples:

```
        MOV.L   #(TARGET-BSRF_PC),R0    ;Sets displacement.
        BRAF    TRGET                   ;Branches to TARGET
        ADD     R0,R1                   ;Executes ADD before branching
    BRAF_PC:                            ;← The PC location is used to calculate the
                                        ;branch destination address of the BRAF
                                        ;instruction
        NOP
    TARGET:                             ;← Branch destination of the BRAF instruction
```

Note: In delayed branching, the branching operation itself takes place after the slot instruction has been executed. However, execution of instructions (register updating, etc.) should always be done in the sequence of delayed branch instruction followed by delayed slot instruction. For example, even if a delayed slot updates a register in which the branching destination address is stored, the contents of the register before updating will be used as the branching destination address.

8.2.9 BSR (Branch to Subroutine): Branch Instruction

Class: Delayed branch instruction

Format		Abstract	Code	Cycle	T Bit
BSR	label	PC → PR, disp × 2 + PC → PC	1011dddddddddddd	2	—

Description: Branches to the subroutine procedure at a specified address after executing the instruction following this BSR instruction. The PC value is stored in the PR, and the program branches to an address specified by PC + displacement. The PC points to the starting address of the second instruction after this BSR instruction. The 12-bit displacement is sign-extended and doubled. Consequently, the relative interval from the branch destination is –4096 to +4094 bytes. If the displacement is too short to reach the branch destination, the JSR instruction must be used instead. With JSR, the destination address must be transferred to a register by using the MOV instruction. This BSR instruction and the RTS instruction are used for a subroutine procedure call.

Note: Since this is a delayed branch instruction, the instruction after BSR is executed before branching. No interrupts are accepted between this instruction and the next instruction. If the next instruction is a branch instruction, it is acknowledged as an illegal slot instruction. If this instruction is located in a delayed slot immediately following a delayed branch instruction, it is acknowledged as an illegal slot instruction.

The PR used by the instruction immediately following this instruction is updated by this instruction.

Also, if the instruction immediately following this instruction generates a re-execution exception other than instruction fetch, the PR is updated by this instruction. Re-execute this instruction to recover.

Operation:

```
BSR(long d)    /* BSR disp */
{
    long disp;

    if ((d&0x800)==0) disp=(0x00000FFF & d);
    else disp=(0xFFFFF000 | d);
    PR=PC;
    PC=PC+(disp<<1)+4;
    Delay_Slot(PR+2);
}
```

Examples:

```
        BSR    TRGET     ;Branches to TRGET
        MOV    R3,R4     ;Executes the MOV instruction before branching
        ADD    R0,R1     ;← The PC location is used to calculate the branch destination
                         ;address of the BSR instruction (return address for when the
                         ;subroutine procedure is completed (PR data))
        .......
        .......
TRGET:                   ;← Procedure entrance
        MOV    R2,R3
        RTS              ;Returns to the above ADD instruction
        MOV    #1,R0     ;Executes MOV before branching
```

Note: In delayed branching, the branching operation itself takes place after the slot instruction has been executed. However, execution of instructions (register updating, etc.) should always be done in the sequence of delayed branch instruction followed by delayed slot instruction. For example, even if a delayed slot updates a register in which the branching destination address is stored, the contents of the register before updating will be used as the branching destination address.

8.2.10 BSRF (Branch to Subroutine Far): Branch Instruction

Class: Delayed branch instruction

Format	Abstract	Code	Cycle	T Bit
BSRF Rm	PC → PR, Rm + PC → PC	0000nnnn00000011	2	—

Description: Branches to the subroutine procedure at a specified address after executing the instruction following this BSRF instruction. The PC value is stored in the PR. The branch destination is PC + the 32-bit contents of the general register Rn. PC is the start address of the second instruction after this instruction. Used as a subroutine call in combination with RTS.

Note: Since this is a delayed branch instruction, the instruction after BSR is executed before branching. No interrupts are accepted between this instruction and the next instruction. If the next instruction is a branch instruction, it is acknowledged as an illegal slot instruction. If this instruction is located in a delayed slot immediately following a delayed branch instruction, it is acknowledged as an illegal slot instruction.

The PR used by the instruction immediately following this instruction is updated by this instruction.

Also, if the instruction immediately following this instruction generates a re-execution exception other than instruction fetch, the PR is updated by this instruction. Re-execute this instruction to recover.

Operation:

```
BSRF(long m)   /* BSRF Rm */
{
    PR=PC;
    PC+=R[m];
    Delay_Slot(PR+2);
}
```

Examples:

```
        MOV.L   #(TARGET-BSRF_PC),R0     ;Sets displacement.
        BRSF    @R0                      ;Branches to TARGET
        MOV     R3,R4                    ;Executes the MOV instruction before
                                         ;branching
BSRF_PC:                                 ;← The PC location is used to calculate the
                                         ;branch destination with BSRF.
        ADD     R0,R1
        .....
        .....
TARGET:         ;←Procedure entrance
        MOV     R2,R3
        RTS                              ;Returns to the above ADD instruction
        MOV     #1,R0                    ;Executes MOV before branching
```

Note: In delayed branching, the branching operation itself takes place after the slot instruction has been executed. However, execution of instructions (register updating, etc.) should always be done in the sequence of delayed branch instruction followed by delayed slot instruction. For example, even if a delayed slot updates a register in which the branching destination address is stored, the contents of the register before updating will be used as the branching destination address.

8.2.11 BT (Branch if True): Branch Instruction

Format		Abstract	Code	Cycle	T Bit
BT	label	When T = 1, disp × 2 + PC → PC; When T = 0, nop	10001001dddddddd	3/1	—

Description: Reads the T bit, and conditionally branches. If T = 1, BT branches. If T = 0, BT executes the next instruction. The branch destination is an address specified by PC + displacement. The PC points to the starting address of the second instruction after the branch instruction. The 8-bit displacement is sign-extended and doubled. Consequently, the relative interval from the branch destination is –256 to +254 bytes. If the displacement is too short to reach the branch destination, use BT with the BRA instruction or the like.

Note: When branching, requires three cycles; when not branching, one cycle. If this instruction is located in a delayed slot immediately following a delayed branch instruction, it is acknowledged as an illegal slot instruction.

Operation:

```
BT(long d) /* BT disp */
{
    long disp;

    if ((d&0x80)==0) disp=(0x000000FF & (long)d);
    else disp=(0xFFFFFF00 | (long)d);
    if (T==1) PC=PC+(disp<<1)+4;
    else PC+=2;
}
```

Examples:

```
          SETT                 ;T is always 1
          BF      TRGET_F      ;Does not branch, because T = 1
          BT      TRGET_T      ;Branches to TRGET_T, because T = 1
          NOP
          NOP                  ;← The PC location is used to calculate the branch destination
                               ;address of the BT instruction
TRGET_T:                       ;← Branch destination of the BT instruction
```

8.2.12 BT/S (Branch if True with Delay Slot): Branch Instruction

Format		Abstract	Code	Cycle	T Bit
BT/S	label	When T = 1, disp × 2 + PC → PC; When T = 0, nop	10001101dddddddd	2/1	—

Description: Reads the T bit, and if T = 1, BT/S branches after the following instruction executes. If T = 0, BT/S executes the next instruction. The branch destination is an address specified by PC + displacement. The PC points to the starting address of the second instruction after the branch instruction. The 8-bit displacement is sign-extended and doubled. Consequently, the relative interval from the branch destination is –256 to +254 bytes. If the displacement is too short to reach the branch destination, use BT/S with the BRA instruction or the like.

Note: The BF/S instruction is a conditional delayed branch instruction:

Taken case: The instruction immediately following is executed before the branch. Between the time this instruction and the instruction immediately following are executed, no interrupts are accepted. When the instruction immediately following is a branch instruction, it is recognized as an illegal slot instruction.

Not taken case: This instruction operates as a nop instruction. Between the time this instruction and the instruction immediately following are executed, interrupts are accepted. When the instruction immediately following is a branch instruction, it is not recognized as an illegal slot instruction.

Operation:

```
BTS(long d)    /* BTS disp */
{
    long disp;
    unsigned   long temp;

    temp=PC;
    if ((d&0x80)==0) disp=(0x000000FF & (long)d);
    else disp=(0xFFFFFF00 | (long)d);
    if (T==1) {
        PC=PC+(disp<<1)+4;
        Delay_Slot(temp+2);
    }
    else PC+=2;
}
```

Examples:

```
        SETT                ;T is always 1
        BF/S   TARGET_F     ;Does not branch, because T = 1
        NOP
        BT/S   TARGET_T     ;Branches to TARGET, because T = 1
        ADD    R0,R1        ;Executes before branching.
        NOP                 ;← The PC location is used to calculate the branch destination
                            ;address of the BT/S instruction
TARGET_T:                   ;← Branch destination of the BT/S instruction
```

Note: In delayed branching, the branching operation itself takes place after the slot instruction has been executed. However, execution of instructions (register updating, etc.) should always be done in the sequence of delayed branch instruction followed by delayed slot instruction. For example, even if a delayed slot updates a register in which the branching destination address is stored, the contents of the register before updating will be used as the branching destination address.

8.2.13 CLRMAC (Clear MAC Register): System Control Instruction

Format	Abstract	Code	Cycle	T Bit
CLRMAC	0 → MACH, MACL	0000000000101000	1	—

Description: Clears the MACH and MACL registers.

Operation:

```
CLRMAC()   /* CLRMAC */
{
    MACH=0;
    MACL=0;
    PC+=2;
}
```

Examples:

```
CLRMAC                ;Initializes the MAC register
MAC.W   @R0+,@R1+     ;Multiply and accumulate operation
MAC.W   @R0+,@R1+
```

8.2.14 CLRS (Clear S Bit): System Control Instruction

Format	Abstract	Code	Cycle	T Bit
CLRS	0 → S	0000000001001000	1	—

Description: Clears the S bit.

Operation:

```
CLRS() /* CLRS */
{
    S=0;
    PC+=2;
}
```

Examples:

```
CLRS        ;Before execution S=1
            ;After execution S=0
```

8.2.15 CLRT (Clear T Bit): System Control Instruction

Format	Abstract	Code	Cycle	T Bit
CLRT	0 → T	0000000000001000	1	0

Description: Clears the T bit.

Operation:

```
CLRT() /* CLRT */
{
    T=0;
    PC+=2;
}
```

Examples:

```
CLRT        ;Before execution   T = 1
            ;After execution    T = 0
```

8.2.16 CMP/cond (Compare Conditionally): Arithmetic Instruction

Format		Abstract	Code	Cycle	T Bit
CMP/EQ	Rm,Rn	When Rn = Rm, 1 → T	0011nnnnmmmm0000	1	Comparison result
CMP/GE	Rm,Rn	When signed and Rn ≥ Rm, 1 → T	0011nnnnmmmm0011	1	Comparison result
CMP/GT	Rm,Rn	When signed and Rn > Rm, 1 → T	0011nnnnmmmm0111	1	Comparison result
CMP/HI	Rm,Rn	When unsigned and Rn > Rm, 1 → T	0011nnnnmmmm0110	1	Comparison result
CMP/HS	Rm,Rn	When unsigned and Rn ≥ Rm, 1 → T	0011nnnnmmmm0010	1	Comparison result
CMP/PL	Rn	When Rn > 0, 1 → T	0100nnnn00010101	1	Comparison result
CMP/PZ	Rn	When Rn ≥ 0, 1 → T	0100nnnn00010001	1	Comparison result
CMP/STR	Rm,Rn	When a byte in Rn equals a byte in Rm, 1 → T	0010nnnnmmmm1100	1	Comparison result
CMP/EQ	#imm,R0	When R0 = imm, 1 → T	10001000iiiiiiii	1	Comparison result

Description: Compares general register Rn data with Rm data, and sets the T bit to 1 if a specified condition (cond) is satisfied. The T bit is cleared to 0 if the condition is not satisfied, and the Rn data does not change. The nine conditions in table 8-1 can be specified. Conditions PZ and PL are the results of comparisons between Rn and 0. Sign-extended 8-bit immediate data can also be compared with R0 by using condition EQ. Here, R0 data does not change. Table 8-1 shows the mnemonics for the conditions.

Table 8-1 CMP Mnemonics

Mnemonics		Condition
CMP/EQ	Rm,Rn	If Rn = Rm, T = 1
CMP/GE	Rm,Rn	If Rn ≥ Rm with signed data, T = 1
CMP/GT	Rm,Rn	If Rn > Rm with signed data, T = 1
CMP/HI	Rm,Rn	If Rn > Rm with unsigned data, T = 1
CMP/HS	Rm,Rn	If Rn ≥ Rm with unsigned data, T = 1
CMP/PL	Rn	If Rn > 0, T = 1
CMP/PZ	Rn	If Rn ≥ 0, T = 1
CMP/STR	Rm,Rn	If a byte in Rn equals a byte in Rm, T = 1
CMP/EQ	#imm,R0	If R0 = imm, T = 1

Operation:

```
CMPEQ(long m,long n)      /* CMP_EQ Rm,Rn */
{
    if (R[n]==R[m]) T=1;
    else T=0;
    PC+=2;
}

CMPGE(long m,long n)      /* CMP_GE Rm,Rn */
{
    if ((long)R[n]>=(long)R[m]) T=1;
    else T=0;
    PC+=2;
}

CMPGT(long m,long n)      /* CMP_GT Rm,Rn */
{
    if ((long)R[n]>(long)R[m]) T=1;
    else T=0;
    PC+=2;
}
```

```
CMPHI(long m,long n)     /* CMP_HI Rm,Rn */
{
    if ((unsigned long)R[n]>(unsigned long)R[m]) T=1;
    else T=0;
    PC+=2;
}

CMPHS(long m,long n)     /* CMP_HS Rm,Rn */
{
    if ((unsigned long)R[n]>=(unsigned long)R[m]) T=1;
    else T=0;
    PC+=2;
}

CMPPL(long n)            /* CMP_PL Rn */
{
    if ((long)R[n]>0) T=1;
    else T=0;
    PC+=2;
}

CMPPZ(long n) /* CMP_PZ Rn */
{
    if ((long)R[n]>=0) T=1;
    else T=0;
    PC+=2;
}

CMPSTR(long m,long n) /* CMP_STR Rm,Rn */
{
    unsigned long temp;
    long HH,HL,LH,LL;

    temp=R[n]^R[m];
    HH=(temp&0xFF000000)>>12;
    HL=(temp&0x00FF0000)>>8;
    LH=(temp&0x0000FF00)>>4; LL=temp&0x000000FF;
    HH=HH&&HL&&LH&&LL;
    if (HH==0) T=1;
    else T=0;
```

```
    PC+=2;
}

CMPIM(long i)             /* CMP_EQ #imm,R0 */
{
    long imm;

    if ((i&0x80)==0) imm=(0x000000FF & (long i));
    else imm=(0xFFFFFF00 | (long i));
    if (R[0]==imm) T=1;
    else T=0;
    PC+=2;
}
```

Examples:

CMP/GE	R0,R1	;R0 = H'7FFFFFFF, R1 = H'80000000
BT	TRGET_T	;Does not branch because T = 0
CMP/HS	R0,R1	;R0 = H'7FFFFFFF, R1 = H'80000000
BT	TRGET_T	;Branches because T = 1
CMP/STR	R2,R3	;R2 = “ABCD”, R3 = “XYCZ”
BT	TRGET_T	;Branches because T = 1

8.2.17 DIV0S (Divide Step 0 as Signed): Arithmetic Instruction

Format	Abstract	Code	Cycle	T Bit
DIV0S Rm,Rn	MSB of Rn → Q, MSB of Rm → M, M^Q → T	0010nnnnmmmm0111	1	Calculation result

Description: DIV0S is an initialization instruction for signed division. It finds the quotient by repeatedly dividing in combination with the DIV1 or another instruction that divides for each bit after this instruction. See the description given with DIV1 for more information.

Operation:

```
DIV0S(long m,long n)     /* DIV0S Rm,Rn */
{
    if ((R[n]&0x80000000)==0) Q=0;
    else Q=1;
    if ((R[m]&0x80000000)==0) M=0;
    else M=1;
    T=!(M==Q);
    PC+=2;
}
```

Examples: See DIV1.

8.2.18 DIV0U (Divide Step 0 as Unsigned): Arithmetic Instruction

Format	Abstract	Code	Cycle	T Bit
DIV0U	0 → M/Q/T	0000000000011001	1	0

Description: DIV0U is an initialization instruction for unsigned division. It finds the quotient by repeatedly dividing in combination with the DIV1 or another instruction that divides for each bit after this instruction. See the description given with DIV1 for more information.

Operation:

```
DIV0U()/* DIV0U */
{
    M=Q=T=0;
    PC+=2;
}
```

Example: See DIV1.

8.2.19 DIV1 (Divide Step 1): Arithmetic Instruction

Format		Abstract	Code	Cycle	T Bit
DIV1	Rm,Rn	1 step division (Rn ÷ Rm)	0011nnnnmmmm0100	1	Calculation result

Description: Uses single-step division to divide one bit of the 32-bit data in general register Rn (dividend) by Rm data (divisor). It finds a quotient through repetition either independently or used in combination with other instructions. During this repetition, do not rewrite the specified register or the M, Q, and T bits.

In one-step division, the dividend is shifted one bit left, the divisor is subtracted and the quotient bit reflected in the Q bit according to the status (positive or negative). To find the remainder in a division, first find the quotient using a DIV1 instruction, then find the remainder as follows:

$$(\text{remainder}) = (\text{dividend}) - (\text{divisor}) \times (\text{quotient})$$

Zero division, overflow detection, and remainder operation are not supported. Check for zero division and overflow division before dividing.

Find the remainder by first finding the sum of the divisor and the quotient obtained and then subtracting it from the dividend. That is, first initialize with DIV0S or DIV0U. Repeat DIV1 for each bit of the divisor to obtain the quotient. When the quotient requires 17 or more bits, place ROTCL before DIV1. For the division sequence, see the following examples.

Operation:

```
DIV1(long m,long n)      /* DIV1 Rm,Rn */
{
    unsigned long tmp0;
    unsigned char  old_q,tmp1;

    old_q=Q;
    Q=(unsigned char)((0x80000000 & R[n])!=0);
    R[n]<<=1;
    R[n]|=(unsigned long)T;
        switch(old_q){
        case 0:switch(M){
            case 0:tmp0=R[n];
                R[n]-=R[m];
                tmp1=(R[n]>tmp0);
                switch(Q){
                case 0:Q=tmp1;
                    break;
                case 1:Q=(unsigned char)(tmp1==0);
                    break;
                }
                break;
            case 1:tmp0=R[n];
                R[n]+=R[m];
                tmp1=(R[n]<tmp0);
                switch(Q){
                case 0:Q=(unsigned char)(tmp1==0);
                    break;
                case 1:Q=tmp1;
                    break;
            }
            break;
        }
        break;
```

```
    case 1:switch(M){
        case 0:tmp0=R[n];
            R[n]+=R[m];
            tmp1=(R[n]<tmp0);
            switch(Q){
            case 0:Q=tmp1;
                break;
            case 1:Q=(unsigned char)(tmp1==0);
                break;
            }
            break;
        case 1:tmp0=R[n];
            R[n]-=R[m];
            tmp1=(R[n]>tmp0);
            switch(Q){
            case 0:Q=(unsigned char)(tmp1==0);
                break;
        case 1:Q=tmp1;
                break;
            }
            break;
        }
        break;
    }
    T=(Q==M);
    PC+=2;
}
```

Example 1:

```
                                ;R1 (32 bits) / R0 (16 bits) = R1 (16 bits):Unsigned
    SHLL16      R0              ;Upper 16 bits = divisor, lower 16 bits = 0
    TST         R0,R0           ;Zero division check
    BT          ZERO_DIV
    CMP/HS      R0,R1           ;Overflow check
    BT          OVER_DIV
    DIV0U                       ;Flag initialization
    .arepeat    16
    DIV1        R0,R1           ;Repeat 16 times
    .aendr
    ROTCL       R1
    EXTU.W      R1,R2           ;R1 = Quotient
```

Example 2:

```
                                ;R1:R2 (64 bits)/R0 (32 bits) = R2 (32 bits): Unsigned
    TST         R0,R0           ;Zero division check
    BT          ZERO_DIV
    CMP/HS      R0,R1           ;Overflow check
    BT          OVER_DIV
    DIV0U                       ;Flag initialization
    .arepeat    32
    ROTCL       R2              ;Repeat 32 times
    DIV1        R0,R1
    .aendr
    ROTCL       R2              ;R2 = Quotient
```

Example 3:

```
                                ;R1 (16 bits)/R0 (16 bits) = R1 (16 bits): Signed
SHLL16      R0                  ;Upper 16 bits = divisor, lower 16 bits = 0
EXTS.W      R1,R1               ;Sign-extends the dividend to 32 bits
XOR         R2,R2               ;R2 = 0
MOV         R1,R3
ROTCL       R3
SUBC        R2,R1               ;Decrements if the dividend is negative
DIV0S       R0,R1               ;Flag initialization
.arepeat    16
DIV1        R0,R1               ;Repeat 16 times
.aendr
EXTS.W      R1,R1
ROTCL       R1                  ;R1 = quotient (ones complement)
ADDC        R2,R1               ;Increments and takes the twos complement if the MSB of the
                                ;quotient is 1
EXTS.W      R1,R1               ;R1 = quotient (two’s complement)
```

Example 4:

```
                                ;R2 (32 bits) / R0 (32 bits) = R2 (32 bits): Signed
MOV         R2,R3
ROTCL       R3
SUBC        R1,R1               ;Sign-extends the dividend to 64 bits (R1:R2)
XOR         R3,R3               ;R3 = 0
SUBC        R3,R2               ;Decrements and takes the ones complement if the dividend is
                                ;negative
DIV0S       R0,R1               ;Flag initialization
.arepeat    32
ROTCL       R2                  ;Repeat 32 times
DIV1        R0,R1
.aendr
ROTCL       R2                  ;R2 = Quotient (one’s complement)
ADDC        R3,R2               ;Increments and takes the two’s complement if the MSB of the
                                ;quotient is 1. R2 = Quotient (two’s complement)
```

8.2.20 DMULS.L (Double-Length Multiply as Signed): Arithmetic Instruction

Format	Abstract	Code	Cycle	T Bit
DMULS.L Rm,Rn	With sign, Rn × Rm → MACH, MACL	0011nnnnmmmm1101	2 (to 5)	—

Description: Performs 32-bit multiplication of the contents of general registers Rn and Rm, and stores the 64-bit results in the MACL and MACH register. The operation is a signed arithmetic operation.

Operation:

```
DMULS(long m,long n)  /* DMULS.L Rm,Rn */
{
    unsigned   long RnL,RnH,RmL,RmH,Res0,Res1,Res2;
    unsigned   long temp0,temp1,temp2,temp3;
    long tempm,tempn,fnLmL;

    tempn=(long)R[n];
    tempm=(long)R[m];
    if (tempn<0) tempn=0-tempn;
    if (tempm<0) tempm=0-tempm;
    if ((long)(R[n]^R[m])<0) fnLmL=-1;
    else fnLmL=0;

    temp1=(unsigned long)tempn;
    temp2=(unsigned long)tempm;

    RnL=temp1&0x0000FFFF;
    RnH=(temp1>>16)&0x0000FFFF;
    RmL=temp2&0x0000FFFF;
    RmH=(temp2>>16)&0x0000FFFF;

    temp0=RmL*RnL;
    temp1=RmH*RnL;
    temp2=RmL*RnH;
    temp3=RmH*RnH;
```

```
    Res2=0
    Res1=temp1+temp2;
    if (Res1<temp1) Res2+=0x00010000;

    temp1=(Res1<<16)&0xFFFF0000;
    Res0=temp0+temp1;
    if (Res0<temp0) Res2++;

    Res2=Res2+((Res1>>16)&0x0000FFFF)+temp3;

    if (fnLmL<0) {
        Res2=~Res2;
        if (Res0==0)
            Res2++;
        else
            Res0=(~Res0)+1;
    }
    MACH=Res2;
    MACL=Res0;
    PC+=2;
}
```

Examples:

`DMULS R0,R1`	;Before execution	R0 = H'FFFFFFFE, R1 = H'00005555
	;After execution	MACH = H'FFFFFFFF, MACL = H'FFFF5556
`STS MACH,R0`	;Operation result (top)	
`STS MACL,R0`	;Operation result (bottom)	

8.2.21 DMULU.L (Double-Length Multiply as Unsigned): Arithmetic Instruction

Format	Abstract	Code	Cycle	T Bit
DMULU.L Rm,Rn	Without sign, Rn × Rm → MACH, MACL	0011nnnnmmmm0101	2 (to 5)	—

Description: Performs 32-bit multiplication of the contents of general registers Rn and Rm, and stores the 64-bit results in the MACL and MACH register. The operation is an unsigned arithmetic operation.

Operation:

```
DMULU(long m,long n)  /* DMULU.L Rm,Rn */
{
    unsigned   long RnL,RnH,RmL,RmH,Res0,Res1,Res2;
    unsigned   long temp0,temp1,temp2,temp3;

    RnL=R[n]&0x0000FFFF;
    RnH=(R[n]>>16)&0x0000FFFF;

    RmL=R[m]&0x0000FFFF;
    RmH=(R[m]>>16)&0x0000FFFF;

    temp0=RmL*RnL;
    temp1=RmH*RnL;
    temp2=RmL*RnH;
    temp3=RmH*RnH;

    Res2=0
    Res1=temp1+temp2;
    if (Res1<temp1) Res2+=0x00010000;

    temp1=(Res1<<16)&0xFFFF0000;
    Res0=temp0+temp1;
    if (Res0<temp0) Res2++;

    Res2=Res2+((Res1>>16)&0x0000FFFF)+temp3;

    MACH=Res2;
```

```
    MACL=Res0;
    PC+=2;
}
```

Examples:

```
DMULU  R0,R1     ;Before execution   R0 = H'FFFFFFFE, R1 = H'00005555
                 ;After execution    MACH = H'FFFFFFFF, MACL = H'FFFF5556
STS    MACH,R0   ;Operation result (top)
STS    MACL,R0   ;Operation result (bottom)
```

8.2.22 DT (Decrement and Test): Arithmetic Instruction

Format		Abstract	Code	Cycle	T Bit
DT	Rn	Rn - 1 → Rn; When Rn is 0, 1 → T, when Rn is nonzero, 0 → T	0100nnnn00010000	1	Comparison result

Description: Decrements the contents of general register Rn by 1 and compares the results to 0 (zero). When the result is 0, the T bit is set to 1. When the result is not zero, the T bit is set to 0.

Operation:

```
DT(long n) /* DT Rn */
{
    R[n]--;
    if (R[n]==0) T=1;
    else T=0;
    PC+=2;
}
```

Example:

```
        MOV     #4,R5       ;Sets the number of loops.
LOOP:
        ADD     R0,R1
        DT      RS          ;Decrements the R5 value and checks whether it has become 0.
        BF      LOOP        ;Branches to LOOP is T=0. (In this example, loops 4 times.)
```

8.2.23 EXTS (Extend as Signed): Arithmetic Instruction

Format		Abstract	Code	Cycle	T Bit
EXTS.B	Rm,Rn	Sign-extend Rm from byte → Rn	0110nnnnmmmm1110	1	—
EXTS.W	Rm,Rn	Sign-extend Rm from word → Rn	0110nnnnmmmm1111	1	—

Description: Sign-extends general register Rm data, and stores the result in Rn. If byte length is specified, the bit 7 value of Rm is copied into bits 8 to 31 of Rn. If word length is specified, the bit 15 value of Rm is copied into bits 16 to 31 of Rn.

Operation:

```
EXTSB(long m,long n)      /* EXTS.B Rm,Rn */
{
    R[n]=R[m];
    if ((R[m]&0x00000080)==0) R[n]&=0x000000FF;
    else R[n]|=0xFFFFFF00;
    PC+=2;
}

EXTSW(long m,long n)      /* EXTS.W Rm,Rn */
{
    R[n]=R[m];
    if ((R[m]&0x00008000)==0) R[n]&=0x0000FFFF;
    else R[n]|=0xFFFF0000;
    PC+=2;
}
```

Examples:

```
EXTS.B  R0,R1     ;Before execution  R0 = H'00000080
                  ;After execution   R1 = H'FFFFFF80
EXTS.W  R0,R1     ;Before execution  R0 = H'00008000
                  ;After execution   R1 = H'FFFF8000
```

8.2.24 EXTU (Extend as Unsigned): Arithmetic Instruction

Format		Abstract	Code	Cycle	T Bit
EXTU.B	Rm,Rn	Zero-extend Rm from byte → Rn	0110nnnnmmmm1100	1	—
EXTU.W	Rm,Rn	Zero-extend Rm from word → Rn	0110nnnnmmmm1101	1	—

Description: Zero-extends general register Rm data, and stores the result in Rn. If byte length is specified, 0s are written in bits 8 to 31 of Rn. If word length is specified, 0s are written in bits 16 to 31 of Rn.

Operation:

```
EXTUB(long m,long n)  /* EXTU.B Rm,Rn */
{
    R[n]=R[m];
    R[n]&=0x000000FF;
    PC+=2;
}

EXTUW(long m,long n)  /* EXTU.W Rm,Rn */
{
    R[n]=R[m];
    R[n]&=0x0000FFFF;
    PC+=2;
}
```

Examples:

```
EXTU.B   R0,R1     ;Before execution    R0 = H'FFFFFF80
                   ;After execution     R1 = H'00000080
EXTU.W   R0,R1     ;Before execution    R0 = H'FFFF8000
                   ;After execution     R1 = H'00008000
```

8.2.25 JMP (Jump): Branch Instruction

Class: Delayed branch instruction

Format	Abstract	Code	Cycle	T Bit
`JMP @Rm`	Rm → PC	0100nnnn00101011	2	—

Description: Branches unconditionally after executing the instruction following this JMP instruction. The branch destination is an address specified by the 32-bit data in general register Rn.

Note: Since this is a delayed branch instruction, the instruction after JMP is executed before branching. No interrupts are accepted between this instruction and the next instruction. If the next instruction is a branch instruction, it is acknowledged as an illegal slot instruction. If this instruction is located in a delayed slot immediately following a delayed branch instruction, it is acknowledged as an illegal slot instruction.

Operation:

```
JMP(long m)    /* JMP @Rm */
{
    unsigned long temp;

    temp=PC;
    PC=R[m]+4;
    Delay_Slot(temp+2);
}
```

Examples:

```
              MOV.L    JMP_TABLE,R0    ;Address of R0 = TRGET
              JMP      @R0             ;Branches to TRGET
              MOV      R0,R1           ;Executes MOV before branching
              .align   4
JMP_TABLE:    .data.l  TRGET           ;Jump table
              .................
TRGET:        ADD      #1,R1           ;← Branch destination
```

Note: In delayed branching, the branching operation itself takes place after the slot instruction has been executed. However, execution of instructions (register updating, etc.) should always be done in the sequence of delayed branch instruction followed by delayed slot instruction. For example, even if a delayed slot updates a register in which the branching destination address is stored, the contents of the register before updating will be used as the branching destination address.

8.2.26 JSR (Jump to Subroutine): Branch Instruction

Class: Delayed branch instruction

Format		Abstract	Code	Cycle	T Bit
JSR	@Rm	PC → Rm, Rm → PC	0100nnnn00001011	2	—

Description: Branches to the subroutine procedure at a specified address after executing the instruction following this JSR instruction. The PC value is stored in the PR. The jump destination is an address specified by the 32-bit data in general register Rn. The PC points to the starting address of the second instruction after JSR. The JSR instruction and RTS instruction are used for subroutine procedure calls.

Note: Since this is a delayed branch instruction, the instruction after JSR is executed before branching. No interrupts are accepted between this instruction and the next instruction. If the next instruction is a branch instruction, it is acknowledged as an illegal slot instruction. If this instruction is located in a delayed slot immediately following a delayed branch instruction, it is acknowledged as an illegal slot instruction.

The PR used by the instruction immediately following this instruction is updated by this instruction.

Also, if the instruction immediately following this instruction generates a re-execution exception other than instruction fetch, the PR is updated by this instruction. Re-execute this instruction to recover.

Operation:

```
JSR(long m)    /* JSR @Rm */
{
    PR=PC;
    PC=R[m]+4;
    Delay_Slot(PR+2);
}
```

Examples:

```
            MOV.L    JSR_TABLE,R0    ;Address of R0 = TRGET
            JSR      @R0             ;Branches to TRGET
            XOR      R1,R1           ;Executes XOR before branching
            ADD      R0,R1           ;← Return address for when the subroutine
                                     ;procedure is completed (PR data)
            ...........
            .align   4
JSR_TABLE:  .data.l  TRGET           ;Jump table
TRGET:      NOP                      ;← Procedure entrance
            MOV      R2,R3
            RTS                      ;Returns to the above ADD instruction
            MOV      #70,R1          ;Executes MOV before RTS
```

Note: In delayed branching, the branching operation itself takes place after the slot instruction has been executed. However, execution of instructions (register updating, etc.) should always be done in the sequence of delayed branch instruction followed by delayed slot instruction. For example, even if a delayed slot updates a register in which the branching destination address is stored, the contents of the register before updating will be used as the branching destination address.

8.2.27 LDC (Load to Control Register): System Control Instruction (Privileged Only)

Format		Abstract	Code	Cycle	T Bit
LDC	Rm,SR	Rm → SR	0100mmmm00001110	5	LSB
LDC	Rm,GBR	Rm → GBR	0100mmmm00011110	1	—
LDC	Rm,VBR	Rm → VBR	0100mmmm00101110	1	—
LDC	Rm,SSR	Rm → SSR	0100mmmm00111110	1	—
LDC	Rm,SPC	Rm → SPC	0100mmmm01001110	1	—
LDC	Rm,MOD*	Rm → MOD	0100mmmm01011110	3	—
LDC	Rm,RE*	Rm → RE	0100mmmm01111110	3	—
LDC	Rm,RS*	Rm → RS	0100mmmm01101110	3	—
LDC	Rm,R0_BANK	Rm → R0_BANK	0100mmmm10001110	1	—
LDC	Rm,R1_BANK	Rm → R1_BANK	0100mmmm10011110	1	—
LDC	Rm,R2_BANK	Rm → R2_BANK	0100mmmm10101110	1	—
LDC	Rm,R3_BANK	Rm → R3_BANK	0100mmmm10111110	1	—
LDC	Rm,R4_BANK	Rm → R4_BANK	0100mmmm11001110	1	—
LDC	Rm,R5_BANK	Rm → R5_BANK	0100mmmm11011110	1	—
LDC	Rm,R6_BANK	Rm → R6_BANK	0100mmmm11101110	1	—
LDC	Rm,R7_BANK	Rm → R7_BANK	0100mmmm11111110	1	—
LDC.L	@Rm+,SR	(Rm) → SR, Rm + 4 → Rm	0100mmmm00000111	7	LSB
LDC.L	@Rm+,GBR	(Rm) → GBR, Rm + 4 → Rm	0100mmmm00010111	1	—
LDC.L	@Rm+,VBR	(Rm) → VBR, Rm + 4 → Rm	0100mmmm00100111	1	—
LDC.L	@Rm+,SSR	(Rm) → SSR, Rm + 4 → Rm	0100mmmm00110111	1	—
LDC.L	@Rm+,SPC	(Rm) → SPC, Rm + 4 → Rm	0100mmmm01000111	1	—
LDC.L	@Rm+,MOD*	(Rm) → MOD, Rm + 4 → Rm	0100mmmm01010111	5	—
LDC.L	@Rm+,RE*	(Rm) → RE, Rm + 4 → Rm	0100mmmm01110111	5	—
LDC.L	@Rm+,RS*	(Rm) → RS, Rm + 4 → Rm	0100mmmm01100111	5	—
LDC.L	@Rm+,R0_BANK	(Rm) → R0_BANK, Rm + 4 → Rm	0100mmmm10000111	1	—
LDC.L	@Rm+,R1_BANK	(Rm) → R1_BANK, Rm + 4 → Rm	0100mmmm10010111	1	—
LDC.L	@Rm+,R2_BANK	(Rm) → R2_BANK, Rm + 4 → Rm	0100mmmm10100111	1	—
LDC.L	@Rm+,R3_BANK	(Rm) → R3_BANK, Rm + 4 → Rm	0100mmmm10110111	1	—

Note: * SH3-DSP only.

Format	Abstract	Code	Cycle	T Bit
LDC.L @Rm+,R4_BANK	(Rm) → R4_BANK, Rm + 4 → Rm	0100mmmm11000111	1	—
LDC.L @Rm+,R5_BANK	(Rm) → R5_BANK, Rm + 4 → Rm	0100mmmm11010111	1	—
LDC.L @Rm+,R6_BANK	(Rm) → R6_BANK, Rm + 4 → Rm	0100mmmm11100111	1	—
LDC.L @Rm+,R7_BANK	(Rm) → R7_BANK, Rm + 4 → Rm	0100mmmm11110111	1	—

Notes: 1. Three cycles on the SH3-DSP.
2. Five cycles on the SH3-DSP.

Description: Stores source operand in control registers SR, GBR, VBR, SSR, SPC, MOD, RE, and RS, or R0_BANK to R7_BANK. LDC and LDC.L, except for LDC Rm, GBR and LDC.L @RM+, GBR, are privileged instructions and can be used in privileged mode only. If used in user mode, they can cause illegal instruction exceptions. Note that LDC Rm, GBR and LDC.L @RM+, GBR can be used in user mode.

The Rm_BANK operand is designated by the RB bit of the SR register. When the value of the RB bit is 1, the R0_BANK1 to R7_BANK1 registers and the R8 to R15 registers are used as the Rn operand, and the R0_BANK0 to R7_BANK0 registers are used as the Rm_BANK operand. When the value of the RB bit is 0, the R0_BANK0 to R7_BANK0 registers and the R8 to R15 registers are used as the Rn operand, and the R0_BANK1 to R7_BANK1 registers are used as the Rm_BANK operand.

If the LDC Rm, SR instruction or LDC.L @RM+, SR instruction is located in a delayed slot immediately following a delayed branch instruction, it is acknowledged as an illegal slot instruction.

Operation:

```
LDCSR(long m)     /* LDC Rm,SR */
{
    SR=R[m]&0x0FFF0FFF;
    PC+=2;
}

LDCGBR(long m) /* LDC Rm,GBR */
{
    GBR=R[m];
    PC+=2;
}
```

```
LDCVBR(long m) /* LDC Rm,VBR */
{
    VBR=R[m];
    PC+=2;
}

LDCSSR(long m)     /* LDC Rm,SSR */
{
    SSR=R[m]&0x700003F3;
    PC+=2;
}

LDCSPC(long m)     /* LDC Rm,SPC */
{
    SPC=R[m];
    PC+=2;
}

LDCRn_BANK(long m)     /* LDC Rm,Rn_BANK */
{                      /* n=0–7, */
    Rn_BANK=R[m];
    PC+=2;
}


LDCMSR(long m) /* LDC.L @Rm+,SR */
{
    SR=Read_Long(R[m])&0x0FFF0FFF;
    R[m]+=4;
    PC+=2;
}

LDCMGBR(long m)    /* LDC.L @Rm+,GBR */
{
    GBR=Read_Long(R[m]);
    R[m]+=4;
    PC+=2;
}
```

```
LDCMVBR(long m)   /* LDC.L @Rm+,VBR */
{
    VBR=Read_Long(R[m]);
    R[m]+=4;
    PC+=2;
}

LDCMSSR(long m)   /* LDC.L @Rm+,SSR */
{
    SSR=Read_Long(R[m])&0x700003F3;
    R[m]+=4;
    PC+=2;
}

LDCMSPC(long m)   /* LDC.L @Rm+,SPC */
{
    SPC=Read_Long(R[m]);
    R[m]+=4;
    PC+=2;
}

LDCMRn_BANK(long m)   /* LDC.L @Rm+,Rn_BANK */
                      /* n=0-7 */
{
    Rn_BANK=Read_Long(R[m]);
    R[m]+=4;
    PC+=2;
}

LDCMOD(long m)    /* LDC Rm,MOD */
{
    MOD=R[m];
    PC+=2;
}

LDCRE(long m)  /* LDC Rm,RE */
{
    RE=R[m];
    PC+=2;
}
```

```
LDCRS(long m)  /* LDC Rm,RS */
{
    RS=R[m];
    PC+=2;
}

LDCMMOD(long m)    /* LDC.L @Rm+,MOD */
{
    MOD=Read_Long(R[m]);
    R[m]+=4;
    PC+=2;
}

LDCMRE(long m) /* LDC.L @Rm+,RE */
{
    RE=Read_Long(R[m]);
    R[m]+=4;
    PC+=2;
}

LDCMRS(long m) /* LDC.L @Rm+,RS */
{
    RS=Read_Long(R[m]);
    R[m]+=4;
    PC+=2;
}
```

Examples:

`LDC`	`R0,SR`	;Before execution	R0 = H'FFFFFFFF, SR = H'00000000
		;After execution	SR = H'700003F3
`LDC.L`	`@R15+,GBR`	;Before execution	R15 = H'10000000, @R15 + H'12345678, GBR = H'EDCBA987
		;After execution	R15 = H'10000004, GBR = @H'10000000

8.2.28 LDRE (Load Effective Address to RE Register): System Control Instruction (SH3-DSP Only)

Format	Abstract	Code	Cycle	T Bit
LDRE @(disp,PC)	disp × 2 + PC → RE	10001110dddddddd	3	—

Description: Stores the effective address of the source operand in the repeat end register RE. The effective address is an address specified by PC + displacement. The PC is the address four bytes after this instruction. The 8-bit displacement is sign-extended and doubled. Consequently, the relative interval from the branch destination is –256 to +254 bytes.

Note: The effective address value designated for the RE reregister is different from the actual repeat end address. Refer to table 8.23, RS and RE Design Rule, for more information. When this instruction is arranged immediately after the delayed branch instruction, PC becomes the "first address +2" of the branch destination.

Operation:

```
LDRE(long d)  /* LDRE @(disp, PC) */
{
    long disp;

    if ((d&0x80)==0) disp=(0x000000FF & (long)d);
    else disp=(0xFFFFFF00 | (long)d);
    RE=PC+(disp<<1);
    PC+=2;
}
```

Example:

```
         LDRS STA         ;Set repeat start address to RS.
         LDRE END         ;Set repeat end address to RE.
         SETRC #32        ;Repeat 32 times from inst.A to inst.C.
         inst.0           ;
STA:          inst.A      ;
              inst.B      ;
         ............
END:          inst.C      ;
         inst.E           ;
         ............
```

8.2.29 LDRS (Load Effective Address to RS Register): System Control Instruction (SH3-DSP Only)

Format	Abstract	Code	Cycle	T Bit
LDRS @(disp,PC)	disp × 2 + PC → RS	10001100dddddddd	3	—

Description: Stores the effective address of the source operand in the repeat start register RS. The effective address is an address specified by PC + displacement. The PC is the address four bytes after this instruction. The 8-bit displacement is sign-extended and doubled. Consequently, the relative interval from the branch destination is –256 to +254 bytes.

Note: When the instructions of the repeat (loop) program are below 3, the effective address value designated for the RS register is different from the actual repeat start address. Refer to Table 8-23. "RS and RE setting rule", for more information. If this instruction is arranged immediately after the delayed branch instruction, the PC becomes "the first address +2" of the branch destination.

Operation:

```
LDRS(long d)   /* LDRS @(disp, PC) */
{
    long disp;

    if ((d&0x80)==0) disp=(0x000000FF & (long)d);
    else disp=(0xFFFFFF00 | (long)d);
    RS=PC+(disp<<1);
    PC+=2;
}
```

Example:

```
         LDRS STA          ;Set repeat start address to RS.
         LDRE END          ;Set repeat end address to RE.
         SETRC #32         ;Repeat 32 times from inst.A to inst.C.
         inst.0            ;
STA:          inst.A       ;
              inst.B       ;
         ............
END:          inst.C       ;
         inst.D            ;
         ............
```

8.2.30 LDS (Load to System Register): System Control Instruction

Format	Abstract	Code	Cycle	T Bit
LDS Rm,MACH	Rm → MACH	0100mmmm00001010	1	—
LDS Rm,MACL	Rm → MACL	0100mmmm00011010	1	—
LDS Rm,PR	Rm → PR	0100mmmm00101010	1	—
LDS Rm,DSR*	Rm → DSR	0100mmmm01101010	1	—
LDS Rm,A0*	Rm → A0	0100mmmm01111010	1	—
LDS Rm,X0*	Rm → X0	0100mmmm10001010	1	—
LDS Rm,X1*	Rm → X1	0100mmmm10011010	1	—
LDS Rm,Y0*	Rm → Y0	0100mmmm10101010	1	—
LDS Rm,Y1*	Rm → Y1	0100mmmm10111010	1	—
LDS.L @Rm+,MACH	(Rm) → MACH, Rm + 4 → Rm	0100mmmm00000110	1	—
LDS.L @Rm+,MACL	(Rm) → MACL, Rm + 4 → Rm	0100mmmm00010110	1	—
LDS.L @Rm+,PR	(Rm) → PR, Rm + 4 → Rm	0100mmmm00100110	1	—
LDS.L @Rm+,DSR*	(Rm) → DSR, Rm + 4 → Rm	0100mmmm01100110	1	—
LDS.L @Rm+,A0*	(Rm) → A0, Rm + 4 → Rm	0100mmmm01110110	1	—
LDS.L@Rm+,X0*	(Rm) → X0,Rm+4 → Rm	0100nnnn10000110	1	—
LDS.L@Rm+,X1*	(Rm) → X1,Rm+4 → Rm	0100nnnn10010110	1	—
LDS.L@Rm+,Y0*	(Rm) → Y0,Rm+4 → Rm	0100nnnn10100110	1	—
LDS.L@Rm+,Y1*	(Rm) → Y1,Rm+4 → Rm	0100nnnn10110110	1	—

Note: * SH3-DSP only.

Description: Stores the source operand into the system registers MACH, MACL, PR, DSR, A0, X0, X1, Y0, or Y1.

Operation:

```
LDSMACH(long m)          /* LDS Rm,MACH */
{
    MACH=R[m];
    if ((MACH&0x00000200)==0) MACH&=0x000003FF;
    else MACH|=0xFFFFFC00;
    PC+=2;
}
```

```
LDSMACL(long m)          /* LDS Rm,MACL */
{
    MACL=R[m];
    PC+=2;
}

LDSPR(long m)            /* LDS Rm,PR */
{
    PR=R[m];
    PC+=2;
}

LDSMMACH(long m)         /* LDS.L @Rm+,MACH */
{
    MACH=Read_Long(R[m]);
    if ((MACH&0x00000200)==0) MACH&=0x000003FF;
    else MACH|=0xFFFFFC00;
    R[m]+=4;
    PC+=2;
}
LDSMMACL(long m)         /* LDS.L @Rm+,MACL */
{
    MACL=Read_Long(R[m]);
    R[m]+=4;
    PC+=2;
}

LDSMPR(long m) /* LDS.L @Rm+,PR */
{
    PR=Read_Long(R[m]);
    R[m]+=4;
    PC+=2;
}
LDSDSR(long m)       /* LDS Rm,DSR */
{
    DSR=R[m]&0x0000000F;
    PC+=2;
}
```

```
LDSA0(long m)            /* LDS Rm,A0 */
{
   A0=R[m];
   if((A0&0x80000000)==0) A0G=0x00;
   else A0G=0xFF;
   PC+=2;
}
LDSX0(long m)            /* LDS Rm, X0 */
{
   X0=R[m];
   PC+=2;
}
LDSX1(long m)            /* LDS Rm, X1 */
{
   X1=R[m];
   PC+=2;
}
LDSY0(long m)            /* LDS Rm, Y0 */
{
   Y0=R[m];
   PC+=2;
}
LDSY1(long m)            /* LDS Rm, Y1 */
{
   Y1=R[m];
   PC+=2;
}
LDSMDSR(long m)          /* LDS.L @Rm+,DSR */
{
   DSR=Read_Long(R[m])&0x0000000F;
   R[m]+=4;
   PC+=2;
}
LDSMA0(long m) /* LDS.L @Rm+,A0 */
{
   A0=Read_Long(R[m]);
   if((A0&0x80000000)==0) A0G=0x00;
   else A0G=0xFF;
```

```
    R[m]+=4;
    PC+=2;
}
LDSMX0(long m)              /* LDS.L @Rm+,X0 */
{
    X0=Read_Long(R[m]);
    R[m]+=4;
    PC+=2;
}
LDSMX1(long m)              /* LDS.L @Rm+,X1 */
{
    X1=Read_Long(R[m]);
    R[m]+=4;
    PC+=2;
}
LDSMY0(long m)              /* LDS.L @Rm+,Y0 */
{
    Y0=Read_Long(R[m]);
    R[m]+=4;
    PC+=2;
}
LDSMY1(long m)              /* LDS.L @Rm+,Y1 */
{
    Y1=Read_Long(R[m]);
    R[m]+=4;
    PC+=2;
}
```

Examples:

LDS	R0,PR	;Before execution	R0 = H'12345678, PR = H'00000000
		;After execution	PR = H'12345678
LDS.L	@R15+,MACL	;Before execution	R15 = H'10000000
		;After execution	R15 = H'10000004, MACL = @H'10000000

8.2.31 LDTLB (Load PTEH/PTEL to TLB): System Control Instruction (Privileged Only)

Format	Abstract	Code	Cycle	T Bit
LDTLB	PTEH/PTEL → TLB	0000000000111000	1	—

Description: Loads PTEH/PTEL registers to the translation lookaside buffer (TLB). The TLB is indexed by the virtual address held in the PTEH register. The loaded set is designated by the MMUCR.RC (MMUCR is an MMU control register and RC is a two bit field for a counter). LDTLB is a privileged instruction and can be used in privileged mode only. If used in user mode, it causes an illegal instruction exception.

Note: As LDTLB is for loading PTEH and PTEL to the TLB, the instruction should be issued when MMU is off (MMUCR.AT = 0) or should be placed in the P1 or P2 space with MMU enabled (see the MMU section of the applicable hardware manual for details). If the instruction is issued in an exception handler, it should be at least two instructions prior to an RTE instruction that terminates the handler.

Operation:

```
LDTLB()/*LDTLB*/
{
    TLB_tag=PTEH;
    TLB_data=PTEL;
    PC+=2;
}
```

Examples:

```
MOV L @R0, R1      ;Load upper bits of page table entry to R1
MOV L R1, @R2      ;Load R1 to PTEH, R2 is PTEH address (H'FFFFFFF0)
MOV L @R3, R4      ;Load lower bits of page table entry to R4
MOV L R4, @R5      ;Load R4 to PTEL, R5 is PTEL address (H'FFFFFFF4)
LDTLB              ;Load PTEH and PTEL registers to TLB
```

8.2.32 MAC.L (Multiply and Accumulate Long): Arithmetic Instruction

Format	Abstract	Code	Cycle	T Bit
`MAC.L @Rm+,@Rn+`	Signed operation, (Rn) × (Rm) + MAC → MAC Rn + 4 → Rn, Rm + 4 → Rm	`0000nnnnmmmm1111`	2 (to 5)	—

Description: Does signed multiplication of 32-bit operands obtained using the contents of general registers Rm and Rn as addresses. The 64-bit result is added to contents of the MAC register, and the final result is stored in the MAC register. Every time an operand is read, RM and Rn are incremented by four.

When the S bit is cleared to 0, the 64-bit result is stored in the coupled MACH and MACL registers. When bit S is set to 1, addition to the MAC register is a saturation operation of 48 bits starting from the LSB. For the saturation operation, only the lower 48 bits of the MACL register are enabled and the result is limited to between H'FFFF800000000000 (minimum) and H'00007FFFFFFFFFFF (maximum).

Operation:

```
MACL(long m,long n)   /* MAC.L @Rm+,@Rn+*/
{
    unsigned long RnL,RnH,RmL,RmH,Res0,Res1,Res2;
    unsigned long temp0,templ,temp2,temp3;
    long tempm,tempn,fnLmL;

    tempn=(long)Read_Long(R[n]);
    R[n]+=4;
    tempm=(long)Read_Long(R[m]);
    R[m]+=4;

    if ((long)(tempn^tempm)<0) fnLmL=-1;
    else fnLmL=0;
    if (tempn<0) tempn=0-tempn;
    if (tempm<0) tempm=0-tempm;

    temp1=(unsigned long)tempn;
    temp2=(unsigned long)tempm;
```

```
RnL=temp1&0x0000FFFF;
RnH=(temp1>>16)&0x0000FFFF;
RmL=temp2&0x0000FFFF;
RmH=(temp2>>16)&0x0000FFFF;

temp0=RmL*RnL;
temp1=RmH*RnL;
temp2=RmL*RnH;
temp3=RmH*RnH;

Res2=0
Res1=temp1+temp2;
if (Res1<temp1) Res2+=0x00010000;

temp1=(Res1<<16)&0xFFFF0000;
Res0=temp0+temp1;
if (Res0<temp0) Res2++;

Res2=Res2+((Res1>>16)&0x0000FFFF)+temp3;

if(fnLm<0){
    Res2=~Res2;
    if (Res0==0) Res2++;
    else Res0=(~Res0)+1;
}
if(S==1){
    Res0=MACL+Res0;
    if (MACL>Res0) Res2++;
    Res2+=(MACH&0x0000FFFF);

    if(((long)Res2<0)&&(Res2<0xFFFF8000)){
        Res2=0x00008000;
        Res0=0x00000000;
    }
```

```
        if(((long)Res2>0)&&(Res2>0x00007FFF)){
            Res2=0x00007FFF;
            Res0=0xFFFFFFFF;
        };

        MACH={Res2;
        MACL=Res0;
    }
    else {
        Res0=MACL+Res0;
        if (MACL>Res0) Res2++;
        Res2+=MACH

        MACH=Res2;
        MACL=Res0;
    }
    PC+=2;
}
```

Examples:

```
        MOVA      TBLM,R0          ;Table address
        MOV       R0,R1
        MOVA      TBLN,R0          ;Table address
        CLRMAC                     ;MAC register initialization
        MAC.L     @R0+,@R1+
        MAC.L     @R0+,@R1+
        STS       MACL,R0          ;Store result into R0
        ...............
        .align    2
TBLM    .data.l   H'1234ABCD
        .data.l   H'5678EF01
TBLN    .data.l   H'0123ABCD
        .data.l   H'4567DEF0
```

8.2.33 MAC (Multiply and Accumulate): Arithmetic Instruction

Format		Abstract	Code	Cycle	T Bit
MAC.W	@Rm+,@Rn+	With sign, (Rn) × (Rm) + MAC → MAC	0100nnnnmmmm1111	2 (to 5)	—
MAC	@Rm+,@Rn+	Rn + 2 → Rn, Rm + 2 → Rm			

Description: Multiplies with sign 16-bit operands obtained using the contents of general registers Rm and Rn as addresses. The 32-bit result is added to the contents of the MAC register, and the final result is stored in the MAC register.

Each time an operand is read, Rm and Rn are each incremented by 2.

When the S bit is cleared to 0, the 64-bit result of the 16-bit (16-bit + 64-bit = 64-bit multiply and accumulate calculation is stored in the coupled MACH and MACL registers.

When the S bit is set to 1, the 16-bit (16-bit + 32-bit = 32-bit multiply and accumulate calculation involves addition to the MAC register using a saturation operation. For the saturation operation, only the MACL register is enabled, and the result is limited to between H'80000000 (minimum) and H'7FFFFFFF (maximum). If an overflow occurs, the LSB of the MACH register is set to 1. If the overflow is in the negative direction, H'80000000 (the minimum value) is stored in the MACL register, and if the overflow is in the positive direction, H'7FFFFFFF (the maximum value) is stored in the MACL register.

Note: The normal number of cycles for execution is 3; however, succeeding instructions can be executed in two cycles.

Operation:

```
MACW(long m,long n)   /* MAC.W @Rm+,@Rn+*/
{
    long tempm,tempn,dest,src,ans;
    unsigned long templ;
    tempn=(long)Read_Word(R[n]);
    R[n]+=2;
    tempm=(long)Read_Word(R[m]);
    R[m]+=2;
    templ=MACL;
    tempm=((long)(short)tempn*(long)(short)tempm);
    if ((long)MACL>=0) dest=0;
    else dest=1;
    if ((long)tempm>=0 {
```

```
        src=0;
        tempn=0;
    }
    else {
        src=1;
        tempn=0xFFFFFFFF;
    }
    src+=dest;
    MACL+=tempm;
    if ((long)MACL>=0) ans=0;
    else ans=1;
    ans+=dest;
    if (S==1) {
        if (ans==1) {
            if (src==0 || src==2) MACH|=0x00000001;
            if (src==0) MACL=0x7FFFFFFF;
            if (src==2) MACL=0x80000000;
        }
    }
    else {
        MACH+=tempn;
        if (templ>MACL) MACH+=1;
        if ((MACH&0x00000200)==0) MACH&=0x000003FF;
        else MACH|=0xFFFFFC00;
    }
    PC+=2;
}
```

Examples:

```
        MOVA      TBLM,R0       ;Table address
        MOV       R0,R1
        MOVA      TBLN,R0       ;Table address
        CLRMAC                  ;MAC register initialization
        MAC.W     @R0+,@R1+
        MAC.W     @R0+,@R1+
        STS       MACL,R0       ;Store result into R0
        ...............
        .align    2
TBLM    .data.w   H'1234
        .data.w   H'5678
TBLN    .data.w   H'0123
        .data.w   H'4567
```

8.2.34 MOV (Move Data): Data Transfer Instruction

Format	Abstract	Code	Cycle	T Bit
MOV Rm,Rn	Rm → Rn	0110nnnnmmmm0011	1	—
MOV.B Rm,@Rn	Rm → (Rn)	0010nnnnmmmm0000	1	—
MOV.W Rm,@Rn	Rm → (Rn)	0010nnnnmmmm0001	1	—
MOV.L Rm,@Rn	Rm → (Rn)	0010nnnnmmmm0010	1	—
MOV.B @Rm,Rn	(Rm) → sign extension → Rn	0110nnnnmmmm0000	1	—
MOV.W @Rm,Rn	(Rm) → sign extension → Rn	0110nnnnmmmm0001	1	—
MOV.L @Rm,Rn	(Rm) → Rn	0110nnnnmmmm0010	1	—
MOV.B Rm,@–Rn	Rn – 1 → Rn, Rm → (Rn)	0010nnnnmmmm0100	1	—
MOV.W Rm,@–Rn	Rn – 2 → Rn, Rm → (Rn)	0010nnnnmmmm0101	1	—
MOV.L Rm,@–Rn	Rn – 4 → Rn, Rm → (Rn)	0010nnnnmmmm0110	1	—
MOV.B @Rm+,Rn	(Rm) → sign extension → Rn, Rm + 1 → Rm	0110nnnnmmmm0100	1	—
MOV.W @Rm+,Rn	(Rm) → sign extension → Rn, Rm + 2 → Rm	0110nnnnmmmm0101	1	—
MOV.L @Rm+,Rn	(Rm) → Rn, Rm + 4 → Rm	0110nnnnmmmm0110	1	—
MOV.B Rm,@(R0,Rn)	Rm → (R0 + Rn)	0000nnnnmmmm0100	1	—
MOV.W Rm,@(R0,Rn)	Rm → (R0 + Rn)	0000nnnnmmmm0101	1	—
MOV.L Rm,@(R0,Rn)	Rm → (R0 + Rn)	0000nnnnmmmm0110	1	—
MOV.B @(R0,Rm),Rn	(R0 + Rm) → sign extension → Rn	0000nnnnmmmm1100	1	—
MOV.W @(R0,Rm),Rn	(R0 + Rm) → sign extension → Rn	0000nnnnmmmm1101	1	—
MOV.L @(R0,Rm),Rn	(R0 + Rm) → Rn	0000nnnnmmmm1110	1	—

Description: Transfers the source operand to the destination. When the operand is stored in memory, the transferred data can be a byte, word, or longword. Loaded data from memory is stored in a register after it is sign-extended to a longword.

Operation:

```
MOV(long m,long n)        /* MOV Rm,Rn */
{
    R[n]=R[m];
    PC+=2;
}
```

```
MOVBS(long m,long n)     /* MOV.B Rm,@Rn */
{
    Write_Byte(R[n],R[m]);
    PC+=2;
}

MOVWS(long m,long n)     /* MOV.W Rm,@Rn */
{
    Write_Word(R[n],R[m]);
    PC+=2;
}

MOVLS(long m,long n)     /* MOV.L Rm,@Rn */
{
    Write_Long(R[n],R[m]);
    PC+=2;
}

MOVBL(long m,long n)     /* MOV.B @Rm,Rn */
{
    R[n]=(long)Read_Byte(R[m]);
    if ((R[n]&0x80)==0) R[n]&0x000000FF;
    else R[n]|=0xFFFFFF00;
    PC+=2;
}

MOVWL(long m,long n)     /* MOV.W @Rm,Rn */
{
    R[n]=(long)Read_Word(R[m]);
    if ((R[n]&0x8000)==0) R[n]&0x0000FFFF;
    else R[n]|=0xFFFF0000;
    PC+=2;
}

MOVLL(long m,long n)     /* MOV.L @Rm,Rn */
{
    R[n]=Read_Long(R[m]);
    PC+=2;
}
```

```
MOVBM(long m,long n)     /* MOV.B Rm,@–Rn */
{
   Write_Byte(R[n]–1,R[m]);
   R[n]–=1;
   PC+=2;
}

MOVWM(long m,long n)     /* MOV.W Rm,@–Rn */
{
   Write_Word(R[n]–2,R[m]);
   R[n]–=2;
   PC+=2;
}

MOVLM(long m,long n)     /* MOV.L Rm,@–Rn */
{
   Write_Long(R[n]–4,R[m]);
   R[n]–=4;
   PC+=2;
}

MOVBP(long m,long n)  /* MOV.B @Rm+,Rn */
{
   R[n]=(long)Read_Byte(R[m]);
   if ((R[n]&0x80)==0) R[n]&0x000000FF;
   else R[n]|=0xFFFFFF00;
   if (n!=m) R[m]+=1;
   PC+=2;
}

MOVWP(long m,long n)     /* MOV.W @Rm+,Rn */
{
   R[n]=(long)Read_Word(R[m]);
   if ((R[n]&0x8000)==0) R[n]&0x0000FFFF;
   else R[n]|=0xFFFF0000;
   if (n!=m) R[m]+=2;
   PC+=2;
}
```

```
MOVLP(long m,long n)      /* MOV.L @Rm+,Rn */
{
    R[n]=Read_Long(R[m]);
    if (n!=m) R[m]+=4;
    PC+=2;
}

MOVBS0(long m,long n)     /* MOV.B Rm,@(R0,Rn) */
{
    Write_Byte(R[n]+R[0],R[m]);
    PC+=2;
}

MOVWS0(long m,long n)     /* MOV.W Rm,@(R0,Rn) */
{
    Write_Word(R[n]+R[0],R[m]);
    PC+=2;
}

MOVLS0(long m,long n) /* MOV.L Rm,@(R0,Rn) */
{
    Write_Long(R[n]+R[0],R[m]);
    PC+=2;
}

MOVBL0(long m,long n) /* MOV.B @(R0,Rm),Rn */
{
    R[n]=(long)Read_Byte(R[m]+R[0]);
    if ((R[n]&0x80)==0) R[n]&0x000000FF;
    else R[n]|=0xFFFFFF00;
    PC+=2;
}

MOVWL0(long m,long n) /* MOV.W @(R0,Rm),Rn */
{
    R[n]=(long)Read_Word(R[m]+R[0]);
    if ((R[n]&0x8000)==0) R[n]&0x0000FFFF;
    else R[n]|=0xFFFF0000;
    PC+=2;
}
```

```
MOVLL0(long m,long n) /* MOV.L @(R0,Rm),Rn */
{
    R[n]=Read_Long(R[m]+R[0]);
    PC+=2;
}
```

Examples:

`MOV`	`R0,R1`	;Before execution	R0 = H'FFFFFFFF, R1 = H'00000000
		;After execution	R1 = H'FFFFFFFF
`MOV.W`	`R0,@R1`	;Before execution	R0 = H'FFFF7F80
		;After execution	@R1 = H'7F80
`MOV.B`	`@R0,R1`	;Before execution	@R0 = H'80, R1 = H'00000000
		;After execution	R1 = H'FFFFFF80
`MOV.W`	`R0,@–R1`	;Before execution	R0 = H'AAAAAAAA, R1 = H'FFFF7F80
		;After execution	R1 = H'FFFF7F7E, @R1 = H'AAAA
`MOV.L`	`@R0+,R1`	;Before execution	R0 = H'12345670
		;After execution	R0 = H'12345674, R1 = @H'12345670
`MOV.B`	`R1,@(R0,R2)`	;Before execution	R2 = H'00000004, R0 = H'10000000
		;After execution	R1 = @H'10000004
`MOV.W`	`@(R0,R2),R1`	;Before execution	R2 = H'00000004, R0 = H'10000000
		;After execution	R1 = @H'10000004

8.2.35 MOV (Move Immediate Data): Data Transfer Instruction

Format		Abstract	Code	Cycle	T Bit
MOV	#imm,Rn	imm → sign extension → Rn	1110nnnniiiiiiii	1	—
MOV.W	@(disp,PC),Rn	(disp × 2 + PC) → sign extension → Rn	1001nnnndddddddd	1	—
MOV.L	@(disp,PC),Rn	(disp × 4 + PC) → Rn	1101nnnndddddddd	1	—

Description: Stores immediate data, which has been sign-extended to a longword, into general register Rn.

If the data is a word or longword, table data stored in the address specified by PC + displacement is accessed. If the data is a word, the 8-bit displacement is zero-extended and doubled. Consequently, the relative interval from the table is up to PC + 510 bytes. The PC points to the starting address of the second instruction after this MOV instruction. If the data is a longword, the 8-bit displacement is zero-extended and quadrupled. Consequently, the relative interval from the table is up to PC + 1020 bytes. The PC points to the starting address of the second instruction after this MOV instruction, but the lowest two bits of the PC are corrected to B'00.

Note: The end address of the program area (module) or the second address after an unconditional branch instruction are suitable for the start address of the table. If suitable table assignment is impossible (for example, if there are no unconditional branch instructions within the area specified by PC + 510 bytes or PC + 1020 bytes), the BRA instruction must be used to jump past the table. When this MOV instruction is placed immediately after a delayed branch instruction, the PC points to an address specified by (the starting address of the branch destination) + 2.

Operation:

```
MOVI(long i,long n)      /* MOV #imm,Rn */
{
    if ((i&0x80)==0) R[n]=(0x000000FF & (long)i);
    else R[n]=(0xFFFFFF00 | (long)i);
    PC+=2;
}
```

```
MOVWI(long d,long n)      /* MOV.W @(disp,PC),Rn */
{
    long disp;

    disp=(0x000000FF & (long)d);
    R[n]=(long)Read_Word(PC+(disp<<1));
    if ((R[n]&0x8000)==0) R[n]&=0x0000FFFF;
    else R[n]|=0xFFFF0000;
    PC+=2;
}

MOVLI(long d,long n)      /* MOV.L @(disp,PC),Rn */
{
    long disp;

    disp=(0x000000FF & (long)d);
    R[n]=Read_Long((PC&0xFFFFFFFC)+(disp<<2));
    PC+=2;
}
```

Examples:

Address				
1000		MOV	#H'80,R1	;R1 = H'FFFFFF80
1002		MOV.W	IMM,R2	;R2 = H'FFFF9ABC, IMM means @(H'08,PC)
1004		ADD	#–1,R0	
1006		TST	R0,R0	;← PC location used for address calculation for ;the MOV.W instruction
1008		MOVT	R13	
100A		BRA	NEXT	;Delayed branch instruction
100C		MOV.L	@(4,PC),R3	;R3 = H'12345678
100E	IMM	*.data.w*	H'9ABC	
1010		*.data.w*	H'1234	
1012	NEXT	JMP	@R3	;Branch destination of the BRA instruction
1014		CMP/EQ	#0,R0	;← PC location used for address calculation for ;the MOV.L instruction
		.align	4	
1018		*.data.l*	H'12345678	

8.2.36 MOV (Move Peripheral Data): Data Transfer Instruction

Format		Abstract	Code	Cycle	T Bit
`MOV.B`	`@(disp,GBR),R0`	(disp + GBR) → sign extension → R0	11000100dddddddd	1	—
`MOV.W`	`@(disp,GBR),R0`	(disp × 2 + GBR) → sign extension → R0	11000101dddddddd	1	—
`MOV.L`	`@(disp,GBR),R0`	(disp × 4 + GBR) → R0	11000110dddddddd	1	—
`MOV.B`	`R0,@(disp,GBR)`	R0 → (disp + GBR)	11000000dddddddd	1	—
`MOV.W`	`R0,@(disp,GBR)`	R0 → (disp × 2 + GBR)	11000001dddddddd	1	—
`MOV.L`	`R0,@(disp,GBR)`	R0 → (disp × 4 + GBR)	11000010dddddddd	1	—

Description: Transfers the source operand to the destination. This instruction is suitable for accessing data in the peripheral module area. The data can be a byte, word, or longword, but only the R0 register can be used.

A peripheral module base address is set to the GBR. When the peripheral module data is a byte, the only change made is to zero-extend the 8-bit displacement. Consequently, an address within +255 bytes can be specified. When the peripheral module data is a word, the 8-bit displacement is zero-extended and doubled. Consequently, an address within +510 bytes can be specified. When the peripheral module data is a longword, the 8-bit displacement is zero-extended and is quadrupled. Consequently, an address within +1020 bytes can be specified. If the displacement is too short to reach the memory operand, the above @(R0,Rn) mode must be used after the GBR data is transferred to a general register. When the source operand is in memory, the loaded data is stored in the register after it is sign-extended to a longword.

Note: The destination register of a data load is always R0. R0 cannot be accessed by the next instruction until the load instruction is finished. The instruction order shown in figure 8-1 will give better results.

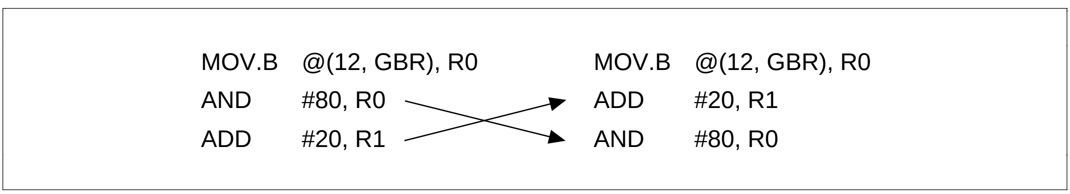


Figure 8-1 Using R0 after MOV

Operation:

```
MOVBLG(long d) /* MOV.B @(disp,GBR),R0 */
{
    long disp;

    disp=(0x000000FF & (long)d);
    R[0]=(long)Read_Byte(GBR+disp);
    if ((R[0]&0x80)==0) R[0]&=0x000000FF;
    else R[0]|=0xFFFFFF00;
    PC+=2;
}

MOVWLG(long d) /* MOV.W @(disp,GBR),R0 */
{
    long disp;

    disp=(0x000000FF & (long)d);
    R[0]=(long)Read_Word(GBR+(disp<<1));
    if ((R[0]&0x8000)==0) R[0]&=0x0000FFFF;
    else R[0]|=0xFFFF0000;
    PC+=2;
}

MOVLLG(long d) /* MOV.L @(disp,GBR),R0 */
{
    long disp;

    disp=(0x000000FF & (long)d);
    R[0]=Read_Long(GBR+(disp<<2));
    PC+=2;
}
```

```
MOVBSG(long d) /* MOV.B R0,@(disp,GBR) */
{
    long disp;

    disp=(0x000000FF & (long)d);
    Write_Byte(GBR+disp,R[0]);
    PC+=2;
}

MOVWSG(long d) /* MOV.W R0,@(disp,GBR) */
{
    long disp;

    disp=(0x000000FF & (long)d);
    Write_Word(GBR+(disp<<1),R[0]);
    PC+=2;
}

MOVLSG(long d) /* MOV.L R0,@(disp,GBR) */
{
    long disp;

    disp=(0x000000FF & (long)d);
    Write_Long(GBR+(disp<<2),R[0]);
    PC+=2;
}
```

Examples:

`MOV.L   @(2,GBR),R0`	;Before execution	@(GBR + 8) = H'12345670
	;After execution	R0 = @H'12345670
`MOV.B   R0,@(1,GBR)`	;Before execution	R0 = H'FFFF7F80
	;After execution	@(GBR + 1) = H'FFFF7F80

8.2.37 MOV (Move Structure Data): Data Transfer Instruction

Format	Abstract	Code	Cycle	T Bit
`MOV.B R0,@(disp,Rn)`	R0 → (disp + Rn)	`10000000`nnnndddd	1	—
`MOV.W R0,@(disp,Rn)`	R0 → (disp × 2 + Rn)	`10000001`nnnndddd	1	—
`MOV.L Rm,@(disp,Rn)`	Rm → (disp × 4 + Rn)	`0001`nnnnmmmmdddd	1	—
`MOV.B @(disp,Rm),R0`	(disp + Rm) → sign extension → R0	`10000100`mmmmdddd	1	—
`MOV.W @(disp,Rm),R0`	(disp × 2 + Rm) → sign extension → R0	`10000101`mmmmdddd	1	—
`MOV.L @(disp,Rm),Rn`	(disp × 4 + Rm) → Rn	`0101`nnnnmmmmdddd	1	—

Description: Transfers the source operand to the destination. This instruction is suitable for accessing data in a structure or a stack. The data can be a byte, word, or longword, but when a byte or word is selected, only the R0 register can be used. When the data is a byte, the only change made is to zero-extend the 4-bit displacement. Consequently, an address within +15 bytes can be specified. When the data is a word, the 4-bit displacement is zero-extended and doubled. Consequently, an address within +30 bytes can be specified. When the data is a longword, the 4-bit displacement is zero-extended and quadrupled. Consequently, an address within +60 bytes can be specified. If the displacement is too short to reach the memory operand, the aforementioned @(R0,Rn) mode must be used. When the source operand is in memory, the loaded data is stored in the register after it is sign-extended to a longword.

Note: When byte or word data is loaded, the destination register is always R0. R0 cannot be accessed by the next instruction until the load instruction is finished. The instruction order in figure 8-2 will give better results.

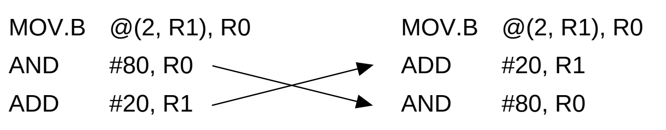


Figure 8-2 Using R0 after MOV

Operation:

```
MOVBS4(long d,long n) /* MOV.B R0,@(disp,Rn) */
{
    long disp;

    disp=(0x0000000F & (long)d);
    Write_Byte(R[n]+disp,R[0]);
    PC+=2;
}

MOVWS4(long d,long n) /* MOV.W R0,@(disp,Rn) */
{
    long disp;

    disp=(0x0000000F & (long)d);
    Write_Word(R[n]+(disp<<1),R[0]);
    PC+=2;
}

MOVLS4(long m,long d,long n)
    /* MOV.L Rm,@(disp,Rn) */
{
    long disp;

    disp=(0x0000000F & (long)d);
    Write_Long(R[n]+(disp<<2),R[m]);
    PC+=2;
}

MOVBL4(long m,long d) /* MOV.B @(disp,Rm),R0 */
{
    long disp;

    disp=(0x0000000F & (long)d);
    R[0]=Read_Byte(R[m]+disp);
    if ((R[0]&0x80)==0) R[0]&=0x000000FF;
    else R[0]|=0xFFFFFF00;
    PC+=2;
}
```

```
MOVWL4(long m,long d) /* MOV.W @(disp,Rm),R0 */
{
    long disp;

    disp=(0x0000000F & (long)d);
    R[0]=Read_Word(R[m]+(disp<<1));
    if ((R[0]&0x8000)==0) R[0]&=0x0000FFFF;
    else R[0]|=0xFFFF0000;
    PC+=2;
}

MOVLL4(long m,long d,long n)
    /* MOV.L @(disp,Rm),Rn */
{
    long disp;

    disp=(0x0000000F & (long)d);
    R[n]=Read_Long(R[m]+(disp<<2));
    PC+=2;
}
```

Examples:

`MOV.L`	`@(2,R0),R1`	;Before execution @(R0 + 8) = H'12345670
		;After execution R1 = @H'12345670
`MOV.L`	`R0,@(H'3C,R1)`	;Before execution R0 = H'FFFF7F80
		;After execution @(R1 + 60) = H'FFFF7F80

8.2.38 MOVA (Move Effective Address): Data Transfer Instruction

Format	Abstract	Code	Cycle	T Bit
`MOVA @(disp,PC),R0`	disp × 4 + PC → R0	11000111dddddddd	1	—

Description: Stores the effective address of the source operand into general register R0. The 8-bit displacement is zero-extended and quadrupled. Consequently, the relative interval from the operand is PC + 1020 bytes. The PC points to the starting address of the second instruction after this MOVA instruction, but the lowest two bits of the PC are corrected to B'00.

Note: If this instruction is placed immediately after a delayed branch instruction, the PC must point to an address specified by (the starting address of the branch destination) + 2.

Operation:

```
MOVA(long d)   /* MOVA @(disp,PC),R0 */
{
    long disp;

    disp=(0x000000FF & (long)d);
    R[0]=(PC&0xFFFFFFFC)+(disp<<2);
    PC+=2;
}
```

Examples:

```
Address .org   H'1006
1006           MOVA    STR,R0       ;Address of STR → R0
1008           MOV.B   @R0,R1       ;R1 = "X"  ← PC location after correcting the lowest
                                    ;two bits
100A           ADD     R4,R5        ;← Original PC location for address calculation for
                                    ;the MOVA instruction
               .align  4
100C    STR:   .sdata  "XYZP12"
...............
2002           BRA     TRGET        ;Delayed branch instruction
2004           MOVA    @(0,PC),R0   ;Address of TRGET + 2 → R0
2006           NOP
```

8.2.39 MOVT (Move T Bit): Data Transfer Instruction

Format	Abstract	Code	Cycle	T Bit
MOVT Rn	T → Rn	0000nnnn00101001	1	—

Description: Stores the T bit value into general register Rn. When T = 1, 1 is stored in Rn, and when T = 0, 0 is stored in Rn.

Operation:

```
MOVT(long n)  /* MOVT Rn */
{
    R[n]=(0x00000001 & SR);
    PC+=2;
}
```

Examples:

```
XOR     R2,R2   ;R2 = 0
CMP/PZ  R2      ;T = 1
MOVT    R0      ;R0 = 1
CLRT            ;T = 0
MOVT    R1      ;R1 = 0
```

8.2.40 MUL.L (Multiply Long): Arithmetic Instruction

Format	Abstract	Code	Cycle	T Bit
`MUL.L Rm,Rn`	Rn × Rm → MACL	`0000nnnnmmmm0111`	2 (to 5)	—

Description: Performs 32-bit multiplication of the contents of general registers Rn and Rm, and stores the bottom 32 bits of the result in the MACL register. The MACH register data does not change.

Operation:

```
MULL(long m,long n)   /* MUL.L Rm,Rn */
{
    MACL=R[n]*R[m];
    PC+=2;
}
```

Examples:

`MULL R0,R1`	;Before execution	R0 = H'FFFFFFFE, R1 = H'00005555
	;After execution	MACL = H'FFFF5556
`STS MACL,R0`	;Operation result	

8.2.41 MULS.W (Multiply as Signed Word): Arithmetic Instruction

Format		Abstract	Code	Cycle	T Bit
MULS.W MULS	Rm,Rn Rm,Rn	Signed operation, Rn × Rm → MACL	0010nnnnmmmm1111	1 (to 3)	—

Description: Performs 16-bit multiplication of the contents of general registers Rn and Rm, and stores the 32-bit result in the MACL register. The operation is signed and the MACH register data does not change.

Operation:

```
MULS(long m,long n)   /* MULS Rm,Rn */
{
   MACL=((long)(short)R[n]*(long)(short)R[m]);
   PC+=2;
}
```

Examples:

```
MULS  R0,R1      ;Before execution   R0 = H'FFFFFFFE, R1 = H'00005555
                 ;After execution    MACL = H'FFFF5556
STS   MACL,R0    ;Operation result
```

8.2.42 MULU.W (Multiply as Unsigned Word): Arithmetic Instruction

Format	Abstract	Code	Cycle	T Bit
`MULU.W Rm,Rn` `MULU Rm,Rn`	Unsigned, Rn × Rm → MACL	`0010nnnnmmmm1110`	1 (to 3)	—

Description: Performs 16-bit multiplication of the contents of general registers Rn and Rm, and stores the 32-bit result in the MACL register. The operation is unsigned and the MACH register data does not change.

Operation:

```
MULU(long m,long n)   /* MULU Rm,Rn */
{
    MACL=((unsigned long)(unsigned short)R[n]
        *(unsigned long)(unsigned short)R[m]);
    PC+=2;
}
```

Examples:

`MULU R0,R1`	;Before execution	R0 = H'00000002, R1 = H'FFFFAAAA
	;After execution	MACL = H'00015554
`STS MACL,R0`	;Operation result	

8.2.43 NEG (Negate): Arithmetic Instruction

Format		Abstract	Code	Cycle	T Bit
NEG	Rm,Rn	0 – Rm → Rn	0110nnnnmmmm1011	1	—

Description: Takes the two's complement of data in general register Rm, and stores the result in Rn. This effectively subtracts Rm data from 0, and stores the result in Rn.

Operation:

```
NEG(long m,long n)    /* NEG Rm,Rn */
{
    R[n]=0-R[m];
    PC+=2;
}
```

Examples:

```
NEG   R0,R1      ;Before execution     R0 = H'00000001
                 ;After execution      R1 = H'FFFFFFFF
```

8.2.44 NEGC (Negate with Carry): Arithmetic Instruction

Format	Abstract	Code	Cycle	T Bit
NEGC Rm,Rn	0 – Rm – T → Rn, Borrow → T	0110nnnnmmmm1010	1	Borrow

Description: Subtracts general register Rm data and the T bit from 0, and stores the result in Rn. If a borrow is generated, T bit changes accordingly. This instruction is used for inverting the sign of a value that has more than 32 bits.

Operation:

```
NEGC(long m,long n)   /* NEGC Rm,Rn */
{
    unsigned long temp;

    temp=0-R[m];
    R[n]=temp-T;
    if (0<temp)    T=1;
    else T=0;
    if (temp<R[n]) T=1;
    PC+=2;
}
```

Examples:

```
CLRT             ;Sign inversion of R1 and R0 (64 bits)
NEGC    R1,R1    ;Before execution    R1 = H'00000001, T = 0
                 ;After execution     R1 = H'FFFFFFFF, T = 1
NEGC    R0,R0    ;Before execution    R0 = H'00000000, T = 1
                 ;After execution     R0 = H'FFFFFFFF, T = 1
```

8.2.45 NOP (No Operation): System Control Instruction

Format	Abstract	Code	Cycle	T Bit
NOP	No operation	0000000000001001	1	—

Description: Increments the PC to execute the next instruction.

Operation:

```
NOP()  /* NOP */
{
    PC+=2;
}
```

Examples:

```
NOP     ;Executes in one cycle
```

8.2.46 NOT (NOT—Logical Complement): Logic Operation Instruction

Format	Abstract	Code	Cycle	T Bit
NOT Rm,Rn	$Rm \rightarrow Rn$	0110nnnnmmmm0111	1	—

Description: Takes the one's complement of general register Rm data, and stores the result in Rn. This effectively inverts each bit of Rm data and stores the result in Rn.

Operation:

```
NOT(long m,long n)    /* NOT Rm,Rn */
{
    R[n]=~R[m];
    PC+=2;
}
```

Examples:

```
NOT     R0,R1   ;Before execution   R0 = H'AAAAAAAA
                ;After execution    R1 = H'55555555
```

8.2.47 OR (OR Logical) Logic Operation Instruction

Format		Abstract	Code	Cycle	T Bit
OR	Rm,Rn	Rn | Rm → Rn	0010nnnnmmmm1011	1	—
OR	#imm,R0	R0 | imm → R0	11001011iiiiiiii	1	—
OR.B	#imm,@(R0,GBR)	(R0 + GBR) | imm → (R0 + GBR)	11001111iiiiiiii	3	—

Description: Logically ORs the contents of general registers Rn and Rm, and stores the result in Rn. The contents of general register R0 can also be ORed with zero-extended 8-bit immediate data, or 8-bit memory data accessed by using indirect indexed GBR addressing can be ORed with 8-bit immediate data.

Operation:

```
OR(long m,long n) /* OR Rm,Rn */
{
    R[n]|=R[m];
    PC+=2;
}

ORI(long i)    /* OR #imm,R0 */
{
    R[0]|=(0x000000FF & (long)i);
    PC+=2;
}

ORM(long i)    /* OR.B #imm,@(R0,GBR) */
{
    long temp;

    temp=(long)Read_Byte(GBR+R[0]);
    temp|=(0x000000FF & (long)i);
    Write_Byte(GBR+R[0],temp);
    PC+=2;
}
```

Examples:

```
OR      R0,R1              ;Before execution   R0 = H'AAAA5555, R1 = H'55550000
                           ;After execution    R1 = H'FFFF5555

OR      #H'F0,R0           ;Before execution   R0 = H'00000008
                           ;After execution    R0 = H'000000F8

OR.B    #H'50,@(R0,GBR)    ;Before execution   @(R0,GBR) = H'A5
                           ;After execution    @(R0,GBR) = H'F5
```

8.2.48 PREF (Prefetch Data to the Cache)

Format	Abstract	Code	Cycle	T Bit
PREF @Rn	(Rn &0xfffffff0) → Cache (Rn &0xfffffff0+4) → Cache (Rn &0xfffffff0+8) → Cache (Rn &0xfffffff0+C) → Cache	0000nnnn10000011	1	—

Description: Loads data to cache on software prefetching. 16-byte data containing the data pointed by Rn (Cache 1 line) is loaded to the cache. Address Rn should be on longword boundary.

No address related error is detected in this instruction. In case of an error, the instruction operates as NOP.

The destination is on-chip cache, therefore this instruction functions as an NOP instruction in effect, that is, it never changes registers or processor status.

Operation:

```
PREF(long n)   /*PREF*/
{
    PC+=2;
}
```

Examples:

```
           MOV.L      SOFT_PF,R1     ;Address of R1 is SOFT_PF
           PREF       @R1            ;Load data from SOFT_PF to on-chip cache

           .align 4

SOFT_PF:   .data.l    H'12345678
           .data.l    H'9ABCDEF0
           .data.l    H'AAAA5555
           .data.l    H'5555AAAA
```

8.2.49 ROTCL (Rotate with Carry Left): Shift Instruction

Format	Abstract	Code	Cycle	T Bit
ROTCL Rn	T ← Rn ← T	0100nnnn00100100	1	MSB

Description: Rotates the contents of general register Rn and the T bit to the left by one bit, and stores the result in Rn. The bit that is shifted out of the operand is transferred to the T bit (figure 8-3).

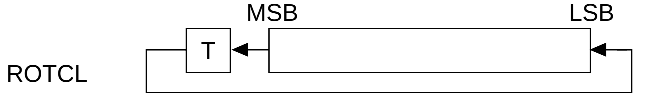


Figure 8-3 Rotate with Carry Left

Operation:

```
ROTCL(long n)  /* ROTCL Rn */
{
    long temp;

    if ((R[n]&0x80000000)==0) temp=0;
    else temp=1;
    R[n]<<=1;
    if (T==1) R[n]|=0x00000001;
    else R[n]&=0xFFFFFFFE;
    if (temp==1) T=1;
    else T=0;
    PC+=2;
}
```

Examples:

ROTCL R0	;Before execution	R0 = H'80000000, T = 0
	;After execution	R0 = H'00000000, T = 1

8.2.50 ROTCR (Rotate with Carry Right): Shift Instruction

Format	Abstract	Code	Cycle	T Bit
ROTCR Rn	T → Rn → T	0100nnnn00100101	1	LSB

Description: Rotates the contents of general register Rn and the T bit to the right by one bit, and stores the result in Rn. The bit that is shifted out of the operand is transferred to the T bit (figure 8-4).

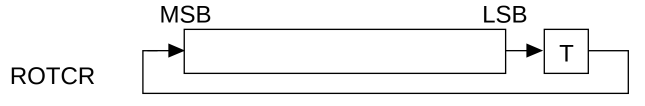


Figure 8-4 Rotate with Carry Right

Operation:

```
ROTCR(long n)  /* ROTCR Rn */
{
    long temp;

    if ((R[n]&0x00000001)==0) temp=0;
    else temp=1;
    R[n]>>=1;
    if (T==1) R[n]|=0x80000000;
    else R[n]&=0x7FFFFFFF;
    if (temp==1) T=1;
    else T=0;
    PC+=2;
}
```

Examples:

ROTCR R0	;Before execution	R0 = H'00000001, T = 1
	;After execution	R0 = H'80000000, T = 1

8.2.51 ROTL (Rotate Left): Shift Instruction

Format		Abstract	Code	Cycle	T Bit
ROTL	Rn	T ← Rn ← MSB	0100nnnn00000100	1	MSB

Description: Rotates the contents of general register Rn to the left by one bit, and stores the result in Rn (figure 8-5). The bit that is shifted out of the operand is transferred to the T bit.

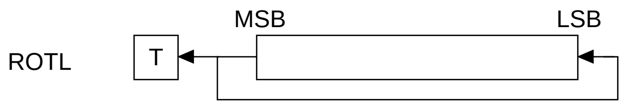


Figure 8-5 Rotate Left

Operation:

```
ROTL(long n)   /* ROTL Rn */
{
    if ((R[n]&0x80000000)==0) T=0;
    else T=1;
    R[n]<<=1;
    if (T==1) R[n]|=0x00000001;
    else R[n]&=0xFFFFFFFE;
    PC+=2;
}
```

Examples:

ROTL	R0	;Before execution	R0 = H'80000000, T = 0
		;After execution	R0 = H'00000001, T = 1

8.2.52 ROTR (Rotate Right): Shift Instruction

Format		Abstract	Code	Cycle	T Bit
ROTR	Rn	LSB → Rn → T	0100nnnn00000101	1	LSB

Description: Rotates the contents of general register Rn to the right by one bit, and stores the result in Rn (figure 8-6). The bit that is shifted out of the operand is transferred to the T bit.

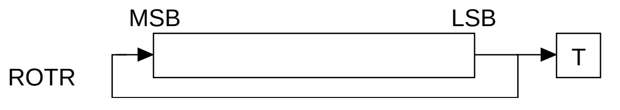


Figure 8-6 Rotate Right

Operation:

```
ROTR(long n)   /* ROTR Rn */
{
    if ((R[n]&0x00000001)==0) T=0;
    else T=1;
    R[n]>>=1;
    if (T==1) R[n]|=0x80000000;
    else R[n]&=0x7FFFFFFF;
    PC+=2;
}
```

Examples:

ROTR	R0	;Before execution	R0 = H'00000001, T = 0
		;After execution	R0 = H'80000000, T = 1

8.2.53 RTE (Return from Exception): System Control Instruction (Privileged Only)

Class: Delayed branch instruction

Format	Abstract	Code	Cycle	T Bit
RTE	SSR → SR, SPC → PC	0000000000101011	4	—

Description: Returns from an exception routine. The PC and SR values are loaded from SPC and SSR. The program continues from the address specified by the loaded PC value. RTE is a privileged instruction and can be used in privileged mode only. If used in user mode, it causes an illegal instruction exception.

Note: Since this is a delayed branch instruction, the instruction after RTE is executed before branching.

No interrupts are accepted between this instruction and the one immediately following it. If the instruction immediately following is a branch instruction, it is acknowledged as an illegal slot instruction.

If this instruction is located in a delayed slot immediately following a delayed branch instruction, it is acknowledged as an illegal slot instruction.

An instruction executed in a delayed slot immediately following this instruction uses the SR restored by this instruction.

Make sure that an instruction executed in a delayed slot immediately following this instruction does not cause an exception. Also, an instruction that manipulates the MD and BL bits of the SR register, as well as the instruction following it, should be used with the multiplier disabled or with fixed physical address space (P1 and P2).

Operation:

```
RTE()  /* RTE */
{
    unsigned long temp;

    temp=PC;
    PC=SPC;
    SR=SSR;
    Delay_Slot(temp+2);
}
```

Examples:

```
RTE              ;Returns to the original routine
ADD    #8,R15    ;Executes ADD before branching
```

Note: In delayed branching, the branching operation itself takes place after the slot instruction has been executed. However, execution of instructions (register updating, etc.) should always be done in the sequence of delayed branch instruction followed by delayed slot instruction. For example, even if a delayed slot updates a register in which the branching destination address is stored, the contents of the register before updating will be used as the branching destination address.

8.2.54 RTS (Return from Subroutine): Branch Instruction

Class: Delayed branch instruction

Format	Abstract	Code	Cycle	T Bit
RTS	PR → PC	0000000000001011	2	—

Description: Returns from a subroutine procedure. The PC values are restored from the PR, and the program continues from the address specified by the restored PC value. This instruction is used to return to the program from a subroutine program called by a BSR or JSR instruction.

Note: Since this is a delayed branch instruction, the instruction after this RTS is executed before branching. No interrupts are accepted between this instruction and the next instruction. If the next instruction is a branch instruction, it is acknowledged as an illegal slot instruction. If this instruction is located in a delayed slot immediately following a delayed branch instruction, it is acknowledged as an illegal slot instruction. An instruction restoring the PR should be prior to an RTS instruction. That restoring instruction should not be the delay slot of the RTS.

Operation:

```
RTS()  /* RTS */
{
    unsigned long temp;

    temp=PC;
    PC=PR+4;
    Delay_Slot(temp+2);
}
```

Examples:

```
         MOV.L    TABLE,R3          ;R3 = Address of TRGET
         JSR      @R3               ;Branches to TRGET
         NOP                        ;Executes NOP before branching
         ADD      R0,R1             ;← Return address for when the subroutine
                                    ;procedure is completed (PR data)
    .............
TABLE:   .data.l  TRGET             ;Jump table
    .............
TRGET:   MOV      R1,R0             ;← Procedure entrance
         RTS                        ;PR data → PC
         MOV      #12,R0            ;Executes MOV before branching
```

Note: In delayed branching, the branching operation itself takes place after the slot instruction has been executed. However, execution of instructions (register updating, etc.) should always be done in the sequence of delayed branch instruction followed by delayed slot instruction. For example, even if a delayed slot updates a register in which the branching destination address is stored, the contents of the register before updating will be used as the branching destination address.

8.2.55 SETRC (Set Repeat Count to RC): System Control Instruction (SH3-DSP Only)

Format		Abstract	Code	Cycle	T Bit
SETRC	Rm	LSW of Rm → RC (MSW of SR), Repeat control flag → RF1, RF0	0100mmmm00010100	3	—
SETRC	#imm	imm → RC (MSW of SR), Repeat control flag → RF1, RF0	10000010iiiiiiii	3	—

Description: Sets the repeat count to the SR register's RC counter. When the operand is a register, the bottom 12 bits are used as the repeat count. When the operand is an immediate data value, 8 bits are used as the repeat count. Set repeat control flags to RF1, RF0 bits of the SR register. Use of the SETRC instruction is subject to any limitations. Refer to section 5.12, DSP Repeat (Loop) Control, for more information.

Operation:

```
SETRC(long m)  /* SETRC Rm */
{
    long temp;

    temp=(R[m] & 0x00000FFF)<<16;
    SR&=0xF000FFF3;
    SR|=temp;
    RF1=Repeat_Control_Flag1;
    RF0=Repeat_Control_Flag0;
    PC+=2;
}

SETRCI(long i) /* SETRC #imm */
{
    long temp;

    temp=((long)i & 0x000000FF)<<16;
    SR&=0xF000FFFF;
    SR|=temp;
    RF1=Repeat_Control_Flag1;
    RF0=Repeat_Control_Flag0;
    PC+=2;
}
```

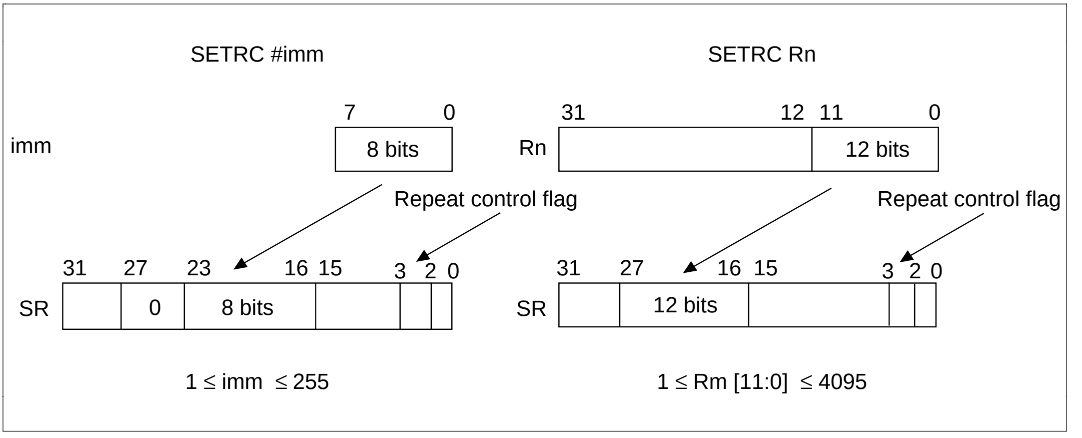


Figure 8-7 SETRC Instruction

Example:

```
            LDRS STA        ;Set repeat start address to RS.
            LDRE END        ;Set repeat end address to RE.
            SETRC #32       ;Repeat 32 times from inst.A to inst.C.
            inst.0          ;
 STA:            inst.A     ;
                 inst.B     ;
            ............
 END:            inst.C     ;
            inst.D          ;
```

8.2.56 SETS (Set S Bit): System Control Instruction

Format	Abstract	Code	Cycle	T Bit
SETS	$1 \rightarrow S$	0000000001011000	1	—

Description: Sets the S bit to 1.

Operation:

```
SETT() /* SETS */
{
    S=1;
    PC+=2;
}
```

Examples:

SETS	;Before execution	S = 0
	;After execution	S = 1

8.2.57 SETT (Set T Bit): System Control Instruction

Format	Abstract	Code	Cycle	T Bit
SETT	1 → T	0000000000011000	1	1

Description: Sets the T bit to 1.

Operation:

```
SETT() /* SETT */
{
    T=1;
    PC+=2;
}
```

Examples:

```
SETT      ;Before execution   T = 0
          ;After execution    T = 1
```

8.2.58 SHAD (Shift Arithmetic Dynamically): Shift Instruction

Format		Abstract	Code	Cycle	T Bit
SHAD	Rm,Rn	Rn << Rm → Rn (Rm ≥ 0) Rn >> Rm → [MSB → Rn]	0100nnnnmmmm1100	2	—

Description: Arithmetically shifts the contents of general register Rn. General register Rm indicates the shift direction and the number of bits to be shifted.

- If the value of the Rm register is positive, the shift is to the left, if it is negative the shift is to the right.
- The number of bits to be shifted is indicated by the five lower bits (bits 4 to 0) of the Rm register. If the value is negative (MSB = 1), the Rm register is indicated with a complement of 2. The magnitude of left shift may be 0 to 31, and the magnitude of right shift may be 1 to 32.

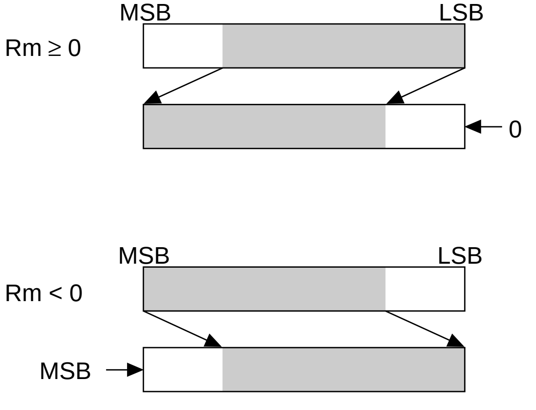


Figure 8-8 Shift Arithmetic Dynamically

Operation:

```
SHAD(long m,n) /* SHAD Rm,Rn */
{
    long cont, sgn;
    sgn = R[m] &0x80000000;
    cnt = R[m] &0x0000001F;
    if (sgn==0)    R[n]<<=cnt;
    else           R[n]=(signed long)R[n]>>((~cnt+1) & 0x1F); /*shift
                   arithmetic right*/
    PC+=2;
}
```

Examples:

SHAD	R1,R2	;Before execution	R1 = H'FFFFFFEC, R2 = H'80180000
		;After execution	R1 = H'FFFFFFEC, R2 = H'FFFFF801
SHAD	R3,R4	;Before execution	R3 = H'00000014, R4 = H'FFFFF801
		;After execution	R3 = H'00000014, R4 = H'80100000

8.2.59 SHAL (Shift Arithmetic Left): Shift Instruction

Format	Abstract	Code	Cycle	T Bit
SHAL Rn	T ← Rn ← 0	0100nnnn00100000	1	MSB

Description: Arithmetically shifts the contents of general register Rn to the left by one bit, and stores the result in Rn. The bit that is shifted out of the operand is transferred to the T bit (figure 8-9).

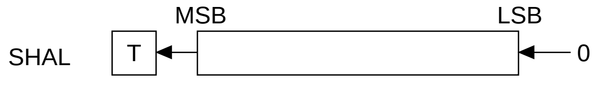


Figure 8-9 Shift Arithmetic Left

Operation:

```
SHAL(long n)   /* SHAL Rn (Same as SHLL) */
{
    if ((R[n]&0x80000000)==0) T=0;
    else T=1;
    R[n]<<=1;
    PC+=2;
}
```

Examples:

SHAL	R0	;Before execution	R0 = H'80000001, T = 0
		;After execution	R0 = H'00000002, T = 1

8.2.60 SHAR (Shift Arithmetic Right): Shift Instruction

Format		Abstract	Code	Cycle	T Bit
SHAR	Rn	MSB → Rn → T	0100nnnn00100001	1	LSB

Description: Arithmetically shifts the contents of general register Rn to the right by one bit, and stores the result in Rn. The bit that is shifted out of the operand is transferred to the T bit (figure 8-10).

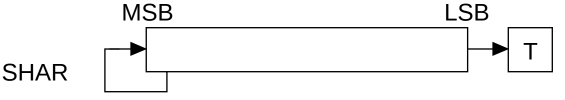


Figure 8-10 Shift Arithmetic Right

Operation:

```
SHAR(long n)   /* SHAR Rn */
{
    long temp;

    if ((R[n]&0x00000001)==0) T=0;
    else T=1;
    if ((R[n]&0x80000000)==0) temp=0;
    else temp=1;
    R[n]>>=1;
    if (temp==1) R[n]|=0x80000000;
    else R[n]&=0x7FFFFFFF;
    PC+=2;
}
```

Examples:

SHAR	R0	;Before execution	R0 = H'80000001, T = 0
		;After execution	R0 = H'C0000000, T = 1

8.2.61 SHLD (Shift Logical Dynamically): Shift Instruction

Format		Abstract	Code	Cycle	T Bit
`SHLD`	`Rm,Rn`	Rn << Rm → Rn (Rm ≥ 0) Rn >> Rm → [0 → Rn] (Rm < 0)	`0100nnnnmmmm1101`	1	—

Description: Arithmetically shifts the contents of general register Rn. General register Rm indicates the shift direction and the number of bits to be shifted. The T bit is the last shifted bit of Rn. If the value of the Rm register is positive, the shift is to the left, if it is negative the shift is to the right. If the shift is to the right, a top bit of 0 is added.

The number of bits to be shifted is indicated by the five lower bits (bits 4 to 0) of the Rm register. If the value is negative (MSB = 1), the Rm register is indicated with a complement of 2. The magnitude of left shift may be 0 to 31, and the magnitude of right shift may be 1 to 32.

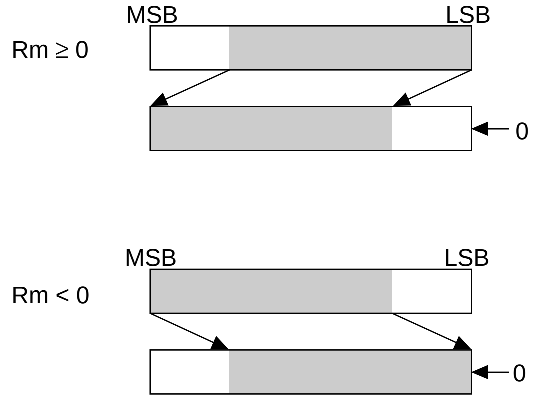


Figure 8-11 Shift Logical Dynamically

Operation:

```
    SHLD(long m,n) /* SHLD Rm,Rn */
{
    long cont, sgn;

    sgn = R[m]&0x80000000;
    cnt = R[m]&0x0000001F);
    if (sgn==0)    R[n]<<=cnt;
    else           R[n]=R[n]>>((~cnt+1)&0x1F);
    PC+=2;
}
```

Examples:

SHLD R1,R2	;Before execution	R1 = H'FFFFFFEC, R2 = H'80180000
	;After execution	R1 = H'FFFFFFEC, R2 = H'00000801
SHLD R3,R4	;Before execution	R3 = H'00000014, R4 = H'FFFFF801
	;After execution	R3 = H'00000014, R4 = H'80100000

8.2.62 SHLL (Shift Logical Left): Shift Instruction

Format	Abstract	Code	Cycle	T Bit
SHLL Rn	T ← Rn ← 0	0100nnnn00000000	1	MSB

Description: Logically shifts the contents of general register Rn to the left by one bit, and stores the result in Rn. The bit that is shifted out of the operand is transferred to the T bit (figure 8-12).

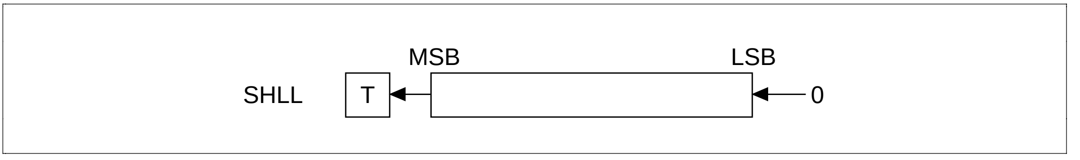


Figure 8-12 Shift Logical Left

Operation:

```
SHLL(long n)   /* SHLL Rn (Same as SHAL) */
{
    if ((R[n]&0x80000000)==0) T=0;
    else T=1;
    R[n]<<=1;
    PC+=2;
}
```

Examples:

SHLL	R0	;Before execution	R0 = H'80000001, T = 0
		;After execution	R0 = H'00000002, T = 1

8.2.63 SHLLn (Shift Logical Left n Bits): Shift Instruction

Format		Abstract	Code	Cycle	T Bit
SHLL2	Rn	Rn << 2 → Rn	0100nnnn00001000	1	—
SHLL8	Rn	Rn << 8 → Rn	0100nnnn00011000	1	—
SHLL16	Rn	Rn << 16 → Rn	0100nnnn00101000	1	—

Description: Logically shifts the contents of general register Rn to the left by 2, 8, or 16 bits, and stores the result in Rn. Bits that are shifted out of the operand are not stored (figure 8-13).

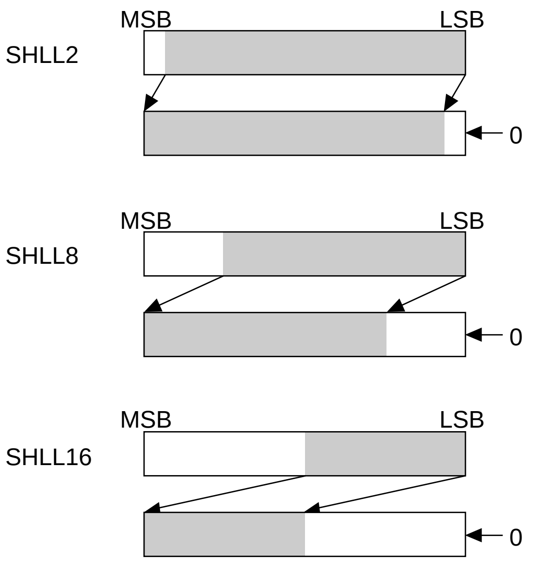


Figure 8-13 Shift Logical Left n Bits

Operation:

```
    SHLL2(long n)  /* SHLL2 Rn */
{
    R[n]<<=2;
    PC+=2;
}
```

```
SHLL8(long n)  /* SHLL8 Rn */
{
    R[n]<<=8;
    PC+=2;
}

SHLL16(long n) /* SHLL16 Rn */
{
    R[n]<<=16;
    PC+=2;
}
```

Examples:

`SHLL2 R0`	;Before execution	R0 = H'12345678
	;After execution	R0 = H'48D159E0
`SHLL8 R0`	;Before execution	R0 = H'12345678
	;After execution	R0 = H'34567800
`SHLL16 R0`	;Before execution	R0 = H'12345678
	;After execution	R0 = H'56780000

8.2.64 SHLR (Shift Logical Right): Shift Instruction

Format	Abstract	Code	Cycle	T Bit
SHLR Rn	0 → Rn → T	0100nnnn00000001	1	LSB

Description: Logically shifts the contents of general register Rn to the right by one bit, and stores the result in Rn. The bit that is shifted out of the operand is transferred to the T bit (figure 8-14).

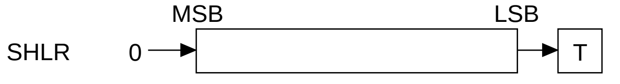


Figure 8-14 Shift Logical Right

Operation:

```
SHLR(long n)   /* SHLR Rn */
{
    if ((R[n]&0x00000001)==0) T=0;
    else T=1;
    R[n]>>=1;
    R[n]&=0x7FFFFFFF;
    PC+=2;
}
```

Examples:

SHLR	R0	;Before execution	R0 = H'80000001, T = 0
		;After execution	R0 = H'40000000, T = 1

8.2.65 SHLRn (Shift Logical Right n Bits): Shift Instruction

Format		Abstract	Code	Cycle	T Bit
SHLR2	Rn	Rn>>2 → Rn	0100nnnn00001001	1	—
SHLR8	Rn	Rn>>8 → Rn	0100nnnn00011001	1	—
SHLR16	Rn	Rn>>16 → Rn	0100nnnn00101001	1	—

Description: Logically shifts the contents of general register Rn to the right by 2, 8, or 16 bits, and stores the result in Rn. Bits that are shifted out of the operand are not stored (figure 8-15).

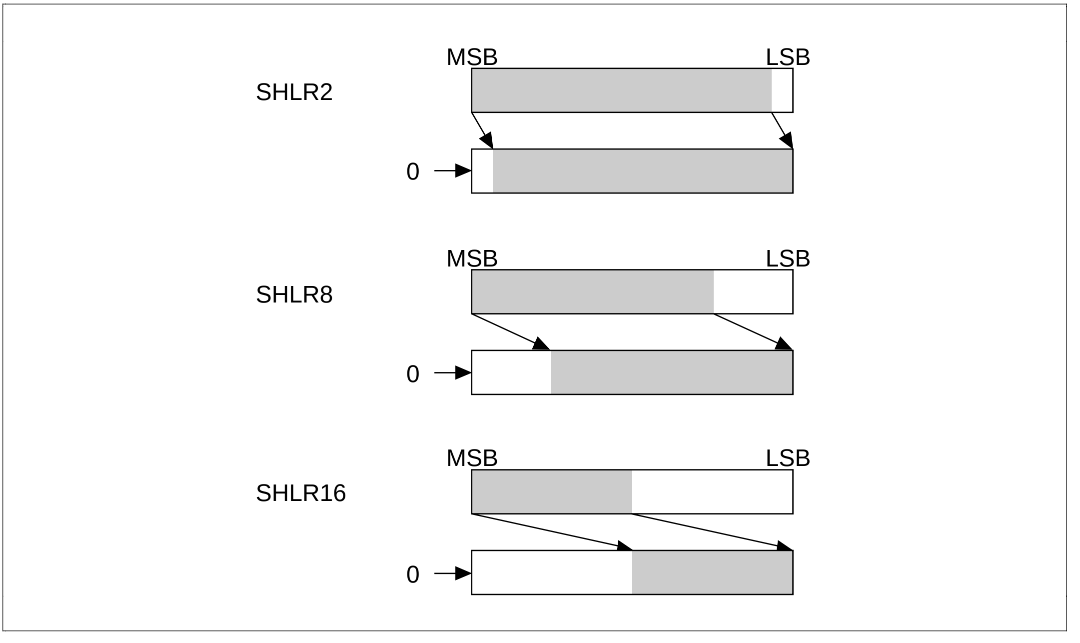


Figure 8-15 Shift Logical Right n Bits

Operation:

```
SHLR2(long n)  /* SHLR2 Rn */
{
    R[n]>>=2;
    R[n]&=0x3FFFFFFF;
    PC+=2;
}
```

```
SHLR8(long n)  /* SHLR8 Rn */
{
    R[n]>>=8;
    R[n]&=0x00FFFFFF;
    PC+=2;
}

SHLR16(long n) /* SHLR16 Rn */
{
    R[n]>>=16;
    R[n]&=0x0000FFFF;
    PC+=2;
}
```

Examples:

SHLR2 R0	;Before execution	R0 = H'12345678
	;After execution	R0 = H'048D159E
SHLR8 R0	;Before execution	R0 = H'12345678
	;After execution	R0 = H'00123456
SHLR16 R0	;Before execution	R0 = H'12345678
	;After execution	R0 = H'00001234

8.2.66 SLEEP (Sleep): System Control Instruction (Privileged Only)

Format	Abstract	Code	Cycle	T Bit
SLEEP	Sleep	0000000000011011	4	—

Description: Sets the CPU into power-down mode. In power-down mode, instruction execution stops, but the CPU module status is maintained, and the CPU waits for an interrupt request. If an interrupt is requested, the CPU exits the power-down mode and begins exception processing.

SLEEP is a privileged instruction and can be used in privileged mode only. If used in user mode, it causes an illegal instruction exception.

Note: The number of cycles given is for the transition to sleep mode.

Operation:

```
SLEEP()/* SLEEP */
{
    PC-=2;
    Error(“Sleep Mode.”);
}
```

Examples:

```
SLEEP      ;Enters power-down mode
```

8.2.67 STC (Store Control Register): System Control Instruction (Privileged Only)

Format		Abstract	Code	Cycle	T Bit
STC	SR,Rn	SR → Rn	0000nnnn00000010	1	—
STC	GBR,Rn	GBR → Rn	0000nnnn00010010	1	—
STC	VBR,Rn	VBR → Rn	0000nnnn00100010	1	—
STC	SSR,Rn	SSR → Rn	0000nnnn00110010	1	—
STC	SPC,Rn	SPC → Rn	0000nnnn01000010	1	—
STC	MOD,Rn*1	MOD → Rn	0000nnnn01010010	1	—
STC	RE,Rn*1	RE → Rn	0000nnnn01110010	1	—
STC	RS,Rn*1	RS → Rn	0000nnnn01100010	1	—
STC	R0_BANK,Rn	R0_BANK → Rn	0000nnnn10000010	1	—
STC	R1_BANK,Rn	R1_BANK → Rn	0000nnnn10010010	1	—
STC	R2_BANK,Rn	R2_BANK → Rn	0000nnnn10100010	1	—
STC	R3_BANK,Rn	R3_BANK → Rn	0000nnnn10110010	1	—
STC	R4_BANK,Rn	R4_BANK → Rn	0000nnnn11000010	1	—
STC	R5_BANK,Rn	R5_BANK → Rn	0000nnnn11010010	1	—
STC	R6_BANK,Rn	R6_BANK → Rn	0000nnnn11100010	1	—
STC	R7_BANK,Rn	R7_BANK → Rn	0000nnnn11110010	1	—
STC.L	SR,@-Rn	Rn – 4 → Rn, SR → (Rn)	0100nnnn00000011	1/2*2	—
STC.L	GBR,@-Rn	Rn – 4 → Rn, GBR → (Rn)	0100nnnn00010011	1/2*2	—
STC.L	VBR,@-Rn	Rn – 4 → Rn, VBR → (Rn)	0100nnnn00100011	1/2*2	—
STC.L	SSR,@-Rn	Rn – 4 → Rn, SSR → (Rn)	0100nnnn00110011	1/2*2	—
STC.L	SPC,@-Rn	Rn – 4 → Rn, SPC → (Rn)	0100nnnn01000011	1/2*2	—
STC.L	MOD,@-Rn*1	Rn – 4 → Rn, MOD → (Rn)	0100nnnn01010011	2	—
STC.L	RE,@-Rn*1	Rn – 4 → Rn, RE → (Rn)	0100nnnn01110011	2	—
STC.L	RS,@-Rn*1	Rn – 4 → Rn, RS → (Rn)	0100nnnn01100011	2	—
STC.L	R0_BANK,@-Rn	Rn – 4 → Rn, R0_BANK → (Rn)	0100nnnn10000011	2	—
STC.L	R1_BANK,@-Rn	Rn – 4 → Rn, R1_BANK → (Rn)	0100nnnn10010011	2	—
STC.L	R2_BANK,@-Rn	Rn – 4 → Rn, R2_BANK → (Rn)	0100nnnn10100011	2	—
STC.L	R3_BANK,@-Rn	Rn – 4 → Rn, R3_BANK → (Rn)	0100nnnn10110011	2	—

Format	Abstract	Code	Cycle	T Bit
STC.L R4_BANK,@-Rn	Rn – 4 → Rn, R4_BANK → (Rn)	0100nnnn11000011	2	—
STC.L R5_BANK,@-Rn	Rn – 4 → Rn, R5_BANK → (Rn)	0100nnnn11010011	2	—
STC.L R6_BANK,@-Rn	Rn – 4 → Rn, R6_BANK → (Rn)	0100nnnn11100011	2	—
STC.L R7_BANK,@-Rn	Rn – 4 → Rn, R7_BANK → (Rn)	0100nnnn11110011	2	—

Notes: 1. SH3-DSP only.
2. Two cycles on the SH3-DSP.

Description: Stores data from control registers SR, GBR, VBR, SSR, SPC, MOD, RE and RS, or R0_BANK to R7_BANK to a specified location. STC and STC.L, except for STC GBR, Rn and STC.L GBR, @-Rn, are privileged instructions and can be used in privileged mode only. If used in user mode, they can cause illegal instruction exceptions. Note that STC GBR, Rn and STC.L GBR, @-Rn can be used in user mode.

The Rm_BANK operand is designated by the RB bit of the SR register. When the value of the RB bit is 1, the R0_BANK1 to R7_BANK1 registers and the R8 to R15 registers are used as the Rn operand, and the R0_BANK0 to R7_BANK0 registers are used as the Rm_BANK operand. When the value of the RB bit is 0, the R0_BANK0 to R7_BANK0 registers and the R8 to R15 registers are used as the Rn operand, and the R0_BANK1 to R7_BANK1 registers are used as the Rm_BANK operand.

Operation:

```
STCSR(long n)     /* STC SR,Rn */
{
    R[n]=SR;
    PC+=2;
}

STCGBR(long n) /* STC GBR,Rn */
{
    R[n]=GBR;
    PC+=2;
}
```

```
STCVBR(long n) /* STC VBR,Rn */
{
    R[n]=VBR;
    PC+=2;
}

STCSSR(long n) /* STC SSR,Rn */
{
    R[n]=SSR;
    PC+=2;
}

STCSPC(long n) /* STC SPC,Rn */
{
    R[n]=SPC;
    PC+=2;
}

STCRn_BANK(long n)    /* STC Rn_BANK,Rm */
{                     /* n=0–7 */
    R[n]=Rn_BANK;
    PC+=2;
}


STCMSR(long n) /* STC.L SR,@-Rn */
{
    R[n]-=4;
    Write_Long(R[n],SR);
    PC+=2;
}

STCMGBR(long n)   /* STC.L GBR,@-Rn */
{
    R[n]-=4;
    Write_Long(R[n],GBR);
    PC+=2;
}

STCMVBR(long n)   /* STC.L VBR,@-Rn */
{
```

```
    R[n]-=4;
    Write_Long(R[n],VBR);
    PC+=2;
}
STCMSSR(long n)    /* STC.L SSR,@-Rn */
{
    R[n]-=4;
    Write_Long(R[n],SSR);
    PC+=2;
}

STCMSPC(long n)    /* STC.L SPC,@-Rn */
{
    R[n]-=4;
    Write_Long(R[n],SPC);
    PC+=2;
}

STCMRm(long n) /* STC.L Rm_BANK,@-Rnn */
                   /* n=0–7 */
{
    R[n]-=4;
    Write_Long(R[n],Rm_BANK);
    PC+=2;
}

STCMOD(long n)     /* STC MOD,Rn */
{
    R[n]=MOD;
    PC+=2;
}

STCRE(long n)  /* STC RE,Rn */
{
    R[n]=RE;
    PC+=2;
}
```

```
STCRS(long n)  /* STC RS,Rn */
{
    R[n]=RS;
    PC+=2;
}

STCMVBR(long n)   /* STC.L VBR,@-Rn */
{
    R[n]-=4;
    Write_Long(R[n],VBR);
    PC+=2;
}

STCMMOD(long n)   /* STC.L MOD,@-Rn */
{
    R[n]-=4;
    Write_Long(R[n],MOD);
    PC+=2;
}

STCMRE(long n) /* STC.L RE,@-Rn */
{
    R[n]-=4;
    Write_Long(R[n],RE);
    PC+=2;
}

STCMRS(long n) /* STC.L RS,@-Rn */
{
    R[n]-=4;
    Write_Long(R[n],SR);
    PC+=2;
}
```

Examples:

STC	SR,R0	;Before execution	R0 = H'FFFFFFFF, SR = H'00000000
		;After execution	R0 = H'00000000
STC.L	GBR,@-R15	;Before execution	R15 = H'10000004
		;After execution	R15 = H'10000000, @R15 = GBR

8.2.68 STS (Store System Register): System Control Instruction

Format		Abstract	Code	Cycle	T Bit
STS	MACH,Rn	MACH → Rn	0000nnnn00001010	1	—
STS	MACL,Rn	MACL → Rn	0000nnnn00011010	1	—
STS	PR,Rn	PR → Rn	0000nnnn00101010	1	—
STS	DSR,Rn*	DSR → Rn	0000nnnn01101010	1	—
STS	A0,Rn*	A0 → Rn	0000nnnn01111010	1	—
STS	X0,Rn*	X0→Rn	0000nnnn10001010	1	—
STS	X1,Rn*	X1→Rn	0000nnnn10011010	1	—
STS	Y0,Rn*	Y0→Rn	0000nnnn10101010	1	—
STS	Y1,Rn*	Y1→Rn	0000nnnn10111010	1	—
STS.L	MACH,@–Rn	Rn – 4 → Rn, MACH → (Rn)	0100nnnn00000010	1	—
STS.L	MACL,@–Rn	Rn – 4 → Rn, MACL → (Rn)	0100nnnn00010010	1	—
STS.L	PR,@–Rn	Rn – 4 → Rn, PR → (Rn)	0100nnnn00100010	1	—
STS.L	DSR,@–Rn*	Rn – 4 → Rn, DSR → (Rn)	0100nnnn01100010	1	—
STS.L	A0,@–Rn*	Rn – 4 → Rn, A0 → (Rn)	0100nnnn01100010	1	—
STS.L	X0,@-Rn*	Rn–4→Rn,X0→(Rn)	0100nnnn10000010	1	—
STS.L	X1,@-Rn*	Rn–4→Rn,X1→(Rn)	0100nnnn10010010	1	—
STS.L	Y0,@-Rn*	Rn–4→Rn,Y0→(Rn)	0100nnnn10100010	1	—
STS.L	Y1,@-Rn*	Rn–4→Rn,Y1→(Rn)	0100nnnn10110010	1	—

Note: * SH3-DSP only.

Description: Stores system registers MACH, MACL, PR, DSP, A0, X0, X1, Y0, and Y1 data into a specified destination.

Note: In the case of system register MACH, the 32-bit contents is stored unchanged.

Operation:

```
STSMACH(long n)   /* STS MACH,Rn */
{
    R[n]=MACH;
    if ((R[n]&0x00000200)==0)
    R[n]&=0x000003FF;
    else R[n]|=0xFFFFFC00;
    PC+=2;
}
```

```
STSMACL(long n)   /* STS MACL,Rn */
{
    R[n]=MACL;
    PC+=2;
}

STSPR(long n)     /* STS PR,Rn */
{
    R[n]=PR;
    PC+=2;
}

STSMMACH(long n)  /* STS.L MACH,@–Rn */
{
    R[n]–=4;
    if ((MACH&0x00000200)==0)
    Write_Long(R[n],MACH&0x000003FF);
    else Write_Long (R[n],MACH|0xFFFFFC00)
    PC+=2;
}

STSMMACL(long n)  /* STS.L MACL,@–Rn */
{
    R[n]–=4;
    Write_Long(R[n],MACL);
    PC+=2;
}

STSMPR(long n) /* STS.L PR,@–Rn */
{
    R[n]–=4;
    Write_Long(R[n],PR);
    PC+=2;
}

STSDSR(long n) /* STS DSR,Rn */
{
    R[n]=DSR;
    PC+=2;
}
```

```
STSA0(long n)     /* STS A0,Rn */
{
    R[n]=A0;
    PC+=2;
}

STSX0(long n)     /* STS X0,Rn */
{
    R[n]=X0;
    PC+=2;
}

STSX1(long n)     /* STS X1,Rn */
{
    R[n]=X1;
    PC+=2;
}

STSY0(long n)     /* STS Y0,Rn */
{
    R[n]=Y0;
    PC+=2;
}

STSY1(long n)     /* STS Y1,Rn */
{
    R[n]=Y1;
    PC+=2;
}

STSMDSR(long n)   /* STS.L DSR,@-Rn */
{
    R[n]-=4;
    Write_Long(R[n],DSR);
    PC+=2;
}
```

```
STSMA0(long n) /* STS.L A0,@–Rn */
{
    R[n]–=4;
    Write_Long(R[n],A0);
    PC+=2;
}

STSMX0(long n) /* STS.L X0,@–Rn */
{
    R[n]–=4;
    Write_Long(R[n],X0);
    PC+=2;
}

STSMX1(long n) /* STS.L X1,@–Rn */
{
    R[n]–=4;
    Write_Long(R[n],X1);
    PC+=2;
}

STSMY0(long n) /* STS.L Y0,@–Rn */
{
    R[n]–=4;
    Write_Long(R[n],Y0);
    PC+=2;
}

STSMY1(long n) /* STS.L Y1,@–Rn */
{
    R[n]–=4;
    Write_Long(R[n],Y1);
    PC+=2;
}
```

Examples:

```
STS     MACH,R0     ;Before execution   R0 = H'FFFFFFFF, MACH = H'00000000
                    ;After execution    R0 = H'00000000

STS.L   PR,@–R15    ;Before execution   R15 = H'10000004
                    ;After execution    R15 = H'10000000, @R15 = PR
```

8.2.69 SUB (Subtract Binary): Arithmetic Instruction

Format		Abstract	Code	Cycle	T Bit
SUB	Rm,Rn	Rn – Rm → Rn	0011nnnnmmmm1000	1	—

Description: Subtracts general register Rm data from Rn data, and stores the result in Rn. To subtract immediate data, use ADD #imm,Rn.

Operation:

```
SUB(long m,long n)    /* SUB Rm,Rn */
{
    R[n]-=R[m];
    PC+=2;
}
```

Examples:

SUB	R0,R1	;Before execution	R0 = H'00000001, R1 = H'80000000
		;After execution	R1 = H'7FFFFFFF

8.2.70 SUBC (Subtract with Carry): Arithmetic Instruction

Format		Abstract	Code	Cycle	T Bit
SUBC	Rm,Rn	Rn – Rm– T → Rn, Borrow → T	0011nnnnmmmm1010	1	Borrow

Description: Subtracts Rm data and the T bit value from general register Rn data, and stores the result in Rn. The T bit changes according to the result. This instruction is used for subtraction of data that has more than 32 bits.

Operation:

```
SUBC(long m,long n)   /* SUBC Rm,Rn */
{
    unsigned long tmp0,tmp1;

    tmp1=R[n]-R[m];
    tmp0=R[n];
    R[n]=tmp1-T;
    if (tmp0<tmp1) T=1;
    else T=0;
    if (tmp1<R[n]) T=1;
    PC+=2;
}
```

Examples:

```
CLRT            ;R0:R1(64 bits) – R2:R3(64 bits) = R0:R1(64 bits)
SUBC    R3,R1   ;Before execution    T = 0, R1 = H'00000000, R3 = H'00000001
                ;After execution     T = 1, R1 = H'FFFFFFFF
SUBC    R2,R0   ;Before execution    T = 1, R0 = H'00000000, R2 = H'00000000
                ;After execution     T = 1, R0 = H'FFFFFFFF
```

8.2.71 SUBV (Subtract with V Flag Underflow Check): Arithmetic Instruction

Format		Abstract	Code	Cycle	T Bit
SUBV	Rm,Rn	Rn – Rm → Rn, Underflow → T	0011nnnnmmmm1011	1	Underflow

Description: Subtracts Rm data from general register Rn data, and stores the result in Rn. If an underflow occurs, the T bit is set to 1.

Operation:

```
SUBV(long m,long n)   /* SUBV Rm,Rn */
{
    long dest,src,ans;

    if ((long)R[n]>=0) dest=0;
    else dest=1;
    if ((long)R[m]>=0) src=0;
    else src=1;
    src+=dest;
    R[n]-=R[m];
    if ((long)R[n]>=0) ans=0;
    else ans=1;
    ans+=dest;
    if (src==1) {
        if (ans==1) T=1;
        else T=0;
    }
    else T=0;
    PC+=2;
}
```

Examples:

SUBV	R0,R1	;Before execution	R0 = H'00000002, R1 = H'80000001
		;After execution	R1 = H'7FFFFFFF, T = 1
SUBV	R2,R3	;Before execution	R2 = H'FFFFFFFE, R3 = H'7FFFFFFE
		;After execution	R3 = H'80000000, T = 1

8.2.72 SWAP (Swap Register Halves): Data Transfer Instruction

Format	Abstract	Code	Cycle	T Bit
`SWAP.B Rm,Rn`	Rm → Swap upper and lower halves of lower 2 bytes → Rn	`0110nnnnmmmm1000`	1	—
`SWAP.W Rm,Rn`	Rm → Swap upper and lower word → Rn	`0110nnnnmmmm1001`	1	—

Description: Swaps the upper and lower bytes of the general register Rm data, and stores the result in Rn. If a byte is specified, bits 0 to 7 of Rm are swapped for bits 8 to 15. The upper 16 bits of Rm are transferred to the upper 16 bits of Rn. If a word is specified, bits 0 to 15 of Rm are swapped for bits 16 to 31.

Operation:

```
SWAPB(long m,long n)  /* SWAP.B Rm,Rn */
{
    unsigned long temp0,temp1;

    temp0=R[m]&0xffff0000;
    temp1=(R[m]&0x000000ff)<<8;
    R[n]=(R[m]&0x0000ff00)>>8;
    R[n]=R[n]|temp1|temp0;
    PC+=2;
}

SWAPW(long m,long n)  /* SWAP.W Rm,Rn */
{
    unsigned long temp;
    temp=(R[m]>>16)&0x0000FFFF;
    R[n]=R[m]<<16;
    R[n]|=temp;
    PC+=2;
}
```

Examples:

```
SWAP.B   R0,R1      ;Before execution     R0 = H'12345678
                    ;After execution      R1 = H'12347856

SWAP.W   R0,R1      ;Before execution     R0 = H'12345678
                    ;After execution      R1 = H'56781234
```

8.2.73 TAS (Test and Set): Logic Operation Instruction

Format	Abstract	Code	Cycle	T Bit
TAS.B @Rn	When (Rn) is 0, 1 → T, 1 → MSB of (Rn)	0100nnnn00011011	3/4*	Test results

Note: * Four cycles on the SH3-DSP.

Description: Reads byte data from the address specified by general register Rn, and sets the T bit to 1 if the data is 0, or clears the T bit to 0 if the data is not 0. Then, data bit 7 is set to 1, and the data is written to the address specified by Rn. During this operation, the bus is not released.

Note: The destination of the TAS instruction should be placed in a non-cacheable space when the cache is enabled.

Operation:

```
TAS(long n)    /* TAS.B @Rn */
{
    long temp;

    temp=(long)Read_Byte(R[n]);    /* Bus Lock enable */
    if (temp==0) T=1;
    else T=0;
    temp|=0x00000080;
    Write_Byte(R[n],temp);         /* Bus Lock disable */
    PC+=2;
}
```

Example:

```
_LOOP  TAS.B  @R7      ;R7 = 1000
       BF     _LOOP    ;Loops until data in address 1000 is 0
```

8.2.74 TRAPA (Trap Always): System Control Instruction

Format	Abstract	Code	Cycle	T Bit
TRAPA #imm	imm → TRA, PC → SPC, SR → SSR, 1 → SR.MD/BL/RB 0x160 → EXPEVT VBR + H'00000100 → PC	11000011iiiiiiii	6/8*	—

Note: * Eight cycles on the SH3-DSP.

Description: Starts the trap exception processing. The PC and SR values are saved in SPC and SSR. Eight-bit immediate data is stored in the TRA registers (TRA9 to TRA2). The processor goes into privileged mode (SR.MD = 1) with SR.BL = 1 and SR.RB = 1, that is, blocking exceptions and masking interrupts, and selecting BANK1 registers (R0_BANK1 to R7_BANK1). Exception code 0x160 is stored in the EXPEVT register (EXPEVT11 to EXPEVT0). The program branches to an address (VBR+H'00000100). TRAPA and RTE are both used together for system calls.

Note: If this instruction is located in a delayed slot immediately following a delayed branch instruction, it is acknowledged as an illegal slot instruction.

Operation:

```
TRAPA(long i)  /* TRAPA #imm */
{
    long imm;
    imm=(0x000000FF & i);
    TRA=imm<<2;
    SSR=SR;
    SPC=PC;
    SR.MD=1
    SR.BL=1
    SR.RB=1
    EXPEVT=0x00000160;
    PC=VBR+H'00000100;
}
```

8.2.75 TST (Test Logical): Logic Operation Instruction

Format	Abstract	Code	Cycle	T Bit
TST Rm,Rn	Rn & Rm, when result is 0, 1 → T	0010nnnnmmmm1000	1	Test results
TST #imm,R0	R0 & imm, when result is 0, 1 → T	11001000iiiiiiii	1	Test results
TST.B #imm,@(R0,GBR)	(R0 + GBR) & imm, when result is 0, 1 → T	11001100iiiiiiii	3	Test results

Description: Logically ANDs the contents of general registers Rn and Rm, and sets the T bit to 1 if the result is 0 or clears the T bit to 0 if the result is not 0. The Rn data does not change. The contents of general register R0 can also be ANDed with zero-extended 8-bit immediate data, or the contents of 8-bit memory accessed by indirect indexed GBR addressing can be ANDed with 8-bit immediate data. The R0 and memory data do not change.

Operation:

```
TST(long m,long n)    /* TST Rm,Rn */
{
    if ((R[n]&R[m])==0) T=1;
    else T=0;
    PC+=2;
}

TSTI(long i)   /* TEST #imm,R0 */
{
    long temp;

    temp=R[0]&(0x000000FF & (long)i);
    if (temp==0) T=1;
    else T=0;
    PC+=2;
}
```

```
TSTM(long i)  /* TST.B #imm,@(R0,GBR) */
{
    long temp;

    temp=(long)Read_Byte(GBR+R[0]);
    temp&=(0x000000FF & (long)i);
    if (temp==0) T=1;
    else T=0;
    PC+=2;
}
```

Examples:

TST	R0,R0	;Before execution	R0 = H'00000000
		;After execution	T = 1
TST	#H'80,R0	;Before execution	R0 = H'FFFFFF7F
		;After execution	T = 1
TST.B	#H'A5,@(R0,GBR)	;Before execution	@(R0,GBR) = H'A5
		;After execution	T = 0

8.2.76 XOR (Exclusive OR Logical): Logic Operation Instruction

Format		Abstract	Code	Cycle	T Bit
XOR	Rm,Rn	Rn ^ Rm → Rn	0010nnnnmmmm1010	1	—
XOR	#imm,R0	R0 ^ imm → R0	11001010iiiiiiii	1	—
XOR.B	#imm,@(R0,GBR)	(R0 + GBR) ^ imm → (R0 + GBR)	11001110iiiiiiii	3	—

Description: Exclusive ORs the contents of general registers Rn and Rm, and stores the result in Rn. The contents of general register R0 can also be exclusive ORed with zero-extended 8-bit immediate data, or 8-bit memory accessed by indirect indexed GBR addressing can be exclusive ORed with 8-bit immediate data.

Operation:

```
XOR(long m,long n)    /* XOR Rm,Rn */
{
    R[n]^=R[m];
    PC+=2;
}

XORI(long i)   /* XOR #imm,R0 */
{
    R[0]^=(0x000000FF & (long)i);
    PC+=2;
}

XORM(long i)   /* XOR.B #imm,@(R0,GBR) */
{
    long temp;

    temp=(long)Read_Byte(GBR+R[0]);
    temp^=(0x000000FF & (long)i);
    Write_Byte(GBR+R[0],temp);
    PC+=2;
}
```

Examples:

```
XOR     R0,R1              ;Before execution  R0 = H'AAAAAAAA, R1 = H'55555555
                           ;After execution   R1 = H'FFFFFFFF

XOR     #H'F0,R0           ;Before execution  R0 = H'FFFFFFFF
                           ;After execution   R0 = H'FFFFFF0F

XOR.B   #H'A5,@(R0,GBR)    ;Before execution  @(R0,GBR) = H'A5
                           ;After execution   @(R0,GBR) = H'00
```

8.2.77 XTRCT (Extract): Data Transfer Instruction

Format	Abstract	Code	Cycle	T Bit
XTRCT Rm,Rn	Rm: Center 32 bits of Rn → Rn	0010nnnnmmmm1101	1	—

Description: Extracts the middle 32 bits from the 64 bits of general registers Rm and Rn, and stores the 32 bits in Rn (figure 8-16).

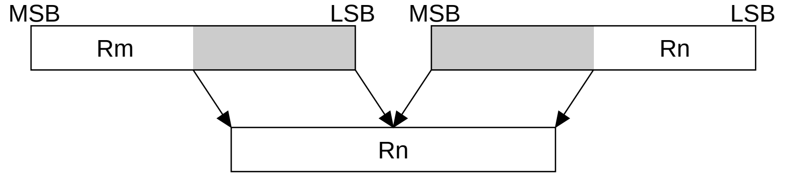


Figure 8-16 Extract

Operation:

```
XTRCT(long m,long n)  /* XTRCT Rm,Rn */
{
    unsigned long temp;

    temp=(R[m]<<16)&0xFFFF0000;
    R[n]=(R[n]>>16)&0x0000FFFF;
    R[n]|=temp;
    PC+=2;
}
```

Example:

XTRCT R0,R1	;Before execution	R0 = H'01234567, R1 = H'89ABCDEF
	;After execution	R1 = H'456789AB

8.3 Floating Point Instructions and FPU Related CPU Instructions (SH-3E Only)

The functions used in the descriptions of the operation of FPU calculations are as follows.

```
long FPSCR;
int T;

int load_long(long *adress, *data)
{
          /* This function is defined in CPU part */
}
int store_long(long *adress, *data)
{
          /* This function is defined in CPU part */
}
int sign_of(long *src)
{
          return(*src >> 31);
}
int data_type_of(long *src)
{
float abs;
          abs = *src & 0x7fffffff;
          if(abs < 0x00800000) {
             if(sign_of (src) == 0) return(PZERO);
             else                   return(NZERO);
}
 else if((0x00800000 <= abs) && (abs < 0x7f800000))
                                    return(NORM);
 else if(0x7f800000 == abs) {
             if(sign_of (src) == 0) return(PINF);
             else                   return(NINF);
          }
 else if(0x00400000 & abs)          return(sNaN);
 else                               return(qNaN);
          }
}
```

```
clear_cause_VZ(){ FPSCR &= (~CAUSE_V & ~CAUSE_Z); }
set_V(){ FPSCR Ω= (CAUSE_V Ω FLAG_V); }
set_Z(){ FPSCR Ω= (CAUSE_Z Ω FLAG_Z); }

invalid(float *dest)
{
 set_V();
 if((FPSCR & ENABLE_V) == 0) qnan(dest);
 }
}
dz(float *dest, int sign)
{
 set_Z();
 if((FPSCR & ENABLE_Z) == 0) inf (dest,sign);
}
zero(float *dest, int sign)
{
             if(sign == 0)          *dest = 0x00000000;
             else                   *dest = 0x80000000;
}
int(float *dest, int sign)
{
             if(sign == 0)          *dest = 0x7f800000;
             else                   *dest = 0xff800000;
}
qnan(float *dest)
{
             *dest = 0x7fbfffff;
}
```

8.3.1 FABS (Floating Point Absolute Value): Floating Point Instruction

Format	Abstract	Code	Latency	Cycles	T Bit
FABS FRn	|FRn| → FRn	1111nnnn01011101	2	1	—

Description: Obtains arithmetic absolute value (as a floating point number) of the contents of floating point register FRn. The calculation result is stored in FRn.

Operation:

```
FABS(float *Frn)  /* FABS FRn */
{
 clear_cause_VZ();
 case(data_type_of(FRn))                        {
         NORM:     if(sign_of(FRn) == 0)     *FRn = *FRn;
                   else                      *FRn = -*FRn;
                                                              break;
         PZERO :
         NZERO :   zero(FRn,0);                               break;
         PINF  :
         NINF  :   inf(FRn,0);                                break;
         qnan  :   qnan(FRn);                                 break;
         sNaN  :   invalid(FRn);                              break;
 }
 pc += 2;
}
```

FABS Special Cases

FRn	NORM	+0	–0	+INF	–INF	qNaN	sNaN
FABS(FRn)	ABS	+0	+0	+INF	+INF	qNaN	Invalid

Note: Non-normalized values are treated as zero.

Exceptions: Invalid operation

Examples:

```
FABS   FR2     ; Floating point absolute value
               ; Before execution  FR2=H'C0800000/*–4 in base 10*/
               ; After execution  FR2=H'40800000/*4 in base 10*/
```

8.3.2 FADD (Floating Point Add): Floating Point Instruction

Format	Abstract	Code	Latency	Cycles	T Bit
FADD FRm,FRn	FRn+FRm → FRn	1111nnnnmmmm0000	2	1	—

Description: Arithmetically adds (as floating point numbers) the contents of floating point registers FRm and FRn. The calculation result is stored in FRn.

Operation:

```
FADD (float *FRm,FRn)                        /* FADD FRm,FRn */
{
       clear_cause_VZ();
       if((data_type_of(FRm) = = sNaN)      ||
                (data_type_of(FRn) = = sNaN))            invalid(FRn);
       else if((data_type_of(FRm) = = qNaN) ||
                (data_type_of(FRn) = = qNaN))            qnan(FRn);
       else case(data_type_of(FRm))         {
    NORM:
      case(data_type_of(FRn))               {
         PINF      :       inf(FRn,0);                   break;
         NINF      :       inf(FRn,1);                   break;
         default   :       *FRn = *FRn + *FRm;           break;
     }                                                   break;
    PZERO:
      case(data_type_of(FRn))               {
         NORM      :       *FRn = *FRn + *FRm;           break;
         PZERO     :
         NZERO     :       zero(FRn,0);                  break;
         PINF      :       inf(FRn,0);                   break;
         NINF      :       inf(FRn,1);                   break;
      }                                                  break;
    NZERO:
      case(data_type_of(FRn)){
         NORM      :       *FRn = *FRn + *FRm;           break;
         PZERO     :       zero(FRn,0);                  break;
         NZERO     :       zero(FRn,1);                  break;
         PINF      :       inf(FRn,0);                   break;
         NINF      :       inf(FRn,1);                   break;
```

```
        }                                                   break;
      PINF:
        case(data_type_of(FRn))                {
           NINF     :       invalid(FRn);                   break;
           default  :       inf(FRn,0);                     break;
        }                                                   break;
      NINF:
        case(data_type_of(FRn)){
           PINF     :       invalid(FRn);                   break;
           default  :       inf(FRn,1);                     break;
        }                                                   break;
     }
       pc += 2;
   }
```

FADD Special Cases

FRm	FRn						
	NORM	+0	–0	+INF	–INF	qNaN	sNaN
NORM	ADD				–INF		
+0		+0					
–0			–0				
+INF				+INF	Invalid		
–INF	–INF			Invalid	–INF		
qNaN						qNaN	
sNaN							Invalid

Note: Non-normalized values are treated as zero.

Exceptions: Invalid operation

Examples:

```
FADD   FR2,FR3     ; Floating point add
                   ; Before execution:  FR2=H'40400000/*3 in base 10*/
                   ;                    FR3=H'3F800000/*1 in base 10*/
                   ; After execution:   FR2=H'40400000
                   ;                    FR3=H'40800000/*4 in base 10*/
FADD   FR5,FR4     ;
                   ; Before execution:  FR5=H'40400000/*3 in base 10*/
                   ;                    FR4=H'C0000000/*–2 in base 10*/
                   ; After execution:   FR5=H'40400000
                   ;                    FR4=H'3F800000/*1 in base 10*/
```

8.3.3 FCMP (Floating Point Compare): Floating Point Instruction

Format	Abstract	Code	Latency	Cycles	T Bit
FCMP/EQ FRm,FRn	(FRn==FRm)? 1:0 → T	1111nnnnmmmm0100	2	1	Comparison result
FCMP/GT FRm,FRn	(FRn> FRm)? 1:0 → T	1111nnnnmmmm0101	2	1	Comparison result

Description: Arithmetically compares (as floating point numbers) the contents of floating point registers FRm and FRn. The calculation result (true/false) is written to the T bit.

Operation:

```
FCMP_EQ(float *FRm,FRn)    /* FCMP/EQ FRm,FRn */
{
           clear_cause_VZ();
           if (fcmp_chk(FRm,FRn) = = INVALID) {fcmp_invalid(0); }
           else if(fcmp_chk(FRm,FRn) = = EQ)       T = 1;
           else                                    T = 0;
           pc += 2;
}
FCMP_GT(float *FRm,FRn)    /* FCMP/GT FRm,FRn */
{
           clear_cause_VZ();
           if (fcmp_chk(FRm,FRn)==INVALID)||{fcmp_chk(FRm,FRn)==UO)){
           fcmp_invalid(0):}
           else if(fcmp_chk(FRm,FRn) = = GT)       T = 1;
           else                             T = 0;
           pc += 2;
}
fcmp_chk(float *FRm,*FRn)
{
           if((data_type_of(FRm) == sNaN) ||
            (data_type_of(FRn) == sNaN))     return(INVALID);
   else    if((data_type_of(FRm) == qNaN) ||
           (data_type_of(FRn) == qNaN))      return(UO);
  else     case(data_type_of(FRm))             {
           NORM             :case(data_type_of(FRn))          {
                            PINF             :return(GT);       break;
```

```
                                NINF                :return(NOTGT);    break;
                                default             :                  break;
                }                                                      break;
                PZERO           :
                NZERO      :    case(data_type_of(FRn))     {
                                PZERO               :
                                NZERO               :return(EQ);       break;
                                PINF                :return(GT);       break;
                                NINF                :return(NOTGT);    break;
                                default             :                  break;
                }                                                      break;
                PINF       :    case(data_type_of(FRn))     {
                                PINF                :return(EQ)        break;
                                default             :return(NOTGT);    break;
                }                                                      break;
                NINF       :    case(data_type_of(FRn))     {
                                NINF                :return(EQ);       break;
                                default             :return(GT);       break;
                }                                                      break;
      }
      if(*FRn = = *FRm)                             return(EQ);
      else if(*FRn > *FRm)                          return(GT);
      else                                          return(NOTGT);
}
fcmp_invalid(int cmp_flag)
{
      set_V();
      if((FPSCR & ENABLE_V) = = 0)  T = cmp_flag;
}
```

FCMP Special Cases

<table>
<tr><th rowspan="2">FRm</th><th colspan="7">FRn</th></tr>
<tr><th>NORM</th><th>+0</th><th>–0</th><th>+INF</th><th>–INF</th><th>qNaN</th><th>sNaN</th></tr>
<tr><td>NORM</td><td colspan="3">CMP</td><td rowspan="3">GT</td><td rowspan="4">!GT</td><td rowspan="5"></td><td rowspan="6"></td></tr>
<tr><td>+0</td><td rowspan="2"></td><td colspan="2" rowspan="2">EQ</td></tr>
<tr><td>–0</td></tr>
<tr><td>+INF</td><td colspan="3">!GT</td><td>EQ</td></tr>
<tr><td>–INF</td><td colspan="4">GT</td><td>EQ</td></tr>
<tr><td>qNaN</td><td colspan="6">UO</td></tr>
<tr><td>sNaN</td><td colspan="7">Invalid</td></tr>
</table>

Notes: 1. UO if result is FCMP/EQ, invalid if result is FCMP/GT.
2. Non-normalized values are treated as zero.

Exceptions: Invalid operation

Note: Four comparison operations that are independent of each other are defined in the IEEE standard, but the SH-3E supports FCMP/EQ and FCMP/GT only. However, all comparison conditions can be supported by using these two FCMP instructions in combination with the BT and BF instructions.

```
      (FRm = = FRn)            fcmp/eq FRm, FRn ; bt
      (FRm ! = FRn)            fcmp/eq FRm, FRn ; bf
      (FRm < FRn)              fcmp/gt FRm, FRn ; bt
      (FRm <= FRn)             fcmp/gt FRn, FRm ; bt
      (FRm > FRn)              fcmp/gt FRn, FRm ; bt
      (FRm >= FRn)             fcmp/gt FRm, FRn ; bf
Unorder FRm, FRn               fcmp/eq FRm, FRm ; bf
```

Examples:

```
FCMP/EQ:
      FLDI1      FR6          ;FR6=H'3F800000/*1 in base 10*/
      FLDI1      FR7          ;FR7=H'3F800000
      CLRT                    ;T Bit =0
      FCMP/EQ    FR6,FR7      ; Floating point compare, equal
      BF         TRGET_F      ; Don't branch (T=1)
      NOP
      BT/S       TRGET_T      ; Branch
      FADD       FR6,FR7      ; Delay slot,  FR7=H'40000000/*2 in base 10*/
```

```
        NOP
TRGET_F FCMP/EQ       FR6,FR7
        BT/S  TRGET_T                  ; Don't branch (T=0)
        FLDI1         FR7              ; Delay slot
TRGET_T FCMP/EQ       FR6,FR7          ; T bit = 0
        BF TRGET_F                     ; Branch first time only
        NOP                            ;FR6=FR7=H'3F800000/*1 in base 10*/
        .END
```

FCMP/GT:

```
        FLDI1         FR2              ;FR2=H'3F800000/*1 in base 10*/
        FLDI1         FR7
        FADD          FR2,FR7          ;FR7=H'40000000/*2 in base 10*/
        CLRT                           ; T bit = 0
        FCMP/GT       FR2,FR7          ; Floating point compare, greater than
        BT/S          TRGET_T          ; Branch (T=1)
        FLDI1         FR7
TRGET_T FCMP/GT       FR2,FR7          ; T bit = 0
        BT            TRGET_T          ; Don't branch (T=0)
        .END
```

8.3.4 FDIV (Floating Point Divide): Floating Point Instruction

Format	Abstract	Code	Latency	Cycles	T Bit
FDIV FRm, FRn	FRn/FRm → FRn	1111nnnnmmmm0011	14	13	—

Description: Arithmetically divides (as floating point numbers) the contents of floating point register FRn by the contents of floating point register FRm. The calculation result is stored in FRn.

Operation:

```
FDIV(float *FRm,*FRn)   /* FDIV FRm,FRn       */
{
       clear_cause_VZ();
       if((data_type_of(FRm) = = sNaN) | |
            (data_type_of(FRn) = = sNaN))     invalid(FRn);
       else if((data_type_of(FRm) = = qNaN) | |
            (data_type_of(FRn) = = qNaN))     qnan(FRn);
       else case((data_type_of(FRm)           {
       NORM  :
       case(data_type_of(FRn))                {
             PINF     :
             NINF     :  inf(FRn,sign_of(FRm)^sign_of(FRn));        break;
             default  :  *FRn =*FRn / *FRm;                         break;
       }                                                            break;
       PZERO :
       NZERO :
       case(data_type_of(FRn))                {
             PZERO    :
             NZERO    :  invalid(FRn);                              break;
             PINF     :
             NINF     :  inf(Fn,Sign_of(FRm)^sign_of(FRn));         break;
             default  :  dz(FRn,sign_of(FRm)^sign_of(FRn));         break;
       }                                                            break;
       PINF  :
       NINF  :
       case(data_type_of(FRn))                {
             PINF     :
             NINF     :  invalid(FRn);                              break;
             default  :zero (FRn,sign_of(FRm)^sign_of(FRn));        break
```

```
                                                break;
        }
        pc += 2;
}
```

FDIV Special Cases

<table>
<tr><td rowspan="2">FRm</td><td colspan="7">FRn</td></tr>
<tr><td>NORM</td><td>+0</td><td>–0</td><td>+INF</td><td>–INF</td><td>qNaN</td><td>sNaN</td></tr>
<tr><td>NORM</td><td>DIV</td><td colspan="2">0</td><td colspan="2" rowspan="3">INF</td><td rowspan="6">qNaN</td><td rowspan="7">Invalid</td></tr>
<tr><td>+0</td><td rowspan="2">DZ</td><td colspan="2" rowspan="2">Invalid</td></tr>
<tr><td>–0</td></tr>
<tr><td>+INF</td><td rowspan="2">0</td><td>+0</td><td>–0</td><td colspan="2" rowspan="2">Invalid</td></tr>
<tr><td>–INF</td><td>–0</td><td>+0</td></tr>
<tr><td>qNaN</td><td colspan="5"></td></tr>
<tr><td>sNaN</td><td colspan="6"></td></tr>
</table>

Note: Non-normalized values are treated as zero.

Exceptions: Invalid operation, divide by zero

Examples:

```
FDIV    FR6, FR5    ; Floating point divide
                    ; Before execution:    ;FR5=H'40800000/*4 in base 10*/
                    ;                      ;FR6=H'40400000/*3 in base 10*/
                    ; After execution:     ;FR5=H'3FAAAAAA/*1.33... in base 10*/
                    ;                      ;FR6=H'40400000
```

8.3.5 FLDI0 (Floating Point Load Immediate 0): Floating Point Instruction

Format	Abstract	Code	Latency	Cycles	T Bit
FLDI0 FRn	H'00000000 → FRn	1111nnnn10001101	2	1	—

Description: Loads the floating point number 0 (0x00000000) in floating point register FRn.

Operation:

```
FLDI0(float *FRn)                               /* FLDI0 FRn */
{
           *FRn = 0x00000000;
           pc += 2;
}
```

Exceptions: None

Examples:

```
FLDI0     FR1          ; Load immediate 0
                       ; Before execution:    FR1=x (don't care)
                       ; After execution:     FR1=00000000
```

8.3.6 FLDI1 (Floating Point Load Immediate 1): Floating Point Instruction

Format	Abstract	Code	Latency	Cycles	T Bit
FLDI1 FRn	H'3F800000 → FRn	1111nnnn10011101	2	1	—

Description: Loads the floating point number 1 (0x3F800000) in floating point register Frn.

Operation:

```
FLDI1(float *FRn)                                /* FLDI1 FRn */
{
         *FRn = 0x3F800000;
         pc += 2;
}
```

Exceptions: None

Examples:

```
FLDI1    FR2               ; Load immediate 1
                           ; Before execution:      FR2=x (don't care)
                           ; After execution:       FR2=H'3F800000/*1 in base 10*/
```

8.3.7 FLDS (Floating Point Load to System Register): Floating Point Instruction

Format	Abstract	Code	Latency	Cycles	T Bit
FLDS FRm,FPUL	FRm → FPUL	1111nnnn00011101	2	1	—

Description: Loads the contents of floating point register FRm to system register FPUL.

Operation:

```
FLDS(float *FRm,*FPUL)                    /* FLDS FRm,FPUL */
{
           *FPUL = *FRm;
           pc += 2;
}
```

Exceptions: None

Examples:

```
                                  ;Before execution of FLDS and FSTS:
FLDI1    FR6                      ;          FR6=H'3F800000/*1 in base 10*/
FLDI0    FR2                      ;          FR2=0
                                  ;After execution of FLDS and FSTS:
FLDS     FR6, FPUL                ;          FPUL=H'3F800000
FSTS     FPUL, FR2                ;          FR2= H'3F800000
```

8.3.8 FLOAT (Floating Point Convert from Integer): Floating Point Instruction

Format	Abstract	Code	Latency	Cycles	T Bit
FLOAT FPUL,FRn	(float)FPUL → FRn	1111nnnn00101101	2	1	—

Description: Interprets the contents of FPUL as an integer value and converts it into a floating point number. The result is stored in floating point register FRn.

Operation:

```
FLOAT(int,*FPUL,float *FRn)            /* FLOAT FRn */
{
            clear_cause_VZ();
            *FRn = (float)*FPUL;
            pc += 2;
}
```

Exceptions: None

Examples:

```
                                   ;Floating Point Convert from Integer
                                   ;Before execution of FLOAT instruction:
MOV.L    #H'00000003,R1            ;          R1=H'00000003
FLDI0    FR2                       ;          FR2=0
                                   ;After execution of FLOAT instruction:
LDS      R1, FPUL                  ;          FPUL=H'00000003
FLOAT    FPUL, FR2                 ;          FR2=H'40400000/*3 in base 10*/
```

8.3.9 FMAC (Floating Point Multiply Accumulate): Floating Point Instruction

Format		Abstract	Code	Latency	Cycles	T Bit
FMAC	FR0, FRm,FRn	FR0 × FRm+FRn → FRn	1111nnnnmmmm1110	2	1	—

Description: Arithmetically multiplies (as floating point numbers) the contents of floating point registers FR0 and FRm. To this calculation result is added the contents of floating point register FRn, and the result is stored in FRn.

Operation:

```
FMAC(float *FR0,*FRm,*FRn)          /* FMAC FR0,FRm,FRn */
{
long        tmp_FPSCR;
float       *tmp_FMUL = *FRm;
            FMUL(F0,tmp_FMUL);
            pc -=2;                 /* correct pc  */
            tmp_FPSCR = FPSCR;      /* save cause field for FR0*FRm */
            FADD(tmp_FMUL,FRn);
            FPSCR |= tmp_FPSCR;     /* reflect cause field for F0*FRm   */
}
```

FMAC Special Cases

FRn	FR0	FRm: +NORM	–NORM	+0	–0	+INF	–INF	qNaN	sNaN
NORM	NORM	MAC				INF			
	0					Invalid			
	+INF	+INF	–INF	Invalid		+INF	–INF		
	–INF	–INF	+INF			–INF	+INF		
+0	NORM	MAC				INF			
	0				+0	Invalid			
	+INF	+INF	–INF	Invalid		+INF	–INF		
	–INF	–INF	+INF			–INF	+INF		
–0	+NORM	MAC		+0	–0	+INF	–INF		
	–NORM			–0	+0	–INF	+INF		
	+0	+0	–0	+0	–0	Invalid			
	–0	–0	+0	–0	+0				
	+INF	+INF	–INF	Invalid		+INF	–INF		
	–INF	–INF	+INF			–INF	+INF		
+INF	+NORM	+INF					Invalid		
	–NORM						+INF		
	0					Invalid			
	+INF			Invalid		+INF			
	–INF	Invalid	+INF				+INF		
–INF	+NORM	–INF					–INF		
	–NORM								
	0								
	+INF	Invalid		Invalid			–INF		
	–INF	–INF				–INF	Invalid		
qNaN	0					Invalid			
	INF			Invalid					
	!sNaN								
!NaN	qNaN						qNaN		
All types	sNaN								
sNaN	All types								Invalid

Note: Non-normalized values are treated as zero.

Exceptions: Invalid operation

Examples:

```
FMAC FR0, FR3, FR5      ;Floating point multiply accumulate
                                          FR0*FR3+FR5->FR5
                        ;Before execution: FR0=H'40000000/*2 in base 10*/
                        ;                  FR3=H'40800000/*4 in base 10*/
                        ;                  FR5=H'3F800000/*1 in base 10*/
                        ;After execution:  FR0=H'40000000/*2 in base 10*/
                        ;                  FR3=H'40800000/*4 in base 10*/
                        ;                  FR5=H'41100000/*9 in base 10*/

FMAC FR0, FR0, FR5      ;FR0*FR0+FR5->FR5
                        ;Before execution: FR0=H'40000000/*2 in base 10*/
                        ;                  FR5=H'3F800000/*1 in base 10*/
                        ;After execution:  FR0=H'40000000/*2 in base 10*/
                        ;                  FR5=H'40A00000/*5 in base 10*/

FMAC FR0, FR5, FR0      ;FR0*FR5+FR0->FR5
                        ;Before execution: FR0=H'40000000/*2 in base 10*/
                        ;                  FR5=H'40A00000/*5 in base 10*/
                        ;After execution:  FR0=H'41400000/*12 in base 10*/
                        ;                  FR5=H'40A00000/*5 in base 10*/
```

8.3.10 FMOV (Floating Point Move): Floating Point Instruction

Format	Abstract	Code	Latency (Wait Time)	Cycles	T Bit
1.FMOV FRm,FRn	FRm → FRn	1111nnnnmmmm1100	2	1	—
2.FMOV.S @Rm,FRn	(Rm) → FRn	1111nnnnmmmm1000	2	1	—
3.FMOV.S FRm, @Rn	FRm → (Rn)	1111nnnnmmmm1010	2	1	—
4.FMOV.S @Rm+,FRn	(Rm) → FRn, Rm+=4	1111nnnnmmmm1001	2	1	—
5.FMOV.S FRm,@-Rn	Rn-=4, FRm → (Rn)	1111nnnnmmmm1011	2	1	—
6.FMOV.S @(R0,Rm),FRn	(R0+Rm) → FRn	1111nnnnmmmm0110	2	1	—
7.FMOV.S FRm, @(R0,Rn)	FRm → (R0+Rn)	1111nnnnmmmm0111	2	1	—

Description:

1. Moves the contents of floating point register FRm to floating point register FRn.
2. Loads the contents of the memory addresses specified by general-use register Rm to floating point register FRn.
3. Stores the contents of floating point register FRm in the memory address position specified by general-use register Rm.
4. Loads the contents of the memory addresses specified by general-use register Rm to floating point register FRn. After the load completes successfully, increments the value of Rm by 4.
5. Stores the contents of floating point register FRm in the memory address position specified by general-use register Rn-4. After the store completes successfully, the decremented value (Rn-4) becomes the value of Rm.
6. Loads the contents of the memory addresses specified by general-use registers Rm and R0 to floating point register FRn.
7. Stores the contents of floating point register FRm in the memory address position specified by general-use registers Rn and R0.

Operation:

```
FMOV(float *FRm,*FRn)       /* FMOV.S FRm,FRn */
{
           *FRn = *FRm;
           pc += 2;
}
FMOV_LOAD(long *Rm,float *FRn)                /* FMOV @Rm,FRn */
{           if(load_long(Rm,FRn)  !=Address_Error)
            load_long(Rm,FRn);
            pc += 2;
}
FMOV_STORE(float *FRm,long *Rn)               /* FMOV.S FRm,@Rn */
{           if(store_long(FRm,tmp_address)  !=Address_Error)
            store_long(FRm,Rn);
            pc += 2;
}
FMOV_RESTORE(long *Rm,float *FRn)             /* FMOV.S @Rm+,FRn */
{           if(load_long(Rm,FRn)  !=Address_Error)
            *Rm += 4;
            pc += 2;
}
FMOV_SAVE(float *FRm,long *Rn)                /*FMOV.S FRm,@-Rn */
{
long        *tmp_address =*Rn -4;
            if(store_long(FRm,tmp_address)  !=Address_Error)
            Rn = tmp_address;
            pc += 2;
}
FMOV_LOAD_index(long *Rm, long *R0, float *FRn)/* FMOV.S @(R0,Rm),FRn*/
{
            if (load_long(&(*Rm+*R0),FRn),  ! = Address_Error);
            pc += 2;
}
FMOV_STORE_index(float *FRm,long *R0, long *Rn)/* FMOV.S FRm,@(R0,Rn)*/
```

```
{
        if (store_long(FRm,&((*Rn+*R0)),  ! = Address_Error);
        pc += 2;
}
```

Exceptions: Address error

Examples:

```
FMOV.S   @R1, FR2          ;Load
                           ;Before execution:     @R1=H'00ABCDEF
                           ;                      FR2=0
                           ;After execution:      @R1=H'00ABCDEF
                           ;                      FR2=H'00ABCDEF

FMOV.S   FR2, @R3          ;Store
                           ;Before execution:     @R3=0
                           ;                      FR2=H'40800000
                           ;After execution:      @R3=H'40800000
                           ;                      FR2=H'40800000

FMOV.S   @R3+,FR3          ;Restore
                           ;Before execution:     R3=H'0C700028
                           ;                      @R3=H'40800000
                           ;                      FR3=0
                           ;After execution:      R3=H'0C70002C
                           ;                      FR3=H'40800000

FMOV.S   FR4, @-R3         ;Save
                           ;Before execution:     R3=H'0C700044
                           ;                      @R3=0
                           ;                      FR4=H'01234567
                           ;After execution:      R3=H'0C700040
                           ;                      @R3=H'01234567
                           ;                      FR4=H'01234567

FMOV.S   @(R0, R3), FR4    ;Load with index
                           ;Before execution:     R0=H'00000004
                           ;                      R3=H'0C700040
```

```
                                ;                    @H'0C700044=H'00ABCDEF
                                ;                    FR=4
                                ;After execution:    R0=H'00000004
                                ;                    R3=H'0C700040
                                ;                    FR4=H'00ABCDEF

FMOV.S   FR5, @(R0,R3)          ;Store with index
                                ;Before execution:   R0=H'00000028
                                ;                    R3=H'0C700040
                                ;                    @H'0C700068=0
                                ;                    FR5=H'76543210
                                ;After execution:    R0=H'00000028
                                ;                    R3=H'0C700040
                                ;                    @H'0C700068=H'76543210

FMOV.S   FR5, FR6               ;Register file contents
                                ;Before execution:   FR5=H'76543210
                                ;                    FR6=x(don't care)
                                ;After execution:    FR5=H'76543210
                                ;                    FR6=H'76543210
```

8.3.11 FMUL (Floating Point Multiply): Floating Point Instruction

Format	Abstract	Code	Latency	Cycles	T Bit
FMUL FRm,FRn	FRn × FRm → FRn	1111nnnnmmmm0010	2	1	—

Description: Arithmetically multiplies (as floating point numbers) the contents of floating point registers FRm and FRn. The calculation result is stored in FRn.

Operation:

```
FMUL(float *FRm,*FRn) /* FMUL FRm,FRn */
{
           clear_cause_VZ();
           if((data_type_of(FRm) = = sNaN)       ||
                      (data_type_of(FRn) = = sNaN))    invalid(FRn);
           else if((data_type_of(FRm) = = qNaN) ||
                      (data_type_of(FRn) = = qNaN))    qnan(FRn);
else case(data_type_of(FRm)  {
           NORM       :
           case(data_type_of(FRn))  {
                    PINF   :
                    NINF   : inf(FRn,sign_of(FRm)^sign_of(FRn)); break;
                    default: *FRn=(*FRn)*(*FRm);                 break;
           }                                                     break;
           PZERO    :
           NZERO    :
           case(data_type_of(FRn))  {
                    PINF   :
                    NINF   : invalid(FRn);                       break;
                    default: zero(FRn,sign_of(FRm)^sign_of(FRn)); break;
           }                                                     break;
           PINF     :
           NINF     :
           case(data_type_of(FRn))  {
                    PZERO  :
                    NZERO  : invalid(FRn);                       break;
                    default:inf (FRn,sign_of(FRm)^sign_of(FRn)); break
           }                                                     break;
  }
```

```
    pc += 2;
  }
```

FMUL Special Cases

<table>
<tr><th rowspan="2">FRm</th><th colspan="7">FRn</th></tr>
<tr><th>NORM</th><th>+0</th><th>–0</th><th>+INF</th><th>–INF</th><th>qNaN</th><th>sNaN</th></tr>
<tr><td>NORM</td><td>MUL</td><td colspan="2">0</td><td colspan="2">INF</td><td rowspan="5"></td><td rowspan="6"></td></tr>
<tr><td>+0</td><td rowspan="2">0</td><td>+0</td><td>–0</td><td colspan="2" rowspan="2">Invalid</td></tr>
<tr><td>–0</td><td>–0</td><td>+0</td></tr>
<tr><td>+INF</td><td rowspan="2">INF</td><td colspan="2" rowspan="2">Invalid</td><td>+INF</td><td>–INF</td></tr>
<tr><td>–INF</td><td>–INF</td><td>+INF</td></tr>
<tr><td>qNaN</td><td colspan="6">qNaN</td></tr>
<tr><td>sNaN</td><td colspan="7">Invalid</td></tr>
</table>

Note: Non-normalized values are treated as zero.

Exceptions: Invalid operation

Examples:

```
FMUL    FR2, FR3      ;Floating point multiply
                      ;Before execution:   FR2=H'40000000/*2 in base 10*/
                      ;                    FR3=H'40800000/*4 in base 10*/
                      ;After execution:    FR2=H'40000000
                      ;                    FR3=H'41000000/*8 in base 10*/
```

8.3.12 FNEG (Floating Point Negate): Floating Point Instruction

Format	Abstract	Code	Latency	Cycles	T Bit
FNEG FRn	-FRn → FRn	1111nnnn01001101	2	1	—

Description: Arithmetically negates (as a floating point number) the contents of floating point register FRn. The calculation result is stored in FRn.

Operation:

```
FNEG(float *FRn)        /* FNEG FRn */
{
           clear_cause_VZ();
           case(data_type_of(FRn))       {
                         qNaN    :       qnan(FRn);        break;
                         sNaN    :       invalid(FRn);     break;
                         default :       *FRn = -(*Frn);   break;
  }
  pc += 2;
}
```

FNEG Special Cases

FRn	NORM	+0	–0	+INF	–INF	qNaN	sNaN
FNEG(FRn)	NEG	–0	+0	–INF	+INF	qNaN	Invalid

Note: Non-normalized values are treated as zero.

Exceptions: Invalid operation

Examples:

```
FNEG    FR2          ;Floating point negate
                     ;Before execution:     FR2=H'40800000/*4 in base 10*/
                     ;After execution:      FR2=H'C0800000/*–4 in base 10*/
```

8.3.13 FSQRT (Floating Point Square Root): Floating Point Instruction

Format	Abstract	Code	Latency	Cycles	T Bit
FSQRT FRn	$\sqrt{FRn} \rightarrow FRn$	1111nnnn01101101	14	13	—

Description: Arithmetically obtains (as a floating point number) the square root of the contents of floating point register FRn. The calculation result is stored in FRn.

Operation:

```
FSQRT(float *FRn)     /* FSQRT FRn */
{
           clear_cause_VZ();
           case(data_type_of(FRn))  {
           NORM      :  if(sign_of(FRn) = = 0)
                               *FRn = sqrt(*FRn);
                            else invalid(FRn);                      break;
           PZERO     :
           NZERO     :
           PINF      :         *FRn = *FRn;                         break;
           NINF      :         invalid(FRn);                        break;
           qNaN      :         qnan(FRn);                           break;
           sNaN      :         invalid(FRn);                        break;
           }
           pc += 2;
}
```

FSQRT Special Cases

FRn	+NORM	–NORM	+0	–0	+INF	–INF	qNaN	sNaN
FSQRT(FRn)	SQRT	Invalid	+0	–0	+INF	Invalid	qNaN	Invalid

Note: Non-normalized values are treated as zero.

Exceptions: Invalid operation

Examples:

```
FSQRT    FR4        ;Floating point square root
                    ;Before execution:      ;FR4=H'40400000/*3 in base 10*/
                    ;After execution:       ;FR4=H'3FDDB3D7/*1.7320 in base 10*/
```

8.3.14 FSTS (Floating Point Store From System Register): Floating Point Instruction

Format	Abstract	Code	Latency	Cycles	T Bit
FSTS FPUL,FRn	FPUL → FRn	1111nnnn00001101	2	1	—

Description: Copies the contents of system register FPUL to floating point register FRn.

Operation:

```
FSTS(float *FRn,*FPUL)        /* FSTS FPUL,FRn */
{
           *FRn = *FPUL;
           pc += 2;
}
```

Exceptions: None

Examples:

MOV.L	#H'00000002, R2	;Before execution of FSTS instruction:	;R2=H'00000002
FLDI0	FR5		;FR5=0
LDS	R2,FPUL	;After execution of FSTS instruction:	;R2=H'00000002
FSTS	FPUL, R5		;FR5= H'00000002

8.3.15 FSUB (Floating Point Subtract): Floating Point Instruction

Format	Abstract	Code	Latency	Cycles	T Bit
FSUB FRm, FRn	FRn-FRm → FRn	1111nnnnmmmm0001	2	1	—

Description: Arithmetically subtracts (as floating point numbers) the contents of floating point register FRm from contents of floating point register FRn. The calculation result is stored in FRn.

Operation:

```
FSUB(float *FRm,FRn)                           /* FSUB FRm,FRn */
{
      clear_cause_VZ();
      if((data_type_of(FRm) = = sNaN)      | |
       (data_type_of(FRn) = = sNaN))                 invalid(FRn);
      else if((data_type_of(FRm) = = qNaN) | |
       (data_type_of(FRn) = = qNaN))                 qnan(FRn);
  else case(data_type_of(FRm))             {
      NORM      :
    case(data_tyoe_of(FRn))  {
      PINF      :            inf(FRn,0);             break;
      NINF      :            inf(FRn,1);             break;
      default   :            *FRn = *FRn - *FRm;     break;
    }                                                break;
    PZERO       :
    case(data_type_of(FRn))  {
      NORM      :            *FRn = *FRn- *FRm;      break;
      PZERO     :            zero(FRn,0);            break;
      NZERO     :            zero(FRn,1);            break;
      PINF      :            inf(FRn,0);             break;
      NINF      :            inf(FRn,1);             break;
    }                                                break;
    NZERO       :
    case(data_type_of(FRn))  {
      NORM      :            *FRn = *FRn - *FRm; break;
      PZERO     :
      NZERO     :            zero(FRn,0);            break;
      PINF      :            inf(FRn,0);             break;
```

```
      NINF      :           inf(FRn,1);            break;
    }                                              break;
    PINF        :
    case(data_type_of(FRn))  {
      NINF      :           invalid(FRn);          break;
      default   :           inf(FRn,1);            break;
    }                                              break;
      NINF      :
    case(data_type_of(FRn))  {
      PINF      :           invalid(FRn);          break;
      default   :           inf(FRn,0);            break;
    }                                              break;
    }
    pc += 2;
}
```

FSUB Special Cases

FRm	FRn						
	NORM	+0	–0	+INF	–INF	qNaN	sNaN
NORM	SUB			+INF	–INF		
+0			–0				
–0		+0					
+INF	–INF			Invalid			
–INF	+INF				Invalid		
qNaN						qNaN	
sNaN							Invalid

Note: Non-normalized values are treated as zero.

Exceptions: Invalid operation

Examples:

```
FSUB    FR0, FR3    ;Floating point subtract
                    ;Before execution:    ;FR0=H'3F800000/*1 in base 10*/
                    ;                     ;FR3=H'40E00000/*7 in base 10*/
                    ;After execution:     ;FR0=H'3F800000/*1 in base 10*/
                    ;                     ;FR3=H'40C00000/*6 in base 10*/

FSUB    FR3, FR2    ;
                    ;Before execution:    ;FR2=H'40800000/*4 in base 10*/
                    ;                     ;FR3=H'40C00000/*6 in base 10*/
                    ;After execution:     ;FR2=H'C0000000/*–2 in base 10*/
                    ;                     ;FR3=H'40C00000/*6 in base 10*/
```

8.3.16 FTRC (Floating Point Truncate And Convert To Integer): Floating Point Instruction

Format	Abstract	Code	Latency	Cycles	T Bit
FTRC FRm, FPUL	(long)FRm → FPUL	1111nnnn00111101	2	1	—

Description: Interprets the contents of floating point register FRm as a floating point number and converts it to an integer by truncating everything after the decimal point. The calculation result is stored in FRn.

Operation:

```
#define N_INT_RANGE 0xCF000000                    /* 01.000000 * 2^16 */
#define P_INT_RANGE 0x47FFFFFF                    /* 1.fffffe * 2^30 */

FTRC(float *FRm,int *FPUL)                        /* FTRC FRm,FPUL */
{
       clear_cause_VZ();
       case(ftrc_type_of(FRm))  {
       NORM      :      *FPUL = (long)(*FRm);   break;
       PINF      :      ftrc_invalid(0);        break;
       NINF      :      ftrc_invalid(1);        break;
  }
  pc += 2;
}
int ftrc_type_of(long *src)
{
long abs;
       abs = *src & 0x7FFFFFF;
  if(sign_of(src) = = 0)       {
        if(abs > 0x7F800000)   return(NINF);    /* NaN*/
   else if(abs > P_INT_RANGE)  return(PINF);    /* out of range,+INF  */
   else                        return(NORM);    /* +0,+NORM    */
   }
  else  {
        if(*src > N_INT_RANGE) return(NINF);/* out of range ,+INF,NaN*/
        else                   return(NORM);    /* -0,-NORM*/
   }
}
```

```
ftrc_invalid(long *dest,int sign)
{
  set_V();
  if((FPSCR & ENABLE_V) = = 0) {
       if(sign = = 0)    *dest = 0x7FFFFFFF;
       else              *dest = 0x80000000;
   }
}
```

FTRC Special Cases

FRn	NORM	+0	–0	positive out of range	negative out of rarge	+INF	-INF	qNaN	sNaN
FTRC (FRn)	TRC	0	0	7FFFFFFF	80000000	Invalid +MAX Invalid	–MAX Invalid	–MAX Invalid	–MAX Invalid

Note: Non-normalized values are treated as zero.

Exceptions: Invalid operation

Examples:

```
MOV.L   #H'402ED9EB, R2
LDS     R2, FPUL
FSTS    FPUL, FR6               ;FR6=H'402ED9EB/*2.7320 in base 10*/
FTRC    FR6, FPUL
STS     FPUL, R2                ;R2=H'00000002/*2 in base 10*/
                                ;Before execution of FTRC and STS:
                                ;     R2=H'402ED9EB
                                ;     FR6=H'402ED9EB
                                ;After execution of FTRC and STS:
                                ;     R2=H'00000002
                                ;     FR6=H'402ED9EB
```

8.3.17 LDS (Load to System Register): FPU Related CPU Instruction

Format	Abstract	Code	Latency	Cycles	T Bit
1.LDS Rm, FPUL	Rm → FPUL	0100nnnn01011010	2	1	—
2.LDS.L@Rm+,FPUL	(Rm) → FPUL, Rm+=4	0100nnnn01010110	2	1	—
3.LDS Rm,FPSCR	Rm → FPSCR	0100nnnn01101010	3	1	—
4.LDS.L @Rm+,FPSCR	(Rm) → FPSCR, Rm+=4	0100nnnn01100110	3	1	—

Description:

1. Moves the contents of general-use register Rm to system register FPUL.
2. Loads the contents of the memory addresses specified by general-use register Rm to system register FPUL. After the load completes successfully, increments the value of Rm by 4.
3. Moves the contents of general-use register Rm to system register FPSCR. Previously defined bits in FPSCR are not changed.
4. Loads the contents of the memory addresses specified by general-use register Rm to system register FPSCR. After the load completes successfully, increments the value of Rm by 4. Previously defined bits in FPSCR are not changed.

Operation:

```
#define FPSCR_MASK 0x00018C60
LDS(long *Rm,*FPUL)                                /* LDS Rm,FPUL */
{
            *FPUL = *Rm;
            pc += 2;
}
LDS_RESTORE(long *Rm, *FPUL)                       /* LDS.L @Rm+,FPUL */
{
           if(load_long(Rm,FPUL) != Address_Error) *Rm += 4 ;
           pc += 2;
}

LDS(long *Rm,*FPSCR)                               /* LDS Rm,FPSCR */
{
           *FPSCR = *Rm & FPSCR_MASK;
           pc += 2;
```

```
}
LDS_RESTORE(long *Rm, *FPSCR)                  /* LDS.L @Rm+,FPSCR */
{
          long *tmp_FPSCR;
          if(load_long(Rm, tmp_FPSCR) != Address_Error){
                           *FPSCR =*tmp_FPSCR & FPSCR_MASK;
                           *Rm += 4 ;
          }
          pc += 2;
}
```

Exceptions: Address error

Examples:

• LDS

Example 1

```
MOV.L    #H'12345678, R2          ;Before execution of LDS and FSTS instructions:
                                  ;              R2=H'12345678
FLDI0    FR3                      ;              FR3=0
LDS      R2, FPUL                 ;After execution of LDS and FSTS instructions:
                                  ;              R2=H'12345678
FSTS     FPUL, FR3                ;              FR3= H'12345678
```

Example 2

```
MOV.L    #H'00040801, R4          ;After execution of LDS instruction:
LDS      R4, FPSCR                ;FPSCR=00040801
```

• LDS.L

Example 1

```
LDI0     FR0                      ;Before execution of LDS.L and FSTS instructions:
MOV.L    #H'87654321, R4          ;              FR0=0
MOV.L    #H'0C700128, R8          ;              R8=0C700128
MOV.L    R4,@R8                   ;After execution of LDS.L and FSTS instructions:
LDS.L    @R8+, FPUL               ;              FR0=87654321
FSTS     FPUL, FR0                ;              R8=0C70012C
```

Example 2

```
MOV.L   #H'00040C01, R4     ;Before execution of LDS.L instruction:
MOV.L   #H'0C700134, R8     ;                    R8=0C700134
MOV.L   R4,@R8              ;After execution of LDS.L instruction:
                            ;                    R8=0C700138
LDS.L   @R8+, FPSCR         ;                    FPSCR=00040C01
```

8.3.18 STS (Store from FPU System Register): FPU Related CPU Instruction

Format	Abstract	Code	Latency (Wait Time)	Cycles	T Bit
1.STS FPUL,Rn	FPUL → Rn	0000nnnn01011010	2	1	—
2.STS.L FPUL,@-Rn	Rn -= 4, FPUL → @(Rn)	0100nnnn01010010	2	1	—
3.STS FPSCR,Rn	FPSCR → Rn	0000nnnn01101010	3	1	—
4.STS.L FPSCR,@-Rn	Rn -= 4, FPSCR → @(Rn)	0100nnnn01100010	3	1	—

Description:

1. Moves the contents of system register FPUL to general-use register Rn.
2. Stores contents of system register FPUL at the memory address position specified by general-use register Rn-4. After the store completes successfully, the decremented value becomes the value of Rn.
3. Moves the contents of system register FPSCR to general-use register Rn.
4. Stores contents of system register FPSCR at the memory address position specified by general-use register Rn-4. After the store completes successfully, the decremented value becomes the value of Rn.

Operation:

```
STS(long *FPUL,*Rn)                              /* STS.L FPUL,Rn */
{
           *Rn = *FPUL;
           pc += 2;
}
STS_SAVE(long *FPUL,*Rn)                         /* STS.L FPUL,@-Rn */
{
long *tmp_address = *Rn - 4;
           if(store_long(FPUL,tmp_address) != Address_Error)
           Rn = tmp_address;
           pc += 2;
}
STS(long *FPSCR,*Rn)                              /* STS FPSCR,Rn */
{
           *Rn = *FPSCR;
```

```
            pc += 2;
}

STS STore from FPU System register

STS_RESTORE long *FPSCR,*Rn)                        /* STS.L FPSCR,@-Rn */
{
long *tmp_address = *Rn - 4;
            if(store_long(FPSCR tmp_address) != Address_Error)
            Rn = tmp_address
            pc += 2;
}
```

Exceptions: Address error

Examples:

- STS

Example 1

```
MOV.L   #H'12ABCDEF, R12
LDS.L   @R12, FPUL
STS     FPUL, R13
                                ;After execution of STS instruction:
                                ;      R13 = 12ABCDEF
```

Example 2

```
STS     FPSCR, R2
                                ;After execution of STS instruction:
                                ;      Contents of FPSCR at that point stored in R2 register
```

- STS.L

Example 1

```
MOV.L    #H'0C700148, R7
STS      FPUL, @-R7
                                ;Before execution of STS.L instruction:
                                ;      R7 = H'0C700148
                                ;After execution of STS.L instruction:
```

```
                        ;       R7 = H'0C700144, contents of FPUL saved at
                                address H'0C700144
                        ;       location H'0C700144
```

Example 2

```
MOV.L    #H'0C700154, R8
STS.L    FPSCR, @-R8
                        ;After execution of STS.L instruction:
                        ;       Contents of FPSCR saved at address H'0C700150
```

8.4 DSP Data Transfer Instructions (SH3-DSP Only)

Table 8-1 lists the DSP data transfer instructions in alphabetical order.

Table 8-1 DSP Data Transfer Instructions in Alphabetical Order

Instruction	Operation	Code	Cycles	DC Bit
MOVS.L @-As,Ds	As–4→As,(As)→Ds	111101AADDDD0010	1	—
MOVS.L @As,Ds	(As)→Ds	111101AADDDD0110	1	—
MOVS.L @As+,Ds	(As)→Ds,As+4→As	111101AADDDD1010	1	—
MOVS.L @As+Ix,Ds	(As)→Ds,As+Ix→As	111101AADDDD1110	1	—
MOVS.L Ds, @-As	As–4→As,Ds→(As)	111101AADDDD0011	1	—
MOVS.L Ds,@As	Ds→(As)	111101AADDDD0111	1	—
MOVS.L Ds,@As+	Ds→(As),As+4→As	111101AADDDD1011	1	—
MOVS.L Ds,@As+Ix	Ds→(As),As+Ix→As	111101AADDDD1111	1	—
MOVS.W @-As,Ds	As–2→As,(As)→MSW of Ds,0→LSW of Ds	111101AADDDD0000	1	—
MOVS.W @As,Ds	(As)→MSW of Ds,0→LSW of Ds	111101AADDDD0100	1	—
MOVS.W @As+,Ds	(As)→MSW of Ds,0→LSW of Ds, As+2→As	111101AADDDD1000	1	—
MOVS.W @As+Ix,Ds	(As)→MSW of Ds,0→LSW of Ds, As+Ix→As	111101AADDDD1100	1	—
MOVS.W Ds,@-As	As–2→As,MSW of Ds→(As)	111101AADDDD0001	1	—
MOVS.W Ds,@As	MSW of Ds→(As)	111101AADDDD0101	1	—
MOVS.W Ds,@As+	MSW of Ds→(As),As+2→As	111101AADDDD1001	1	—
MOVS.W Ds,@As+Ix	MSW of Ds→(As),As+Ix→As	111101AADDDD1101	1	—
MOVX.W @Ax,Dx	(Ax)→MSW of Dx,0→LSW of Dx	111100A*D*0*01**	1	—
MOVX.W @Ax+,Dx	(Ax)→MSW of Dx,0→LSW of Dx,Ax+2→Ax	111100A*D*0*10**	1	—

Table 8-1 DSP Data Transfer Instructions in Alphabetical Order (cont)

Instruction	Operation	Code	Cycles	DC Bit
MOVX.W @Ax+Ix,Dx	(Ax)→MSW of Dx,0→LSW of Dx,Ax+Ix→Ax	111100A*D*0*11**	1	—
MOVX.W Da,@Ax	MSW of Da→(Ax)	111100A*D*1*01**	1	—
MOVX.W Da,@Ax+	MSW of Da→(Ax),Ax+2→Ax	111100A*D*1*10**	1	—
MOVX.W Da,@Ax+Ix	MSW of Da→(Ax),Ax+Ix→Ax	111100A*D*1*11**	1	—
MOVY.W @Ay,Dy	(Ay)→MSW of Dy,0→LSW of Dy	111100*A*D*0**01	1	—
MOVY.W @Ay+,Dy	(Ay)→MSW of Dy,0→LSW of Dy, Ay+2→Ay	111100*A*D*0**10	1	—
MOVY.W @Ay+Iy,Dy	(Ay)→MSW of Dy,0→LSW of Dy, Ay+Iy→Ay	111100*A*D*0**11	1	—
MOVY.W Da,@Ay	MSW of Da→(Ay)	111100*A*D*1**01	1	—
MOVY.W Da,@Ay+	MSW of Da→(Ay),Ay+2→Ay	111100*A*D*1**10	1	—
MOVY.W Da,@Ay+Iy	MSW of Da→(Ay),Ay+Iy→Ay	111100*A*D*1**11	1	—
NOPX	No Operation	1111000*0*0*00**	1	—
NOPY	No Operation	111100*0*0*0**00	1	—

Note: MSW = High-order word of operand
LSW = Low-order word of operand

X and Y Data Transfers (MOVX.W and MOVY.W)

These instructions use the XDB and YDB buses to access X and Y memory. Areas other than X and Y memory cannot be accessed. Memory is accessed in word units. Since independent bus is used, it does not create access contention with instruction fetches (using the LDB bus).

X and Y data transfer instructions are executed regardless of conditions even when the data operation instruction executed in parallel has conditions.

Figure 8-17 shows the load and store operations in X and Y data transfers.

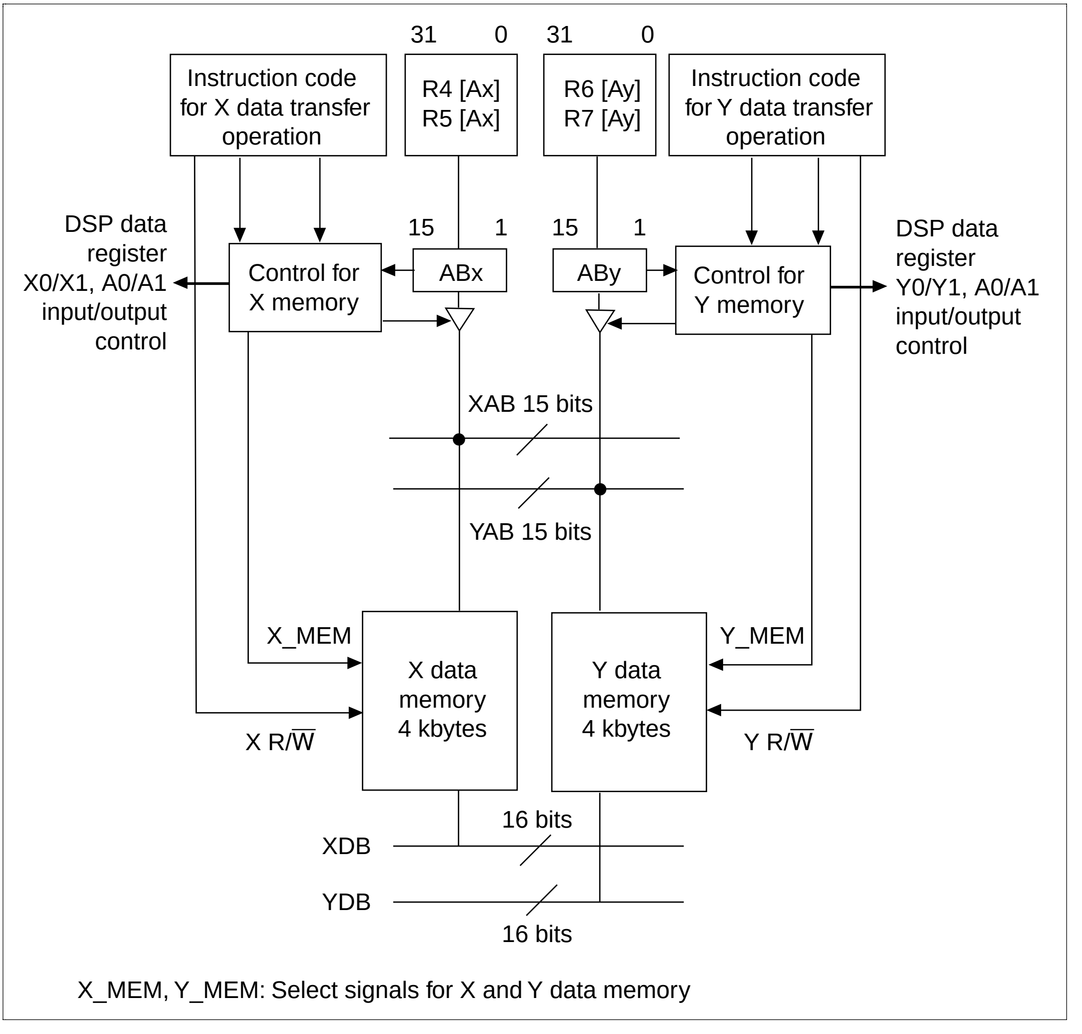


Figure 8-17 Load and Store Operations in X and Y Data Transfers

X memory data transfer operation is shown below. Y memory data transfers are the same.

```
if ( !NOP ) {
    X_MEM=1; XAB=ABx; X R/W=1;
    if ( load operation ) {
        DX[31:16]=XDB;
        DX[15:0] =0x0000; /* Dx is X0 or X1 */
    }
    else {XDB=Dx[31:16];X R/W=0;}  /* Dx is A0 or A1 */
}
else { X_MEM=0; XAB=Unknown; }
```

Single Data Transfers (MOVS.W and MOVS.L)

Single data transfers are instructions that load to and store from the DSP register. They are like system register load and store instructions. Data transfers between the DSP register and memory use the LAB and LDB buses. Like CPU core instructions, data accesses can create access contention with instruction memory accesses.

Single data transfers can use either word or longword data. Figure 8-18 shows the load and store operations in single data transfers.

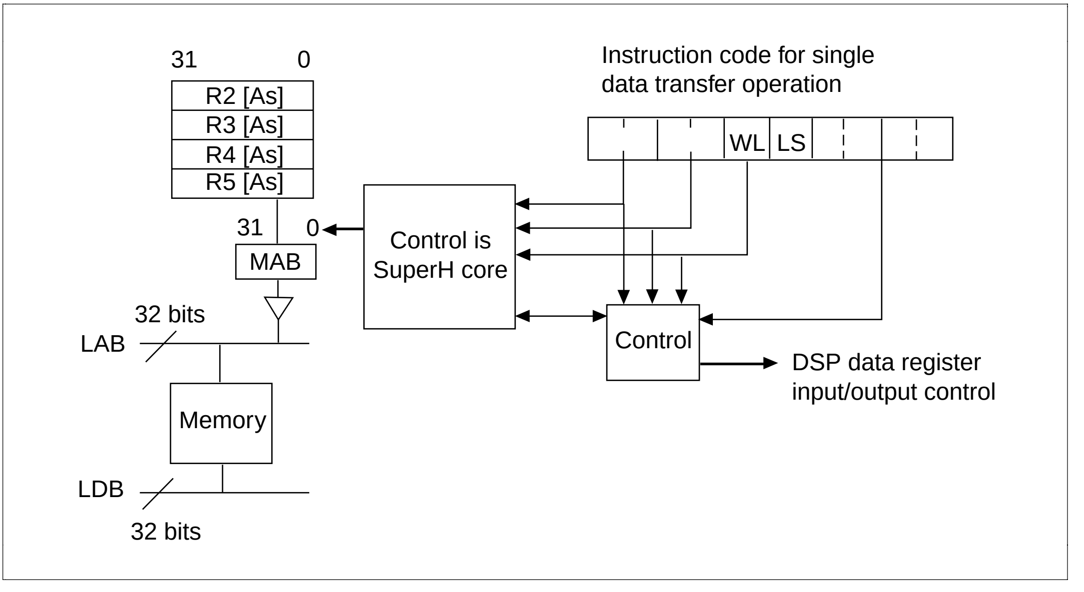


Figure 8-18 Load and Store Operations in Single Data Transfers

Load and store operations in single data transfers are shown below.

```
LAB = MAB;
if ( Ms!=NLS @@ W/L is word access {/* MOVS.W */
    if (LS==load) {
        if (DS!=A0G @@ Ds!=A1G){
                Ds[31:16] = LDB[15:0]; Ds[15:0] = 0x0000;
                if (Ds==A0) A0G[7:0] = LDB[15];
                if (Ds==A1) A1G[7:0] = LDB[15];
        }
        else Ds[7:0] = LDB[7:0]            /* Ds is A0G or A1G */
    }
    else { /* Store */
            if (DS!=A0G @@ Ds!=A1G) LDB[15:0] = Ds[31:16];
            /* Ds is A0G or A1G */
            else LDB[15:0] = Ds[7:0] with 8-bit sign extension
    }
}
else    if ( MA!=NLS @@ W/L is longword access ) { /* MOVS.L */
        if (LS==load {
                if (Ds!=A0G @@ Ds!=A1G) {
                Ds[31:0] = LDB[31:0];
                if (Ds==A0) A0G[7:0] = LDB[31];
                if (Ds==A1) A1G[7:0] = LDB[31];
        }
        else Ds[7:0] = LDB[7:0]            /* Ds is A0G or A1G */
    }
    else { /* Store */
            if (DS!=A0G @@ Ds!=A1G) LDB[31:0] = Ds[31:0]
            /* Ds is A0G or A1G */
            else LDB[31:0] = Ds[7:0] with 24-bit sign extension
    }
}
```

This section explains the breakdown of instructions, descriptions, etc. given in the rest of this section.

Table 8-2 Sample Description (Name): Classification

Format	Abstract	Code	Cycle	DC Bit
Assembler input format.	A brief description of operation	Displayed in order MSB ↔ LSB	All DSP instructions execute in 1 cycle	The status of the DC bit after the instruction is executed

Format:

[if cc] OP.Sz SRC1,SRC2,DEST

[if cc]: Condition (unconditional, DCT, or DCF)
OP: Operation code
Sz: Size
SRC1: Source 1 operand
SRC2: Source 2 operand
DEST: Destination

Table 8-3 Operation Summary

<table>
<tr><th>Operation</th><th>Description</th></tr>
<tr><td>→, ←</td><td>Direction of transfer</td></tr>
<tr><td>(xx)</td><td>Memory operand</td></tr>
<tr><td>DC</td><td>Flag bits in the DSR</td></tr>
<tr><td>&</td><td>Logical AND of each bit</td></tr>
<tr><td>|</td><td>Logical OR of each bit</td></tr>
<tr><td>^</td><td>Exclusive OR of each bit</td></tr>
<tr><td>~</td><td>Logical NOT of each bit</td></tr>
<tr><td><<n, >>n</td><td>n-bit shift</td></tr>
<tr><td>MSW</td><td>Most significant word (bits 16-31)</td></tr>
<tr><td>LSW</td><td>Least significant word (bits 0-15)</td></tr>
<tr><td>[n1:n2]</td><td>Bits n1 to n2</td></tr>
</table>

Instruction Code: Shows the source register and destination register.

X Data Transfer Instructions:

A(Ax): 0=R4, 1=R5
D(destination, Dx): 0=X0, 1=X1
D (source, Da): 0=A0, 1=A1

Y Data Transfer Instructions:

A(Ay): 0=R6, 1=R7
D(destination, Dy): 0=Y0, 1=Y1
D (source, Da): 0=A0, 1=A1

Single Data Transfer Instructions:

AA(As): 0=R4, 1=R5, 2=R2, 3=R3
DDDD(Ds): 5=A1, 7=A0, 8=X0, 9=X1, A=Y0, B=Y1, C=M0, D=A1G, E=M1
F=A0G

DSP Operation Instructions:

iiiiiii(imm): –32 to +32
ee(Se): 0=X0, 1=X1, 2=Y0, 3=A1
ff(Sf): 0=Y0, 1=Y1, 2=X0, 3=A1
xx(Sx): 0=X0, 1=X1, 2=A0, 3=A1
yy(Sy): 0=Y0, 1=Y1, 2=M0, 3=M1
gg(Dg): 0=M0, 1=M1, 2=A0, 3=A1
uu(Du): 0=X0, 1=Y0, 2=A0, 3=A1
zzzz(Dz): 5=A1, 7=A0, 8=X0, 9=X1, A=Y0, B=Y1, C=M0, E=M1

DC Bit:

Update: Updated according to the operation result and the specifications of the CS (condition select) bits.
—: Not updated.

Description: Description of operation

Notes: Notes on using the instruction

Operation: Operation written in C language.

Examples: Examples are written in assembler mnemonics and describe status before and after executing the instruction.

8.4.1 MOVS (Move Single Data between Memory and DSP Register): DSP Data Transfer Instruction

Format	Abstract	Code	Cycle	DC Bit
MOVS.W @-As,Ds	As–2→As,(As)→MSW of Ds,0→LSW of Ds	111101AADDDD0000	1	—
MOVS.W @As,Ds	(As)→MSW of Ds,0→LSW of Ds	111101AADDDD0100	1	—
MOVS.W @As+,Ds	(As)→MSW of Ds,0→LSW of Ds, As+2→As	111101AADDDD1000	1	—
MOVS.W @As+Ix,Ds	(As)→MSW of Ds,0→LSW of Ds, As+Ix→As	111101AADDDD1100	1	—
MOVS.W Ds,@-As	As–2→As,MSW of Ds→(As)	111101AADDDD0001	1	—
MOVS.W Ds,@As	MSW of Ds→(As)	111101AADDDD0101	1	—
MOVS.W Ds,@As+	MSW of Ds→(As),As+2→As	111101AADDDD1001	1	—
MOVS.W Ds,@As+Ix	MSW of Ds→(As),As+Ix→As	111101AADDDD1101	1	—
MOVS.L @-As,Ds	As–4→As,(As)→Ds	111101AADDDD0010	1	—
MOVS.L @As,Ds	(As)→Ds	111101AADDDD0110	1	—
MOVS.L @As+,Ds	(As)→Ds,As+4→As	111101AADDDD1010	1	—
MOVS.L @As+Ix,Ds	(As)→Ds,As+Ix→As	111101AADDDD1110	1	—
MOVS.L Ds, @-As	As–4→As,Ds→(As)	111101AADDDD0011	1	—
MOVS.L Ds,@As	Ds→(As)	111101AADDDD0111	1	—
MOVS.L Ds,@As+	Ds→(As),As+4→As	111101AADDDD1011	1	—
MOVS.L Ds,@As+Ix	Ds→(As),As+Ix→As	111101AADDDD1111	1	—

Description: Transfers the source operand data to the destination. Transfer can be from memory to register or register to memory. The transferred data can be a word or longword. When a word is transferred, the source operand is in memory, and the destination operand is a register, the word data is loaded to the top word of the register and the bottom word is cleared with zeros. When the source operand is a register and the destination operand is memory, the top word of the register is stored as the word data . In a longword transfer, the longword data is transferred. When the destination operand is a register with guard bits, the sign is extended and stored in the guard bits.

Note: When one of the guard bit registers A0G and A1G is the source operand for store processing, the data is output to the bottom 8 bits (bits 0–7) and the top 24 bits (bits 31–8) become undefined.

Operation: See figure 8-19.

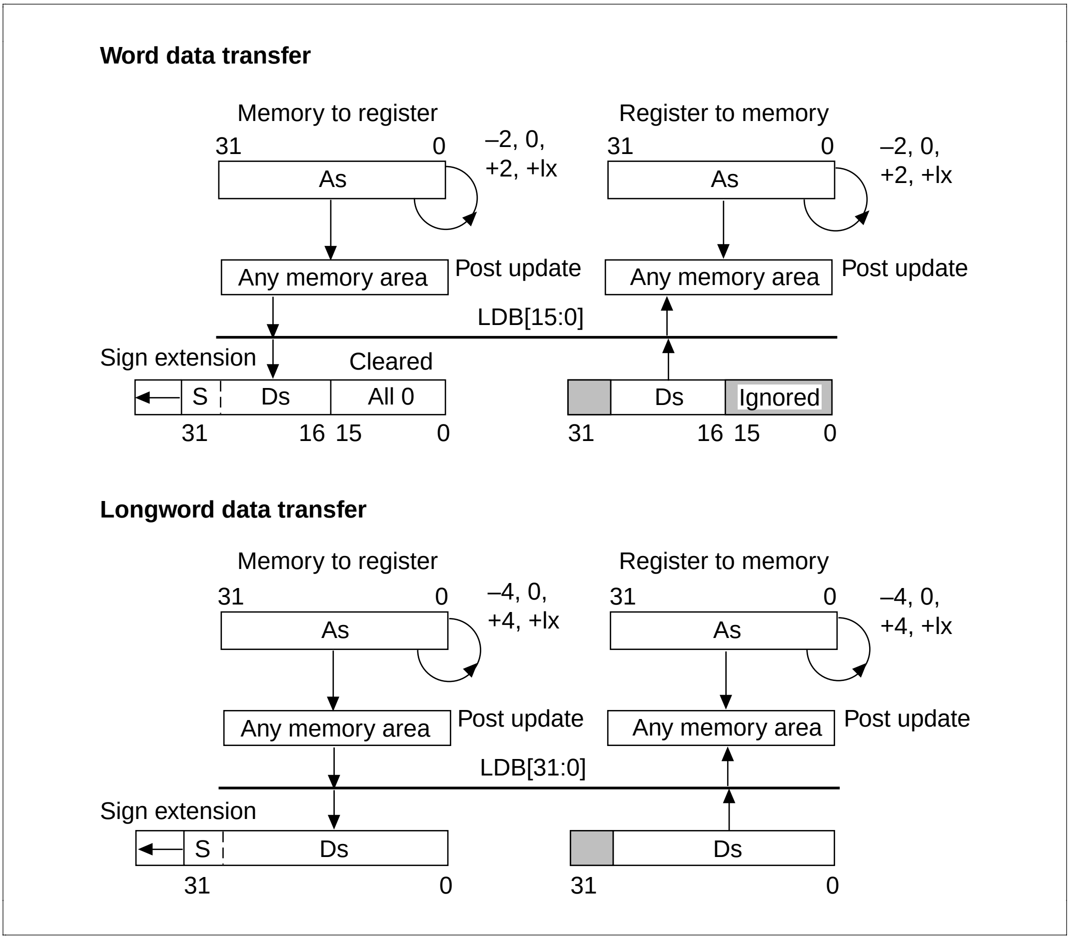


Figure 8-19 The MOVS Instruction

Examples:

`MOVS.W @R4+,A0`	;Before execution:	R4=H'00000400, @R4=H'8765, A0=H'123456789A
	;After execution:	R4=H'00000402, A0=H'FF87650000
`MOVS.L A1, @-R3`	;Before execution:	R3=H'00000800, A1=H'123456789A
	;After execution:	R3=H'000007FC, @(H'000007FC)=H'3456789A

8.4.2 MOVX (Move between X Memory and DSP Register): DSP Data Transfer Instruction

Format	Abstract	Code	Cycle	DC Bit
`MOVX.W @Ax,Dx`	(Ax)→MSW of Dx,0→LSW of Dx	`111100A*D*0*01**`	1	—
`MOVX.W @Ax+,Dx`	(Ax)→MSW of Dx,0→LSW of Dx,Ax+2→Ax	`111100A*D*0*10**`	1	—
`MOVX.W @Ax+Ix,Dx`	(Ax)→MSW of Dx,0→LSW of Dx,Ax+Ix→Ax	`111100A*D*0*11**`	1	—
`MOVX.W Da,@Ax`	MSW of Da→(Ax)	`111100A*D*1*01**`	1	—
`MOVX.W Da,@Ax+`	MSW of Da→(Ax),Ax+2→Ax	`111100A*D*1*10**`	1	—
`MOVX.W Da,@Ax+Ix`	MSW of Da→(Ax),Ax+Ix→Ax	`111100A*D*1*11**`	1	—

Note: "*" of the instruction code is MOVY instruction designation area.

Description: Transfers the source operand data to the destination operand. Transfer can be from memory to register or register to memory. The transferred data can only be word length for X memory. When the source operand is in memory, and the destination operand is a register, the word data is loaded to the top word of the register and the bottom word is cleared with zeros. When the source operand is a register and the destination operand is memory, the word data is stored in the top word of the register.

Operation: See figure 8-20.

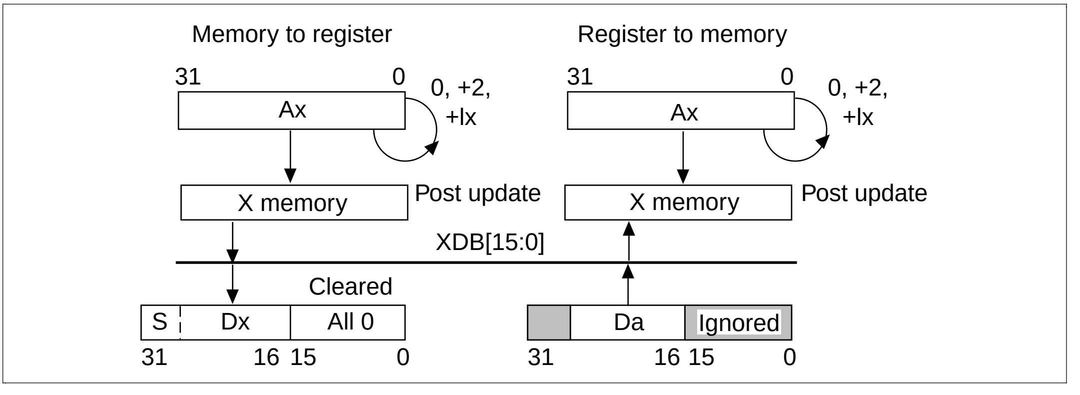


Figure 8-20 The MOVX Instruction

Examples:

`MOVX.W @R4+,X0` ;Before execution: R4=H'08010000, @R4=H'5555, X0=H'12345678

;After execution: R4=H'08010002, X0=H'55550000

8.4.3 MOVY (Move between Y Memory and DSP Register): DSP Data Transfer Instruction

Format	Abstract	Code	Cycle	DC Bit
MOVY.W @Ay,Dy	(Ay)→MSW of Dy,0→LSW of Dy	111100*A*D*0**01	1	—
MOVY.W @Ay+,Dy	(Ay)→MSW of Dy,0→LSW of Dy, Ay+2→Ay	111100*A*D*0**10	1	—
MOVY.W @Ay+Iy,Dy	(Ay)→MSW of Dy,0→LSW of Dy, Ay+Iy→Ay	111100*A*D*0**11	1	—
MOVY.W Da,@Ay	MSW of Da→(Ay)	111100*A*D*1**01	1	—
MOVY.W Da,@Ay+	MSW of Da→(Ay),Ay+2→Ay	111100*A*D*1**10	1	—
MOVY.W Da,@Ay+Iy	MSW of Da→(Ay),Ay+Iy→Ay	111100*A*D*1**11	1	—

Note: "*" of the instruction code is MOVX instruction designation area.

Description: Transfers the source operand data to the destination operand. Transfer can be from memory to register or register to memory. The transferred data can only be word length for Y memory. When the source operand is in memory, and the destination operand is a register, the word data is loaded to the top word of the register and the bottom word is cleared with zeros. When the source operand is a register and the destination operand is memory, the word data is stored in the top word of the register.

Operation:

See figure 8-21.

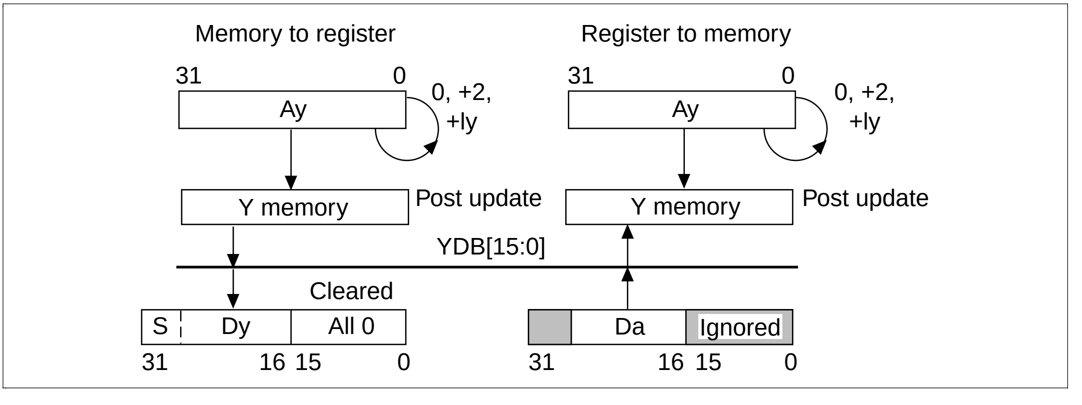


Figure 8-21 The MOVY Instruction

Examples:

`MOVY.W A0, @R6+,R9`	;Before execution:	R6=H'08020000, R9=H'00000006, A0=H'123456789A
	;After execution:	R6=H'08020006, @(H'08020000)=H'3456

8.4.4 NOPX (No Access Operation for X Memory): DSP Data Transfer Instruction

Format	Abstract	Code	Cycle	DC Bit
NOPX	No Operation	1111000*0*0*00**	1	—

Description: No access operation for X memory.

8.4.5 NOPY (No Access Operation for Y Memory): DSP Data Transfer Instruction

Format	Abstract	Code	Cycle	DC Bit
NOPY	No Operation	111100*0*0*0**00	1	—

Description: No access operation for Y memory.

8.5 DSP Operation Instructions

The DSP operation instructions are listed below in alphabetical order. See section 8.4, DSP Data Transfer Instructions: Classification, for an explanation of the format and symbols used in this description.

Table 8-4 Alphabetical Listing of DSP Operation Instructions

Instruction	Operation	Code	Cycles	DC Bit
`PABS Sx,Dz`	If Sx≥0, Sx→Dz If Sx<0, 0–Sx→Dz	111110********** 10001000xx00zzzz	1	Update
`PABS Sy,Dz`	If Sy≥0, Sy→Dz If Sy<0, 0–Sy→Dz	111110********** 1010100000yyzzzz	1	Update
`PADD Sx,Sy,Dz`	Sx + Sy→Dz	111110********** 10110001xxyyzzzz	1	Update
`DCT PADD Sx,Sy,Dz`	If DC = 1, Sx + Sy→Dz; if 0, nop	111110********** 10110010xxyyzzzz	1	—
`DCF PADD Sx,Sy,Dz`	If DC = 0, SX + Sy–Dz; if 1, nop	111110********** 10110011xxyyzzzz	1	—
`PADD Sx,Sy,Du` `PMULS Se,Sf,Dg`	Sx + Sy→Du; MSW of Se × MSW of Sf→Dg	111110********** 0111eeffxxyygguu	1	Update*
`PADDC Sx,Sy,Dz`	Sx + Sy + DC→Dz	111110********** 10110000xxyyzzzz	1	Update
`PAND Sx,Sy,Dz`	Sx & Sy→Dz; clear LSW of Dz	111110********** 10010101xxyyzzzz	1	Update
`DCT PAND Sx,Sy,Dz`	If DC = 1, SX & SY→Dz, clear LSW of Dz; if 0, nop	111110********** 10010110xxyyzzzz	1	—
`DCF PAND Sx,Sy,Dz`	If DC = 0, SX & SY→Dz, clear LSW of Dz; if 1, nop	111110********** 10010111xxyyzzzz	1	—
`PCLR Dz`	H'00000000→Dz	111110********** 100011010000zzzz	1	Update
`DCT PCLR Dz`	If DC = 1, H'00000000 →Dz; if 0, nop	111110********** 100011100000zzzz	1	—
`DCF PCLR Dz`	If DC = 0, H'00000000→Dz; if 1, nop	111110********** 100011110000zzzz	1	—

Table 8-4 Alphabetical Listing of DSP Operation Instructions (cont)

Instruction	Operation	Code	Cycles	DC Bit
PCMP Sx,Sy	Sx – Sy	111110********** 10000100xxyy0000	1	Update
PCOPY Sx,Dz	Sx→Dz	111110********** 11011001xx00zzzz	1	Update
PCOPY Sy,Dz	Sy→Dz	111110********** 1111100100yyzzzz	1	Update
DCT PCOPY Sx,Dz	If DC = 1, Sx→Dz; if 0, nop	111110********** 11011010xx00zzzz	1	—
DCT PCOPY Sy,Dz	If DC = 1, Sy→Dz; if 0, nop	111110********** 1111101000yyzzzz	1	—
DCF PCOPY Sx,Dz	If DC = 0, Sx→Dz; if 1, nop	111110********** 11011011xx00zzzz	1	—
DCF PCOPY Sy,Dz	If DC = 0, Sy→Dz; if 1, nop	111110********** 1111101100yyzzzz	1	—
PDEC Sx,Dz	MSW of Sx–1→MSW of Dz, clear LSW of Dz	111110********** 10001001xx00zzzz	1	Update
PDEC Sy,Dz	MSW of Sy–1→MSW of Dz, clear LSW of Dz	111110********** 10101001xx00zzzz	1	Update
DCT PDEC Sx,Dz	If DC = 1, MSW of Sx–1→ MSW of Dz, clear LSW of Dz; if 0, nop	111110********** 10001010xx00zzzz	1	—
DCT PDEC Sy,Dz	If DC = 1, MSW of Sy–1→ MSW of Dz, clear LSW of Dz; if 0, nop	111110********** 10101010xx00zzzz	1	—
DCF PDEC Sx,Dz	If DC = 0, MSW of Sx–1→ MSW of Dz, clear LSW of Dz; if 1, nop	111110********** 10001011xx00zzzz	1	—
DCF PDEC Sy,Dz	If DC = 0, MSW of Sy–1→ MSW of Dz, clear LSW of Dz; if 1, nop	111110********** 10101011xx00zzzz	1	—
PDMSB Sx,Dz	Sx data MSB position → MSW of Dz, clear LSW of Dz	111110********** 10011101xx00zzzz	1	Update
PDMSB Sy,Dz	Sy data MSB position → MSW of Dz, clear LSW of Dz	111110********** 1011110100yyzzzz	1	Update

Table 8-4 Alphabetical Listing of DSP Operation Instructions (cont)

Instruction	Operation	Code	Cycles	DC Bit
`DCT PDMSB Sx,Dz`	If DC = 1, Sx data MSB position → MSW of Dz, clear LSW of Dz; if 0, nop	111110********** 10011110xx00zzzz	1	—
`DCT PDMSB Sy,Dz`	If DC = 1, Sy data MSB position → MSW of Dz, clear LSW of Dz; if 0, nop	111110********** 1011111000yyzzzz	1	—
`DCF PDMSB Sx,Dz`	If DC = 0, Sx data MSB position → MSW of Dz, clear LSW of Dz; if 1, nop	111110********** 10011111xx00zzzz	1	—
`DCF PDMSB Sy,Dz`	If DC = 0, Sy data MSB position → MSW of Dz, clear LSW of Dz; if 1, nop	111110********** 1011111100yyzzzz	1	—
`PINC Sx,Dz`	MSW of Sx + 1→ MSW of Dz, clear LSW of Dz	111110********** 10011001xx00zzzz	1	Update
`PINC Sy,Dz`	MSW of Sy + 1→ MSW of Dz, clear LSW of Dz	111110********** 1011100100yyzzzz	1	Update
`DCT PINC Sx,Dz`	If DC = 1, MSW of Sx + 1→ MSW of Dz, clear LSW of Dz; if 0, nop	111110********** 10011010xx00zzzz	1	—
`DCT PINC Sy,Dz`	If DC = 1, MSW of Sy + 1→ MSW of Dz, clear LSW of Dz; if 0, nop	111110********** 1011101000yyzzzz	1	—
`DCF PINC Sx,Dz`	If DC = 0, MSW of Sx + 1→ MSW of Dz, clear LSW of Dz; if 1, nop	111110********** 10011011xx00zzzz	1	—
`DCF PINC Sy,Dz`	If DC = 0, MSW of Sy + 1→ MSW of Dz, clear LSW of Dz; if 1, nop	111110********** 1011101100yyzzzz	1	—
`PLDS Dz,MACH`	Dz→MACH	111110********** 111011010000zzzz	1	—
`PLDS Dz,MACL`	Dz→MACL	111110********** 111111010000zzzz	1	—
`DCT PLDS Dz,MACH`	If DC = 1, Dz→MACH; if 0, nop	111110********** 111011100000zzzz	1	—
`DCT PLDS Dz,MACL`	If DC = 1, Dz→MACL; if 0, nop	111110********** 111111100000zzzz	1	—
`DCF PLDS Dz,MACH`	If DC = 0, Dz→MACH; if 1, nop	111110********** 111011110000zzzz	1	—

Table 8-4 Alphabetical Listing of DSP Operation Instructions (cont)

Instruction	Operation	Code	Cycles	DC Bit
DCF PLDS Dz,MACL	If DC = 0, Dz→MACL; if 1, nop	111110********** 111111110000zzzz	1	—
PMULS Se,Sf,Dg	MSW of Se × MSW of Sf→Dg	111110********** 0100eeff0000gg00	1	—
PNEG Sx,Dz	0 – Sx → Dz	111110********** 11001001xx00zzzz	1	Update
PNEG Sy,Dz	0 – Sy → Dz;	111110********** 1110100100yyzzzz	1	Update
DCT PNEG Sx,Dz	If DC = 1, 0 – Sx→Dz; if 0, nop	111110********** 11001010xx00zzzz	1	—
DCT PNEG Sy,Dz	If DC = 1, 0 – Sy→Dz; if 0, nop	111110********** 1110101000yyzzzz	1	—
DCF PNEG Sx,Dz	If DC = 0, 0 – Sx→Dz; if 1, nop	111110********** 11001011xx00zzzz	1	—
DCF PNEG Sy,Dz	If DC = 0, 0 – Sy→Dz; if 1, nop	111110********** 1110101100yyzzzz	1	—
POR Sx,Sy,Dz	Sx | Sy→Dz, clear LSW of Dz	111110********** 10110101xxyyzzzz	1	Update
DCT POR Sx,Sy,Dz	If DC = 1, Sx|Sy→Dz, clear LSW of Dz; if 0, nop	111110********** 10110110xxyyzzzz	1	—
DCF POR Sx,Sy,Dz	If DC = 0, Sx|Sy→Dz, clear LSW of Dz; if 1, nop	111110********** 10110111xxyyzzzz	1	—
PRND Sx,Dz	Sx + H'00008000→Dz, clear LSW of Dz	111110********** 10011000xx00zzzz	1	Update
PRND Sy,Dz	Sy + H'00008000→Dz, clear LSW of Dz	111110********** 1011100000yyzzzz	1	Update
PSHA Sx,Sy,Dz	If Sy≥0, Sx<<Sy→Dz; if Sy<0, Sx>>Sy→Dz	111110********** 10010001xxyyzzzz	1	Update
DCT PSHA Sx,Sy,Dz	If DC = 1 & Sy≥0, Sx<<Sy→Dz; if DC = 1 & Sy<0, Sx>>Sy→Dz; if DC = 0, nop	111110********** 10010010xxyyzzzz	1	—

Table 8-4 Alphabetical Listing of DSP Operation Instructions (cont)

Instruction	Operation	Code	Cycles	DC Bit
DCF PSHA Sx,Sy,Dz	If DC = 0 & Sy≥0, Sx<<Sy→Dz; if DC = 0 & Sy<0, Sx>>Sy→Dz; if DC = 1, nop	111110********** 10010011xxyyzzzz	1	—
PSHA #imm,Dz	If imm≥0, Dz<<imm→Dz; if imm<0, Dz>>imm→Dz	111110********** 00001iiiiiiizzzz	1	Update
PSHL Sx,Sy,Dz	If Sy≥0, Sx<<Sy → Dz, clear LSW of Dz; if Sy<0, Sx>>Sy → Dz, clear LSW of Dz	111110********** 10000001xxyyzzzz	1	Update
DCT PSHL Sx,Sy,Dz	If DC=1 & Sy≥0, Sx<<Sy → Dz, clear LSW of Dz; if DC=1 & Sy<0, Sx>>Sy → Dz, clear LSW of Dz; if DC=0, nop	111110********** 10000010xxyyzzzz	1	—
DCF PSHL Sx,Sy,Dz	If DC=0 & Sy≥0, Sx<<Sy → Dz, clear LSW of Dz; if DC=0 & Sy<0, Sx>>Sy → Dz, clear LSW of Dz; if DC=1, nop	111110********** 10000011xxyyzzzz	1	—
PSHL #imm,Dz	If imm≥0, Dz<<imm → Dz, clear LSW of Dz; if imm<0, Dz>>imm → Dz, clear LSW of Dz	111110********** 00000iiiiiiizzzz	1	Update
PSTS MACH,Dz	MACH → Dz	111110********** 110011010000zzzz	1	—
PSTS MACL,Dz	MACL → Dz	111110********** 110111010000zzzz	1	—
DCT PSTS MACH,Dz	If DC=1, MACH → Dz; if 0, nop	111110********** 110011100000zzzz	1	—
DCT PSTS MACL,Dz	If DC=1, MACL → Dz; if 0, nop	111110********** 110111100000zzzz	1	—
DCF PSTS MACH,Dz	If DC = 0, MACH→Dz; if 1, nop	111110********** 110011110000zzzz	1	—
DCF PSTS MACL,Dz	If DC = 0, MACL→Dz; if 1, nop	111110********** 110111110000zzzz	1	—

Table 8-4 Alphabetical Listing of DSP Operation Instructions (cont)

Instruction	Operation	Code	Cycles	DC Bit
PSUB Sx,Sy,Dz	Sx–Sy→Dz	111110********** 10100001xxyyzzzz	1	Update
DCT PSUB Sx,Sy,Dz	If DC = 1, Sx – Sy→Dz; if 0, nop	111110********** 10100010xxyyzzzz	1	—
DCF PSUB Sx,Sy,Dz	If DC = 0, Sx – Sy→Dz; if 1, nop	111110********** 10100011xxyyzzzz	1	—
PSUB Sx,Sy,Du PMULS Se,Sf,Dg	Sx – Sy→Du; MSW of Se × MSW of Sf→Dg	111110********** 0110eeffxxyygguu	1	Update
PSUBC Sx,Sy,Dz	Sx–Sy–DC→Dz	111110********** 10100000xxyyzzzz	1	Update
PXOR Sx,Sy,Dz	Sx ^ Sy→Dz, clear LSW of Dz	111110********** 10100101xxyyzzzz	1	Update
DCT PXOR Sx,Sy,Dz	If DC = 1, Sx ^ Sy→Dz, clear LSW of Dz; if 0, nop	111110********** 10100110xxyyzzzz	1	—
DCF PXOR Sx,Sy,Dz	If DC = 0, Sx ^ Sy→Dz, clear LSW of Dz; if 1, nop	111110********** 10100111xxyyzzzz	1	—

Note: Updated based on the PADD operation results

The DC bit in the DSR register is updated in accordance with the result of a DSP instruction and the specification of the status selection bit (CS). In addition to the DC bit, the DSR register also contains four status indication flags (V, N, Z, and GT). The operation of each bit is described below. In the later descriptions of instruction operation for each DSP operation, the following operation contents are used as subroutine modules.

Operation contents (1) Fix-point borrow DC bit

```
/*  SH-DSP: DSP Engine:  fixed_pt_dc_always_borrow.c
    Set DSR's DC Bit to borrow bit regardless the status of CS[2:0] bits */


{
    /* DC update policy: don't care the status of DSPCSBITS */
    DSPDCBIT = borrow_bit;
    DSPGTBIT = ~((negative_bit ^ overflow_bit) | zero_bit);
    DSPZBIT  = zero_bit;
    DSPNBIT  = negative_bit;
    DSPVBIT  = overflow_bit;
```

```
}
```

Operation contents (2) Fixed-point carry DC bit

```
/*  SH-DSP: DSP Engine:  fixed_pt_dc_always_carry.c
    Set DSR's DC Bit to carry bit regardless the status of CS[2:0] bits */

{
    /* DC update policy: don't care the status of DSPCSBITS */
    DSPDCBIT = carry_bit;
    DSPGTBIT = ~((negative_bit ^ overflow_bit) | zero_bit);
    DSPZBIT  = zero_bit;
    DSPNBIT  = negative_bit;
    DSPVBIT  = overflow_bit;
}
```

Operation contents (3) Fixed-point negative value DC bit

```
/*  SH-DSP: DSP Engine:  fixed_pt_minus_dc_bit.c
    Fixed Point Minus(-) Operation: Set DC Bit in DSR */

{
    switch (DSPCSBITS) {
        case 0x0:  /* Borrow Mode */
            DSPDCBIT = borrow_bit;
            break;
        case 0x1:  /* Negative Value Mode */
            DSPDCBIT = negative_bit;
            break;
        case 0x2:  /* Zero Value Mode */
            DSPDCBIT = zero_bit;
            break;
        case 0x3:  /* Overflow Mode */
            DSPDCBIT = overflow_bit;
            break;
        case 0x4:  /* Signed Greater Than Mode */
            DSPDCBIT = ~((negative_bit ^ overflow_bit) | zero_bit);
            break;
        case 0x5:  /* Signed Greater Than or Equal Mode */
            DSPDCBIT = ~(negative_bit ^ overflow_bit);
```

```
            break;
        case 0x6:  /* Reserved */
        case 0x7:  /* Reserved */
            break;
    }
    DSPGTBIT = ~((negative_bit ^ overflow_bit) | zero_bit);
    DSPZBIT  = zero_bit;
    DSPNBIT  = negative_bit;
    DSPVBIT  = overflow_bit;
}
```

Operation contents (4) Fixed-point overflow prevention function (saturated operation)

```
/*  SH-DSP: DSP Engine:  Set to maximum non-overflow value if overlow
    fixed_pt_overflow_protection.c */

{
    if(SBIT && overflow_bit) {     /* Overflow Protection Enable & overflow */
    if(DSP_ALU_DSTG_BIT7==0) { /* positive value */
        if((DSP_ALU_DSTG_LSB8!=0x0) || (DSP_ALU_DST_MSB!=0)) {
            DSP_ALU_DSTG= 0x0;
            DSP_ALU_DST = 0x7fffffff;
        }
    }
    else {        /* negative value */
        if((DSP_ALU_DSTG_LSB8!=0xff) || (DSP_ALU_DST_MSB!=1)) {
            DSP_ALU_DSTG= 0xff;
            DSP_ALU_DST = 0x80000000;
        }
    }
    overflow_bit = 0; /* No more overflow when protected */
    }
}
```

Operation contents (5) Fixed-point positive value DC bit

```
/*  SH-DSP: DSP Engine:  fixed_pt_plus_dc_bit.c
    Fixed Point Plus(+) Operation: Set DC Bit in DSR /*
{
    switch (DSPCSBITS) {
```

```
        case 0x0:  /* Carry Mode */
            DSPDCBIT = carry_bit;
            break;
        case 0x1:  /* Negative Value Mode */
            DSPDCBIT = negative_bit;
            break;
        case 0x2:  /* Zero Value Mode */
            DSPDCBIT = zero_bit;
            break;
        case 0x3:  /* Overflow Mode */
            DSPDCBIT = overflow_bit;
            break;
        case 0x4:  /* Signed Greater Than Mode */
            DSPDCBIT = ~((negative_bit ^ overflow_bit) | zero_bit);
            break;
        case 0x5:  /* Signed Greater Than or Equal Mode */
            DSPDCBIT = ~(negative_bit ^ overflow_bit);
            break;
        case 0x6:  /* Reserved */
        case 0x7:  /* Reserved */
            break;
    }
    DSPGTBIT = ~((negative_bit ^ overflow_bit) | zero_bit);
    DSPZBIT  = zero_bit;
    DSPNBIT  = negative_bit;
    DSPVBIT  = overflow_bit;
}
```

Operation contents (6) Fixed-point operation unconditional DC bit update

```
/*  SH-DSP: DSP Engine: Fixed Point Unconditional Update
    fixed_pt_unconditional_update.c
        1. Write back to the Destination Register
        2. update negative_bit and zero_bit. */
/*  negative_bit = MSB of ALU's 40-bit result.
    zero_bit     = if(ALU's 40-bit result==0)
    sign-extend to A0/1G[31:8] */

{
```

```
    DSP_REG[ex2_dz_no] = DSP_ALU_DST;
    if (ex2_dz_no==0) {
        A0G = DSP_ALU_DSTG & MASK000000FF;
        if(DSP_ALU_DSTG_BIT7) A0G = A0G | MASKFFFFFF00;
    }
    else if (ex2_dz_no==1) {
        A1G = DSP_ALU_DSTG & MASK000000FF;
        if(DSP_ALU_DSTG_BIT7) A1G = A1G | MASKFFFFFF00;
    }
    negative_bit = DSP_ALU_DSTG_BIT7;
    zero_bit = (DSP_ALU_DST==0) & (DSP_ALU_DSTG_LSB8==0);
}
```

Operation contents (7) Integer negative value DC bit

```
/*  SH-DSP: DSP Engine:  integer_minus_dc_bit.c
    Integer Minus(-) Operation: Set DC Bit in DSR */

#include "fixed_pt_minus_dc_bit.c"
```

Operation contents (8) Integer overflow prevention function (saturated operation)

```
/*  SH-DSP: DSP Engine:  Set to maximum non-overflow value if overlow
    integer_overflow_protection.c */

#include "fixed_pt_overflow_protection.c"
```

Operation contents (9) Integer positive value DC bit

```
/*  SH-DSP: DSP Engine: integer_plus_dc_bit.c
    Integer Plus(+) Operation: Set DC Bit in DSR */

#include "fixed_pt_plus_dc_bit.c"
```

Operation contents (10) Integer unconditional DC bit update

```
/*  SH-DSP: DSP Engine: Integer Operation Unconditional Update
    integer_unconditional_update.c
        1. Write back to the Destination Register
        2. update negative_bit and zero_bit.
    negative_bit = MSB of ALU's 24-bit(g-bit and hw) result.
    zero_bit = if(ALU's g-bit & hw==0)
```

```
    Spec 1.1: Clear ALU Integer operation's LSW. */

{
    DSP_REG_WD[ex2_dz_no*2] = DSP_ALU_DST_HW;
    DSP_REG_WD[ex2_dz_no*2+1] = 0x0;      /* clear LSW */
    if (ex2_dz_no==0) {
        A0G = DSP_ALU_DSTG & MASK000000FF;
        if(DSP_ALU_DSTG_BIT7) A0G = A0G | MASKFFFFFF00;
    }
    else if (ex2_dz_no==1) {
        A1G = DSP_ALU_DSTG & MASK000000FF;
        if(DSP_ALU_DSTG_BIT7) A1G = A1G | MASKFFFFFF00;
    }
    negative_bit = DSP_ALU_DSTG_BIT7;
    zero_bit = (DSP_ALU_DST_HW==0) & (DSP_ALU_DSTG_LSB8==0);
}
```

Operation contents (11) Logical operation DC bit

```
/*  SH-DSP: DSP Engine: logical_dc_bit.c
    Logical Operation: Set DC Bit in DSR */

{
    switch (DSPCSBITS) {
        case 0x0:  /* Carry Mode */
            DSPDCBIT = 0;
            break;
        case 0x1:  /* Negative Value Mode */
            DSPDCBIT = negative_bit;
            break;
        case 0x2:  /* Zero Value Mode */
            DSPDCBIT = zero_bit;
            break;
        case 0x3:  /* Overflow Mode */
            DSPDCBIT = 0;
            break;
        case 0x4:  /* Signed Greater Than Mode */
            DSPDCBIT = 0;
            break;
```

```
        case 0x5:  /* Signed Greater Than or Equal Mode */
            DSPDCBIT = 0;
            break;
        case 0x6:  /* Reserved */
        case 0x7:  /* Reserved */
            break;
    }
         DSPGTBIT = 0;
         DSPZBIT  = zero_bit;
         DSPNBIT  = negative_bit;
         DSPVBIT  = 0;
}
```

Operation contents (12) Shift operation DC bit

```
/*  SH-DSP: DSP Engine: Shift_dc_bit.c
    Shift Operation: Set DC Bit in DSR */

{
    switch (DSPCSBITS) {
        case 0x0:  /* Carry Mode */
            DSPDCBIT = carry_bit;
            break;
        case 0x1:  /* Negative Value Mode */
            DSPDCBIT = negative_bit;
            break;
        case 0x2:  /* Zero Value Mode */
            DSPDCBIT = zero_bit;
            break;
        case 0x3:  /* Overflow Mode */
            DSPDCBIT = overflow_bit;
            break;
        case 0x4:  /* Signed Greater Than Mode */
            DSPDCBIT = 0;
            break;
        case 0x5:  /* Signed Greater Than or Equal Mode */
            DSPDCBIT = 0;
            break;
        case 0x6:  /* Reserved */
```

```
        case 0x7:  /* Reserved */
            break;
    }
    DSPGTBIT = 0;
    DSPZBIT  = zero_bit;
    DSPNBIT  = negative_bit;
    DSPVBIT  = overflow_bit;
}
```

8.5.1 PABS (Absolute): DSP Arithmetic Operation Instruction

Format	Abstract	Code	Cycle	DC Bit
PABS Sx,Dz	If Sx≥0,Sx→Dz If Sx<0,0–Sx→Dz	111110********** 10001000xx00zzzz	1	Update
PABS Sy,Dz	If Sy≥0,Sy→Dz If Sy<0,0–Sy→Dz	111110********** 1010100000yyzzzz	1	Update

Description: Finds absolute values. When the Sx and Sy operands are positive, the contents of the operands are transferred to the Dz operand. If the value is negative, the amounts of the Sx and Sy operand contents are subtracted from 0 and stored in the Dz operand.

The DC bit of the DSR register are updated according to the specifications of the CS bits. The N, Z, V, and GT bits of the DSR register are updated.

Operation:

```
{
    DSP_ALU_SRC1 = 0;
    DSP_ALU_SRC1G= 0;
    if (EX2_DSP_BIT13==0) {        /* 0 +/- Sx -> Dz */
        switch (EX2_SX) {
            case 0x0:  DSP_ALU_SRC2  = X0;
                    if (DSP_ALU_SRC2_MSB) DSP_ALU_SRC2G = 0xff;
                    else   DSP_ALU_SRC2G = 0x0;
                    break;
            case 0x1:  DSP_ALU_SRC2  = X1;
                    if (DSP_ALU_SRC2_MSB) DSP_ALU_SRC2G = 0xff;
                    else     DSP_ALU_SRC2G = 0x0;
                    break;
            case 0x2:  DSP_ALU_SRC2  = A0;
                    DSP_ALU_SRC2G = A0G;
                    break;
            case 0x3:  DSP_ALU_SRC2  = A1;
                    DSP_ALU_SRC2G = A1G;
                    break;
        }
    }

    else  {        /* 0 +/- Sy -> Dz */
```

```
    switch (EX2_SY) {
            case 0x0:  DSP_ALU_SRC2  = Y0;
                    break;
            case 0x1:  DSP_ALU_SRC2  = Y1;
                    break;
            case 0x2:  DSP_ALU_SRC2  = M0;
                    break;
            case 0x3:  DSP_ALU_SRC2  = M1;
                    break;
    }
    if (DSP_ALU_SRC2_MSB)   DSP_ALU_SRC2G = 0xff;
    else                  DSP_ALU_SRC2G = 0x0;
}

if(DSP_ALU_SRC2G_BIT7==0) {       /* positive value */
    DSP_ALU_DST = 0x0 + DSP_ALU_SRC2;
    carry_bit = 0;
    DSP_ALU_DSTG_LSB8= 0x0 + DSP_ALU_SRC2G_LSB8 + carry_bit;
}
else {                    /* negative value */
        DSP_ALU_DST = 0x0 - DSP_ALU_SRC2;
        borrow_bit = 1;
        DSP_ALU_DSTG_LSB8= 0x0 - DSP_ALU_SRC2G_LSB8 - borrow_bit;
}

        overflow_bit= PLUS_OP_G_OV || !(POS_NOT_OV || NEG_NOT_OV);
#include "fixed_pt_overflow_protection.c"
#include "fixed_pt_unconditional_update.c"

    if(DSP_ALU_SRC2G_BIT7==0) {
#include "fixed_pt_plus_dc_bit.c"
    }
    else   {
        overflow_bit= MINUS_OP_G_OV || !(POS_NOT_OV || NEG_NOT_OV);
#include "fixed_pt_minus_dc_bit.c"
    }
}
```

```
break;
```

Examples:

`PABS X0, M0 NOPX NOPY`	;Before execution:	X0 = H'33333333, M0 = H'12345678
	;After execution:	X0 = H'33333333, M0 = H'33333333
`PABS X1, X1 NOPX NOPY`	;Before execution:	X1 = H'DDDDDDDD
	;After execution:	X1 = H'22222223
	DC bit is updated depending on the state of CS [2:0].	

8.5.2 [if cc]PADD (Addition with Condition): DSP Arithmetic Operation Instruction

Format	Abstract	Code	Cycle	DC Bit
`PADD Sx,Sy,Dz`	Sx+Sy→Dz	111110********** 10110001xxyyzzzz	1	Update
`DCT PADD Sx,Sy,Dz`	if DC=1,Sx+Sy→Dz if 0,nop	111110********** 10110010xxyyzzzz	1	—
`DCF PADD Sx,Sy,Dz`	if DC=0,Sx+Sy→Dz if 1,nop	111110********** 10110011xxyyzzzz	1	—

Description: Adds the contents of the Sx and Sy operands and stores the result in the Dz operand. When conditions are specified for DCT and DCF, the instruction is executed when those conditions are TRUE. When they are FALSE, the instruction is not executed.

When conditions are not specified, the DC bit of the DSR register is updated according to the specifications for the CS bits. The N, Z, V, and GT bits of the DSR register are also updated. The DC, N, Z, V, and GT bits are not updated when conditions are specified, even if the conditions are TRUE.

Operation:

```
{
    switch (EX2_SX) {
         case 0x0: DSP_ALU_SRC1  = X0;
            if (DSP_ALU_SRC1_MSB) DSP_ALU_SRC1G = 0xff;
            else   DSP_ALU_SRC1G = 0x0;
            break;
         case 0x1: DSP_ALU_SRC1  = X1;
            if (DSP_ALU_SRC1_MSB) DSP_ALU_SRC1G = 0xff;
            else   DSP_ALU_SRC1G = 0x0;
                   break;
         case 0x2: DSP_ALU_SRC1  = A0;
            DSP_ALU_SRC1G = A0G;
            break;
        case 0x3:  DSP_ALU_SRC1  = A1;
            DSP_ALU_SRC1G = A1G;
            break;
    }
    switch (EX2_SY) {
```

```
        case 0x0:  DSP_ALU_SRC2  = Y0;
            break;
        case 0x1:  DSP_ALU_SRC2  = Y1;
            break;
        case 0x2:  DSP_ALU_SRC2  = M0;
            break;
        case 0x3:  DSP_ALU_SRC2  = M1;
            break;
    }
    if (DSP_ALU_SRC2_MSB) DSP_ALU_SRC2G = 0xff;
    else       DSP_ALU_SRC2G = 0x0;

    DSP_ALU_DST = DSP_ALU_SRC1 + DSP_ALU_SRC2;
    carry_bit = ((DSP_ALU_SRC1_MSB | DSP_ALU_SRC2_MSB) & !DSP_ALU
_DST_MSB) |
        (DSP_ALU_SRC1_MSB & DSP_ALU_SRC2_MSB);
    DSP_ALU_DSTG_LSB8 = DSP_ALU_SRC1G_LSB8 + DSP_ALU_SRC2G_LSB8 + carry_bit;

    overflow_bit= PLUS_OP_G_OV || !(POS_NOT_OV || NEG_NOT_OV);
#include "fixed_pt_overflow_protection.c"

    if(DSP_UNCONDITIONAL_UPDATE) { /* unconditional operation */
#include "fixed_pt_unconditional_update.c"
#include "fixed_pt_plus_dc_bit.c"
    }
    else if(DSP_CONDITION_MATCH) { /* conditional operation and match */
        DSP_REG[ex2_dz_no] = DSP_ALU_DST;
        if(ex2_dz_no==0) {
            A0G = DSP_ALU_DSTG & MASK000000FF;
            if(DSP_ALU_DSTG_BIT7) A0G = A0G | MASKFFFFFF00;
        }
        else if(ex2_dz_no==1) {
            A1G = DSP_ALU_DSTG & MASK000000FF;
            if(DSP_ALU_DSTG_BIT7) A1G = A1G | MASKFFFFFF00;
        }
    }
}
```

```
break;
```

Examples:

```
PADD X0,Y0,A0 NOPX NOPY   ;Before execution:   X0 = H'22222222, Y0 = H'33333333,
                                               A0 = H'123456789A
                          ;After execution:    X0 = H'22222222, Y0 = H'33333333,
                                               A0 = H'0055555555
```

In case of unconditional execution, the DC bit is updated depending on the state of the CS [2:0] bit immediately before the operation.

8.5.3 PADD PMULS (Addition & Multiply Signed by Signed): DSP Arithmetic Operation Instruction

Format	Abstract	Code	Cycle	DC Bit
PADD Sx,Sy,Du	Sx + Sy→Du	111110**********	1	Update
PMULS Se,Sf,Dg	MSW of Se × MSW of Sf→Dg	0111eeffxxyygguu		

Description: Adds the contents of the Sx and Sy operands and stores the result in the Du operand. The contents of the top word of the Se and Sf operands are multiplied as signed and the result stored in the Dg operand. These two processes are executed simultaneously in parallel.

The DC bit of the DSR register is updated according to the results of the ALU operation and the specifications for the CS bits. The N, Z, V, and GT bits of the DSR register are also updated according to the results of the ALU operation.

Note: Since the PMULS is fixed decimal point multiplication, the operation result is different from that of MULS even though the source data is the same.

Operation:

```
{
    DSP_ALU_DST = DSP_ALU_SRC1 + DSP_ALU_SRC2;
    carry_bit=((DSP_ALU_SRC1_MSB | DSP_ALU_SRC2_MSB) & !DSP_ALU
_DST_MSB) |
        (DSP_ALU_SRC1_MSB & DSP_ALU_SRC2_MSB);
    DSP_ALU_DSTG_LSB8=DSP_ALU_SRC1G_LSB8 + DSP_ALU_SRC2G_LSB8 + carry
_bit;

    overflow_bit= PLUS_OP_G_OV || !(POS_NOT_OV || NEG_NOT_OV);
#include "../d_3operand.d/fixed_pt_overflow_protection.c"
    switch (EX2_DU) {
         case 0x0:
            X0  = DSP_ALU_DST;
            negative_bit = DSP_ALU_DSTG_BIT7
            zero_bit = (DSP_ALU_DST==0)&(DSP_ALU_DSTG_LSB8==0);
            break;
         case 0x1:
            Y0  = DSP_ALU_DST;
            negative_bit = DSP_ALU_DSTG_BIT7;
            zero_bit = (DSP_ALU_DST==0)&(DSP_ALU_DSTG_LSB8==0);
```

```
            break;

         case 0x2:
            A0  = DSP_ALU_DST;
            A0G = DSP_ALU_DSTG & MASK000000FF;
            if(DSP_ALU_DSTG_BIT7) A0G = A0G | MASKFFFFFF00;
            negative_bit = DSP_ALU_DSTG_BIT7;
            zero_bit = (DSP_ALU_DST==0) & (DSP_ALU_DSTG
_LSB8==0);
            break;
         case 0x3:
            A1  = DSP_ALU_DST;
            A1G = DSP_ALU_DSTG & MASK000000FF;
            if(DSP_ALU_DSTG_BIT7) A1G = A1G | MASKFFFFFF00;
            negative_bit = DSP_ALU_DSTG_BIT7;
            zero_bit = (DSP_ALU_DST==0) & (DSP_ALU_DSTG
_LSB8==0);
            break;
    }
#include "../d_3operand.d/fixed_pt_plus_dc_bit.c"
}

break;
```

Examples:

```
PADD A0,M0,A0 PMULS X0,YO,MO NOPX NOPY
```

;Before execution: X0 = H'00020000, Y0 = H'00030000,
M0 = H'22222222, A0 = H'0055555555

;After execution: X0 = H'00020000, Y0 = H'00030000,
M0 = H'0000000C, A0 = H'0077777777

The DC bit is updated based on the result of the PADD operation, depending on the state of CD [2:0].

8.5.4 PADDC (Addition with Carry): DSP Arithmetic Operation Instruction

Format	Abstract	Code	Cycle	DC Bit
PADDC Sx, Sy, Dz	Sx + Sy + DC → Dz	111110********** 10110000xxyyzzzz	1	Carry

Description: Adds the contents of the Sx and Sy operands to the DC bit and stores the result in the Dz operand. The DC bit of the DSR register is updated as the carry flag. The N, Z, V, and GT bits of the DSR register are also updated.

Note: The DC bit is updated as the carry flag after execution of the PADDC instruction regardless of the CS bits.

Operation:

```
{
    switch (EX2_SX) {
         case 0x0: DSP_ALU_SRC1  = X0;
                   if (DSP_ALU_SRC1_MSB) DSP_ALU_SRC1G = 0xff;
                   else   DSP_ALU_SRC1G = 0x0;
                   break;
         case 0x1: DSP_ALU_SRC1  = X1;
                   if (DSP_ALU_SRC1_MSB) DSP_ALU_SRC1G = 0xff;
                   else   DSP_ALU_SRC1G = 0x0;
                   break;
         case 0x2: DSP_ALU_SRC1  = A0;
            DSP_ALU_SRC1G = A0G;
            break;
         case 0x3: DSP_ALU_SRC1  = A1;
            DSP_ALU_SRC1G = A1G;
            break;

    }
    switch (EX2_SY) {
        case 0x0:  DSP_ALU_SRC2  = Y0;
            break;
        case 0x1:  DSP_ALU_SRC2  = Y1;
            break;
        case 0x2:  DSP_ALU_SRC2  = M0;
            break;
```

```
        case 0x3:  DSP_ALU_SRC2  = M1;
            break;
    }
    if (DSP_ALU_SRC2_MSB) DSP_ALU_SRC2G = 0xff;
    else          DSP_ALU_SRC2G = 0x0;

    DSP_ALU_DST = DSP_ALU_SRC1 + DSP_ALU_SRC2 + DSPDCBIT;
    carry_bit = ((DSP_ALU_SRC1_MSB | DSP_ALU_SRC2_MSB) & !DSP_ALU
_DST_MSB) |
            (DSP_ALU_SRC1_MSB & DSP_ALU_SRC2_MSB);
    DSP_ALU_DSTG_LSB8 = DSP_ALU_SRC1G_LSB8 + DSP_ALU_SRC2G_LSB8 + carry_bit;

    overflow_bit= PLUS_OP_G_OV || !(POS_NOT_OV || NEG_NOT_OV);
#include "fixed_pt_overflow_protection.c"

#include "fixed_pt_unconditional_update.c"
#include "fixed_pt_dc_always_carry.c"
}

break;
```

Example:

CS[2:0]=***: Always operate as Carry or Borrow mode, regardless of the status of the DC bit.

PADDC X0,Y0,M0 NOPX NOPY	;Before execution:	X0 = H'B3333333, Y0 = H'55555555
		M0 = H'12345678, DC = 0
	;After execution:	X0 = H'B3333333, Y0 = H'55555555
		M0 = H'08888888, DC = 1
PADDC X0,Y0,M0 NOPX NOPY	;Before execution:	X0 = H'33333333, Y0 = H'55555555
		M0 = H'12345678, DC = 1
	;After execution:	X0 = H'33333333, Y0 = H'55555555
		M0 = H'88888889, DC = 0

DC bit is updated depending on the state of CS [2:0].

8.5.5 [if cc] PAND (Logical AND): DSP Logical Operation Instruction

Format	Abstract	Code	Cycle	DC Bit
`PAND Sx,Sy,Dz`	Sx & Sy→Dz; clear LSW of Dz	111110********** 10010101xxyyzzzz	1	
`DCT PAND Sx,Sy,Dz`	If DC = 1, SX & SY→Dz, clear LSW of Dz; if 0, nop	111110********** 10010110xxyyzzzz	1	—
`DCF PAND Sx,Sy,Dz`	If DC = 0, SX & SY→Dz, clear LSW of Dz; if 1, nop	111110********** 10010111xxyyzzzz	1	—

Description: Does an AND of the upper word of the Sx operand and the upper word of the Sy operand, stores the result in the upper word of the Dz operand, and clears the bottom word of the Dz operand with zeros. When Dz is a register that has guard bits, the guard bits are also zeroed. When conditions are specified for DCT and DCF, the instruction is executed when those conditions are TRUE. When they are FALSE, the instruction is not executed.

When conditions are not specified, the DC bit of the DSR register is updated according to the specifications for the CS bits. The N, Z, V, and GT bits of the DSR register are also updated. The DC, N, Z, V, and GT bits are not updated when conditions are specified, even if the conditions are TRUE.

Note: The bottom word of the destination register and the guard bits are ignored when the DC bit is updated.

Operation:

```
{
    switch (EX2_SX) {
        case 0x0:    DSP_ALU_SRC1  = X0;
              break;
        case 0x1:    DSP_ALU_SRC1  = X1;
              break;
        case 0x2:    DSP_ALU_SRC1  = A0;
              break;
        case 0x3:    DSP_ALU_SRC1  = A1;
              break;
    }
    switch (EX2_SY) {
        case 0x0:    DSP_ALU_SRC2  = Y0;
              break;
```

```
        case 0x1:    DSP_ALU_SRC2  = Y1;
              break;
        case 0x2:    DSP_ALU_SRC2  = M0;
              break;
        case 0x3:    DSP_ALU_SRC2  = M1;
              break;
    }

    DSP_ALU_DST_HW = DSP_ALU_SRC1_HW & DSP_ALU_SRC2_HW;

    if(DSP_UNCONDITIONAL_UPDATE)   { /* unconditional operation */
       DSP_REG_WD[ex2_dz_no*2] = DSP_ALU_DST_HW;
       DSP_REG_WD[ex2_dz_no*2+1] = 0x0;  /* clear LSW */
       if (ex2_dz_no==0) A0G = 0x0;      /* clear Guard
bits */
       else if (ex2_dz_no==1)     A1G = 0x0;

       carry_bit    = 0x0;
       negative_bit = DSP_ALU_DST_MSB;
       zero_bit     = (DSP_ALU_DST_HW==0);
       overflow_bit = 0x0;
#include "logical_dc_bit.c"
    }
    else if(DSP_CONDITION_MATCH)   { /* conditional operation and match */
       DSP_REG_WD[ex2_dz_no*2] = DSP_ALU_DST_HW;
       DSP_REG_WD[ex2_dz_no*2+1] = 0x0;  /* clear LSW */
       if (ex2_dz_no==0) A0G = 0x0;      /* clear Guard
bits */
       else if (ex2_dz_no==1)     A1G = 0x0;
    }
}

break;
```

Example:

```
PAND X0,Y0,A0   NOPX   NOPY
```

;Before execution: X0 = H'33333333, Y0 = H'55555555

A0 = H'123456789A

;After execution: X0 = H'33333333, Y0 = H'55555555

A0 = H'0011110000

In case of unconditional execution, the DC bit is updated depending on the state of the CS [2:0] bit immediately before the operation.

8.5.6 [if cc] PCLR (Clear): DSP Arithmetic Operation Instruction

Format	Abstract	Code	Cycle	DC Bit
PCLR Dz	H'00000000→Dz	111110********** 100011010000zzzz	1	Update
DCT PCLR Dz	if DC = 1, H'00000000→Dz if 0, nop	111110********** 100011100000zzzz	1	—
DCF PCLR Dz	if DC = 0, H'00000000→Dz if 1, nop	111110********** 100011110000zzzz	1	—

Description: Clears the Dz operand. When conditions are specified for DCT and DCF, the instruction is executed when those conditions are TRUE. When they are FALSE, the instruction is not executed.

When conditions are not specified, the DC bit of the DSR register is updated according to the specifications for the CS bits. The Z bit of the DSR register is set to 1. The N, V, and GT bits are cleared to 0. The DC, N, Z, V, and GT bits are not updated when conditions are specified, even if the conditions are TRUE.

Operation:

```
{   /* 0 + 0 -> Dz */
    if(DSP_UNCONDITIONAL_UPDATE) { /* unconditional operation */
        DSP_REG[ex2_dz_no] = 0x0;
        if (ex2_dz_no==0) A0G = 0x0;
        else if (ex2_dz_no==1)     A1G = 0x0;

        carry_bit  = 0;
        negative_bit   = 0;
        zero_bit   = 1;
        overflow_bit   = 0;

#include "fixed_pt_plus_dc_bit.c"
    }
    else if(DSP_CONDITION_MATCH) { /* conditional operation and match */
        DSP_REG[ex2_dz_no] = 0x0;
    }
}
```

```
break;
```

Example:

`PCLR A0   NOPX   NOPY`	;Before execution:	A0 = H'FF87654321
	;After execution:	A0 = H'0000000000

In case of unconditional execution, the DC bit is updated depending on the state of the CS [2:0].

8.5.7 PCMP (Compare Two Data): DSP Arithmetic Operation Instruction

Format	Abstract	Code	Cycle	DC Bit
PCMP Sx, Sy	Sx–Sy	111110********** 10000100xxyy0000	1	Update

Description: Subtracts the contents of the Sy operand from the Sx operand. The DC bit of the DSR register is updated according to the specifications for the CS bits. The N, Z, V, and GT bits of the DSR register are also updated.

Operation:

```
{
    switch (EX2_SX)   {
        case 0x0:  DSP_ALU_SRC1  = X0;
            if (DSP_ALU_SRC1_MSB) DSP_ALU_SRC1G = 0xff;
            else    DSP_ALU_SRC1G = 0x0;
            break;
        case 0x1:  DSP_ALU_SRC1  = X1;
            if (DSP_ALU_SRC1_MSB)   DSP_ALU_SRC1G = 0xff;
            else    DSP_ALU_SRC1G = 0x0;
            break;
        case 0x2:  DSP_ALU_SRC1  = A0;
            DSP_ALU_SRC1G = A0G;
            break;
        case 0x3:  DSP_ALU_SRC1  = A1;
            DSP_ALU_SRC1G = A1G;
            break;
    }
    switch (EX2_SY)   {
        case 0x0:  DSP_ALU_SRC2  = Y0;
            break;
        case 0x1:  DSP_ALU_SRC2  = Y1;
            break;
        case 0x2:  DSP_ALU_SRC2  = M0;
            break;
        case 0x3:  DSP_ALU_SRC2  = M1;
            break;
    }
```

```
    if (DSP_ALU_SRC2_MSB)   DSP_ALU_SRC2G = 0xff;
          else     DSP_ALU_SRC2G = 0x0;
    DSP_ALU_DST = DSP_ALU_SRC1 - DSP_ALU_SRC2;
    carry_bit =((DSP_ALU_SRC1_MSB | !DSP_ALU_SRC2_MSB) && !DSP_ALU_DST_MSB) |
           (DSP_ALU_SRC1_MSB & !DSP_ALU_SRC2_MSB);
    borrow_bit = !carry_bit;
    DSP_ALU_DSTG_LSB8 = DSP_ALU_SRC1G_LSB8 - DSP_ALU_SRC2G_LSB8 - borrow_bit;

    negative_bit = DSP_ALU_DSTG_BIT7;
    zero_bit = (DSP_ALU_DST==0) & (DSP_ALU_DSTG_LSB8==0);
    overflow_bit= MINUS_OP_G_OV || !(POS_NOT_OV || NEG_NOT_OV);

#include "fixed_pt_overflow_protection.c"
#include "fixed_pt_minus_dc_bit.c"
}

break;
```

Examples:

`PCMP X0, Y0 NOPX NOPY`	;Before execution:	X0 = H'22222222, Y0 = H'33333333
	;After execution:	X0 = H'22222222, Y0 = H'33333333
		N = 1, Z = 0, V = 0, GT = 0
	DC bit is updated depending on the state of CS [2:0].	

8.5.8 [if cc] PCOPY (Copy with Condition): DSP Arithmetic Operation Instruction

Format	Abstract	Code	Cycle	DC Bit
PCOPY Sx,Dz	Sx→Dz	111110********** 11011001xx00zzzz	1	Update
PCOPY Sy,Dz	Sy→Dz	111110********** 1111100100yyzzzz	1	Update
DCT PCOPY Sx,Dz	if DC = 1, Sx→Dz if 0, nop	111110********** 11011010xx00zzzz	1	—
DCT PCOPY Sy,Dz	if DC = 1, Sy→Dz if 0, nop	111110********** 1111101000yyzzzz	1	—
DCF PCOPY Sx,Dz	if DC = 0, Sx→Dz if 1, nop	111110********** 11011011xx00zzzz	1	—
DCF PCOPY Sy,Dz	if DC = 0, Sy→Dz if 1, nop	111110********** 1111101100yyzzzz	1	—

Description: Stores the Sx and Sy operands in the Dz operand. When conditions are specified for DCT and DCF, the instruction is executed when those conditions are TRUE. When they are FALSE, the instruction is not executed.

When conditions are not specified, the DC bit of the DSR register is updated according to the specifications for the CS bits. The N, Z, V, and GT bits are also updated. The DC, N, Z, V, and GT bits are not updated when conditions are specified, even if the conditions are TRUE.

Operation:

```
{   /* Sx + 0 -> Dz */
    if (EX2_DSP_BIT13==0) {        /* Sx + 0 -> Dz */
        switch (EX2_SX) {
            case 0x0:  DSP_ALU_SRC1  = X0;
                    if (DSP_ALU_SRC1_MSB) DSP_ALU_SRC1G = 0xff;
                    else             DSP_ALU_SRC1G = 0x0;
                    break;
            case 0x1:  DSP_ALU_SRC1  = X1;
                    if (DSP_ALU_SRC1_MSB) DSP_ALU_SRC1G = 0xff;
                    else             DSP_ALU_SRC1G = 0x0;
                    break;
            case 0x2:  DSP_ALU_SRC1  = A0;
                    DSP_ALU_SRC1G = A0G;
```

```
                break;

            case 0x3:  DSP_ALU_SRC1  = A1;
                DSP_ALU_SRC1G = A1G;
                break;
        }
        DSP_ALU_SRC2 = 0;
        DSP_ALU_SRC2G= 0;
    }
    else  {        /* 0 + Sy -> Dz */
        DSP_ALU_SRC1 = 0;
        DSP_ALU_SRC1G= 0;
        switch (EX2_SY) {
            case 0x0:  DSP_ALU_SRC2  = Y0;
                break;
            case 0x1:  DSP_ALU_SRC2  = Y1;
                break;
            case 0x2:  DSP_ALU_SRC2  = M0;
                break;
            case 0x3:  DSP_ALU_SRC2  = M1;
                break;
        }
        if (DSP_ALU_SRC2_MSB)      DSP_ALU_SRC2G = 0xff;
        else              DSP_ALU_SRC2G = 0x0;
    }

    DSP_ALU_DST = DSP_ALU_SRC1 + DSP_ALU_SRC2;
    carry_bit = ((DSP_ALU_SRC1_MSB | DSP_ALU_SRC2_MSB) & !DSP_ALU
_DST_MSB) |
         (DSP_ALU_SRC1_MSB & DSP_ALU_SRC2_MSB);
    DSP_ALU_DSTG_LSB8 = DSP_ALU_SRC1G_LSB8 + DSP_ALU_SRC2G_LSB8 + carry_bit;

    overflow_bit= PLUS_OP_G_OV || !(POS_NOT_OV || NEG_NOT_OV);
#include "fixed_pt_overflow_protection.c"

    if(DSP_UNCONDITIONAL_UPDATE) { /* unconditional operation */
#include "fixed_pt_unconditional_update.c"
#include "fixed_pt_plus_dc_bit.c"
```

```
    }
    else if(DSP_CONDITION_MATCH) { /* conditional operation and match */
        DSP_REG[ex2_dz_no] = DSP_ALU_DST;
        if(ex2_dz_no==0) {
            A0G = DSP_ALU_DSTG & MASK000000FF;
            if(DSP_ALU_DSTG_BIT7) A0G = A0G | MASKFFFFFF00;
        }
        else if(ex2_dz_no==1) {
            A1G = DSP_ALU_DSTG & MASK000000FF;
            if(DSP_ALU_DSTG_BIT7) A1G = A1G | MASKFFFFFF00;
        }
    }
}

break;
```

Examples:

`PCOPY X0, A0 NOPX NOPY`	;Before execution:	X0 = H'55555555, A0 = H'FFFFFFFF
	;After execution:	X0 = H'55555555, A0 = H'0055555555

In case of unconditional execution, the DC bit is updated depending on the state of CS [2:0].

8.5.9 [if cc] PDEC (Decrement by 1): DSP Arithmetic Operation Instruction

Format	Abstract	Code	Cycle	DC Bit
PDEC Sx,Dz	MSW of Sx–1→MSW of Dz, clear LSW of Dz	111110********** 10001001xx00zzzz	1	Update
PDEC Sy,Dz	MSW of Sy–1→MSW of Dz, clear LSW of Dz	111110********** 1010100100yyzzzz	1	Update
DCT PDEC Sx,Dz	If DC = 1, MSW of Sx–1→ MSW of Dz, clear LSW of Dz; if 0, nop	111110********** 10001010xx00zzzz	1	—
DCT PDEC Sy,Dz	If DC = 1, MSW of Sy–1→ MSW of Dz, clear LSW of Dz; if 0, nop	111110********** 1010101000yyzzzz	1	—
DCF PDEC Sx,Dz	If DC = 0, MSW of Sx–1→ MSW of Dz, clear LSW of Dz; if 1, nop	111110********** 10001011xx00zzzz	1	—
DCF PDEC Sy,Dz	If DC = 0, MSW of Sy–1→ MSW of Dz, clear LSW of Dz; if 1, nop	111110********** 1010101100yyzzzz	1	—

Description: Subtracts 1 from the top word of the Sx and Sy operands, stores the result in the upper word of the Dz operand, and clears the bottom word of the Dz operand with zeros. When conditions are specified for DCT and DCF, the instruction is executed when those conditions are TRUE. When they are FALSE, the instruction is not executed.

When conditions are not specified, the DC bit of the DSR register is updated according to the specifications for the CS bits. The N, Z, V, and GT bits of the DSR register are also updated. The DC, N, Z, V, and GT bits are not updated when conditions are specified, even if the conditions are TRUE.

Note: The bottom word of the destination register is ignored when the DC bit is updated.

Operation:

```
{
    DSP_ALU_SRC2 = 0x1;
    DSP_ALU_SRC2G= 0x0;
    if (EX2_DSP_BIT13==0) {  /* MSW of Sx -1 -> Dz */
        switch (EX2_SX) {
        case 0x0:  DSP_ALU_SRC1  = X0;
                if (DSP_ALU_SRC1_MSB) DSP_ALU_SRC1G = 0xff;
                else    DSP_ALU_SRC1G = 0x0;
```

```
                break;
        case 0x1:  DSP_ALU_SRC1  = X1;
                if (DSP_ALU_SRC1_MSB) DSP_ALU_SRC1G = 0xff;
                else       DSP_ALU_SRC1G = 0x0;
                break;
        case 0x2:  DSP_ALU_SRC1  = A0;
                DSP_ALU_SRC1G = A0G;
                break;
        case 0x3:  DSP_ALU_SRC1  = A1;
                DSP_ALU_SRC1G = A1G;
                break;
    }
}
else {                /* MSW of Sy -1 -> Dz */
        switch (EX2_SY) {
        case 0x0:  DSP_ALU_SRC1  = Y0;
                break;
        case 0x1:  DSP_ALU_SRC1  = Y1;
                break;
        case 0x2:  DSP_ALU_SRC1  = M0;
                break;
        case 0x3:  DSP_ALU_SRC1  = M1;
                break;
        }

        if (DSP_ALU_SRC1_MSB) DSP_ALU_SRC1G = 0xff;
        else       DSP_ALU_SRC1G = 0x0;
}

    DSP_ALU_DST_HW = DSP_ALU_SRC1_HW - 1;
    carry_bit =((DSP_ALU_SRC1_MSB | !DSP_ALU_SRC2_MSB) && !DSP_ALU
_DST_MSB) |
         (DSP_ALU_SRC1_MSB & !DSP_ALU_SRC2_MSB);
    borrow_bit = !carry_bit;
    DSP_ALU_DSTG_LSB8 = DSP_ALU_SRC1G_LSB8 - DSP_ALU_SRC2G_LSB8 -
borrow_bit;

    overflow_bit= PLUS_OP_G_OV || !(POS_NOT_OV || NEG_NOT_OV);
```

```
#include "integer_overflow_protection.c"

    if(DSP_UNCONDITIONAL_UPDATE) { /* unconditional operation */
#include "integer_unconditional_update.c"
#include "integer_minus_dc_bit.c"
    }
    else if(DSP_CONDITION_MATCH) { /* conditional operation and match */
        DSP_REG_WD[ex2_dz_no*2] = DSP_ALU_DST_HW;
        DSP_REG_WD[ex2_dz_no*2+1] = 0x0;  /* clear LSW */
        if(ex2_dz_no==0) {
            A0G = DSP_ALU_DSTG & MASK000000FF;
            if(DSP_ALU_DSTG_BIT7) A0G = A0G | MASKFFFFFF00;
        }
        else if(ex2_dz_no==1) {
            A1G = DSP_ALU_DSTG & MASK000000FF;
            if(DSP_ALU_DSTG_BIT7) A1G = A1G | MASKFFFFFF00;
        }
    }
}

break;
```

Example:

`PDEC X0,M0  NOPX  NOPY`	;Before execution:	X0 = H'0052330F, M0 = H'12345678
	;After execution:	X0 = H'0052330F, M0 = H'00510000
`PDEC X1,X1  NOPX  NOPY`	;Before execution:	X1 = H'FC342855
	;After execution:	X1 = H'FC330000

In case of unconditional execution, the DC bit is updated depending on the state of CS [2:0].

8.5.10 [if cc] PDMSB (Detect MSB with Condition): DSP Arithmetic Operation Instruction

Format	Abstract	Code	Cycle	DC Bit
`PDMSB Sx,Dz`	Sx data MSB position → MSW of Dz, clear LSW of Dz	111110********** 10011101xx00zzzz	1	Update
`PDMSB Sy,Dz`	Sy data MSB position → MSW of Dz, clear LSW of Dz	111110********** 1011110100yyzzzz	1	Update
`DCT PDMSB Sx,Dz`	If DC = 1, Sx data MSB position → MSW of Dz, clear LSW of Dz; if 0, nop	111110********** 10011110xx00zzzz	1	—
`DCT PDMSB Sy,Dz`	If DC = 1, Sy data MSB position → MSW of Dz, clear LSW of Dz; if 0, nop	111110********** 1011111000yyzzzz	1	—
`DCF PDMSB Sx,Dz`	If DC = 0, Sx data MSB position → MSW of Dz, clear LSW of Dz; if 1, nop	111110********** 10011111xx00zzzz	1	—
`DCF PDMSB Sy,Dz`	If DC = 0, Sy data MSB position → MSW of Dz, clear LSW of Dz; if 1, nop	111110********** 1011111100yyzzzz	1	—

Description: Finds the first position to change in the lineup of Sx and Sy operand bits and stores the bit position in the Dz operand. When conditions are specified for DCT and DCF, the instruction is executed when those conditions are TRUE. When they are FALSE, the instruction is not executed.

When conditions are not specified, the DC bit of the DSR register is updated according to the specifications for the CS bits. The N, Z, V, and GT bits of the DSR register are also updated. The DC, N, Z, V, and GT bits are not updated when conditions are specified, even if the conditions are TRUE.

Operation:

```
{
    DSP_ALU_SRC2 = 0x0;
    DSP_ALU_SRC2G= 0x0;
    if (EX2_DSP_BIT13==0) { /*  msb(Sx) -> Dz */
        switch (EX2_SX) {
        case 0x0:  DSP_ALU_SRC1  = X0;
                   if (DSP_ALU_SRC1_MSB) DSP_ALU_SRC1G = 0xff;
                   else   DSP_ALU_SRC1G = 0x0;
                   break;
```

```
    case 0x1:  DSP_ALU_SRC1  = X1;
               if (DSP_ALU_SRC1_MSB) DSP_ALU_SRC1G = 0xff;
               else   DSP_ALU_SRC1G = 0x0;
               break;
    case 0x2:  DSP_ALU_SRC1  = A0;
               DSP_ALU_SRC1G = A0G;
               break;
    case 0x3:  DSP_ALU_SRC1  = A1;
               DSP_ALU_SRC1G = A1G;
               break;
    }
}
else {         /* msb(Sy) -> Dz */
    switch (EX2_SY) {
    case 0x0:  DSP_ALU_SRC1  = Y0;
               break;
    case 0x1:  DSP_ALU_SRC1  = Y1;
               break;
    case 0x2:  DSP_ALU_SRC1  = M0;
               break;
    case 0x3:  DSP_ALU_SRC1  = M1;
               break;
}
if (DSP_ALU_SRC1_MSB) DSP_ALU_SRC1G = 0xff;
else                 DSP_ALU_SRC1G = 0x0;
}
{
short int i;
unsigned char msb, src1g;
unsigned long src1=DSP_ALU_SRC1;
msb= DSP_ALU_SRC1G_BIT7;
src1g=(DSP_ALU_SRC1G_LSB8 << 1);
for(i=38;((msb==(src1g>>7))&&(i>=32));i--) { src1g <<= 1; }
if(i==31)  {
    for(i;((msb==(src1>>31))&&(i>=0));i--) { src1 <<= 1; }
}

    DSP_ALU_DST = 0x0;
```

```
        DSP_ALU_DST_HW = (short int) (30-i);
        if (DSP_ALU_DST_MSB)       DSP_ALU_DSTG_LSB8 = 0xff;
        else              DSP_ALU_DSTG_LSB8 = 0x0;
    }

    carry_bit = 0;

    if(DSP_UNCONDITIONAL_UPDATE) { /* unconditional operation */
        overflow_bit= 0;
#include "integer_unconditional_update.c"
#include "integer_plus_dc_bit.c"
    }
    else if(DSP_CONDITION_MATCH) { /* conditional operation and match */
        DSP_REG_WD[ex2_dz_no*2] = DSP_ALU_DST_HW;
        DSP_REG_WD[ex2_dz_no*2+1] = 0x0;        /* clear LSW */
        if(ex2_dz_no==0) {
                A0G = DSP_ALU_DSTG & MASK000000FF;
                if(DSP_ALU_DSTG_BIT7) A0G = A0G | MASKFFFFFF00;
        }
        else if(ex2_dz_no==1) {
                A1G = DSP_ALU_DSTG & MASK000000FF;
                if(DSP_ALU_DSTG_BIT7) A1G = A1G | MASKFFFFFF00;
        }
    }
}

break;
```

Example:

PDMSB X0,M0 NOPX NOPY	;Before execution:	X0 = H'0052330F, M0 = H'12345678
	;After execution:	X0 = H'0052330F, M0 = H'00080000
PDMSB X1,X1 NOPX NOPY	;Before execution:	X1 = H'FC342855
	;After execution:	X1 = H'00050000

In case of unconditional execution, the DC bit is updated depending on the state of CS [2:0].

8.5.11 [if cc] PINC (Increment by 1 with Condition): DSP Arithmetic Operation Instruction

Format	Abstract	Code	Cycle	DC Bit
`PINC Sx,Dz`	MSW of Sx + 1→ MSW of Dz, clear LSW of Dz	111110********** 10011001xx00zzzz	1	Update
`PINC Sy,Dz`	MSW of Sy + 1→ MSW of Dz, clear LSW of Dz	111110********** 1011100100yyzzzz	1	Update
`DCT PINC Sx,Dz`	If DC = 1, MSW of Sx + 1→ MSW of Dz, clear LSW of Dz; if 0, nop	111110********** 10011010xx00zzzz	1	—
`DCT PINC Sy,Dz`	If DC = 1, MSW of Sy + 1→ MSW of Dz, clear LSW of Dz; if 0, nop	111110********** 1011101000yyzzzz	1	—
`DCF PINC Sx,Dz`	If DC = 0, MSW of Sx + 1→ MSW of Dz, clear LSW of Dz; if 1, nop	111110********** 10011011xx00zzzz	1	—
`DCF PINC Sy,Dz`	If DC = 0, MSW of Sy + 1→ MSW of Dz, clear LSW of Dz; if 1, nop	111110********** 1011101100yyzzzz	1	—

Description: Adds 1 to the top word of the Sx and Sy operands, stores the result in the upper word of the Dz operand, and clears the bottom word of the Dz operand with zeros. When conditions are specified for DCT and DCF, the instruction is executed when those conditions are TRUE. When they are FALSE, the instruction is not executed.

When conditions are not specified, the DC bit of the DSR register is updated according to the specifications for the CS bits. The N, Z, V, and GT bits of the DSR register are also updated. The DC, N, Z, V, and GT bits are not updated when conditions are specified, even if the conditions are TRUE.

Note: The bottom word of the destination register is ignored when the DC bit is updated.

Operation:

```
{
    DSP_ALU_SRC2 = 0x1;
    DSP_ALU_SRC2G= 0x0;
    if (EX2_DSP_BIT13==0) { /* MSW of Sx +1 -> Dz */
        switch (EX2_SX) {
            case 0x0:  DSP_ALU_SRC1 = X0;
                    if (DSP_ALU_SRC1_MSB) DSP_ALU_SRC1G = 0xff;
```

```
                else     DSP_ALU_SRC1G = 0x0;
                break;

        case 0x1:  DSP_ALU_SRC1    = X1;
                if (DSP_ALU_SRC1_MSB) DSP_ALU_SRC1G = 0xff;
                else     DSP_ALU_SRC1G = 0x0;
                break;
        case 0x2:  DSP_ALU_SRC1    = A0;
                DSP_ALU_SRC1G   = A0G;
                break;
        case 0x3:  DSP_ALU_SRC1    = A1;
                DSP_ALU_SRC1G   = A1G;
                break;
        }
    }
    else {            /* MSW of Sy +1 -> Dz */
        switch (EX2_SY) {
        case 0x0:  DSP_ALU_SRC1  = Y0;
              break;
        case 0x1:  DSP_ALU_SRC1  = Y1;
              break;
        case 0x2:  DSP_ALU_SRC1  = M0;
              break;
        case 0x3:  DSP_ALU_SRC1  = M1;
              break;
         }
         if (DSP_ALU_SRC1_MSB)           DSP_ALU_SRC1G = 0xff;
         else            DSP_ALU_SRC1G = 0x0;
    }

    DSP_ALU_DST_HW = DSP_ALU_SRC1_HW + 1;
    carry_bit = ((DSP_ALU_SRC1_MSB | DSP_ALU_SRC2_MSB) & !DSP_ALU
_DST_MSB) |
        (DSP_ALU_SRC1_MSB & DSP_ALU_SRC2_MSB);
    DSP_ALU_DSTG_LSB8 = DSP_ALU_SRC1G_LSB8 + DSP_ALU_SRC2G_LSB8 +
carry_bit;

    overflow_bit= PLUS_OP_G_OV || !(POS_NOT_OV || NEG_NOT_OV);
```

```
#include "integer_overflow_protection.c"
    if(DSP_UNCONDITIONAL_UPDATE) { /* unconditional operation */
#include "integer_unconditional_update.c"
#include "integer_plus_dc_bit.c"
    }
    else if(DSP_CONDITION_MATCH) { /* conditional operation and match
*/
        DSP_REG_WD[ex2_dz_no*2] = DSP_ALU_DST_HW;
        DSP_REG_WD[ex2_dz_no*2+1] = 0x0;        /* clear LSW */
        if(ex2_dz_no==0) {
                A0G = DSP_ALU_DSTG & MASK000000FF;
                if(DSP_ALU_DSTG_BIT7) A0G = A0G | MASKFFFFFF00;
        }
        else if(ex2_dz_no==1) {
                A1G = DSP_ALU_DSTG & MASK000000FF;
                if(DSP_ALU_DSTG_BIT7) A1G = A1G | MASKFFFFFF00;
        }
    }
}

break;
```

Example:

PINC X0,M0 NOPX NOPY	;Before execution:	X0 = H'0052330F, M0 = H'12345678
	;After execution:	X0 = H'0052330F, M0 = H'00530000
PINC X1,X1 NOPX NOPY	;Before execution:	X1 = H'FC342855
	;After execution:	X1 = H'FC350000

In case of unconditional execution, the DC bit is updated depending on the state of CS [2:0].

8.5.12 [if cc] PLDS (Load System Register): DSP System Control Instruction

Format	Abstract	Code	Cycle	DC Bit
PLDS Dz,MACH	Dz→MACH	111110********** 111011010000zzzz	1	—
PLDS Dz,MACL	Dz→MACL	111110********** 111111010000zzzz	1	—
DCT PLDS Dz,MACH	if DC = 1, Dz→MACH if 0, nop	111110********** 111011100000zzzz	1	—
DCT PLDS Dz,MACL	if DC = 1, Dz→MACL if 0, nop	111110********** 111111100000zzzz	1	—
DCF PLDS Dz,MACH	if DC = 0, Dz→MACH if 1, nop	111110********** 111011110000zzzz	1	—
DCF PLDS Dz,MACL	if DC = 0, Dz→MACL if 1, nop	111110********** 111111110000zzzz	1	—

Description: Stores the Dz operand in the MACH and MACL registers. When conditions are specified for DCT and DCF, the instruction is executed when those conditions are TRUE. When they are FALSE, the instruction is not executed.

The DC, N, Z, V, and GT bits of the DSR register are not updated.

Note: Though PSTS, MOVX, and MOVY can be designated in parallel, their execution may take two cycles.

Operation:

```
{   /* Dz -> MACH */
    if(DSP_UNCONDITIONAL_UPDATE) { /* unconditional operation */
        MACH = DSP_REG[ex2_dz_no] ;
    }
    else if(DSP_CONDITION_MATCH) { /* conditional operation and match */
        MACH = DSP_REG[ex2_dz_no] ;
    }
}

break;

/*  SH-DSP: DSP Engine:  Local Data Move Operation: LoaD System register
```

```
    plds_macl.c
    rev 1.0 24 May 1995, EY  */

{   /* Dz -> MACL */
    if(DSP_UNCONDITIONAL_UPDATE) { /* unconditional operation */
        MACL = DSP_REG[ex2_dz_no] ;
    }
    else if(DSP_CONDITION_MATCH) { /* conditional operation and match */
        MACL = DSP_REG[ex2_dz_no] ;
    }
}

break;
```

Example:

```
PLDS A0,MACH  NOPX NOPY   ;Before execution:   A0 = H'123456789A, MACH = H'66666666
                          ;After execution:    A0 = H'123456789A, MACH = H'3456789A
```

8.5.13 PMULS (Multiply Signed by Signed): DSP Arithmetic Operation Instruction

Format	Abstract	Code	Cycle	DC Bit
PMULS Se,Sf,Dg	MSW of Se × MSW of Sf→Dg	111110********** 0100eeff0000gg00	1	—

Description: The contents of the top word of the Se and Sf operands are multiplied as signed and the result stored in the Dg operand. The DC, N, Z, V, and GT bits of the DSR register are not updated.

Note: Since PMULS is fixed decimal point multiplication, the operation result is different from that of MULS even thou**gh the source data is the same.**

Examples:

PMULS X0,Y0,M0 NOPX NOPY	;Before execution:	X0 = H'00010000, Y0 = H'00020000, M0 = H'33333333
	;After execution:	X0 = H'00010000, Y0 = H'00020000, M0 = H'00000004
PMULS X1,Y1,A0 NOPX NOPY	;Before execution:	X1 = H'FFFE2222, Y1 = H'0001AAAA, A0 = H'4444444444
	;After execution:	X1 = H'FFFE2222, Y1 = H'0001AAAA, A0 = H'FFFFFFFFFC

8.5.14 [if cc] PNEG (Negate): DSP Arithmetic Operation Instruction

Format	Abstract	Code	Cycle	DC Bit
PNEG Sx,Dz	0 – Sx→Dz	111110********** 11001001xx00zzzz	1	Update
PNEG Sy,Dz	0 – Sy→Dz	111110********** 1110100100yyzzzz	1	Update
DCT PNEG Sx,Dz	if DC = 1, 0 – Sx→Dz if 0, nop	111110********** 11001010xx00zzzz	1	—
DCT PNEG Sy,Dz	if DC = 1, 0 – Sy→Dz if 0, nop	111110********** 1110101000yyzzzz	1	—
DCF PNEG Sx,Dz	if DC = 0, 0 – Sx→Dz if 1, nop	111110********** 11001011xx00zzzz	1	—
DCF PNEG Sy,Dz	if DC = 0, 0 – Sy→Dz if 1, nop	111110********** 1110101100yyzzzz	1	—

Description: Reverses the sign. Subtracts the Sx and Sy operands from 0 and stores the result in the Dz operand. When conditions are specified for DCT and DCF, the instruction is executed when those conditions are TRUE. When they are FALSE, the instruction is not executed.

When conditions are not specified, the DC bit of the DSR register is updated according to the specifications for the CS bits. The N, Z, V, and GT bits of the DSR register are also updated. The DC, N, Z, V, and GT bits are not updated when conditions are specified, even if the conditions are TRUE.

Operation:

```
{
    DSP_ALU_SRC1 = 0;
    DSP_ALU_SRC1G= 0;
    if (EX2_DSP_BIT13==0) {         /* 0 - Sx -> Dz */
        switch (EX2_SX) {
            case 0x0:  DSP_ALU_SRC2  = X0;
                    if (DSP_ALU_SRC2_MSB) DSP_ALU_SRC2G = 0xff;
                    else                  DSP_ALU_SRC2G = 0x0;
                    break;

            case 0x1:  DSP_ALU_SRC2  = X1;
                    if (DSP_ALU_SRC2_MSB) DSP_ALU_SRC2G = 0xff;
```

```
                else                DSP_ALU_SRC2G = 0x0;
                break;
            case 0x2:  DSP_ALU_SRC2  = A0;
                DSP_ALU_SRC2G = A0G;
                break;
            case 0x3:  DSP_ALU_SRC2  = A1;
                DSP_ALU_SRC2G = A1G;
                break;
        }
    }
    else  {        /* 0 - Sy -> Dz */
        switch (EX2_SY) {
            case 0x0:  DSP_ALU_SRC2  = Y0;
                break;
            case 0x1:  DSP_ALU_SRC2  = Y1;
                break;
            case 0x2:  DSP_ALU_SRC2  = M0;
                break;
            case 0x3:  DSP_ALU_SRC2  = M1;
                break;
        }
        if (DSP_ALU_SRC2_MSB)      DSP_ALU_SRC2G = 0xff;
        else           DSP_ALU_SRC2G = 0x0;
    }

        DSP_ALU_DST = DSP_ALU_SRC1 - DSP_ALU_SRC2;
        carry_bit =((DSP_ALU_SRC1_MSB | !DSP_ALU_SRC2_MSB)
&& !DSP_ALU_DST_MSB) |
                (DSP_ALU_SRC1_MSB & !DSP_ALU_SRC2_MSB);
        borrow_bit = !carry_bit;
        DSP_ALU_DSTG_LSB8 = DSP_ALU_SRC1G_LSB8 - DSP_ALU_SRC2G_LSB8 -
borrow_bit;
        overflow_bit= MINUS_OP_G_OV || !(POS_NOT_OV || NEG_NOT_OV);
#include "fixed_pt_overflow_protection.c"

        if(DSP_UNCONDITIONAL_UPDATE) { /* unconditional operation */
#include "fixed_pt_unconditional_update.c"
#include "fixed_pt_minus_dc_bit.c"
```

```
        }
        else if(DSP_CONDITION_MATCH) { /* conditional operation and match */
        DSP_REG[ex2_dz_no] = DSP_ALU_DST;
        if(ex2_dz_no==0) {
                A0G = DSP_ALU_DSTG & MASK000000FF;
                if(DSP_ALU_DSTG_BIT7) A0G = A0G | MASKFFFFFF00;
        }
        else if(ex2_dz_no==1) {
                A1G = DSP_ALU_DSTG & MASK000000FF;
                if(DSP_ALU_DSTG_BIT7) A1G = A1G | MASKFFFFFF00;
        }
        }
}

break;
```

Examples:

`PNEG X0,A0 NOPX NOPY`	;Before execution:	X0 = H'55555555, A0 = H'A987654321
	;After execution:	X0 = H'55555555, A0 = H'FFAAAAAAAB
`PNEG X1,Y1 NOPX NOPY`	;Before execution:	Y1 = H'99999999
	;After execution:	Y1 = H'66666667

In case of unconditional execution, the DC bit is updated depending on the state of CS [2:0].

8.5.15 [if cc] POR (Logical OR): DSP Logical Operation Instruction

Format	Abstract	Code	Cycle	DC Bit
`POR Sx,Sy,Dz`	Sx | Sy→Dz, clear LSW of Dz	111110********** 10110101xxyyzzzz	1	Update
`DCT POR Sx,Sy,Dz`	If DC = 1, Sx | Sy→Dz, clear LSW of Dz; if 0, nop	111110********** 10110110xxyyzzzz	1	—
`DCF POR Sx,Sy,Dz`	If DC = 0, Sx | Sy→Dz, clear LSW of Dz; if 1, nop	111110********** 10110111xxyyzzzz	1	—

Description: Takes the OR of the top word of the Sx operand and the top word of the Sy operand, stores the result in the top word of the Dz operand, and clears the bottom word of Dz with zeros. When Dz is a register that has guard bits, the guard bits are also zeroed. When conditions are specified for DCT and DCF, the instruction is executed when those conditions are TRUE. When they are FALSE, the instruction is not executed.

When conditions are not specified, the DC bit of the DSR register is updated according to the specifications for the CS bits. The N, Z, V, and GT bits of the DSR register are also updated. The DC, N, Z, V, and GT bits are not updated when conditions are specified, even if the conditions are TRUE.

Note: The bottom word of the destination register and the guard bits are ignored when the DC bit is updated.

Operation:

```
{
    switch (EX2_SX) {
        case 0x0:  DSP_ALU_SRC1  = X0;
                   break;
        case 0x1:  DSP_ALU_SRC1  = X1;
                   break;
        case 0x2:  DSP_ALU_SRC1  = A0;
                   break;
        case 0x3:  DSP_ALU_SRC1  = A1;
                   break;
    }

    switch (EX2_SY) {
        case 0x0:  DSP_ALU_SRC2  = Y0;
```

```
                break;
        case 0x1:  DSP_ALU_SRC2  = Y1;
                break;
        case 0x2:  DSP_ALU_SRC2  = M0;
                break;
        case 0x3:  DSP_ALU_SRC2  = M1;
                break;
    }

    DSP_ALU_DST_HW = DSP_ALU_SRC1_HW | DSP_ALU_SRC2_HW;

    if(DSP_UNCONDITIONAL_UPDATE) { /* unconditional operation */
        DSP_REG_WD[ex2_dz_no*2] = DSP_ALU_DST_HW;
        DSP_REG_WD[ex2_dz_no*2+1] = 0x0;        /* clear LSW */
        if (ex2_dz_no==0) A0G = 0x0;    /* clear Guard
bits */
        else if (ex2_dz_no==1)     A1G = 0x0;

        carry_bit      = 0x0;
        negative_bit   = DSP_ALU_DST_MSB;
        zero_bit       = (DSP_ALU_DST_HW==0);
        overflow_bit   = 0x0;
#include "logical_dc_bit.c"
    }
    else if(DSP_CONDITION_MATCH) { /* conditional operation and match */
        DSP_REG_WD[ex2_dz_no*2] = DSP_ALU_DST_HW;
        DSP_REG_WD[ex2_dz_no*2+1] = 0x0;  /* clear LSW */
        if (ex2_dz_no==0) A0G = 0x0;      /* clear Guard
bits */
        else if (ex2_dz_no==1)     A1G = 0x0;
    }
}

break;
```

Example:

```
POR X0,Y0,A0  NOPX NOPY   ;Before execution:  X0 = H'33333333, Y0 = H'55555555
                                              A0 = H'123456789A
                          ;After execution:   X0 = H'33333333, Y0 = H'55555555
                                              A0 = H'127777789A
```

In case of unconditional execution, the DC bit is updated depending on the state of CS [2:0].

8.5.16 PRND (Rounding): DSP Arithmetic Operation Instruction

Format	Abstract	Code	Cycle	DC Bit
PRND Sx,Dz	Sx + H'00008000→Dz clear LSW of Dz	111110********** 10011000xx00zzzz	1	Update
PRND Sy,Dz	Sy + H'00008000→Dz clear LSW of Dz	111110********** 1011100000yyzzzz	1	Update

Description: Does rounding. Adds the immediate data H'00008000 to the contents of the Sx and Sy operands, stores the result in the upper word of the Dz operand, and clears the bottom word of Dz with zeros.

The DC bit of the DSR register is updated according to the specifications for the CS bits. The N, Z, V, and GT bits of the DSR register are also updated.

Operation:

```
{
    DSP_ALU_SRC2 = 0x00008000;
    DSP_ALU_SRC2G= 0x0;
    if (EX2_DSP_BIT13==0) { /* Sx + H'00008000 -> Dz; clr Dz
LW */
        switch (EX2_SX) {
            case 0x0:  DSP_ALU_SRC1  = X0;
                    if (DSP_ALU_SRC1_MSB) DSP_ALU_SRC1G = 0xff;
                    else             DSP_ALU_SRC1G = 0x0;
                    break;
            case 0x1:  DSP_ALU_SRC1  = X1;
                    if (DSP_ALU_SRC1_MSB) DSP_ALU_SRC1G = 0xff;
                    else             DSP_ALU_SRC1G = 0x0;
                    break;
            case 0x2:  DSP_ALU_SRC1  = A0;
                    DSP_ALU_SRC1G = A0G;
                    break;
            case 0x3:  DSP_ALU_SRC1  = A1;
                    DSP_ALU_SRC1G = A1G;
                    break;
        }
    }
```

```
    else  {       /* Sy + H'00008000 -> Dz; clr Dz LW */
        switch (EX2_SY) {
            case 0x0:  DSP_ALU_SRC1  = Y0;
                    break;
            case 0x1:  DSP_ALU_SRC1  = Y1;
                    break;
            case 0x2:  DSP_ALU_SRC1  = M0;
                    break;
            case 0x3:  DSP_ALU_SRC1  = M1;
                    break;
        }
        if (DSP_ALU_SRC1_MSB)      DSP_ALU_SRC1G = 0xff;
        else           DSP_ALU_SRC1G = 0x0;
    }

    DSP_ALU_DST = (DSP_ALU_SRC1 + DSP_ALU_SRC2) & MASKFFFF0000;
    carry_bit = ((DSP_ALU_SRC1_MSB | DSP_ALU_SRC2_MSB) & !DSP_ALU
_DST_MSB) |
        (DSP_ALU_SRC1_MSB & DSP_ALU_SRC2_MSB);
    DSP_ALU_DSTG_LSB8 = DSP_ALU_SRC1G_LSB8 + DSP_ALU_SRC2G_LSB8 +
carry_bit;

        overflow_bit= PLUS_OP_G_OV || !(POS_NOT_OV || NEG_NOT_OV);

#include "fixed_pt_overflow_protection.c"
#include "fixed_pt_unconditional_update.c"
#include "fixed_pt_plus_dc_bit.c"
}

break;
```

Example:

```
PRND X0,M0  NOPX  NOPY    ;Before execution:   X0 = H'0052330F, M0 = H'12345678
                          ;After execution:    X0 = H'0052330F, M0 = H'00520000
PRND X1,X1  NOPX  NOPY    ;Before execution:   X1 = H'FC34C087
                          ;After execution:    X1 = H'FC350000
                          DC bit is updated depending on the state of CS [2:0].
```

8.5.17 [if cc] PSHA (Shift Arithmetically with Condition): DSP Arithmetic Shift Instruction

Format	Abstract	Code	Cycle	DC Bit
`PSHA Sx,Sy,Dz`	if Sy> = 0, Sx<<Sy→Dz if Sy<0, Sx>>Sy–>Dz	`111110**********` `10010001xxyyzzzz`	1	Update
`DCT PSHA Sx,Sy,Dz`	if DC = 1 & Sy> = 0, Sx<<Sy→Dz if DC = 1 & Sy<0, Sx>>Sy→Dz if DC = 0, nop	`111110**********` `10010010xxyyzzzz`	1	Update
`DCF PSHA Sx,Sy,Dz`	if DC = 0 & Sy> = 0, Sx<<Sy–>Dz if DC = 0 & Sy<0, Sx>>Sy→Dz if DC = 1, nop	`111110**********` `10010011xxyyzzzz`	1	—
`PSHA #Imm,Dz`	if Imm> = 0, Dz<<Imm→Dz if Imm<0, Dz>>Imm→Dz	`111110**********` `00010iiiiiiizzzz`	1	—

Description: Arithmetically shifts the contents of the Sx or Dz operand and stores the result in the Dz operand. The amount of the shift is specified by the Sy operand or the immediate value Imm operand. When the shift amount is positive, it shifts left. When the shift amount is negative, it shifts right. When conditions are specified for DCT and DCF, the instruction is executed when those conditions are TRUE. When they are FALSE, the instruction is not executed.

When conditions are not specified, the DC bit of the DSR register is updated according to the specifications for the CS bits. The N, Z, V, and GT bits of the DSR register are also updated. The DC, N, Z, V, and GT bits are not updated when conditions are specified, even if the conditions are TRUE.

Operation:

< When register operand is used >

```
{
    switch (EX2_SX) {
        case 0x0:  DSP_ALU_SRC1     = X0;
                   if (DSP_ALU_SRC1_MSB) DSP_ALU_SRC1G = 0xff;
                   else             DSP_ALU_SRC1G = 0x0;
                   break;
```

```
        case 0x1:  DSP_ALU_SRC1    = X1;
                   if (DSP_ALU_SRC1_MSB) DSP_ALU_SRC1G = 0xff;
                   else              DSP_ALU_SRC1G = 0x0;
                   break;
        case 0x2:  DSP_ALU_SRC1    = A0;
                   DSP_ALU_SRC1G   = A0G;
                   break;
        case 0x3:  DSP_ALU_SRC1    = A1;
                   DSP_ALU_SRC1G   = A1G;
                   break;
    }
    switch (EX2_SY) {
        case 0x0:  DSP_ALU_SRC2    = Y0 & MASK007F0000;
                   break;
        case 0x1:  DSP_ALU_SRC2    = Y1 & MASK007F0000;
                   break;
        case 0x2:  DSP_ALU_SRC2    = M0 & MASK007F0000;
                   break;
        case 0x3:  DSP_ALU_SRC2    = M1 & MASK007F0000;
                   break;
    }
    if (DSP_ALU_SRC2_MSB) DSP_ALU_SRC2G = 0xff;
    else              DSP_ALU_SRC2G = 0x0;

    if((DSP_ALU_SRC2_HW & MASK0040)==0) {        /* Left Shift
0<=cnt<=32 */
        char cnt = (DSP_ALU_SRC2_HW & MASK003F);
        if(cnt > 32) {
            printf("¥nPSHA Sz,Sy,Dz Error! Shift %2X exceed range.
¥n",cnt);
            exit();
        }
        DSP_ALU_DST  = DSP_ALU_SRC1 << cnt;
        DSP_ALU_DSTG = ((DSP_ALU_SRC1G << cnt) |
                   (DSP_ALU_SRC1 >> (32-cnt))) & MASK000000FF;
        carry_bit = ((DSP_ALU_DSTG & MASK00000001)==0x1);
    }
```

```
    else   {                /* Right Shift 0< cnt <=32 */
        char cnt = ((~DSP_ALU_SRC2_HW & MASK003F)+1);
        if(cnt > 32) {
            printf("¥nPSHA Sz,Sy,Dz Error! shift -%2X exceed range.¥n",cnt);
            exit();
        }
        if((cnt>8) && DSP_ALU_SRC1G_BIT7) { /* MSB copy */
            DSP_ALU_DST=((DSP_ALU_SRC1>>8) | (DSP_ALU_SRC1G<<
(32-8)));
            DSP_ALU_DST=(long) DSP_ALU_DST >> (cnt-8);
        }
        else {
            DSP_ALU_DST=((DSP_ALU_SRC1>>cnt)|(DSP_ALU_SRC1G<<
(32-cnt)));
        }
        DSP_ALU_DSTG_LSB8 = (char) DSP_ALU_SRC1G_LSB8 >> cnt-- ;
        carry_bit = (((DSP_ALU_SRC1 >> cnt) & MASK00000001)==0x1);
    }

/*  overflow_bit = !(POS_NOT_OV || NEG_NOT_OV); /* do overflow detection */
/* #include "fixed_pt_overflow_protection.c" /* do overflow protection; V=0
*/

    if(DSP_UNCONDITIONAL_UPDATE) { /* unconditional operation */
#include "fixed_pt_unconditional_update.c"
#include "shift_dc_bit.c"
    }
    else if(DSP_CONDITION_MATCH) { /* conditional operation and match */
        DSP_REG[ex2_dz_no] = DSP_ALU_DST;
        if(ex2_dz_no==0) {
                A0G = DSP_ALU_DSTG & MASK000000FF;
                if(DSP_ALU_DSTG_BIT7) A0G = A0G | MASKFFFFFF00;
        }
        else if(ex2_dz_no==1) {
                A1G = DSP_ALU_DSTG & MASK000000FF;
                if(DSP_ALU_DSTG_BIT7) A1G = A1G | MASKFFFFFF00;
        }
    }
```

```
}

break;
```

<When according to immediate operand>

```
{
    unsigned short tmp_imm;
    DSP_ALU_SRC1=DSP_REG[ex2_dz_no];
    switch (ex2_dz_no) {
        case 0x0:  DSP_ALU_SRC1G = A0G;
                   break;
        case 0x1:  DSP_ALU_SRC1G = A1G;
                   break;
        default:   if (DSP_ALU_SRC1_MSB) DSP_ALU_SRC1G = 0xff;
                       else                DSP_ALU_SRC1G =
0x0;
    }

    tmp_imm = ((EX2_LW >> 4) & MASK0000007F); /* bit[10:4] */

    if((tmp_imm & MASK0040)==0) {  /* Left Shift 0<= cnt <=32 */
        char cnt = (tmp_imm & MASK003F);
        if(cnt > 32) {
            printf("¥nPSHA Dz,#Imm,Dz Error! #Imm=%7X exceed range
¥n",tmp_imm);
            exit();
        }
        DSP_ALU_DST  = DSP_ALU_SRC1 << cnt;
        DSP_ALU_DSTG = ((DSP_ALU_SRC1G << cnt) |
                   (DSP_ALU_SRC1 >> (32-cnt))) & MASK000000FF;
        carry_bit = ((DSP_ALU_DSTG & MASK00000001)==0x1);
    }

    else   {                /* Right Shift 0< cnt <=32 */
        char cnt = ((~tmp_imm & MASK003F)+1);
        if(cnt > 32) {
            printf("¥nPSHL Dz,#Imm,Dz Error! #Imm=%7X exceed range
¥n",tmp_imm);
```

```
            exit();
        }
        if((cnt>8) && DSP_ALU_SRC1G_BIT7) { /* MSB copy */
            DSP_ALU_DST=((DSP_ALU_SRC1>>8) | (DSP_ALU_SRC1G<<
(32-8)));
            DSP_ALU_DST=(long) DSP_ALU_DST >> (cnt-8);
        }
        else {
            DSP_ALU_DST=((DSP_ALU_SRC1>>cnt)|(DSP_ALU_SRC1G<<
(32-cnt)));
        }
        DSP_ALU_DSTG_LSB8 = (char) DSP_ALU_SRC1G_LSB8 >>  cnt-
-;
        carry_bit = (((DSP_ALU_SRC1 >> cnt) & MASK00000001)==0x1);
    }

/*  overflow_bit = !(POS_NOT_OV || NEG_NOT_OV); /* do overflow detection */
/* #include "fixed_pt_overflow_protection.c"  /* do overflow
protection; V=0 */

    { /* unconditional operation */
#include "fixed_pt_unconditional_update.c"
#include "shift_dc_bit.c"
    }
}

break;
```

Examples:

`PSHA X0,Y0,A0 NOPX NOPY`	;Before execution:	X0 = H'88888888, Y0 = H'00020000, A0 = H'123456789A
	;After execution:	X0 = H'88888888, Y0 = H'00020000, A0 = H'FE22222222
`PSHA X0,Y0,X0 NOPX NOPY`	;Before execution:	X0 = H'33333333, Y0 = H'FFFF0000
	;After execution:	X0 = H'19999999, Y0 = H'FFFE0000
`PSHA #-5,A1 NOPX NOPY`	;Before execution:	A1 = H'AAAAAAAAAA
	;After execution:	A1 = H'FD55555555

In case of unconditional execution, the DC bit is updated depending on the state of CS [2:0].

8.5.18 [if cc] PSHL (Shift Logically with Condition): DSP Logical Shift Instruction

Format	Abstract	Code	Cycle	DC Bit
`PSHL Sx,Sy,Dz`	If Sy≥0, Sx<<Sy → Dz, clear LSW of Dz; if Sy<0, Sx>>Sy → Dz, clear LSW of Dz	111110********** 10000001xxyyzzzz	1	Update
`DCT PSHL Sx,Sy,Dz`	If DC=1 & Sy≥0, Sx<<Sy → Dz, clear LSW of Dz; if DC=1 & Sy<0, Sx>>Sy → Dz, clear LSW of Dz; if DC=0, nop	111110********** 10000010xxyyzzzz	1	—
`DCF PSHL Sx,Sy,Dz`	If DC=0 & Sy≥0, Sx<<Sy → Dz, clear LSW of Dz; if DC=0 & Sy<0, Sx>>Sy → Dz, clear LSW of Dz; if DC=1, nop	111110********** 10000011xxyyzzzz	1	—
`PSHL #imm,Dz`	If imm≥0, Dz<<imm → Dz, clear LSW of Dz; if imm<0, Dz>>imm → Dz, clear LSW of Dz	111110********** 00000iiiiiiizzzz	1	Update

Description: Logically shifts the top word contents of the Sx or Dz operand, stores the result in the top word of the Dz operand, and clears the bottom word of the Dx operand with zeros. When Dz is a register that has guard bits, the guard bits are also zeroed. The amount of the shift is specified by the Sy operand or the immediate value Imm operand. When the shift amount is positive, it shifts left. When the shift amount is negative, it shifts right. When conditions are specified for DCT and DCF, the instruction is executed when those conditions are TRUE. When they are FALSE, the instruction is not executed.

When conditions are not specified, the DC bit of the DSR register is updated according to the specifications for the CS bits. The N, Z, V, and GT bits of the DSR register are also updated. The DC, N, Z, V, and GT bits are not updated when conditions are specified, even if the conditions are TRUE.

Operation:

<When register operand is used>

```
{
    switch (EX2_SX) {
        case 0x0:  DSP_ALU_SRC1  = X0;
                   break;
        case 0x1:  DSP_ALU_SRC1  = X1;
                   break;
```

```
        case 0x2:  DSP_ALU_SRC1  = A0;
                   break;
        case 0x3:  DSP_ALU_SRC1  = A1;
                   break;
    }
    switch (EX2_SY) {
        case 0x0:  DSP_ALU_SRC2  = Y0 & MASK003F0000;
                   break;
        case 0x1:  DSP_ALU_SRC2  = Y1 & MASK003F0000;
                   break;
        case 0x2:  DSP_ALU_SRC2  = M0 & MASK003F0000;
                   break;
        case 0x3:  DSP_ALU_SRC2  = M1 & MASK003F0000;
                   break;
    }
    if((DSP_ALU_SRC2_HW & MASK0020)==0) {        /* Left Shift
0<=cnt<=16 */
        char cnt = (DSP_ALU_SRC2_HW & MASK001F);
        if(cnt > 16) {
            printf("PSHL Sx,Sy,Dz Error! Shift %2X exceed range
¥n",cnt);
            exit();
            }
        DSP_ALU_DST_HW = DSP_ALU_SRC1_HW << cnt--;
        carry_bit = (((DSP_ALU_SRC1_HW << cnt) & MASK8000)==
0x8000);
    }
    else   {                /* Right Shift 0<cnt<=16 */
        char cnt = ((~DSP_ALU_SRC2_HW & MASK000F)+1);
        if(cnt > 16) {
            printf("PSHL Sx,Sy,Dz Error! Shift -%2X exceed range
¥n",cnt);
            exit();
            }
        DSP_ALU_DST_HW = DSP_ALU_SRC1_HW >> cnt--;
        carry_bit = (((DSP_ALU_SRC1_HW >> cnt) & MASK0001)==0x1);
    }
```

```
    if(DSP_UNCONDITIONAL_UPDATE) { /* unconditional operation */
        DSP_REG_WD[ex2_dz_no*2] = DSP_ALU_DST_HW;
        DSP_REG_WD[ex2_dz_no*2+1] = 0x0;  /* clear LSW */
        if (ex2_dz_no==0) A0G = 0x0;      /* clear Guard bits */
        else if (ex2_dz_no==1)            A1G = 0x0;

        negative_bit  = DSP_ALU_DST_MSB;
        zero_bit   = (DSP_ALU_DST_HW==0);
        overflow_bit  = 0x0;
#include "shift_dc_bit.c"
    }
    else if(DSP_CONDITION_MATCH) { /* conditional operation and match */
        DSP_REG_WD[ex2_dz_no*2] = DSP_ALU_DST_HW;
        DSP_REG_WD[ex2_dz_no*2+1] = 0x0;  /* clear LSW */
        if (ex2_dz_no==0)   A0G = 0x0;    /* clear Guard bits */
        else if (ex2_dz_no==1)            A1G = 0x0;
    }
}

break;
```

<When according to immediate operand>

```
{
    unsigned short tmp_imm;
    DSP_ALU_SRC1=DSP_REG[ex2_dz_no];
    switch (ex2_dz_no) {
        case 0x0:  DSP_ALU_SRC1G = A0G;
                   break;
        case 0x1:  DSP_ALU_SRC1G = A1G;
                   break;
        default:   if (DSP_ALU_SRC1_MSB) DSP_ALU_SRC1G = 0xff;
                       else              DSP_ALU_SRC1G =
0x0;
    }

    tmp_imm = ((EX2_LW >> 4) & MASK0000003F); /* bit[9:4] */

    if((tmp_imm & MASK0020)==0) {  /* Left Shift 0<= cnt <16 */
```

```
        char cnt = (tmp_imm & MASK001F);
        if(cnt > 16) {
            printf("PSHL Dz,#Imm,Dz Error! #Imm=%6X exceed range
¥n",tmp_imm);
            exit();
        }
        DSP_ALU_DST_HW = DSP_ALU_SRC1_HW << cnt--;
        carry_bit = (((DSP_ALU_SRC1_HW << cnt) & MASK8000)==
0x8000);
    }

    else   {              /* Right Shift 0< cnt <=16 */
        char cnt = ((~tmp_imm & MASK001F)+1);
        if(cnt > 16) {
            printf("PSHL Dz,#Imm,Dz Error! #Imm=%6X exceed range
¥n",tmp_imm);
            exit();
        }
        DSP_ALU_DST_HW = DSP_ALU_SRC1_HW >> cnt--;
        carry_bit = (((DSP_ALU_SRC1_HW >> cnt) & MASK0001)==0x1);
    }

    { /* unconditional operation */
        DSP_REG_WD[ex2_dz_no*2] = DSP_ALU_DST_HW;
        DSP_REG_WD[ex2_dz_no*2+1] =       0x0;   /* clear LSW */
        if (ex2_dz_no==0) A0G = 0x0;      /* clear Guard bits */
        else if (ex2_dz_no==1)     A1G = 0x0;

        negative_bit  = DSP_ALU_DST_MSB;
        zero_bit   = (DSP_ALU_DST_HW==0);
        overflow_bit  = 0x0;
#include "shift_dc_bit.c"
    }
}

break;
```

Examples:

`PSHL X0,Y0,A0 NOPX NOPY`	;Before execution:	X0 = H'22222222, Y0 = H'00030000, A0 = H'123456789A
	;After execution:	X0 = H'22222222, Y0 = H'00030000, A0 = H'0011100000
`PSHL X1,Y1,X1 NOPX NOPY`	;Before execution:	X1 = H'CCCCCCCC, Y1 = H'FFFE0000
	;After execution:	X1 = H'33330000, Y1 = H'FFFE0000
`PSHL #7,A1 NOPX NOPY`	;Before execution:	A1 = H'55555555
	;After execution:	A1 = H'AA800000

In case of unconditional execution, the DC bit is updated depending on the state of CS [2:0].

8.5.19 [if cc] PSTS (Store System Register): DSP System Control Instruction

Format	Abstract	Code	Cycle	DC Bit
PSTS MACH,Dz	MACH→Dz	111110********** 110011010000zzzz	1	—
PSTS MACL,Dz	MACL→Dz	111110********** 110111010000zzzz	1	—
DCT PSTS MACH,Dz	if DC = 1, MACH→Dz if 0, nop	111110********** 110011100000zzzz	1	—
DCT PSTS MACL,Dz	if DC = 1, MACL→Dz if 0, nop	111110********** 110111100000zzzz	1	—
DCF PSTS MACH,Dz	if DC = 0, MACH→Dz if 1, nop	111110********** 110011110000zzzz	1	—
DCF PSTS MACL,Dz	if DC = 0, MACL→Dz if 1, nop	111110********** 110111110000zzzz	1	—

Description: Stores the contents of the MACH and MACL registers in the Dz operand. When conditions are specified for DCT and DCF, the instruction is executed when those conditions are TRUE. When they are FALSE, the instruction is not executed. The DC, N, Z, V, and GT bits of the DSR register are not updated.

Note: Though PSTS, MOVX and MOVY can be designated in parallel, their execution may take 2 cycles.

Operation:

```
     /* MACH -> Dz */
  {
     if(DSP_UNCONDITIONAL_UPDATE) { /* unconditional operation */
        DSP_REG[ex2_dz_no] = MACH;
        if(ex2_dz_no==0) {
               A0G = DSP_ALU_DSTG & MASK000000FF;
               if(DSP_ALU_DSTG_BIT7) A0G = A0G | MASKFFFFFF00;
        }
        else if(ex2_dz_no==1) {
               A1G = DSP_ALU_DSTG & MASK000000FF;
               if(DSP_ALU_DSTG_BIT7) A1G = A1G | MASKFFFFFF00;
        }
     }
```

```
    else if(DSP_CONDITION_MATCH) { /* conditional operation and match */
        DSP_REG[ex2_dz_no] = MACH;
        if(ex2_dz_no==0) {
                A0G = DSP_ALU_DSTG & MASK000000FF;
                if(DSP_ALU_DSTG_BIT7) A0G = A0G | MASKFFFFFF00;
        }
        else if(ex2_dz_no==1) {
                A1G = DSP_ALU_DSTG & MASK000000FF;
                if(DSP_ALU_DSTG_BIT7) A1G = A1G | MASKFFFFFF00;
        }
    }
}

break;

    /* MACL -> Dz */
{
    if(DSP_UNCONDITIONAL_UPDATE) { /* unconditional operation */
        DSP_REG[ex2_dz_no] = MACL;
        if(ex2_dz_no==0) {
                A0G = DSP_ALU_DSTG & MASK000000FF;
                if(DSP_ALU_DSTG_BIT7) A0G = A0G | MASKFFFFFF00;
        }
        else if(ex2_dz_no==1) {
                A1G = DSP_ALU_DSTG & MASK000000FF;
                if(DSP_ALU_DSTG_BIT7) A1G = A1G | MASKFFFFFF00;
        }
    }
    else if(DSP_CONDITION_MATCH) { /* conditional operation and match */
        DSP_REG[ex2_dz_no] = MACL;
        if(ex2_dz_no==0) {
                A0G = DSP_ALU_DSTG & MASK000000FF;
                if(DSP_ALU_DSTG_BIT7) A0G = A0G | MASKFFFFFF00;
        }
        else if(ex2_dz_no==1) {
                A1G = DSP_ALU_DSTG & MASK000000FF;
                if(DSP_ALU_DSTG_BIT7) A1G = A1G | MASKFFFFFF00;
```

```
        }
    }
}

break;
```

Examples:

`PSTS MACH,A0 NOPX NOPY`	;Before execution:	A0 = H'123456789A, MACH = H'88888888
	;After execution:	A0 = H'FF88888888, MACH = H'88888888

8.5.20 [if cc]PSUB (Subtract with Condition): DSP Arithmetic Operation Instruction

Format	Abstract	Code	Cycle	DC Bit
`PSUB Sx,Sy,Dz`	Sx – Sy→Dz	111110********** 10100001xxyyzzzz	1	Update
`DCT PSUB Sx,Sy,Dz`	if DC = 1, Sx – Sy→Dz if 0, nop	111110********** 10100010xxyyzzzz	1	—
`DCF PSUB Sx,Sy,Dz`	if DC = 0, Sx – Sy→Dz if 1, nop	111110********** 10100011xxyyzzzz	1	—

Description: Subtracts the contents of the Sy operand from the Sx operand and stores the result in the Dz operand. When conditions are specified for DCT and DCF, the instruction is executed when those conditions are TRUE. When they are FALSE, the instruction is not executed.

When conditions are not specified, the DC bit of the DSR register is updated according to the specifications for the CS bits. The N, Z, V, and GT bits of the DSR register are updated. The DC, N, Z, V, and GT bits are not updated when conditions are specified, even if the conditions are TRUE.

Operation:

```
{
    switch (EX2_SX) {
        case 0x0:  DSP_ALU_SRC1  = X0;
                   if (DSP_ALU_SRC1_MSB) DSP_ALU_SRC1G = 0xff;
                   else     DSP_ALU_SRC1G = 0x0;
                   break;
        case 0x1:  DSP_ALU_SRC1  = X1;
                   if (DSP_ALU_SRC1_MSB) DSP_ALU_SRC1G = 0xff;
                   else     DSP_ALU_SRC1G = 0x0;
                   break;
        case 0x2:  DSP_ALU_SRC1  = A0;
                   DSP_ALU_SRC1G = A0G;
                   break;
        case 0x3:  DSP_ALU_SRC1  = A1;
                   DSP_ALU_SRC1G = A1G;
                   break;
    }
    switch (EX2_SY) {
```

```
        case 0x0:  DSP_ALU_SRC2  = Y0;
                   break;
        case 0x1:  DSP_ALU_SRC2  = Y1;
                   break;
        case 0x2:  DSP_ALU_SRC2  = M0;
                   break;
        case 0x3:  DSP_ALU_SRC2  = M1;
                   break;
    }
    if (DSP_ALU_SRC2_MSB) DSP_ALU_SRC2G = 0xff;
    else              DSP_ALU_SRC2G = 0x0;

    DSP_ALU_DST = DSP_ALU_SRC1 - DSP_ALU_SRC2;
    carry_bit =((DSP_ALU_SRC1_MSB | !DSP_ALU_SRC2_MSB) && !DSP_ALU_DST_MSB) |
        (DSP_ALU_SRC1_MSB & !DSP_ALU_SRC2_MSB);
    borrow_bit = !carry_bit;
    DSP_ALU_DSTG_LSB8 = DSP_ALU_SRC1G_LSB8 - DSP_ALU_SRC2G_LSB8 - borrow_bit;

    overflow_bit= MINUS_OP_G_OV || !(POS_NOT_OV || NEG_NOT_OV);
#include "fixed_pt_overflow_protection.c"
    if(DSP_UNCONDITIONAL_UPDATE) { /* unconditional operation */
#include "fixed_pt_unconditional_update.c"
#include "fixed_pt_minus_dc_bit.c"
    }
    else if(DSP_CONDITION_MATCH) { /* conditional operation and match */
        DSP_REG[ex2_dz_no] = DSP_ALU_DST;
        if(ex2_dz_no==0) {
                A0G = DSP_ALU_DSTG & MASK000000FF;
                if(DSP_ALU_DSTG_BIT7) A0G = A0G | MASKFFFFFF00;
        }
        else if(ex2_dz_no==1) {
                A1G = DSP_ALU_DSTG & MASK000000FF;
                if(DSP_ALU_DSTG_BIT7) A1G = A1G | MASKFFFFFF00;
        }
    }
}

break;
```

Examples:

`PSUB X0,Y0,A0 NOPX NOPY`	;Before execution:	X0 = H'55555555, Y0 = H'33333333, A0 = H'123456789A
	;After execution:	X0 = H'55555555, Y0 = H'33333333, A0 = H'0022222222

In case of unconditional execution, the DC bit is updated depending on the state of CS [2:0].

8.5.21 PSUB PMULS (Subtraction & Multiply Signed by Signed): DSP Arithmetic Operation Instruction

Format	Abstract	Code	Cycle	DC Bit
PSUB Sx,Sy,Du	Sx – Sy→Du	111110**********	1	Update
PMULS Se,Sf,Dg	MSW of Se × MSW of Sf→Dg	0110eeffxxyygguu		

Description: Subtracts the contents of the Sy operand from the Sx operand and stores the result in the Du operand. The contents of the top word of the Se and Sf operands are multiplied as signed and the result stored in the Dg operand. These two processes are executed simultaneously in parallel.

The DC bit of the DSR register is updated according to the results of the ALU operation and the specifications for the CS bits. The N, Z, V, and GT bits of the DSR register are also updated according to the results of the ALU operation.

Operation:

```
{
    DSP_ALU_DST = DSP_ALU_SRC1 - DSP_ALU_SRC2;
    carry_bit=((DSP_ALU_SRC1_MSB | !DSP_ALU_SRC2_MSB)&& !DSP_ALU_DST_MSB)|
          (DSP_ALU_SRC1_MSB & !DSP_ALU_SRC2_MSB);
    borrow_bit = !carry_bit;
    DSP_ALU_DSTG_LSB8=DSP_ALU_SRC1G_LSB8 - DSP_ALU_SRC2G_LSB8 -
borrow_bit;

    overflow_bit= MINUS_OP_G_OV || !(POS_NOT_OV || NEG_NOT_OV);
#include "../d_3operand.d/fixed_pt_overflow_protection.c"
    switch (EX2_DU) {
        case 0x0:
                X0  = DSP_ALU_DST;
                negative_bit = DSP_ALU_DST_MSB;
                zero_bit = (DSP_ALU_DST==0);
                break;
        case 0x1:
                Y0  = DSP_ALU_DST;

                negative_bit = DSP_ALU_DST_MSB;
                zero_bit = (DSP_ALU_DST==0);
```

```
                break;
        case 0x2:
                A0  = DSP_ALU_DST;
                A0G = DSP_ALU_DSTG & MASK000000FF;
                if(DSP_ALU_DSTG_BIT7) A0G = A0G | MASKFFFFFF00;
                negative_bit = DSP_ALU_DSTG_BIT7;
                zero_bit = (DSP_ALU_DST==0) & (DSP_ALU_DSTG_
LSB8==0);
                break;
        case 0x3:
                A1  = DSP_ALU_DST;
                A1G = DSP_ALU_DSTG & MASK000000FF;
                if(DSP_ALU_DSTG_BIT7) A1G = A1G | MASKFFFFFF00;
                negative_bit = DSP_ALU_DSTG_BIT7;
                zero_bit = (DSP_ALU_DST==0) & (DSP_ALU_DSTG
_LSB8==0);
                break;
    }

#include "../d_3operand.d/fixed_pt_minus_dc_bit.c"
}

break;
```

Examples:

```
PSUB A0,M0,A0 PMULS X0,Y0, M0 NOPX NOPY
```

;Before execution: X0 = H'00020000, Y0 = H'FFFE0000, M0 = H'33333333, A0 = H'0022222222

;After execution: X0 = H'00020000, Y0 = H'FFFE0000, M0 = H'FFFFFFF8, A0 = H'55555555

8.5.22 PSUBC (Subtraction with Carry): DSP Arithmetic Operation Instruction

Format	Abstract	Code	Cycle	DC Bit
PSUBC Sx,Sy,Dz	Sx – Sy – DC→Dz	111110********** 10100000xxyyzzzz	1	Borrow

Description: Subtracts the contents of the Sy operand and the DC bit from the Sx operand and stores the result in the Dz operand. The DC bit of the DSR register is updated as the borrow flag. The N, Z, V, and GT bits of the DSR register are also updated.

Note: After the PSUBC instruction is executed, the DC bit is updated as the borrow flag without regard to the CS bit.

Operation:

```
{
    switch (EX2_SX) {
        case 0x0:  DSP_ALU_SRC1  = X0;
                   if (DSP_ALU_SRC1_MSB) DSP_ALU_SRC1G = 0xff;
                   else               DSP_ALU_SRC1G = 0x0;
                   break;
        case 0x1:  DSP_ALU_SRC1  = X1;
                   if (DSP_ALU_SRC1_MSB) DSP_ALU_SRC1G = 0xff;
                   else               DSP_ALU_SRC1G = 0x0;
                   break;
        case 0x2:  DSP_ALU_SRC1    = A0;
                   DSP_ALU_SRC1G   = A0G;
                   break;
        case 0x3:  DSP_ALU_SRC1    = A1;
                   DSP_ALU_SRC1G   = A1G;
                   break;
    }
    switch (EX2_SY) {
        case 0x0:  DSP_ALU_SRC2    = Y0;
                   break;
        case 0x1:  DSP_ALU_SRC2    = Y1;
                   break;
        case 0x2:  DSP_ALU_SRC2    = M0;
                   break;
```

```
        case 0x3:  DSP_ALU_SRC2    = M1;
                   break;
    }
    if (DSP_ALU_SRC2_MSB)  DSP_ALU_SRC2G = 0xff;
    else              DSP_ALU_SRC2G = 0x0;

    DSP_ALU_DST = DSP_ALU_SRC1 - DSP_ALU_SRC2 - DSPDCBIT;
    carry_bit =((DSP_ALU_SRC1_MSB | !DSP_ALU_SRC2_MSB) && !DSP_ALU
_DST_MSB)
          | (DSP_ALU_SRC1_MSB & !DSP_ALU_SRC2_MSB);
    borrow_bit = !carry_bit;
    DSP_ALU_DSTG_LSB8 = DSP_ALU_SRC1G_LSB8 - DSP_ALU_SRC2G_LSB8 -
borrow_bit;

    overflow_bit= MINUS_OP_G_OV || !(POS_NOT_OV || NEG_NOT_OV);
#include "fixed_pt_overflow_protection.c"
#include "fixed_pt_unconditional_update.c"
#include "fixed_pt_dc_always_borrow.c"
}

break;
```

Example:

```
CS[2:0]=***: Always Carry or Borrow Mode
PSUBC X0,Y0,M0  NOPX  NOPY     ;Before execution:   X0 = H'33333333, Y0 = H'55555555
                                                    M0 = H'0012345678, DC = 0
                               ;After execution:    X0 = H'33333333, Y0 = H'55555555
                                                    M0 = H'FFDDDDDDDE, DC = 1
PSUBC X0,Y0,M0  NOPX  NOPY     ;Before execution:   X0 = H'33333333, Y0 = H'55555555
                                                    M0 = H'0012345678, DC = 1
                               ;After execution:    X0 = H'33333333, Y0 = H'55555555
                                                    M0 = H'FFDDDDDDDD, DC = 1
```

8.5.23 [if cc] PXOR (Logical Exclusive OR): DSP Logical Operation Instruction

Format	Abstract	Code	Cycle	DC Bit
`PXOR Sx,Sy,Dz`	Sx ^ Sy→Dz, clear LSW of Dz	`111110**********` `10100101xxyyzzzz`	1	Update
`DCT PXOR Sx,Sy,Dz`	if DC = 1, Sx^Sy→Dz, clear LSW of Dz; if 0, nop	`111110**********` `10100110xxyyzzzz`	1	—
`DCF PXOR Sx,Sy,Dz`	if DC = 0, Sx^Sy→Dz clear LSW of Dz; if 1, nop	`111110**********` `10100111xxyyzzzz`	1	—

Description: Takes the exclusive OR of the top word of the Sx operand and the top word of the Sy operand, stores the result in the top word of the Dz operand, and clears the bottom word of Dz with zeros. When Dz is a register that has guard bits, the guard bits are also zeroed. When conditions are specified for DCT and DCF, the instruction is executed when those conditions are TRUE. When they are FALSE, the instruction is not executed.

When conditions are not specified, the DC bit of the DSR register is updated according to the specifications for the CS bits. The N, Z, V, and GT bits of the DSR register are also updated. The DC, N, Z, V, and GT bits are not updated when conditions are specified, even if the conditions are TRUE.

Note: The bottom word of the destination register and the guard bits are ignored when the DC bit is updated.

Operation:

```
{
    switch (EX2_SX) {
        case 0x0:  DSP_ALU_SRC1  = X0;
                   break;
        case 0x1:  DSP_ALU_SRC1  = X1;
                   break;
        case 0x2:  DSP_ALU_SRC1  = A0;
                   break;
        case 0x3:  DSP_ALU_SRC1  = A1;
                   break;
    }
    switch (EX2_SY) {
        case 0x0:  DSP_ALU_SRC2  = Y0;
                   break;
```

```
        case 0x1:  DSP_ALU_SRC2  = Y1;
                   break;
        case 0x2:  DSP_ALU_SRC2  = M0;
                   break;
        case 0x3:  DSP_ALU_SRC2  = M1;
                   break;
    }

    DSP_ALU_DST_HW = DSP_ALU_SRC1_HW ^ DSP_ALU_SRC2_HW;

    if(DSP_UNCONDITIONAL_UPDATE) { /* unconditional operation */
        DSP_REG_WD[ex2_dz_no*2] = DSP_ALU_DST_HW;
        DSP_REG_WD[ex2_dz_no*2+1] = 0x0;  /* clear LSW */
        if (ex2_dz_no==0) A0G = 0x0;      /* clear Guard bits */
        else if (ex2_dz_no==1)  A1G = 0x0;

        carry_bit  = 0x0;
        negative_bit   = DSP_ALU_DST_MSB;
        zero_bit   = (DSP_ALU_DST_HW==0);
        overflow_bit   = 0x0;

#include "logical_dc_bit.c"
    }
    else if(DSP_CONDITION_MATCH) { /* conditional operation and match */
        DSP_REG_WD[ex2_dz_no*2] = DSP_ALU_DST_HW;
        DSP_REG_WD[ex2_dz_no*2+1] = 0x0;         /* clear LSW */
        if (ex2_dz_no==0) A0G = 0x0;      /* clear Guard bits */
        else if (ex2_dz_no==1)  A1G = 0x0;
    }
}

break;
```

Example:

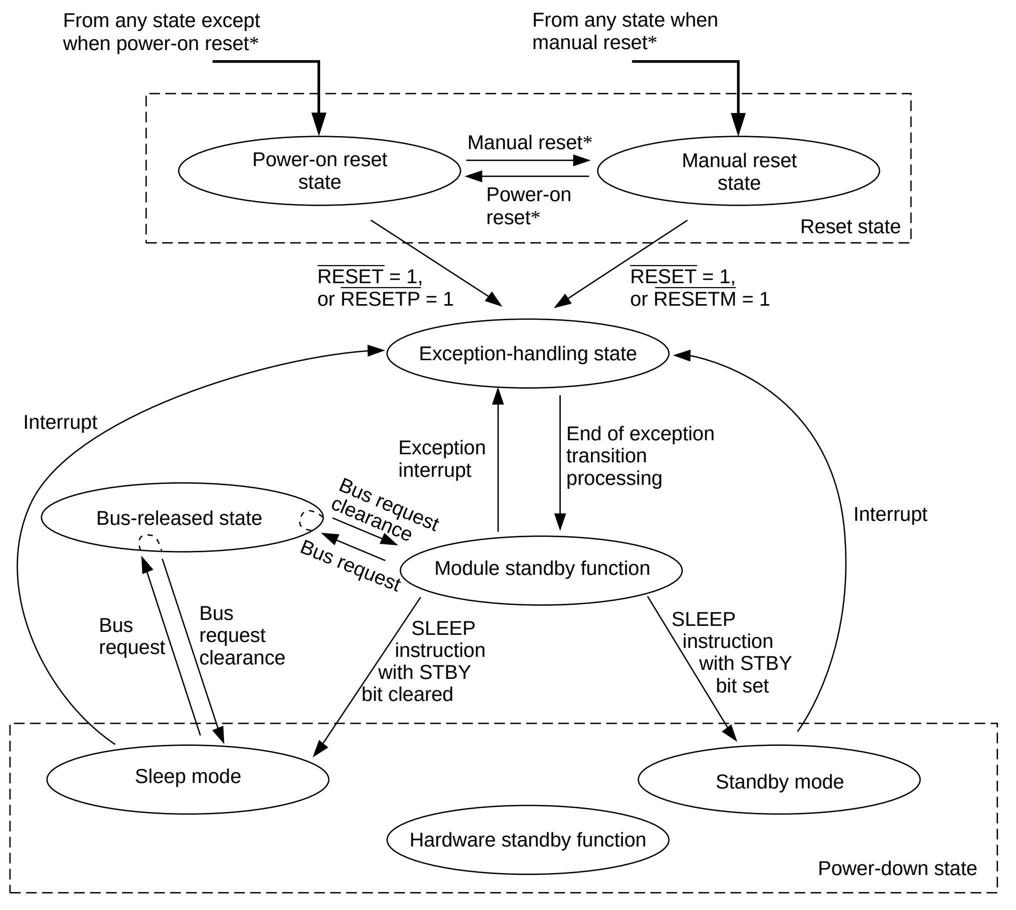
```
PXOR X0,Y0,A0  NOPX  NOPY    ;Before execution:  X0 = H'33333333, Y0 = H'55555555
                                                 A0 = H'123456789A
                             ;After execution:   X0 = H'33333333, Y0 = H'55555555
                                                 A0 = H'0066660000
```

In case of unconditional execution, the DC bit is updated depending on the state of CS [2:0].

Section 9 Processing States

9.1 State Transitions

The CPU has five processing states: reset, exception processing, bus release, program execution and power-down. The transitions between the states are shown in figure 9-1.



Note: SH-3 (SH7702, SH7707, SH7708), SH-3E:
Power-on reset: RESET = 0, BREQ = 1
Manual reset: RESET = 0, BREQ = 0
SH-3 (SH7709), SH3-DSP:
Power-on reset: RESETP = 0
Manual reset: RESETM = 0

Figure 9-1 Transitions between Processing States

9.1.1 Reset State

In the reset state, the CPU is reset. On the SH-3 (SH7702, SH7707, SH7708) and SH-3E, this occurs when the $\overline{\text{RESET}}$ pin goes low. When the $\overline{\text{BREQ}}$ pin is high, the result is a power-on reset; when it is low, a manual reset occurs. On the SH-3 (SH7709) and SH3-DSP, a power-on reset occurs when the $\overline{\text{RESETP}}$ pin is low, and a manual reset occurs when the $\overline{\text{RESETM}}$ pin is low.

9.1.2 Exception Processing State

The exception processing state is a transient state that occurs when the CPU's processing state flow is altered by exception processing sources such as resets, general exceptions, or interrupts.

For a reset, the CPU branches to H'A0000000 and starts executing the user-created exception process program.

For a general exception or interrupt, the program counter (PC) is saved in the save program counter (SPC), and the status register (SR) is saved in the save status register (SSR). The CPU then branches to the starting address of the user-created exception service routine by adding the content of the vector base address and the vector offset, thereby starting program execution state.

9.1.3 Program Execution State

In the program execution state, the CPU sequentially executes the program.

9.1.4 Power-Down State

In the power-down state, the CPU operation halts and power consumption declines. The SLEEP instruction places the CPU in the power-down state. This state has four modes and function: sleep mode, standby mode, hardware standby mode, and module standby function. See section 9.2 for more details.

9.1.5 Bus Release State

In the bus release state, the CPU releases access rights to the bus to the device that has requested them.

9.2 Power-Down State

In addition to the ordinary program execution states, the CPU also has a power-down state in which CPU operation halts and power consumption is lowered (table 9-1). There are four power-down state modes and function: sleep mode, standby mode, hardware standby mode, and module standby function.

9.2.1 Sleep Mode

When the standby bit (STBY) of the standby control register (STBCR) is cleared to 0 and the SLEEP instruction executed, the CPU enters the sleep mode. In sleep mode, the CPU halts but the contents of the CPU and cache registers are maintained. Operation of the on-chip peripheral modules continues.

Returning from the sleep mode is accomplished using a reset or an interrupt. The CPU first enters the exception processing mode and then makes the transition to the normal program execution mode.

9.2.2 Standby Mode

When the standby bit (STBY) of the standby control register (STBCR) is set to 1 and the SLEEP instruction executed, the CPU enters the standby mode. In standby mode, the functioning of the CPU, the on-chip peripheral modules, and oscillator halt. However, the contents of the CPU and cache registers are maintained.

Returning from the standby mode is accomplished using a reset or an interrupt. If a reset is used, the CPU enters the exception processing mode after the oscillator stabilization time has elapsed and then makes the transition to the normal program execution mode. If an interrupt is used, the CPU enters the exception processing mode after the oscillator stabilization time set in WDT has elapsed and then makes the transition to the normal program execution mode.

In this mode, power consumption drops markedly, since the oscillator stops.

9.2.3 Hardware Standby Mode

The CPU enters the hardware standby mode when the CA pin is set to low level. As with the standby modes initiated using the SLEEP command, the hardware standby mode, all modules other than those which function using the RTC clock halt.

9.2.4 Module Standby Function

The timer (TMU), real-time clock (RTC), and serial communication interface (SCI) each have a module standby function.

When the module stop bit of the standby control register (STBCR) is set to 1, the supply of the clock to the corresponding modules is halted. This function can be used to reduce power consumption both in the normal program execution mode and in the sleep mode.

When the module standby function is being used, the status of the external pins of the on-chip peripheral modules differs depending on the module. The external pins of the TMU maintain their status prior to standby. The external pins of the SCI are reset.

To cancel the module standby function, either clear the MSTP bits to 0 or perform a reset.

Table 9-1 Power-Down State

Mode	Entering Procedure	State: Oscil-lator	CPU	CPU Reg-ister	On-Chip Memory	On-Chip Peripheral Modules	Pins	External Memory	Canceling Procedure
Sleep mode	Execute SLEEP instruction when STBY bit of STBCR is cleared to 0	Run	Halt	Held	Held	Run	Held	Refresh	1. Interrupt 2. Reset
Standby mode	Execute SLEEP instruction with STBY bit set to 1 in STBCR	Halt	Halt	Held	Held	Halt*	Held	Self-refresh	1. Interrupt 2. Reset
Hardware standby mode	Set CA pin to low level	Halt	Halt	Held	Held	Halt*	Held	Self-refresh	
Module standby function	Set MSTP bit of STBCR to 1	Run	Run	Held	Held	Specified module halts	Held	Refresh	1. Set MSTP bit to 0 2. Reset

Note: * Differs depending on the on-chip peripheral module. Refer to the Hardware Manual for the SH-3, SH-3E, and SH3-DSP for details.

Section 10 Pipeline Operation

This section describes the operation of the pipelines for each instruction. This information is provided to allow calculation of the required number of CPU instruction execution states (system clock cycles).

10.1 Basic Configuration of Pipelines

10.1.1 Five-Stage Pipeline

Pipelines are composed of the following five stages:

- IF (Instruction fetch) — Fetches instruction from the memory stored in the program.
- ID (Instruction decode) — Decodes the instruction fetched.
- EX (Instruction execution) — Does data operations and address calculations according to the results of decoding.
- MA (Memory access) — Accesses data in memory in conjunction with instructions that involve memory access.
  For instructions that do not involve memory access, the resulting data is maintained as is and MA is expressed in lowercase letters as "ma".
- WB (Write back) — Returns the results of the memory access (data) to a register in conjunction with instructions that involve memory access.
  For instructions that do not involve memory access, the data maintained in the ma stage is returned to the register.

Instructions are executed using a pipeline consisting of five stages. The various instruction stages flow with the execution of the instructions and form this pipeline. This means that at any given moment, five instructions are being executed simultaneously. The basic flow of the pipeline is shown in Figure 10-1. Each period during which a single stage is executed is called a slot and is indicated using the “↔” symbol.

All instructions have at least three stages: IF, ID, and EX. Some also have stages MA and WB. Also, the way the pipeline flows varies with the type of instruction, with some containing two MA stages, some including access to the multiplier (mm), and so on. There can also be contention, for example, between IF and MA. If contention occurs, the flow of the pipeline changes.

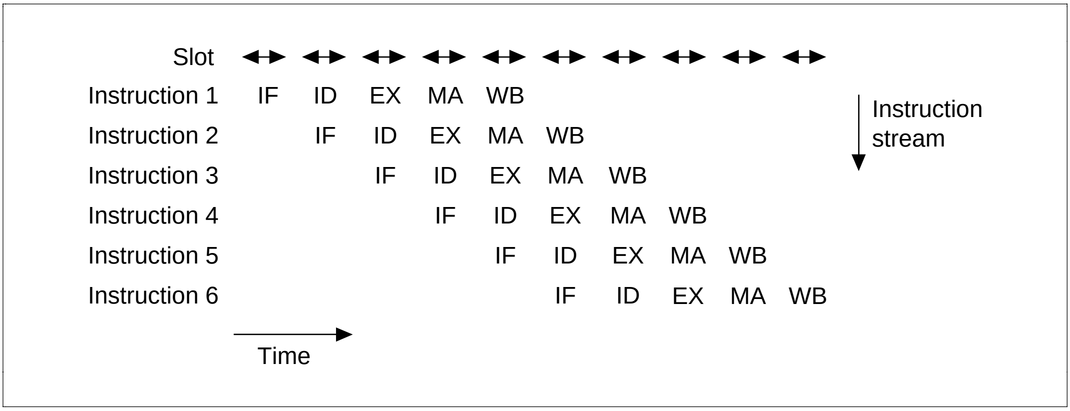


Figure 10-1 Basic Structure of Pipeline Flow

10.1.2 Slot and Pipeline Flow

The time period in which a single stage operates is called a slot. Slots must follow the rules described below.

Instruction Execution

Each stage (IF, ID, EX, MA, WB) of an instruction must be executed in one slot. Two or more stages cannot be executed within one slot (figure 10-2), with exception of WB and MA.

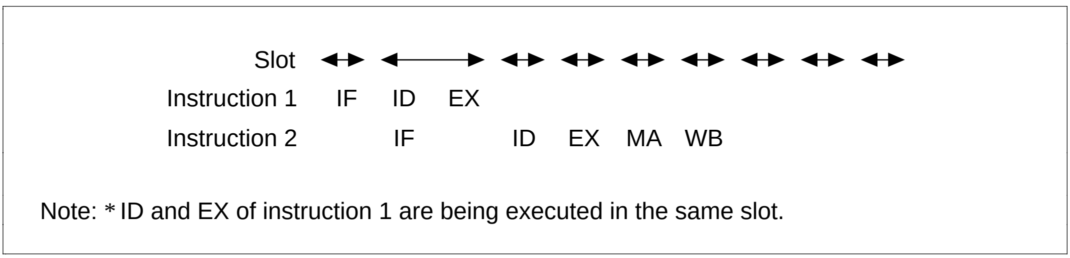


Figure 10-2 Impossible Pipeline Flow 1

Slot Sharing

A maximum of one stage from another instruction may be set per slot, and that stage must be different from the stage of the first instruction. Identical stages from two different instructions may never be executed within the same slot (figure 10-3).

Slot	↔	↔	↔	↔	↔	↔	↔	↔	↔	↔
Instruction 1	IF	ID	EX	MA	WB					
Instruction 2	IF	ID	EX	MA	WB					
Instruction 3		IF	ID	EX	MA	WB				
Instruction 4			IF	ID	EX	MA	WB			
Instruction 5			IF	ID	EX	MA	WB			

Note: * Same stage of another instruction is being executed in same slot.

Figure 10-3 Impossible Pipeline Flow 2

10.1.3 Number of Cycles Required for Execution of One Slot

The number of states (system clock cycles) S for the execution of one slot is calculated with the following conditions:

- S = (the cycles of the stage with the highest number of cycles of all instruction stages contained in the slot)

  This means that the instruction with the longest stage stalls others with shorter stages.

- The number of execution cycles for each stage:

  — IF The number of memory access cycles for instruction fetch
  — ID Always one cycle
  — EX Always one cycle
  — MA The number of memory access cycles for data access
  — WB Always one cycle

As an example, figure 10-4 shows the flow of a pipeline in which the IF (memory access for instruction fetch) of instructions 1 and 2 are two cycles, the MA (memory access for data access) of instruction 1 is three cycles and all others are one cycle. The dashes indicate the instruction is being stalled. Refer to the Hardware Manual for information on the number of clock cycles in each case.

Slot	↔		↔		↔	↔			↔	↔
Number of cycles	(2)		(2)		(1)	(3)			(1)	(1)
Instruction 1	IF	IF	ID	—	EX	MA	MA	MA	WB	
Instruction 2			IF	IF	ID	EX	—	—	MA	WB

Figure 10-4 Slots Requiring Multiple Cycles

10.1.4 Number of Instruction Execution Cycles

The number of instruction execution cycles is counted based on the interval between execution of EX stages. The number of cycles between the start of the EX stage for instruction 1 and the start of the EX stage for the following instruction (instruction 2) is the execution time for instruction 1. Figure 10-5 shows an example of the way in which the number of instruction execution cycles is counted.

In this example, the flow of the pipeline is such that the EX stage interval between instructions 1 and 2 is two cycles. Therefore, the execution time for instruction 1 is two cycles. Also, the EX stage interval between instructions 2 and 3 is three cycles, so the execution time for instruction 2 is three cycles. If a program ends with instruction 3, the execution time for instruction 3 would be calculated as the interval between the EX stage of instruction 3 and the EX stage of a hypothetical instruction 4 following instruction 3, using MOV Rm, Rn. In this example, the execution time for instruction 3 is two cycles. The execution time for instructions 1 through 3 is therefore seven cycles (2 + 3 + 2 = 7).

In this example, the MA of instruction 1 and the IF of instruction 4 are in contention. For information on operation when MA and IF are in contention, refer to section 10.2.1, Contention between Instruction Fetch (IF) and Memory Access (MA).

Slot												
Number of cycles	(2)		(2)		(2)		(4)			(1)	(1)	(1)
Instruction 1	IF	IF	ID	—	[EX]	—	MA	MA	MA	WB		
Instruction 2			IF	IF	ID	—	[EX]	—	—	ma	WB	
Instruction 3					IF	IF	ID	—	—	[EX]	MA	WB
(Instruction 4: MOV Rm, Rn							—	—	—	IF	ID	[EX])

Figure 10-5 How Instruction Execution Cycles Are Counted

10.2 Contention

Contention occurs in the following seven situations. When contention occurs in a particular stage, that stage is stored and the next and subsequent slots are executed.

(1) Contention between instruction fetch (IF) and memory access (MA)

(2) Contention caused by a memory load instruction

(3) Contention caused by an SR update instruction

(4) Contention caused by accessing the multiplier

(5) FPU contention (SH-3E only)

(6) Contention between DSP data operation instruction and store instruction (SH3-DSP only)

(7) Contention between a transfer between DSP registers and a memory load or store operation (SH3-DSP only)

10.2.1 Contention between Instruction Fetch (IF) and Memory Access (MA)

Basic Operation when IF and MA Are in Contention

The IF and MA stages both access memory, so they cannot operate simultaneously. If the IF and MA stages both try to access memory within the same slot, the IF stage is stored and the next slot is executed. However, if contention with another MA stage occurs in the next slot, the IF stage is again stored and the next slot is executed. Figure 10-6 illustrates operation when IF and MA are in contention.

(a) When there is no subsequent MA stage

Slot	A	B	C	D	E	F	G		
Instruction 1	if	ID	EX	MA	WB				
Instruction 2		IF	ID	EX	ma	WB			
Instruction 3			if	ID	EX	ma	WB		
Instruction 4				IF	ID	EX	ma	WB	
Instruction 5					if	ID	EX		

• MA of instruction 1 and IF of instruction 4 contend at D

↓ : (When MA and IFare in contention, the following occurs:)

Slot	A	B	C	D	E	F	G		
Instruction 1	if	ID	EX	MA	WB				
Instruction 2		IF	ID	EX	ma	WB			
Instruction 3			if	ID	EX	ma	WB		
Instruction 4				—	IF	ID	EX	ma	WB
Instruction 5						if	ID	EX	

• IF stored at D

(b) When there is a subsequent MA stage

Slot	A	B	C	D	E	F	G		
Instruction 1	if	ID	EX	MA	WB				
Instruction 2		IF	ID	EX	MA	WB			
Instruction 3			if	ID	EX	ma	WB		
Instruction 4				IF	ID	EX	ma	WB	
Instruction 5					if	ID	EX		

• MA of instruction 1 and IF of instruction 4 contend at D

↓ : (When MA and IFare in contention, the following occurs:)

Slot	A	B	C	D	E	F	G			
Instruction 1	if	ID	EX	MA	WB					
Instruction 2		IF	ID	EX	MA	WB				
Instruction 3			if	—	ID	EX	ma	WB		
Instruction 4				—	—	IF	ID	EX	ma	WB
Instruction 5							if	ID	EX	

• ID and IF stored at D
• IF stored at E

Figure 10-6 Operation when IF and MA Are in Contention

The operation when there is contention between IF and MA and no subsequent MA stage is shown in (a) of Figure 10-6. IF and MA are in contention in slot D. In this case, the IF stage is stored and the following slot, E, is executed. In slot E ma and IF are in contention, but the IF stage is not stored because the ma stage does not generate a bus cycle.

The operation when there is contention between IF and MA and there is a subsequent MA stage is shown in (b) of Figure 10-6. There are MA stages in slots D and E, and MA is in contention with IF in slot D. In this case, the ID and IF of slot D are stored and then executed in slot E. However, contention between IF and MA occurs again in slot E, so the IF stage is stored again and then executed in the next slot, F.

Relationship between IF and the Location of Instructions in Memory

When the instruction is located in memory, the SuperH microcomputer accesses the memory in 32-bit units. The SuperH microcomputer instructions are all fixed at 16 bits, so basically 2 instructions can be fetched in a single IF stage access. Whether an IF fetches one or two instructions depends on the memory location (word or longword boundary).

If an instruction is located on a longword boundary, an IF can get two instructions at each instruction fetch. The IF of the next instruction does not generate a bus cycle to fetch an instruction from memory. Since the next instruction IF also fetches two instructions, the instruction IFs after that do not generate a bus cycle either.

This means that IFs of instructions that are located so they start from the longword boundaries within instructions located in memory (the position when the bottom two bits of the instruction address are 00 is A1 = 0 and A0 = 0) also fetch two instructions. The IF of the next instruction does not generate a bus cycle. IFs that do not generate bus cycles are written in lower case as “if”. These ifs always take one cycle.

When branching results in a fetch from an instruction located so it starts from the word boundaries (the position when the bottom two bits of the instruction address are 10 is A1 = 1, A0 = 0), the bus cycle of the IF fetches only the specified instruction more than one of said instructions. The IF of the next instruction thus generates a bus cycle, and fetches two instructions. Figure 10-7 illustrates these operations.

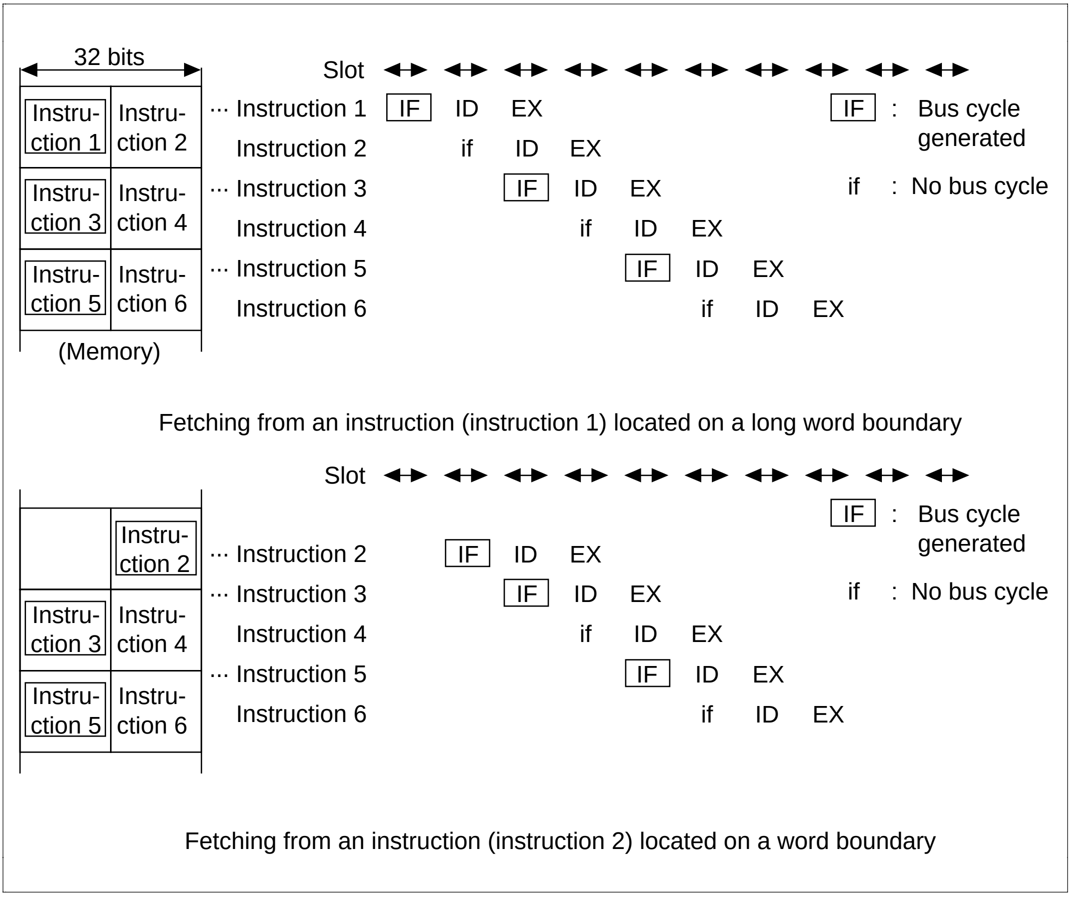
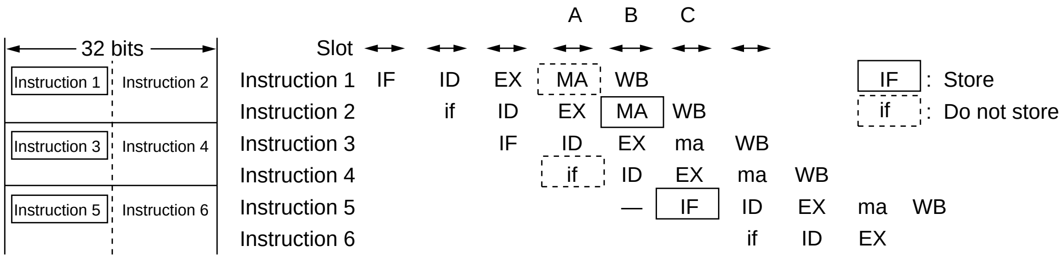


Figure 10-7 Relationship between IF and Location of Instructions in Memory

Relationship between Position of Instructions Located in Memory and Contention between IF and MA

When an instruction is located in memory, there are instruction fetch stages ("if", written in lower case) that do not generate bus cycles as explained in section 10.4.2 above. When an if is in contention with an MA, the slot will not split, as it does when an IF and an MA are in contention, because ifs and MAs can be executed simultaneously. Such slots execute in the number of cycles the MA requires for memory access, as illustrated in figure 10-8.

When programming, avoid contention of MA and IF whenever possible and pair MAs with ifs to increase the instruction execution speed. Instructions that have 4 (5)-stage pipelines of IF, ID, EX, MA, (WB) prevent stalls when they are located, so they start from the longword boundaries in memory (the position when the bottom 2 bits of instruction address are 00 is A1 = 0 and A0 = 0) because the MA of the instruction falls in the same slot as ifs that follow.



Notes: 1. MA in slot A is in contention with if, so no store occurs; MA in slot B is in contention with IF, so a store occurs.

2. In slot C ma and IF are in contention, so no store occurs.

Figure 10-8 Relationship between the Location of Instructions in Memory and Contention between IF and MA

10.2.2 Effects of Memory Load Instructions on Pipelines

Instructions that involve loading from memory access data in memory at the MA stage of the pipeline. In the case of a load instruction (instruction 1) and the following instruction (instruction 2), the EX stage of instruction 2 starts before the MA stage of instruction 1 ends.

When instruction 2 uses the same data that instruction 1 is loading, the contents of that register will not be ready, so any slot containing the MA of instruction and EX of instruction 2 will split. No split occurs, however, when instruction 2 is MAC @Rm+,@Rn+ and the destinations of Rm and load instruction 1 were the same.

The number of cycles in the slot generated by the split is the number of MA cycles plus the number of IF (or if) cycles, as illustrated in figure 10-9. This means the execution speed will be lowered if the instruction that will use the results of the load instruction is placed immediately after the load instruction. The instruction that uses the result of the load instruction will not slow down the program if placed one or more instructions after the load instruction.

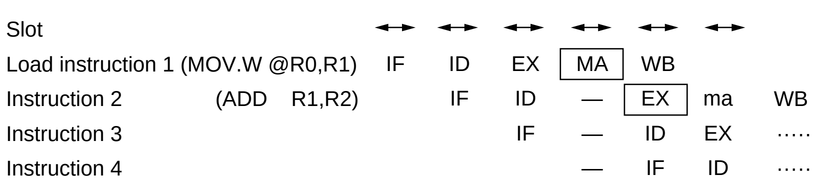


Figure 10-9 Effects of Memory Load Instructions on the Pipeline

10.2.3 Contention due to SR Update Instructions

Instructions (SR update instructions) that overwrite the M, Q, S, and T bits of the status register (SR) use the WB stage of the pipeline. If an instruction (instruction 2) that reads SR comes immediately after such an instruction, the data to be read is not yet ready and the EX stage of instruction 2 is stalled until the overwriting of the data in SR is complete. However, in the case of instructions that overwrite all the bits of SR, such as LDC Rm,SR; LDC.L@Rm+,SR; or RTE, no stall occurs due to the contention. The instructions that reads SR are STC SR,Rn; STC.L SR,@-Rn; and TRAPA. The status of the pipeline when a stall occurs is shown in Figure 10-10.

As the above makes clear, writing a program in such a way that an instruction that reads SR occurs immediately after an instruction that updates SR will cause the speed of execution to be reduced. If the instruction that reads SR occurs at least three instructions after the instruction that updates SR, no slowdown results.

				A	B	C		
Slot	◄►	◄►	◄►	◄►	◄►	◄►		
SR update instruction 1 (SETT)	IF	ID	EX	ma	[WB]			
Instruction 2 (STC SR, R1)		IF	ID	—	—	[EX]	ma	WB
Instruction 3			IF	—	—	ID	EX	· · ·
Instruction 4				—	—	IF	ID	· · ·

Figure 10-10 Affect on Pipeline of SR Update Instructions

10.2.4 Multiplier Access Contention

A multiplier-type instruction (multiply/accumulate calculations, multiplier instructions), an instruction in which the multiply and accumulate registers (MACH, MACL) are accessed, can cause a contention in the multiplier access.

In the multiplier instruction, the multiplier takes action regardless of the slots after the ending of the last MA. In the double precision (64 bytes) type multiplier instruction and the multiply/accumulate calculations instruction, the multiplier takes action in three states. In the single precision (32 bytes) type multiplier instruction, the action is taken in two states.

When MA (when there are two, the first MA takes precedence) of the multiplier instruction (multiply/accumulate calculations, multiplier instruction) contends with the multiplier access (mm) of the preceding multiplier instruction, the MA bus cycle is extended until the mm ends. The extended MA then becomes one slot.

The MA instruction which accesses the multiply/accumulate register (MACH, MACL) also accesses the multiplier. Similar to the multiplier instruction, the MA bus cycle is extended until the mm of the preceding multiplier-type instruction ends, and the extended MA becomes one slot. In particular, in the instruction (STS, STS.L), which reads out the multiply/accumulate register

(MACH, MACL,MA) is extended until one slot has elapsed after the ending of the mm, the extended MA becomes one slot.

On the other hand, when the instruction has two MAs, the succeeding ID instruction is stalled for a one-slot period.

Because the multiplier-type instruction and the multiply/accumulate register access instruction both have MA cycles, a contention with IF may develop.

Examples of multiplier access contention are shown in figures 10-10 and 10-11. In these cases, the contention between MA and IF is not taken into consideration.

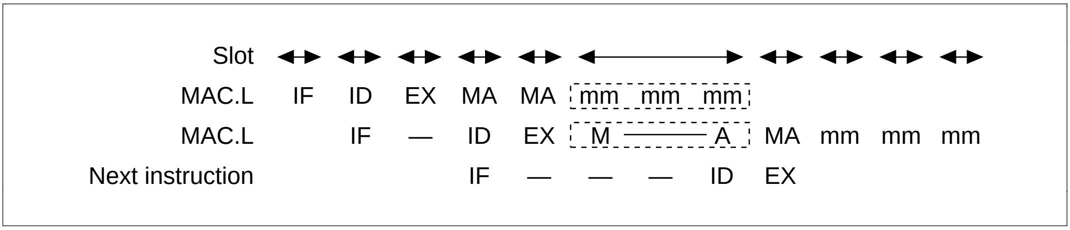


Figure 10-11 Contention between Two MAC.L Instructions

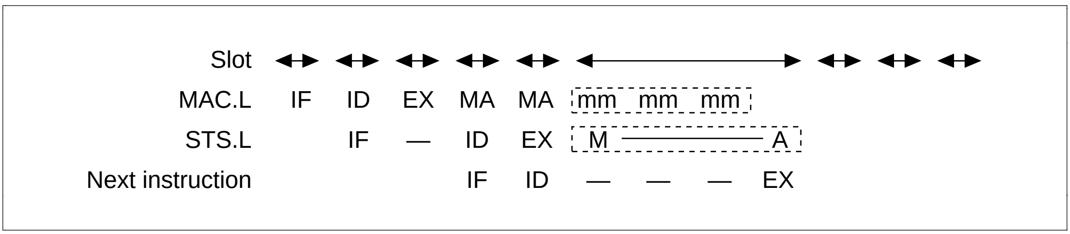


Figure 10-12 Contention between the MAC.L and STS.L Instructions

10.2.5 FPU Contention (SH-3E Only)

In addition to the LDS and STS instructions, which move data between the CPU and FPU, loading and storing floating point numbers also uses the MA stage of the pipeline. Consequently, such instructions create contention with the IF stage.

If the register to which the result of a floating point arithmetic calculation instruction, the FMOV instruction, or a floating point number load instruction is stored is read by the next instruction, the execution of this instruction (the next instruction) is delayed by one slot cycle (Figure 10-13).

Slot	◄►	◄►	◄►	◄►	◄►	◄►	◄►	◄►	◄►
Floating point arithmetic calculation instruction (FADD FR1, FR2)	IF	ID	E1	E2	SF				
Next floating point instruction (FMOV FR2, FR2)		IF	DF	—	E1	E2	SF		

Figure 10-13 FPU Contention 1

If the LDS or LDS.L instruction is used to change the value of FPSCR, the execution of the next instruction (if it is a floating point instruction) is delayed by one slot cycle (Figure 10-14).

Slot	◄►	◄►	◄►	◄►	◄►	◄►	◄►	◄►	◄►
Instruction 1 (LDS R2, FPSCR)	IF	ID	E1	E2	SF				
Floating point arithmetic calculation instruction (FADD FR4, FR5)		IF	DF	—	—	E1	E2	SF	

Figure 10-14 FPU Contention 2

If the preceding instruction was a floating point arithmetic calculation instruction (using the STS or STS.L instruction), the execution of an instruction that reads the value of FPSCR is delayed by one slot cycle (Figure 10-15).

Slot	◄►	◄►	◄►	◄►	◄►	◄►	◄►	◄►	◄►
Floating point arithmetic calculation instruction (FADD FR6, FR9)	IF	ID	E1	E2	SF				
Instruction 2 (STS FPSCR, R3)		IF	DF	—	—	E1	E2	SF	

Figure 10-15 FPU Contention 3

The FDIV and FSQRT instructions require 13 cycles in the E1 stage. During this period, no other floating point instruction may enter the E1 stage. If another floating point instruction is encountered before the FDIV or FSQRT instruction has finished using the E1 stage, the fixed slot duration for the execution of that instruction is delayed, and the instruction enters the E1 stage only after the FDIV or FSQRT instruction has entered the E2 stage (Figure 10-16).

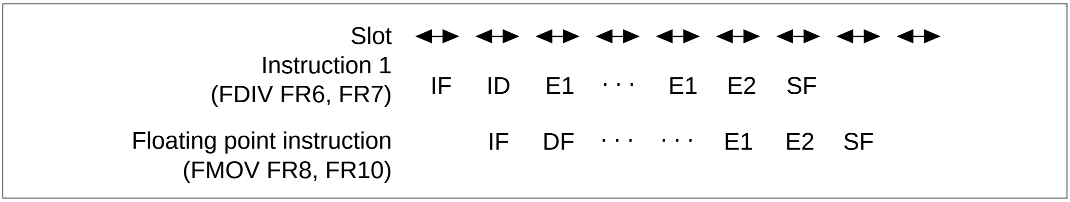


Figure 10-16 FPU Contention 4

However, if contention arises because the preceding FDIV or FSQRT instruction and the FPU calculation which follows it use the same register, the FDIV or FSQRT instruction enters the E1 stage after the execution of the SF instruction.

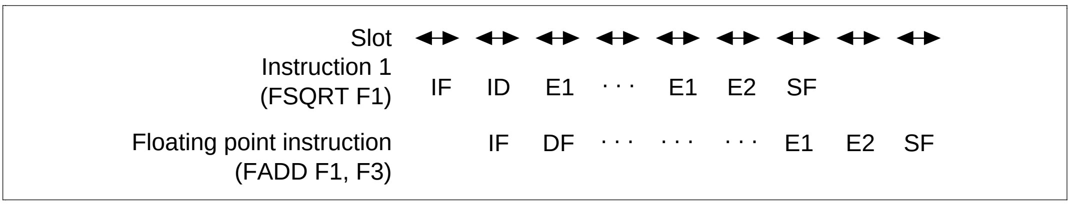


Figure 10-17 FPU Contention 5

10.2.6 Contention between DSP Data Operation Instructions and Store Instructions (SH3-DSP Only)

When DSP operations are executed by the DSP unit and the results are stored in memory by the next instruction, contention occurs just as with memory load instructions. In such cases, the data store of the MA stage of the following instruction is extended until the data operation of the WB/DSP stage of the previous instruction ends. Since the operation is executed in the EX stage by the CPU core, however, no stall cycle is produced. Figure 10-18 shows the relationship between DSP unit data operation instructions and store instructions; figure 10-19 shows the relationship to the CPU core.

(↔ : Slot)									
Instruction 1 (PADD X0,Y0,A0)	IF	ID	EX	MA	**W/D**				
Instruction 2 (MOVX A0,@Ra)		IF	ID	EX	—	**MA**	W/D		
Instruction 3			IF	ID	—	EX	MA	W/D	
Instruction 4				IF	—	ID	EX	MA	W/D

Figure 10-18 Relationship between DSP Engine Operation Instructions and Store Instructions

(↔ : Slot)								
Instruction 1 (ADD Ra,Rb)	IF	ID	**EX**	MA	W/D			
Instruction 2 (MOV Rb,@Rc)		IF	ID	**EX**	MA	W/D		
Instruction 3			IF	ID	EX	MA	W/D	
Instruction 4				IF	ID	EX	MA	W/D

Figure 10-19 Relationship between CPU Core Operation Instructions and Store Instructions

10.2.7 Relationship between Load and Store Instructions (SH3-DSP Only)

When data is loaded from memory to the destination register and the register is then specified as the source operand for a following store instruction, the preceding instruction's load is executed in the WB/DSP stage and the following instruction's store is executed in the MA stage. These stages are executed in exactly the same cycle. Nevertheless, they do not contend. The CPU core and DSP unit use the same data transfer method. In this case, when the data input to the internal bus is stored to the destination register, the same data is simultaneously output again to the internal bus. In the end, the store instruction's output operation never actually happens.

	◄►	◄►	◄►	◄►	◄►	◄►	◄►	◄►	◄►	◄►	: Slot
Instruction 1 (MOV.L @Ra,Rn)	IF	ID	EX	MA	**W/D**						
Instruction 2 (MOV.L Rn,@Rb)		IF	ID	EX	**MA**	W/D					
Instruction 3			IF	ID	EX	MA	W/D				
Instruction 4				IF	ID	EX	MA	W/D			

Figure 10-20 Relationship between Load and Store Instructions in the CPU Core

	◄►	◄►	◄►	◄►	◄►	◄►	◄►	◄►	◄►	◄►	: Slot
Instruction 1 (MOVS.L @R4,Ds)	IF	ID	EX	MA	**W/D**						
Instruction 2 (MOVS.L Ds,@R5)		IF	ID	EX	**MA**	W/D					
Instruction 3			IF	ID	EX	MA	W/D				
Instruction 4				IF	ID	EX	MA	W/D			

Figure 10-21 Relationship between Load and Store Instructions in the DSP Unit

10.3 Programming Guidelines

10.3.1 Correspondence between Contention and Instructions

The types of correspondence between contention and instructions can be summarized as follows.

(1) Instructions that do not cause contention

(2) Instructions where a memory access (MA) causes contention with an instruction fetch (IF)

(3) Instructions where a write back (WB) to SR causes contention with a SR update

(4) Instructions where a memory access (MA) causes contention with an instruction fetch (IF), and in addition a write back (WB) to memory causes contention with a memory load

(5) Instructions where a memory access (MA) causes contention with an instruction fetch (IF), and in addition a write back (WB) to SR causes contention with a SR update

(6) Instructions where a memory access (MA) causes contention with an instruction fetch (IF), and in addition a multiplier access (mm) causes contention with the multiplier.

(7) Instructions where a memory access (MA) causes contention with an instruction fetch (IF), a multiplier access (mm) causes contention with the multiplier, and in addition a write back (WB) causes contention with a memory load

(8) Instructions that cause contention with the MOVX.W, MOVS.W, or MOVS.L instruction

Table 10-1 shows the correspondence between types of contention and instructions.

Table 10-1 Types of Contention and Instructions

Contention	Cycles	Stages	Instructions
None	1	5	Inter-register transfer instructions
	1	5	Inter-register operations (except multiplier type instructions)
	1	5	Inter-register logic operation instructions
	1	5	Shift instructions
	3/1	3	Conditional branch instructions
	2/1	3	Delayed conditional branch instruction
	2	3	Unconditional branch instructions
	2	5	Unconditional branch instructions (PR)
	1	5	System control instructions
	1	3	NOP instruction
	5	5	LDC instruction (SR)
	7	7	LDC.L instruction
	4	5	RTE instruction
	6	6	TRAP instruction
	4	6	SLEEP instruction
	1	5	DSP data operation instructions MOVX.W (load) and MOVY.W (load) instructions
• MA contends with IF	1	4	Memory store instructions
	1	5	Memory store instructions (pre-decrement)
	1	4	Cache instruction
	3	6	Memory logic operation instruction
	1	4	LDTLB instruction
	1	5	STS.L instruction (PR)
	1	5	STC.L instruction (excluding bank registers)
	2	6	STC.L instruction (bank registers)
	1	5	MOVS.W (load) and MOVS.L (load) instructions
• Causes DSP operation contention	1	4	MOVX.W (store) and MOVS.L (store) instructions

Table 10-1 Types of Contention and Instructions (cont)

Contention	Cycles	Stages	Instructions
• Contention caused by SR update	1	5	Arithmetic calculation instructions between SR updated registers (excluding instructions involving multiplication)
	1	5	Logical calculation instructions between SR updated registers
	1	5	SR update shift instructions
	1	5	SR update system control instructions
• MA contends with IF • Causes memory load contention	1	5	Memory load instructions
	1	5	LDS.L instruction (PR)
	1	5	LDC.L instruction
• MA contends with IF • Contention caused by SR update	3	7	SR update memory logical calculation instructions
	3	7	TAS instruction
• MA contends with IF • Causes multiplier contention	2 (to 5)*	8	Multiply and accumulate calculation instructions
	2 (to 5)*	8	Double-length multiply and accumulate calculation instructions
	1 (to 3)*	6	Multiplication instructions (excluding PWULS)
	2 (to 5)*	8	Double-length multiplication instructions
	1	4	Register to MAC transfer instructions
	1	4	Memory to MAC transfer instructions
	1	5	MAC to memory transfer instructions
• MA contends with IF • Causes DSP operation contention	1	4	MOVS.W (store) and MOVS.L (store) instructions
• MA contends with IF • Causes multiplier contention • Causes memory load contention • Causes DSP operation contention	1	5	MAC/DSP to register transfer instructions
• Causes MOVX.W, MOVS.W, or MOVS.L instruction	1	5	PLDS and PSTS instructions

Note: * Indicates the normal number of cycles. The figures in parentheses are the cycles when contention also occurs with the previous instruction.

10.3.2 Increasing Instruction Execution Speed

To improve instruction execution speed, consider the following when programming:

- To prevent contention between MA and IF, locate instructions that have MA stages so they start from the longword boundaries of on-chip memory (the position when the bottom two bits of the instruction address are 00 is A1 = 0 and A0 = 0) wherever possible.
- The instruction that immediately follows an instruction that loads from memory should not use the same destination register as the load instruction. This will avoid causing contention with the memory load triggered by the write back (WB).
- Locate two instructions that do not read SR immediately after any instruction that overwrites the M, Q, S, and T bits of SR. This will prevent contention with SR update instructions from occurring.
- Locate instructions that use the multiplier nonconsecutively (excluding PWULS).
- Immediately following a data operation using the DSP unit, do not use an instruction that transfers data to memory or the CPU core from the register where the operation result is stored. By placing some other instruction in between, contention can be avoided.
- Do not use MOVX.W, MOVS.W, or MOVS.L to perform a memory store immediately following a PLDS or PSTS instruction using the DSP unit. Also, do not specify a PLDS or PSTS instruction in parallel with a memory store instruction using MOVX.W.

10.3.3 Number of Cycles

These instructions are designed to require only one cycle for execution. Of these one-cycle instructions, some never cause contention and some can cause contention.

Some instructions may require two or more cycles even if no contention occurs. Instructions that require two or more cycles include instructions that execute access memory twice or more, such as branching instructions that update the branching destination address, memory logical calculation instructions, and certain system control instructions. Further examples include instructions that access both memory and the multiplier, such as multiplication instructions and accumulate-and-add instructions.

Among instructions that require two or more cycles, some never cause contention and some can cause contention.

In order to create efficient programs, it is essential to keep in mind the need to increase execution speed by avoiding contention and also to use instructions that require few cycles to execute.

10.4 Operation of Instruction Pipelines

This section describes the operation of the instruction pipelines. By combining these with the rules described so far, the way pipelines flow in a program and the number of instruction execution cycles can be calculated.

In the following figures, “Instruction A” refers to the instruction being discussed. When “IF” is written in the instruction fetch stage, it may refer to either “IF” or “if”. When there is contention between IF and MA, the slot will split, but the manner of the split is not discussed in the tables, with a few exceptions. When a slot has split, see section 10.2, Contention between Instruction Fetch (IF) and Memory Access (MA). Base your response on the rules for pipeline operation given there.

Table 10-2 shows the number of instruction stages and number of execution cycles as follows:

- Type: Given by function
- Category: Categorized by differences in instruction operation
- Instructions: Gives a mnemonic for the instruction concerned
- Cycles: The number of execution cycles when there is no contention
- Stages: The number of stages in the instruction
- Contention: Indicates the contention that occurs

Table 10-2 Number of Instruction Stages and Execution Cycles

Type	Category	Instruction		Cycles	Stages	Contention
Data transfer instructions	Register-register transfer instructions	MOV	#imm,Rn	1	5	—
		MOV	Rm,Rn			
		MOVA	@(disp,PC),R0			
		MOVT	Rn			
		SWAP.B	Rm,Rn			
		SWAP.W	Rm,Rn			
		XTRCT	Rm,Rn			
	Memory load instructions	MOV.W	@(disp,PC),Rn	1	5	• Contention occurs if the instruction placed immediately after this one uses the same destination register • MA contends with IF
		MOV.L	@(disp,PC),Rn			
		MOV.B	Rm,@Rn			
		MOV.W	Rm,@Rn			
		MOV.L	Rm,@Rn			
		MOV.B	@Rm+,Rn			
		MOV.W	@Rm+,Rn			
		MOV.L	@Rm+,Rn			
		MOV.B	@(disp,Rm),R0			
		MOV.W	@(disp,Rm),R0			
		MOV.L	@(disp,Rm),Rn			
		MOV.B	@(R0,Rm),Rn			
		MOV.W	@(R0,Rm),Rn			
		MOV.L	@(R0,Rm),Rn			
		MOV.B	@(disp,GBR),R0			
		MOV.W	@(disp,GBR),R0			
		MOV.L	@(disp,GBR),R0			
	Memory store instructions	MOV.B	@Rm,Rn	1	4	• MA contends with IF
		MOV.W	@Rm,Rn			
		MOV.L	@Rm,Rn			
		MOV.B	R0,@(disp,Rn)			
		MOV.W	R0,@(disp,Rn)			
		MOV.L	Rm,@(disp,Rn)			

Table 10-2 Number of Instruction Stages and Execution Cycles (cont)

Type	Category	Instruction		Cycles	Stages	Contention
Data transfer instructions (cont)	Memory store instructions (cont)	MOV.B	Rm,@(R0,Rn)	1	4	• MA contends with IF
		MOV.W	Rm,@(R0,Rn)			
		MOV.L	Rm,@(R0,Rn)			
		MOV.B	R0,@(disp,GBR)			
		MOV.W	R0,@(disp,GBR)			
		MOV.L	R0,@(disp,GBR)			
	Memory store instructions (pre-decrement)	MOV.B	Rm,@-Rm	1	5	• MA contends with IF
		MOV.W	Rm,@-Rm			
		MOV.L	Rm,@-Rm			
	Cache instruction	PREF	@Rn	1/2*[1]	4	• MA contends with IF
Arithmetic instructions	Arithmetic operation instruction between registers (excluding multiply instructions)	ADD	Rm,Rn	1	5	—
		ADD	#imm,Rn			
		EXTS.B	Rm,Rn			
		EXTS.W	Rm,Rn			
		EXTU.B	Rm,Rn			
		EXTU.W	Rm,Rn			
		NEG	Rm,Rn			
		SUB	Rm,Rn			
	SR update arithmetic operation instruction between registers (excluding multiply instructions)	ADDC	Rm,Rn	1	5	• Contention occurs if the instruction following this instruction, or the instruction after that, reads from SR.
		ADDV	Rm,Rn			
		CMP/EQ	#imm,R0			
		CMP/EQ	Rm,Rn			
		CMP/HS	Rm,Rn			
		CMP/GE	Rm,Rn			
		CMP/HI	Rm,Rn			
		CMP/GT	Rm,Rn			
		CMP/PL	Rn			
		CMP/PZ	Rn			
		CMP/STR	Rm,Rn			

Table 10-2 Number of Instruction Stages and Execution Cycles (cont)

Type	Category	Instruction	Cycles	Stages	Contention

Arithmetic instructions (cont)	SR update arithmetic operation instruction between registers (excluding multiply instructions)	`DIV1 Rm,Rn`
		`DIV0S Rm,Rn`
		`DIV0U`
		`DT Rn`
		`NEGC Rm,Rn`
		`SUBC Rm,Rn`
		`SUBV Rm,Rn`
			1	5	• Contention occurs if the instruction following this instruction, or the instruction after that, reads from SR.

	Multiply/ accumulate instruction	`MAC.W @Rm+,@Rn+`	2 (to 5)*[2]	8	• Causes multiplier contention • MA contends with IF
	Double length/ multiply accumulate instruction	`MAC.L @Rm+,@Rn+`	2 (to 5)*[2]	8	• Causes multiplier contention • MA contends with IF
	Multiplic-ation instruction	`MULS.W Rm,Rn` `MULU.W Rm,Rn`	1 (to 3)*[2]	6	• Causes multiplier contention • MA contends with IF
	Double length multipli-cation instructions	`DMULS.L Rm,Rn` `DMULU.L Rm,Rn`	2 (to 5)*[2]	8	• Causes multiplier contention • MA contends with IF

Table 10-2 Number of Instruction Stages and Execution Cycles (cont)

Type	Category	Instruction		Cycles	Stages	Contention
Logic operation instructions	Register to register logic operation instructions	AND	Rm,Rn	1	5	—
		AND	#imm,R0			
		NOT	Rm,Rn			
		OR	Rm,Rn			
		OR	#imm,R0			
		XOR	Rm,Rn			
		XOR	#imm,R0			
	Logical calculation instructions between SR updated registers	TST	Rm,Rn	1	5	• Contention occurs if the instruction following this instruction, or the instruction after that, reads from SR
		TST	#imm,R0			
	Memory logic operations instructions	AND.B	#imm,@(R0,GBR)	3	6	• MA contends with IF
		OR.B	#imm,@(R0,GBR)			
		XOR.B	#imm,@(R0,GBR)			
	SR update memory logical calculation instructions	TST.B	#imm,@(R0,GBR)	3	7	• Contention occurs if the instruction following this instruction, or the instruction after that, reads from SR • MA contends with IF
	TAS instruction	TAS.B	@Rn	3/4*3	7	• MA contends with IF

Table 10-2 Number of Instruction Stages and Execution Cycles (cont)

Type	Category	Instruction	Cycles	Stages	Contention
Shift instructions	Shift instructions	SHLL2 Rn SHLR2 Rn SHLL8 Rn SHLR8 Rn SHLL16 Rn SHLR16 Rn SHAD Rm,Rn SHLD Rm,Rn	1	5	—
	SR update shift instructions	ROTL Rn ROTR Rn ROTCL Rn ROTCR Rn SHAL Rn SHAR Rn SHLL Rn SHLR Rn	1	5	• Contention occurs if the instruction following this instruction, or the instruction after that, reads from SR
Branch instructions	Conditional branch instructions	BF label BT label	3/1*[4]	3	—
	Delayed conditional branch instructions	BF/S label BT/S label	2/1*[4]	3	—
	Uncondi-tional branch instructions	BRA label BRAF Rm JMP @Rm RTS	2	3	—
	Uncondi-tional branch instructions (PR)	BSR label BSRF Rm JSR @Rm	2	5	—

Table 10-2 Number of Instruction Stages and Execution Cycles (cont)

Type	Category	Instruction		Cycles	Stages	Contention
System control instructions	System control ALU instructions	LDC	Rm,GBR	1/3*5	5	—
		LDC	Rm,VBR			
		LDC	Rm,SSR			
		LDC	Rm,SPC			
		LDC	Rm,MOD			
		LDC	Rm,RE			
		LDC	Rm,RS			
		LDC	Rm,R0_BANK			
		LDC	Rm,R1_BANK			
		LDC	Rm,R2_BANK			
		LDC	Rm,R3_BANK			
		LDC	Rm,R4_BANK			
		LDC	Rm,R5_BANK			
		LDC	Rm,R6_BANK			
		LDC	Rm,R7_BANK			
		SETRC	Rm	3	5	
		SETRC	#imm			
		LDRE	@(disp,PC)			
		LDRS	@(disp,PC)			
		LDS	Rm,PR	1	5	
		STC	SR,Rn			
		STC	GBR,Rn			
		STC	VBR,Rn			
		STC	SSR,Rn			
		STC	SPC,Rn			
		STC	MOD,Rn			
		STC	RE,Rn			
		STC	RS,Rn			
		STC	R0_BANK,Rn			
		STC	R1_BANK,Rn			
		STC	R2_BANK,Rn			
		STC	R3_BANK,Rn			
		STC	R4_BANK,Rn			

Table 10-2 Number of Instruction Stages and Execution Cycles (cont)

Type	Category	Instruction		Cycles	Stages	Contention
System control instructions (cont)	System control ALU instructions	STC STC STC STS	R5_BANK,Rn R6_BANK,Rn R7_BANK,Rn PR,Rn	1	5	—
	SR update system control instructions	CLRS CLRT SETS SETT		1	5	• Contention occurs if the instruction following this instruction, or the instruction after that, reads from SR
	LDTLB instruction	LDTLB		1	4	• MA contends with IF
	NOP instruction	NOP		1	3	—
	LDC instructions (SR)	LDC	Rm,SR	5	5	—
	LDC.L instructions (SR)	LDC.L	@Rm+,SR	7	7	—
	LDS.L instructions (PR)	LDS.L	@Rm+,PR	1	5	• Contention occurs when an instruction that uses the same destination register is placed immediately after this instruction • MA contends with IF
	STS.L instruction (PR)	STS.L	PR,@–Rn	1	5	• MA contends with IF

Table 10-2 Number of Instruction Stages and Execution Cycles (cont)

Type	Category	Instruction		Cycles	Stages	Contention
System control instructions (cont)	LDC.L instructions	LDC.L	@Rm+,GBR	1/5*6	5	• Contention occurs when an instruction that uses the same destination register is placed immediately after this instruction • MA contends with IF
		LDC.L	@Rm+,VBR			
		LDC.L	@Rm+,SSR			
		LDC.L	@Rm+,SPC			
		LDC.L	@Rm+,MOD			
		LDC.L	@Rm+,RE			
		LDC.L	@Rm+,RS			
		LDC.L	@Rm+,R0_BANK			
		LDC.L	@Rm+,R1_BANK			
		LDC.L	@Rm+,R2_BANK			
		LDC.L	@Rm+,R3_BANK			
		LDC.L	@Rm+,R4_BANK			
		LDC.L	@Rm+,R5_BANK			
		LDC.L	@Rm+,R6_BANK			
		LDC.L	@Rm+,R7_BANK			
	STC.L instructions	STC.L	SR,@–Rn	1/2*1	5	• MA contends with IF
		STC.L	GBR,@–Rn			
		STC.L	VBR,@–Rn			
		STC.L	SSR,@–Rn			
		STC.L	SPC,@–Rn			
		STC.L	MOD,@-Rn			
		STC.L	RE,@-Rn			
		STC.L	RS,@-Rn			
		STC.L	R0_BANK,@–Rn	2	6	• MA contends with IF
		STC.L	R1_BANK,@–Rn			
		STC.L	R2_BANK,@–Rn			
		STC.L	R3_BANK,@–Rn			
		STC.L	R4_BANK,@–Rn			
		STC.L	R5_BANK,@–Rn			
		STC.L	R6_BANK,@–Rn			
		STC.L	R7_BANK,@–Rn			

Table 10-2 Number of Instruction Stages and Execution Cycles (cont)

Type	Category	Instruction		Cycles	Stages	Contention
System control instructions (cont)	Register → MAC/DSP transfer instruction	CLRMAC		1	4	• Contention occurs with multiplier • MA contends with IF
		LDS	Rm,MACH			
		LDS	Rm,MACL			
		LDS	Rm,DSR			
		LDS	Rm,A0			
		LDS	Rm,X0			
		LDS	Rm,X1			
		LDS	Rm,Y0			
		LDS	Rm,Y1			
	Memory → MAC/DSP transfer instructions	LDS.L	@Rm+,MACH	1	4	• Contention occurs with multiplier • MA contends with IF
		LDS.L	@Rm+,MACL			
		LDS.L	@Rm+,DSR			
		LDS.L	@Rm+,A0			
		LDS.L	@Rm+,X0			
		LDS.L	@Rm+,X1			
		LDS.L	@Rm+,Y0			
		LDS.L	@Rm+,Y1			
	MAC/DSP → register transfer instruction	STS	MACH,Rn	1	5	• Contention occurs with multiplier • Contention occurs when an instruction that uses the same destination register is placed immediately after this instruction • MA contends with IF
		STS	MACL,Rn			
		STS	DSR,Rn			
		STS	A0,Rn			
		STS	X0,Rn			
		STS	X1,Rn			
		STS	Y0,Rn			
		STS	Y1,Rn			
	MAC/DSP → memory transfer instruction	STS.L	MACH,@–Rn	1	5	• Contention occurs with multiplier • MA contends with IF
		STS.L	MACL,@–Rn			
		STS.L	DSR,@–Rn			
		STS.L	A0,@–Rn			
		STS.L	X0,@–Rn			
		STS.L	X1,@–Rn			
		STS.L	Y0,@–Rn			
		STS.L	Y1,@–Rn			

Table 10-2 Number of Instruction Stages and Execution Cycles (cont)

Type	Category	Instruction	Cycles	Stages	Contention
System control instructions (cont)	RTE instruction	`RTE`	4	5	—
	TRAP instruction	`TRAPA #imm`	6/8*7	6/8*7	—
	SLEEP instruction	`SLEEP`	4	6	—

	Register → MAC/DSP transfer instruction	`CLRMAC`
		`LDS Rm,MACH`
		`LDS Rm,MACL`
		`LDS Rm,DSR`
		`LDS Rm,A0`
		`LDS Rm,X0`
		`LDS Rm,X1`
		`LDS Rm,Y0`
		`LDS Rm,Y1`
			4	1	• Causes multiplier contention
					• MA contends with IF

	Memory → MAC/DSP transfer instructions	`LDS.L @Rm+,MACH`
		`LDS.L @Rm+,MACL`
		`LDS.L @Rm+,DSR`
		`LDS.L @Rm+,A0`
		`LDS.L @Rm+,X0`
		`LDS.L @Rm+,X1`
		`LDS.L @Rm+,Y0`
		`LDS.L @Rm+,Y1`
			4	1	• Causes multiplier contention
					• MA contends with IF

	MAC/DSP → register transfer instruction	`STS MACH,Rn`
		`STS MACL,Rn`
		`STS DSR,Rn`
		`STS A0,Rn`
		`STS X0,Rn`
		`STS X1,Rn`
		`STS Y0,Rn`
		`STS Y1,Rn`
			5	1	• Causes multiplier contention
					• Contention occurs when an instruction that uses the same destination register is placed immediately after this instruction
					• MA contends with IF

Table 10-2 Number of Instruction Stages and Execution Cycles (cont)

Type	Category	Instruction	Cycles	Stages	Contention
System control instructions (cont)	MAC/DSP → memory transfer instruction	STS.L MACH,@–Rn STS.L MACL,@–Rn STS.L DSR,@–Rn STS.L A0,@–Rn STS.L X0,@–Rn STS.L X1,@–Rn STS.L Y0,@–Rn STS.L Y1,@–Rn	4	1	• Causes multiplier contention • MA contends with IF
	RTE instruction	RTE	5	4	—
	TRAP instruction	TRAPA #imm	9	8	—
	SLEEP instruction	SLEEP	3	3	—

	Register → DSP transfer instructions	CLRMAC
		LDS Rm,MACH
		LDS Rm,MACL
			1	4	• Causes multiplier contention
					• MA contends with IF

		LDS Rm,DSR
		LDS Rm,A0
		LDS Rm,X0
		LDS Rm,X1
		LDS Rm,Y0
		LDS Rm,Y1
			1	4	—

	Memory → DSP transfer instructions	LDS.L @Rm+,MACH LDS.L @Rm+,MACL	1	4	• Causes multiplier contention • MA contends with IF
		DS.L @Rm+,DSR DS.L @Rm+,A0 DS.L @Rm+,X0 DS.L @Rm+,X1 DS.L @Rm+,Y0 DS.L @Rm+,Y1	1	4	—

Table 10-2 Number of Instruction Stages and Execution Cycles (cont)

Type	Category	Instruction	Cycles	Stages	Contention
System control instructions (cont)	DSP → register transfer instructions	STS MACH,Rn STS MACL,Rn STS DSR,Rn STS A0,Rn STS X0,Rn STS X1,Rn STS Y0,Rn STS Y1,Rn	1	5	• Causes multiplier contention • Contention occurs when an instruction that uses the same destination register is placed immediately after this instruction • MA contends with IF • Causes contention with DSP operation.
	DSP → memory transfer instructions	STS.L MACH,@-Rn STS.L MACL,@-Rn	1	4	• Causes multiplier contention • MA contends with IF
		STS.L DSR,@-Rn STS.L A0,@-Rn STS.L X0,@-Rn STS.L X1,@-Rn STS.L Y0,@-Rn STS.L Y1,@-Rn	1	4	—
	RTE instruction	RTE	4	5	—
	TRAP instruction	TRAPA #imm	8	9	—
	SLEEP instruction	SLEEP	3	3	—
DSP data transfer instructions	X memory load instructions	NOPX MOVX.W @Ax,Dx MOVX.W @Ax+,Dx MOVX.W @Ax+Ix,Dx	1	5	—
	X memory store instructions	MOVX.W Da,@Ax MOVX.W Da,@Ax+ MOVX.W Da,@Ax+Ix	1	4	• Causes contention with DSP operation.

Table 10-2 Number of Instruction Stages and Execution Cycles (cont)

Type	Category	Instruction	Cycles	Stages	Contention
DSP data transfer instructions (cont)	Y memory load instructions	NOPY	1	5	—
		MOVY.W @Ay,Dy			
		MOVY.W @Ay+,Dy			
		MOVY.W @Ay+Ix,Dy			
	Y memory store instructions	MOVY.W Da,@Ay	1	4	• Causes contention with DSP operation.
		MOVY.W Da,@Ay+			
		MOVY.W Da,@Ay+Iy			
	Single load instructions	MOVS.W @-As,Ds	1	5	• MA contends with IF
		MOVS.W @As,Ds			
		MOVS.W @As+,Ds			
		MOVS.W @As+Is,Ds			
		MOVS.L @-As,Ds			
		MOVS.L @As,Ds			
		MOVS.L @As+,Ds			
		MOVS.L @As+Is,Ds			
	Single store instructions	MOVS.W Ds,@-As	1	5	• MA contends with IF • Causes contention with DSP operation.
		MOVS.W Ds,@As			
		MOVS.W Ds,@As+			
		MOVS.W Ds,@As+Is			
		MOVS.L Ds,@-As			
		MOVS.L Ds,@As			
		MOVS.L Ds,@As+			
		MOVS.L Ds,@As+Is			
DSP operation instructions		PADD Sx,Sy,Dz(Du)	1	5	—
		DCT PADD Sx,Sy,Dz			
		DCF PADD Sx,Sy,Dz			
		PSUB Sx,Sy,Dz(Du)			
		DCT PSUB Sx,Sy,Dz			
		DCF PSUB Sx,Sy,Dz			
		PCOPY Sx,Dz			
		DCT PCOPY Sx,Dz			
		DCF PCOPY Sx,Dz			

Table 10-2 Number of Instruction Stages and Execution Cycles (cont)

Type	Category	Instruction	Cycles	Stages	Contention
DSP operation instructions (cont)		PCOPY Sy,Dz	1	5	—
		DCT PCOPY Sy,Dz			
		DCF PCOPY Sy,Dz			
		PDMSB Sx,Dz			
		DTC PDMSB Sx,Dz			
		DCF PDMSB Sx,Dz			
		PDMSB Sy,Dz			
		DCT PDMSB Sy,Dz			
		DCF PDMSB Sy,Dz			
		PINC Sx,Dz			
		DCT PINC Sx,Dz			
		DCF PINC Sx,Dz			
		PINC Sy,Dz			
		DCT PINC Sy,Dz			
		DCF PINC Sy,Dz			
		PNEG Sx,Dz			
		DCT PNEG Sx,Dz			
		DCF PNEG Sx,Dz			
		PNEG Sy,Dz			
		DCT PNEG Sy,Dz			
		DCF PNEG Sy,Dz			
		PDEC Sx,Dz			
		DTC PDEC Sx,Dz			
		DCF PDEC Sx,Dz			
		PDEC Sy,Dz			
		DTC PDEC Sy,Dz			
		DCF PDEC Sy,Dz			
		PCLR Dz			
		DCT PCLR Dz			
		DCF PCLR DZ			

Table 10-2 Number of Instruction Stages and Execution Cycles (cont)

Type	Category	Instruction	Cycles	Stages	Contention
DSP operation instructions (cont)		PADDC Sx,Sy,Dz	1	5	—
		PSUBC Sx,Sy,Dz			
		PCMP Sx,Sy			
		PABS Sx,Dz			
		PABS Sy,Dz			
		PRNDSx,Dz			
		PRNDSy,Dz			
		POR Sx,Sy,Dz	1	5	—
		DCT POR Sx,Sy,Dz			
		DCF POR Sx,Sy,Dz			
		PAND Sx,Sy,Dz			
		DCT PAND Sx,Sy,Dz			
		DCF PAND Sx,Sy,Dz			
		PXOR Sx,Sy,Dz			
		DCT PXOR Sx,Sy,Dz			
		DCF PXOR Sx,Sy,Dz			
	Shift instructions	PSHA Sx,Sy,Dz	1	5	—
		DCT PSHA Sx,Sy,Dz			
		DCF PSHA Sx,Sy,Dz			
		PSHA #imm,Dz			
		PSHL Sx,Sy,Dz			
		DCT PSHL Sx,Sy,Dz			
		DCF PSHL Sx,Sy,Dz			
		PSHL #imm,Dz			
		PMULS Se,Sf,Dg	1	5	—

Table 10-2 Number of Instruction Stages and Execution Cycles (cont)

Type	Category	Instruction	Cycles	Stages	Contention
DSP operation instructions (cont)		PSTS MACH,Dz DTC PSTS MACH,Dz DCF PSTS MACH,Dz PSTS MACL,Dz DCT PSTS MACL,Dz DCF PSTS MACL,Dz PLDS Dz,MACH DCT PLDS Dz,MACH DCF PLDS Dz,MACH PLDS Dz,MACL DCT PLDS Dz,MACL DCF PLDS Dz,MACL	1	5	• Contends with MOVX.W, MOVS.W, and MOVS.L

Notes:
1. Two cycles on the SH3-DSP.
2. Indicates the normal minimum number of execution states (the number in parentheses is the number of cycles when there is contention with following instructions).
3. Four cycles on the SH3-DSP.
4. One state when there is no branch.
5. Three cycles on the SH3-DSP.
6. Five cycles on the SH3-DSP.
7. Eight cycles and eight stages on the SH3-DSP.

10.4.1 Data Transfer Instructions

(1) Register to Register Transfer Instructions

Instruction Types:

```
MOV      #imm, Rn
MOV      Rm, Rn
MOVA     @(disp, PC), R0
MOVT     Rn
SWAP.B   Rm, Rn
SWAP.W   Rm, Rn
XTRCT    Rm, Rn
```

Pipeline:

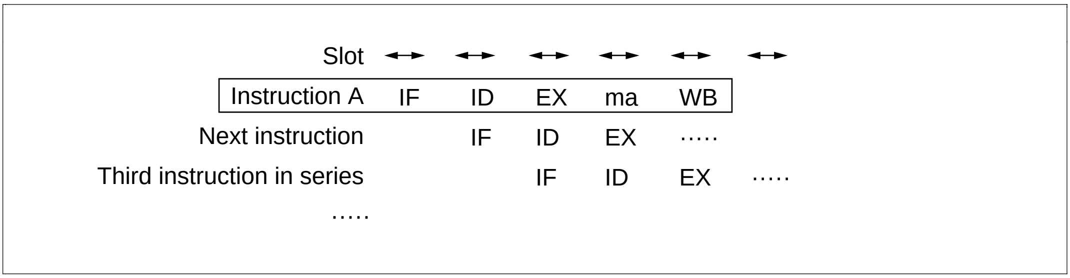


Figure 10-22 Register to Register Transfer Instruction Pipeline

Operation Description:

The pipeline ends after five stages: IF, ID, Ex, ma, and WB. In the ma stage nothing happens and the data is retained. The data is written to the register in the WB stage.

(2) Memory Load Instructions

Instruction Types:

```
MOV.W     @(disp, PC), Rn
MOV.L     @(disp, PC), Rn
MOV.B     @Rm, Rn
MOV.W     @Rm, Rn
MOV.L     @Rm, Rn
MOV.B     @Rm+, Rn
MOV.W     @Rm+, Rn
MOV.L     @Rm+, Rn
MOV.B     @(disp, Rm), R0
MOV.W     @(disp, Rm), R0
MOV.L     @(disp, Rm), Rn
MOV.B     @(R0, Rm), Rn
MOV.W     @(R0, Rm), Rn
MOV.L     @(R0, Rm), Rn
MOV.B     @(disp, GBR), R0
MOV.W     @(disp, GBR), R0
MOV.L     @(disp, GBR), R0
```

Pipeline:

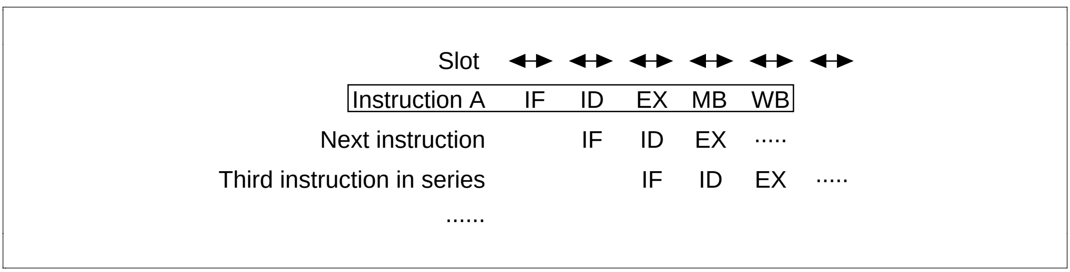


Figure 10-23 Memory Load Instruction Pipeline

Operation Description:

The pipeline has five stages: IF, ID, EX, MA, and WB (figure 10-23). If an instruction that uses the same destination register as this instruction is placed immediately after it, contention will occur (see section 10.2.2, Effects of Memory Load Instructions on Pipelines). Also, see section 10.2.1, Contention between Instruction Fetch (IF) and Memory Access (MA), with reference to contention between the MA and IF stages of these instructions.

(3) Memory Store Instructions

Instruction Types:

```
MOV.B       Rm, @Rn
MOV.W       Rm, @Rn
MOV.L       Rm, @Rn
MOV.B       R0, @(disp, Rn)
MOV.W       R0, @(disp, Rn)
MOV.L       Rm, @(disp, Rn)
MOV.B       Rm, @(R0, Rn)
MOV.W       Rm, @(R0, Rn)
MOV.L       Rm, @(R0, Rn)
MOV.B       R0, @(disp, GBR)
MOV.W       R0, @(disp, GBR)
MOV.L       R0, @(disp, GBR)
```

Pipeline:

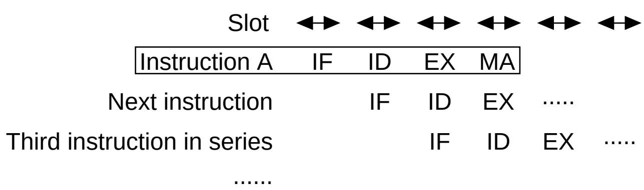


Figure 10-24 Memory Store Instructions Pipeline

Operation Description:

The pipeline has four stages: IF, ID, EX, and MA (figure 10-24). Data is not returned to the register so there is no WB stage. See section 10.2.1, Contention between Instruction Fetch (IF) and Memory Access (MA), with reference to contention between the MA and IF stages of these instructions.

(4) Memory Store Instruction (Pre-decrement)

Instruction Types:

```
MOV.B    Rm,@-Rn
MOV.W    Rm,@-Rn
MOV.L    Rm,@-Rn
```

Pipeline:

Slot	←→	←→	←→	←→	←→	
Instruction A	IF	ID	EX	MA	WB	
Next instruction		IF	ID	EX	·····	
Third instruction in series			IF	ID	EX	·····
·····						

Figure 10-25 Memory Store Instruction (Pre–decrement) Pipeline

Operation Description:

The pipeline ends after five stages: IF, ID, EX, MA, and WB. In the WB stage the decremented value is written to the register. See section 10.2.1, Contention between Instruction Fetch (IF) and Memory Access (MA), with reference to contention between the MA and IF stages of these instructions.

(5) Cache Instruction

Instruction Types:

PREF @Rn

Pipeline:

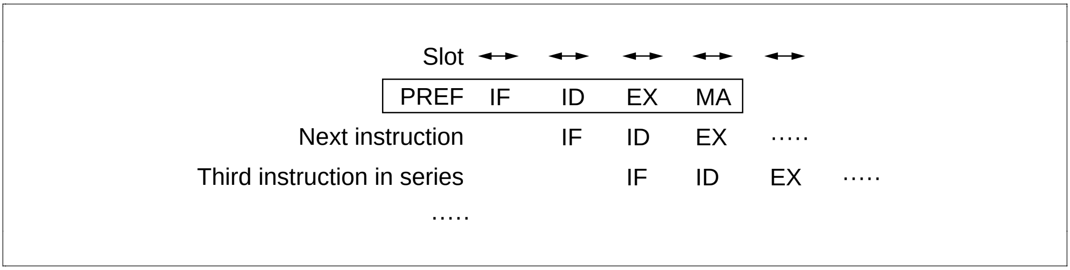


Figure 10-26 Cache Instruction Pipeline

Operation Description:

The pipeline ends after four stages: IF, ID, EX, and MA. There is no WB stage because no data is returned to the register. See section 10.2.1, Contention between Instruction Fetch (IF) and Memory Access (MA), with reference to contention between the MA and IF stages of these instructions.

On the SH3-DSP, the ID of the next instruction is stored one slot behind.

10.4.2 Arithmetic Instructions

(1) Arithmetic Instructions between Registers (Except Multiplication Instructions)

Instruction Types:

```
ADD       Rm, Rn          EXTU.B    Rm, Rn
ADD       #imm, Rn        EXTU.W    Rm, Rn
EXTS.B    Rm, Rn          NEG       Rm, Rn
EXTS.W    Rm, Rn          SUB       Rm, Rn
```

Pipeline:

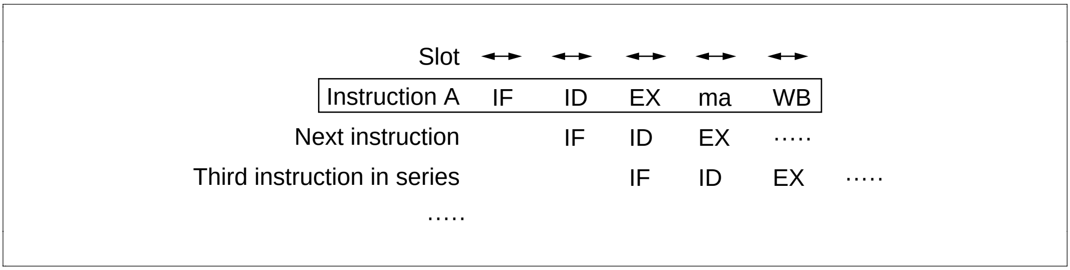


Figure 10-27 Arithmetic Instructions between Registers (Except Multiplication Instructions) Pipeline

Operation Description:

The pipeline ends after five stages: IF, ID, EX, ma, and WB. In the ma stage nothing happens and the calculation result is retained. The result is written to the register in the WB stage.

(2) Arithmetic Calculation Instructions between SR Updated Registers (Excluding Instructions Involving Multiplication)

Instruction Types:

ADDC	Rm,Rn	CMP/PZ	Rn
ADDV	Rm,Rn	CMP/STR	Rm,Rn
CMP/EQ	#imm,R0	DIV1	Rm,Rn
CMP/EQ	Rm,Rn	DIV0S	Rm,Rn
CMP/HS	Rm,Rn	DIV0U	
CMP/GE	Rm,Rn	DT	Rn
CMP/HI	Rm,Rn	NEGC	Rm,Rn
CMP/GT	Rm,Rn	SUBC	Rm,Rn
CMP/PL	Rn	SUBV	Rm,Rn

Pipeline:

Slot	←→	←→	←→	←→	←→	
Instruction A	IF	ID	EX	ma	WB	
Next instruction		IF	ID	EX	·····	
Third instruction in series			IF	ID	EX	·····
·····						

Figure 10-28 Pipeline for Arithmetic Calculation Instructions between SR Updated Registers (Excluding Instructions Involving Multiplication)

Operation Description:

The pipeline ends after five stages: IF, ID, EX, ma, and WB. In the ma stage nothing happens and the calculation result is retained. The result is written to the register in the WB stage. Contention occurs if the instruction immediately following this instruction, or the instruction after that, reads from SR. (See section 10.2.3, Contention due to SR Update Instructions.)

(3) Multiply/Accumulate Instruction

Instruction Type:

```
MAC.W      @Rm+, @Rn+
```

Pipeline:

Slot	↔	↔	↔	↔	↔	↔	↔	↔
MAC.W	IF	ID	EX	MA	MA	mm	mm	mm
Next instruction		IF	—	ID	EX	MA	WB	
Third instruction in series				IF	ID	EX	MA	WB
.....								

Figure 10-29 Multiply/Accumulate Instruction Pipeline

The pipeline has eight stages*: IF, ID, EX, MA, MA, mm, mm, and mm (figure 10-29). The second MA reads the memory and accesses the multiplier. The mm indicates that the multiplier is operating. The mm operates for three cycles after the final MA ends, regardless of slot. The ID of the instruction after the MAC.W instruction is stalled for one slot. The two MAs of the MAC.W instruction, when they contend with IF, split the slots as described in section 10.2.1, Contention between Instruction Fetch (IF) and Memory Access (MA).

When an instruction that does not use the multiplier follows the MAC.W instruction, the MAC.W instruction may be considered to be a five-stage pipeline instruction of IF, ID, EX, MA, MA. In such cases, the ID of the next instruction simply stalls one slot and thereafter the pipeline operates normally. When an instruction that uses the multiplier comes after the MAC instruction, contention occurs with the multiplier, so operation is not as normal (see 10.2.4 Multiplier Access Contention).

Note: * On the SH3-DSP there are seven stages: IF, ID, EX, MA, MA, mm, and mm.

(4) Double-Length Multiply/Accumulate Instruction

Instruction Type:

```
MAC.L      @Rm+, @Rn+
```

Pipeline:

Slot	↔	↔	↔	↔	↔	↔	↔	↔	↔	↔	↔
MAC.L	IF	ID	EX	MA	MA	mm	mm	mm			
Next instruction		IF	—	ID	EX	MA	WB				
Third instruction				IF	ID	EX	MA	WB			
......											

Figure 10-30 Multiply/Accumulate Instruction Pipeline

Operation Description:

The pipeline has eight stages*: IF, ID, EX, MA, MA, mm, mm, and mm (figure 10-30). The second MA reads the memory and accesses the multiplier. The mm indicates that the multiplier is operating. The mm operates for three cycles after the final MA ends, regardless of slot. The ID of the instruction after the MAC.L instruction is stalled for one slot. The two MAs of the MAC.L instruction, when they contend with IF, split the slots as described in section 10.2.1, Contention between Instruction Fetch (IF) and Memory Access (MA).

When an instruction that does not use the multiplier follows the MAC.L instruction, the MAC.L instruction may be considered to be a five-stage pipeline instruction of IF, ID, EX, MA, MA. In such cases, the ID of the next instruction simply stalls one slot and thereafter the pipeline operates normally. When an instruction that uses the multiplier comes after the MAC.L instruction, contention occurs with the multiplier, so operation is not as normal (see 10.2.4 Multiplier Access Contention).

Note: * On the SH3-DSP there are nine stages: IF, ID, EX, MA, MA, mm, mm, mm, and mm.

(5) Multiplication Instructions

Instruction Types:

```
MULS.W      Rm, Rn
MULU.W      Rm, Rn
```

Pipeline:

Slot								
Instruction A	IF	ID	EX	MA	mm	mm		
Next instruction		IF	ID	EX	MA	WB		
Third instruction			IF	ID	EX	MA	WB	
......								

Figure 10-31 Multiplication Instruction Pipeline

Operation Description:

The pipeline has six stages: IF, ID, EX, MA, mm, and mm (figure 10-31). The MA accesses the multiplier. The mm indicates that the multiplier is operating. The mm operates for two cycles after the MA ends, regardless of the slot. The MA of the MULS.W instruction, if it contends with IF, operates as described in section 10.2.1, Contention between Instruction Fetch (IF) and Memory Access (MA).

When an instruction that does not use the multiplier comes after the MULS.W instruction, the MULS.W instruction may be considered to be a four-stage pipeline instruction of IF, ID, EX, and MA. In such cases, it operates like a normal pipeline. When an instruction that uses the multiplier come after the MULS.W instruction, however, contention occurs with the multiplier, so operation is not as normal (see 10.2.4 Multiplier Access Contention).

(6) Double-Length Multiplication Instructions

Instruction Types:

```
DMULS.L     Rm, Rn
DMULU.L     Rm, Rn
MUL.L       Rm, Rn
```

Pipeline:

Slot	←→	←→	←→	←→	←→	←→	←→	←→	←→
Instruction A	IF	ID	EX	MA	MA	mm	mm	mm	
Next instruction		IF	—	ID	EX	MA	WB		
Third instruction				IF	ID	EX	MA	WB	
......									

Figure 10-32 Multiplication Instruction Pipeline

Operation Description:

The pipeline has eight stages*: IF, ID, EX, MA, MA, mm, mm, and mm (figure 10-32). The MA accesses the multiplier. The mm indicates that the multiplier is operating. The mm operates for three cycles after the MA ends, regardless of slot. The ID of the instruction following the DMULS.L instruction is stalled for 1 slot (see the description of the multiply/accumulate instruction). The two MA stages of the DMULS.L instruction, when they contend with IF, split the slot as described in section 10.2.1, Contention between Instruction Fetch (IF) and Memory Access (MA).

When an instruction that does not use the multiplier comes after the DMULS.L instruction, the DMULS.L instruction may be considered to be a five-stage pipeline instruction of IF, ID, EX, MA, and MA. In such cases, it operates like a normal pipeline. When an instruction that uses the multiplier come after the DMULS.L instruction, however, contention occurs with the multiplier, so operation is not as normal (see 10.2.4 Multiplier Access Contention).

Note: * On the SH3-DSP there are nine stages: IF, ID, EX, MA, MA, mm, mm, mm, and mm.

10.4.3 Logic Operation Instructions

(1) Register to Register Logic Operation Instructions

Instruction Types:

```
AND     Rm, Rn
AND     #imm, R0
NOT     Rm, Rn
OR      Rm, Rn
OR      #imm, R0
XOR     Rm, Rn
XOR     #imm, R0
```

Pipeline:

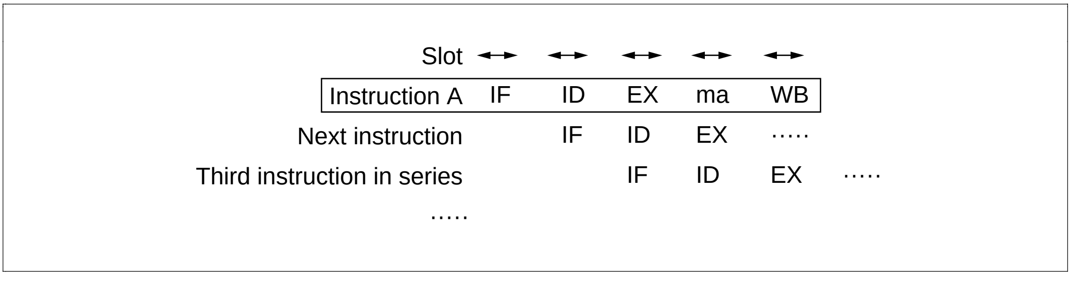


Figure 10-33 Register to Register Logic Operation Instruction Pipeline

Operation Description:

The pipeline ends after five stages: IF, ID, EX, ma, and WB. In the ma stage nothing happens and the calculation result is retained. The result is written to the register in the WB stage.

(2) Logical Calculation Instructions between SR Updated Registers

Instruction Types:

```
TST          Rm,Rn
TST          #imm,R0
```

Pipeline:

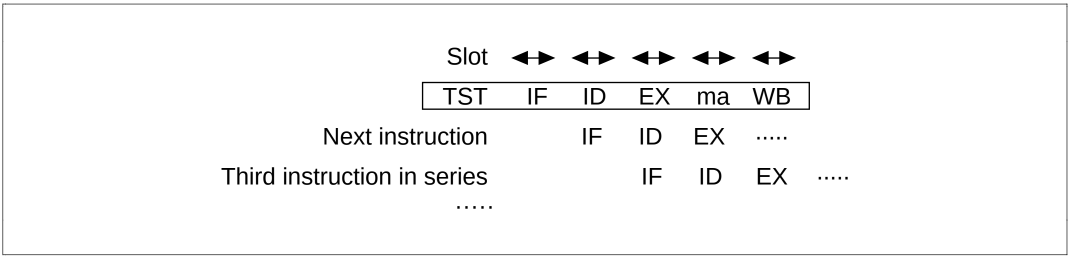


Figure 10-34 Pipeline for Logical Calculation Instructions between SR Updated Registers

Operation Description:

The pipeline ends after five stages: IF, ID, EX, ma, and WB. In the ma stage nothing happens and the calculation result is retained. The result is written to the register in the WB stage. Contention occurs if the instruction immediately following this instruction, or the instruction after that, reads from SR (see section 10.2.3, Contention due to SR Update Instructions).

(3) Memory Logic Operations Instructions

Instruction Types:

```
AND.B       #imm, @(R0, GBR)
OR.B        #imm, @(R0, GBR)
XOR.B       #imm, @(R0, GBR)
```

Pipeline:

Slot	←→	←→	←→	←→	←→	←→	←→	←→	←→
Instruction A	IF	ID	EX	MA	EX	MA			
Next instruction		IF	—	—	ID	EX	·····		
Third instruction in series					IF	ID	EX	·····	
·····									

Figure 10-35 Memory Logic Operation Instruction Pipeline

Operation Description:

The pipeline has six stages: IF, ID, EX, MA, EX, and MA (figure 10-35). The ID of the next instruction stalls for 2 slots. The MAs of these instructions contend with IF (see 10.2.1 Contention between Instruction Fetch (IF) and Memory Access (MA).

(4) SR Update Memory Logical Calculation Instructions

Instruction Type:

```
TST.B       #imm,@(R0,GBR)
```

Pipeline:

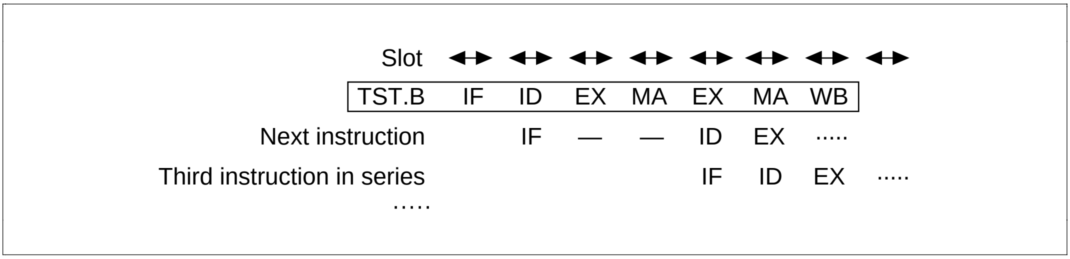


Figure 10-36 SR Updated Memory Logical Calculation Instruction Pipeline

Operation Description:

The pipeline ends after seven stages: IF, ID, EX, MA, EX, MA, and WB. The result is written to the T bit of SR in the WB stage. The MA of the TST instruction contends with IF. (See section 10.2.1, Contention between Instruction Fetch (IF) and Memory Access (MA).) Also, contention occurs if the instruction immediately following this instruction, or the instruction after that, reads from SR (see section 10.2.3, Contention due to SR Update Instructions).

(5) TAS Instruction

Instruction Type:

```
TAS.B       @Rn
```

Pipeline:

Slot	↔	↔	↔	↔	↔	↔	↔	↔	↔
TAS.B	IF	ID	EX	MA	MA	MA	WB		
Next instruction		IF	—	—	ID	EX	·····		
Third instruction in series						IF	ID	EX	·····
·····									

Figure 10-37 TAS Instruction Pipeline

Operation Description:

The pipeline ends after seven stages: IF, ID, EX, MA, MA, MA, and WB. The result is written to the T bit of SR in the WB stage. The MA of the TST instruction contends with IF (see section 10.2.1, Contention between Instruction Fetch (IF) and Memory Access (MA)). Also, contention occurs if the instruction immediately following this instruction, or the instruction after that, reads from SR (see section 10.2.3, Contention due to SR Update Instructions).

On the SH3-DSP, the ID of the next instruction is stored three slots behind.

10.4.4 Shift Instructions

(1) Shift Instructions

Instruction Types:

```
SHLL2     Rn
SHLR2     Rn
SHLL8     Rn
SHLR8     Rn
SHLL16    Rn
SHLR16    Rn
SHAD      Rm,Rn
SHLD      Rm,Rn
```

Pipeline:

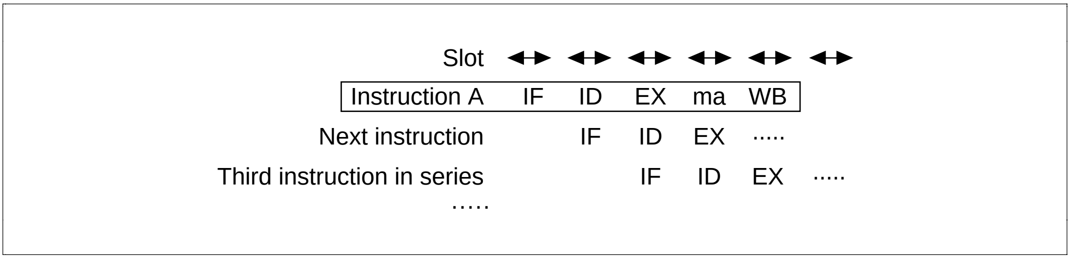


Figure 10-38 Shift Instruction Pipeline

Operation Description:

The pipeline ends after five stages: IF, ID, EX, ma, and WB. In the ma stage nothing happens and the shift result is retained. The result is written to the register in the WB stage.

(2) SR Update Shift Instructions

Instruction Types:

```
ROTL    Rn
ROTR    Rn
ROTCL   Rn
ROTCR   Rn
SHAL    Rn
SHAR    Rn
SHLL    Rn
SHLR    Rn
```

Pipeline:

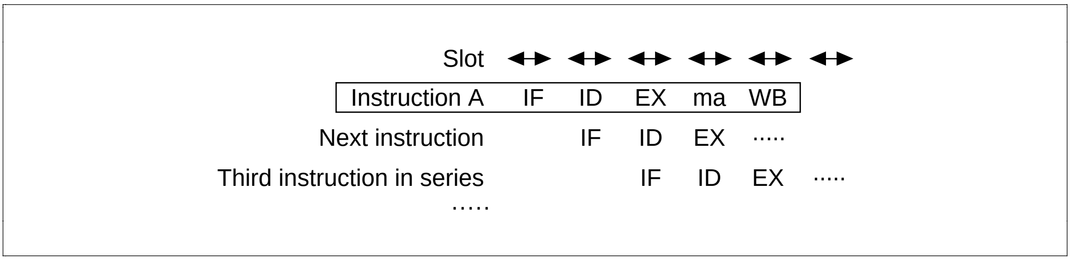


Figure 10-39 SR Updated Shift Instruction Pipeline

Operation Description:

The pipeline ends after five stages: IF, ID, EX, ma, and WB. In the ma stage nothing happens and the result is retained. The result is written to the register in the WB stage. Contention occurs if the instruction immediately following this instruction, or the instruction after that, reads from SR (see section 10.2.3, Contention due to SR Update Instructions).

10.4.5 Branch Instructions

(1) Conditional Branch Instructions

Instruction Types:

```
BF        label
BT        label
```

Pipeline/Operation Description:

The pipeline has three stages: IF, ID, and EX. Condition verification is performed in the ID stage. Conditionally branched instructions are not delay branched.

1. When condition is satisfied

   The branch destination address is calculated in the EX stage. The two instructions after the conditional branch instruction (instruction A) are fetched but discarded. The branch destination instruction begins its fetch from the slot following the slot which has the EX stage of instruction A (figure 10-40).

Slot	←→	←→	←→	←→	←→	←→	←→	←→	←→
Instruction A	IF	ID	EX						
Next instruction		IF	—		(Fetched but discarded)				
Third instruction in series			IF	—	(Fetched but discarded)				
Branch destination			—	IF	ID	EX	.....		
.....					IF	ID	EX	.....	
.....									

Figure 10-40 Branch Instruction when Condition is Satisfied

2. When condition is not satisfied

If it is determined that conditions are not satisfied at the ID stage, the EX stage proceeds without doing anything. The next instruction also executes a fetch (figure 10-41).

Slot	←→	←→	←→	←→	←→	←→	←→	←→	←→
Instruction A	IF	ID	EX						
Next instruction		IF	ID	EX	.....				
Third instruction in series			IF	ID	EX	.....			
.....				IF	ID	EX	.....		
.....									

Figure 10-41 Branch Instruction when Condition is Not Satisfied

(2) Delayed Conditional Branch Instructions

Instruction Types:

```
BF/S        label
BT/S        label
```

Pipeline/Operation Description:

The pipeline has three stages: IF, ID, and EX. Condition verification is performed in the ID stage.

1. When condition is satisfied

   The branch destination address is calculated in the EX stage. The instruction after the conditional branch instruction (instruction A) is fetched and executed, but the instruction after that is fetched and discarded. The branch destination instruction begins its fetch from the slot following the slot which has the EX stage of instruction A (figure 10-42).

Slot	←→	←→	←→	←→	←→	←→	←→	←→	←→
Instruction A	IF	ID	EX						
Next instruction		IF	—	ID	EX	·····			
Third instruction in series			IF	—	(Fetched but discarded)				
Branch destination				IF	ID	EX	·····		
·····					IF	ID	EX	·····	
·····									

Figure 10-42   Branch Instruction when Condition is Satisfied

2. When condition is not satisfied

If it is determined that conditions are not satisfied at the ID stage, the EX stage proceeds without doing anything. The next instruction also executes a fetch (figure 10-43).

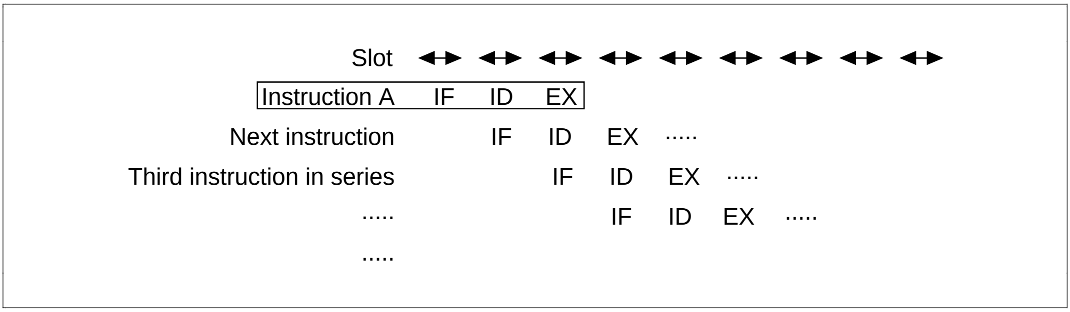


Figure 10-43 Branch Instruction when Condition is Not Satisfied

(3) Unconditional Branch Instructions

Instruction Types:

```
BRA     label
BRAF    Rm
JMP     @Rm
RTS
```

Pipeline:

Slot	←→	←→	←→	←→	←→			
Instruction A	IF	ID	EX					
Next instruction		IF	—	ID	EX	·····		
Third instruction in series			IF	—	(Fetch but then data is discarded)			
Branch destination				IF	ID	EX	·····	
·····					IF	ID	EX	·····

Figure 10-44 Unconditional Branch Instruction Pipeline

Operation Description:

The pipeline has three stages: IF, ID, and EX (figure 10-44). Unconditionally branched instructions are delay branched. The branch destination address is calculated in the EX stage. The instruction following the unconditional branch instruction (instruction A), that is, the delay slot instruction is not fetched and discarded as the conditional branch instructions are, but is then executed. Note that the ID slot of the delay slot instruction does stall for one cycle. The branch destination instruction starts its fetch from the slot after the slot that has the EX stage of instruction A.

(4) Unconditional Branch Instructions (PR)

Instruction Types:

```
BSR      label
BSRF     Rm
JSR      @Rm
```

Pipeline:

Slot	↔	↔	↔	↔	↔		
Instruction A	IF	ID	EX	ma	WB		
Next instruction		IF	—	ID	EX	·····	
Third instruction in series			IF	—	(Fetch but then data is discarded)		
·····				IF	ID	EX	·····

Figure 10-45 Unconditional Branch Instruction (PR) Pipeline

Operation Description:

The pipeline ends after five stages: IF, ID, EX, ma, and WB. Unconditionally branched instructions are delay branching. The instruction following the unconditional branch instruction (instruction A), that is, the delay slot instruction, is fetched and executed. However, the instruction after that is fetched and discarded. The branch destination instruction starts its fetch from the slot after the slot that has the EX stage of instruction A.

10.4.6 System Control Instructions

(1) System Control ALU Instructions

Instruction Types:

`LDC`	`Rm, GBR`	`STC`	`SR, Rn`	`LDRE`	`@(disp,PC)`(SH3-DSP only)
`LDC`	`Rm, VBR`	`STC`	`GBR, Rn`	`LDRS`	`@(disp,PC)`(SH3-DSP only)
`LDC`	`Rm, SSR`	`STC`	`VBR, Rn`	`SETRC`	`Rm` (SH3-DSP only)
`LDC`	`Rm, SPC`	`STC`	`SSR, Rn`	`SETRC`	`#imm` (SH3-DSP only)
`LDC`	`Rm, MOD`	`STC`	`SPC, Rn`		
`LDC`	`Rm, RE`	`STC`	`MOD, Rn`		
`LDC`	`Rm, RS`	`STC`	`RE, Rn`		
`LDC`	`Rm, R0_BANK`	`STC`	`RS, Rn`		
`LDC`	`Rm, R1_BANK`	`STC`	`R0_BANK, Rn`		
`LDC`	`Rm, R2_BANK`	`STC`	`R1_BANK, Rn`		
`LDC`	`Rm, R3_BANK`	`STC`	`R2_BANK, Rn`		
`LDC`	`Rm, R4_BANK`	`STC`	`R3_BANK, Rn`		
`LDC`	`Rm, R5_BANK`	`STC`	`R4_BANK, Rn`		
`LDC`	`Rm, R6_BANK`	`STC`	`R5_BANK, Rn`		
`LDC`	`Rm, R7_BANK`	`STC`	`R6_BANK, Rn`		
`LDS`	`Rm, PR`	`STC`	`R7_BANK, Rn`		
		`STS`	`PR, Rn`		

Pipeline:

Slot	←→	←→	←→	←→	←→	
Instruction A	IF	ID	EX	ma	WB	
Next instruction		IF	ID	EX	·····	
Third instruction in series			IF	ID	EX	·····
·····						

Figure 10-46 System Control ALU Instruction Pipeline

Operation Description:

The pipeline ends after five stages: IF, TD, EX, ma, and WB. In the EX stage, the data calculation is completed via ALU. In the ma stage nothing happens and the result is retained. The result is written to the register in the WB stage.

On the SH3-DSP, the ID of the instruction following the LDC, LDRE, LDRS, and SETRC instruction is stored two slots behind.

(2) SR Update System Control Instructions

Instruction Types:

```
CLRS
CLRT
SETS
SETT
```

Pipeline:

Slot	←→	←→	←→	←→	←→		
Instruction A	IF	ID	EX	ma	WB		
Next instruction		IF	ID	EX	·····		
Third instruction in series			IF	ID	EX	·····	
·····				IF	ID	EX	·····

Figure 10-47 SR Update System Control Instruction Pipeline

Operation Description:

The pipeline ends after five stages: IF, ID, EX, ma, and WB. In the ma stage nothing happens and the data to be transferred is retained. The data is written to the register in the WB stage. Contention occurs if the instruction immediately following this instruction, or the instruction after that, reads from SR (see section 10.2.3, Contention due to SR Update Instructions).

(3) LDTLB Instruction

Instruction Type:

LDTLB

Pipeline:

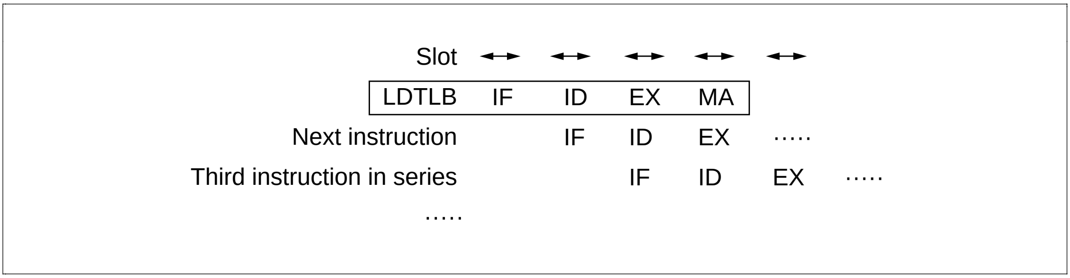


Figure 10-48 LDTLB Instruction Pipeline

Operation Description:

The pipeline ends after four stages: IF, ID, EX, and MA. There is no WB stage because no data is returned to the register. See section 10.2.1, Contention between Instruction Fetch (IF) and Memory Access (MA), with reference to contention between the MA and IF stages of these instructions.

(4) NOP Instruction

Instruction Type:

NOP

Pipeline:

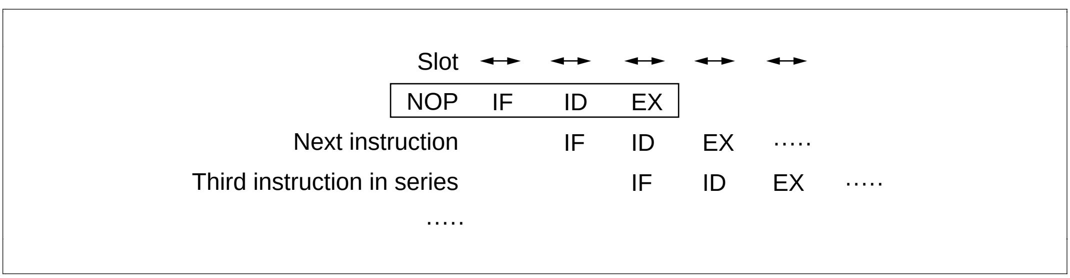


Figure 10-49 NOP Instruction Pipeline

Operation Description:

The pipeline ends after three stages: IF, ID, and EX.

(5) LDC Instruction (SR)

Instruction Type:

```
LDC        Rm,SR
```

Pipeline:

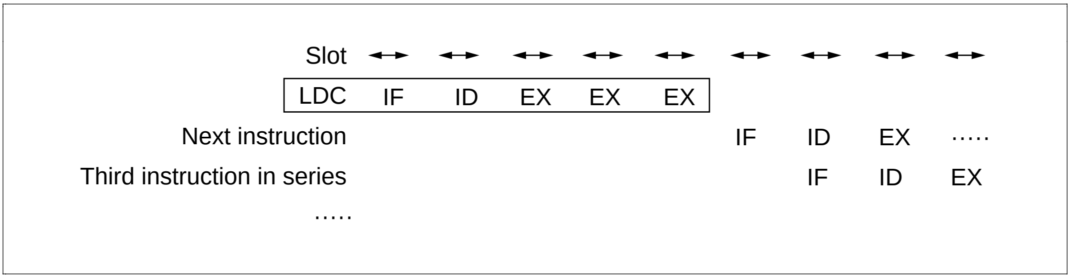


Figure 10-50 LDC Instruction (SR) Pipeline

Operation Description:

The pipeline ends after five stages: IF, ID, EX, EX, and EX. The data is written to SR in the last EX stage. The IF of the next instruction starts from the slot after the slot that has the EX stage of instruction A.

(6) LDC.L Instructions (SR)

Instruction Type:

```
LDC.L      @Rm+, SR
```

Pipeline:

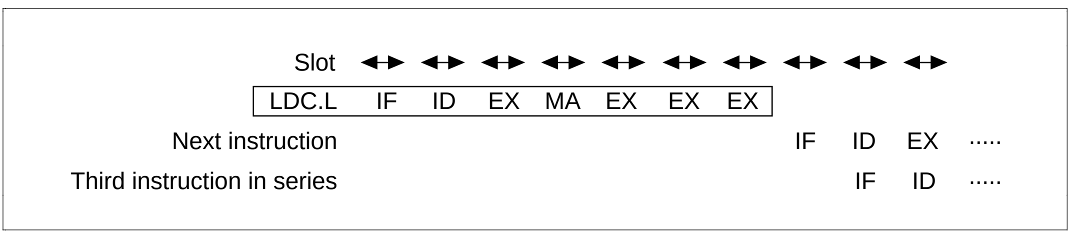


Figure 10-51 LDC.L Instruction (SR) Pipeline

Operation Description:

The pipeline ends after seven stages: IF, ID, EX, MA, EX, EX, and EX. The data is written to SR in the last EX stage. The IF of the next instruction starts from the slot after the slot that has the final EX stage of instruction A.

(7) LDS.L Instruction (PR)

Instruction Type:

```
LDS.L      @Rm+, PR
```

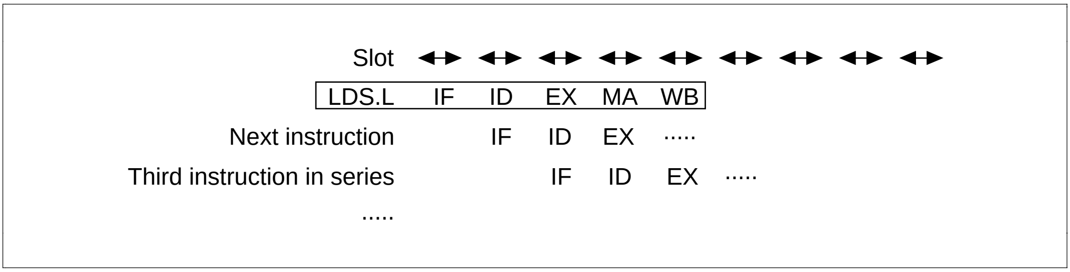


Figure 10-52 LDS.L Instructions (PR) Pipeline

Operation Description:

The pipeline ends after five stages: IF, ID, EX, MA and WB. Contention occurs if this instruction is followed by an instruction that uses the same destination register (see section 10.2.2, Effects of Memory Load Instructions on Pipelines). Also, The MA of this instruction contends with IF (see section 10.2.1, Contention between Instruction Fetch (IF) and Memory Access (MA)).

(8) STS.L Instruction (PR)

Instruction Type:

```
STS.L      PR, @–Rn
```

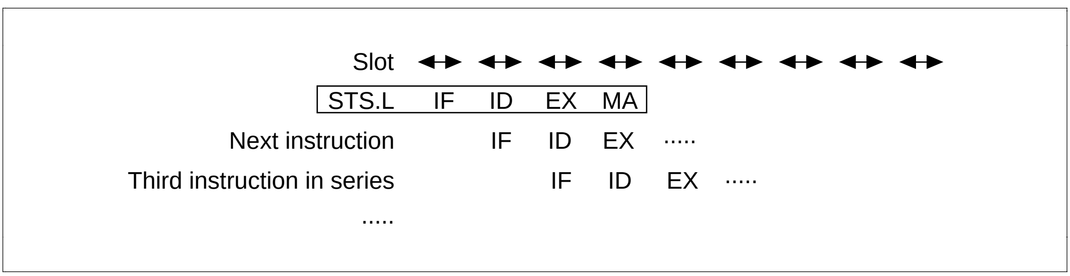


Figure 10-53 STS.L Instruction (PR) Pipeline

Operation Description:

The pipeline ends after five stages: IF, ID, EX, MA and WB. The WB stage writes the decremented value to the register. The MA of this instruction contends with IF (see section 10.2.1, Contention between Instruction Fetch (IF) and Memory Access (MA)).

(9) LDC.L Instructions

Instruction Types:

```
LDC.L     @Rm+, GBR
LDC.L     @Rm+, VBR
LDC.L     @Rm+, SSR
LDC.L     @Rm+, SPC
LDC.L     @Rm+, MOD (SH3-DSP only)
LDC.L     @Rm+, RE (SH3-DSP only)
LDC.L     @Rm+, RS (SH3-DSP only)
LDC.L     @Rm+, R0_BANK
LDC.L     @Rm+, R1_BANK
LDC.L     @Rm+, R2_BANK
LDC.L     @Rm+, R3_BANK
LDC.L     @Rm+, R4_BANK
LDC.L     @Rm+, R5_BANK
LDC.L     @Rm+, R6_BANK
LDC.L     @Rm+, R7_BANK
```

Pipeline:

Slot	↔	↔	↔	↔	↔	↔
LDC.L	IF	ID	EX	MA	WB	
Next instruction		IF	ID	EX	·····	·····
Third instruction in series			IF	ID	EX	·····

Figure 10-54   LDC.L Instruction Pipeline

Operation Description:

The pipeline ends after five stages: IF, ID, EX, MA and WB. Contention occurs if this instruction is followed by an instruction that uses the same destination register (see section 10.2.2, Effects of Memory Load Instructions on Pipelines). Also, The MA of this instruction contends with IF (see section 10.2.1, Contention between Instruction Fetch (IF) and Memory Access (MA)).

On the SH3-DSP, the ID of the instruction following the LDC instruction is stored four slots behind.

(10) STC.L Instructions (Excluding Bank Registers)

Instruction Types:

```
STC.L   SR, @–Rn          STC.L   MOD,@-Rn (SH3-DSP only)
STC.L   GBR, @–Rn         STC.L   RE,@-Rn (SH3-DSP only)
STC.L   VBR, @–Rn         STC.L   RS,@-Rn (SH3-DSP only)
STC.L   SSR, @–Rn
STC.L   SPC, @–Rn
```

Pipeline:

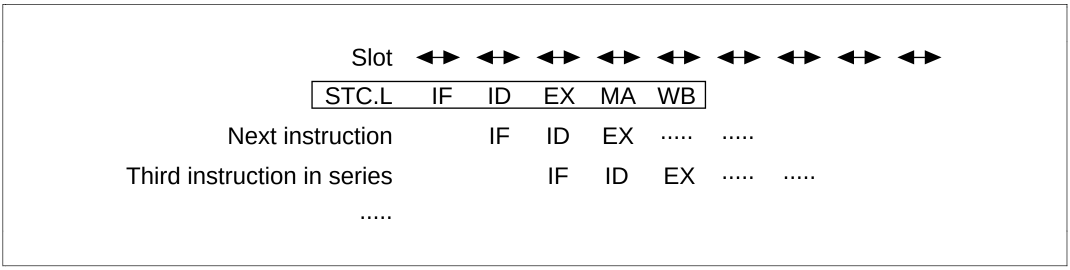


Figure 10-55 STC.L Instruction (Excluding Bank Register) Pipeline

Operation Description:

The pipeline ends after five stages: IF, ID, EX, MA and WB. The WB stage writes the decremented value to the register. The MA of this instruction contends with IF (see section 10.2.1, Contention between Instruction Fetch (IF) and Memory Access (MA)).

On the SH3-DSP, the ID of the instruction following the LDC instruction is stored one slot behind.

(11) STC.L Instructions (Bank Registers)

Instruction Types:

```
STC.L       R0_BANK,@–Rn
STC.L       R1_BANK,@–Rn
STC.L       R2_BANK,@–Rn
STC.L       R3_BANK,@–Rn
STC.L       R4_BANK,@–Rn
STC.L       R5_BANK,@–Rn
STC.L       R6_BANK,@–Rn
STC.L       R7_BANK,@–Rn
```

Pipeline:

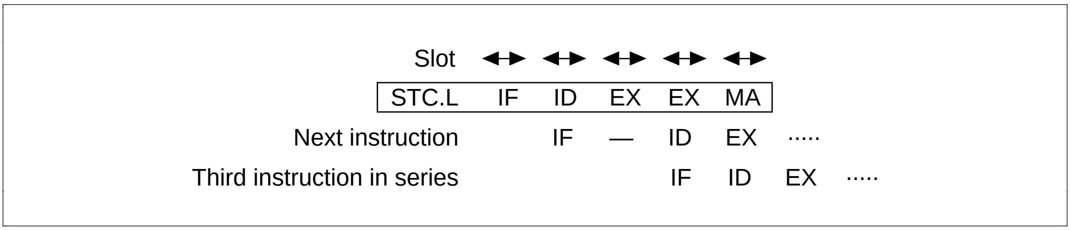


Figure 10-56 STC.L Instruction (Bank Register) Pipeline

Operation Description:

The pipeline ends after six stages: IF, ID, EX, EX, MA and WB. The ID of the next instruction is stalled one cycle. These instructions cause contention with IF (see section 10.2.1, Contention between Instruction Fetch (IF) and Memory Access (MA)).

(12) Register → MAC/DSP Transfer Instructions

Instruction Types:

```
CLRMAC
LDS        Rm, MACH
LDS        Rm, MACL
```
`LDS        Rm, DSR` (SH3-DSP only)
`LDS        Rm, A0` (SH3-DSP only)
`LDS        Rm, X0` (SH3-DSP only)
`LDS        Rm, X1` (SH3-DSP only)
`LDS        Rm, Y0` (SH3-DSP only)
`LDS        Rm, Y1` (SH3-DSP only)

Pipeline:

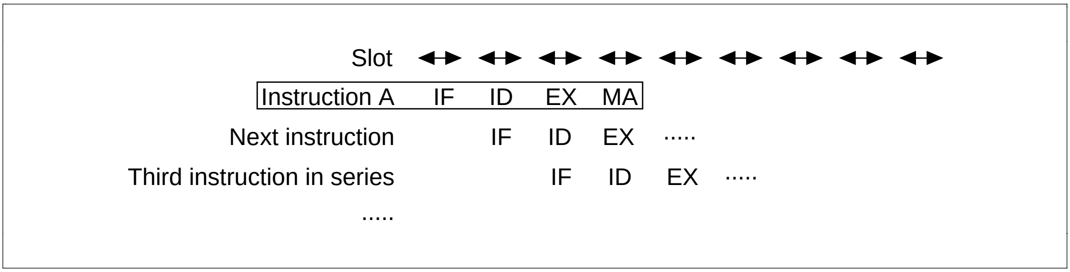


Figure 10-57 Register → MAC Transfer Instruction Pipeline

Operation Description:

The pipeline ends after four stages: IF, ID, EX, and MA. The MA stage is used to access the multiplier. This MA contends with IF (see section 10.2.1, Contention between Instruction Fetch (IF) and Memory Access (MA)). Also, if one of these instructions is followed by an instruction that uses the multiplier, multiplier contention will result (see section 10.2.4, Multiplier Access Contention).

(13) Memory → MAC Transfer Instructions

Instruction Types:

```
LDS.L        @Rm+, MACH
LDS.L        @Rm+, MACL
LDS.L        @Rm+, DSR (SH3-DSP only)
LDS.L        @Rm+, A0 (SH3-DSP only)
LDS.L        @Rm+, X0 (SH3-DSP only)
LDS.L        @Rm+, X1 (SH3-DSP only)
LDS.L        @Rm+, Y0 (SH3-DSP only)
LDS.L        @Rm+, Y1 (SH3-DSP only)
```

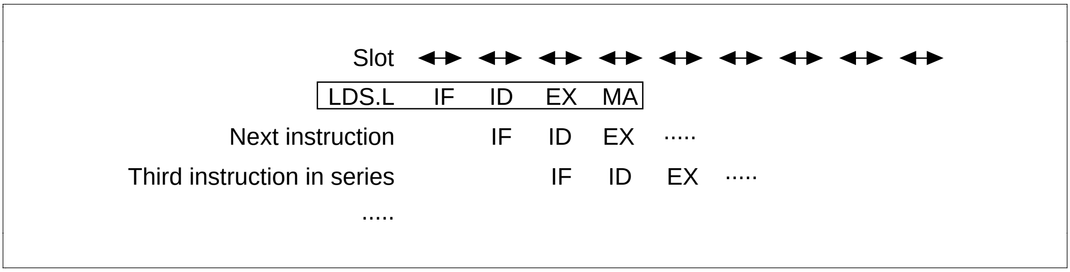


Figure 10-58 Memory → MAC Transfer Instruction Pipeline

The pipeline ends after four stages: IF, ID, EX, and MA. The MA stage is used to access memory and the multiplier. This MA contends with IF. (See section 10.2.1, Contention between Instruction Fetch (IF) and Memory Access (MA).) Also, if one of these instructions is followed by an instruction that uses the multiplier, multiplier contention will result (see section 10.2.4, Multiplier Access Contention).

(14) MAC/DSP → Register Transfer Instructions

Instruction Types:

```
STS        MACH, Rn
STS        MACL, Rn
STS        DSR,  Rn (SH3-DSP only)
STS        A0,  Rn (SH3-DSP only)
STS        X0,  Rn (SH3-DSP only)
STS        X1,  Rn (SH3-DSP only)
STS        Y0,  Rn (SH3-DSP only)
STS        Y1,  Rn (SH3-DSP only)
```

Pipeline:

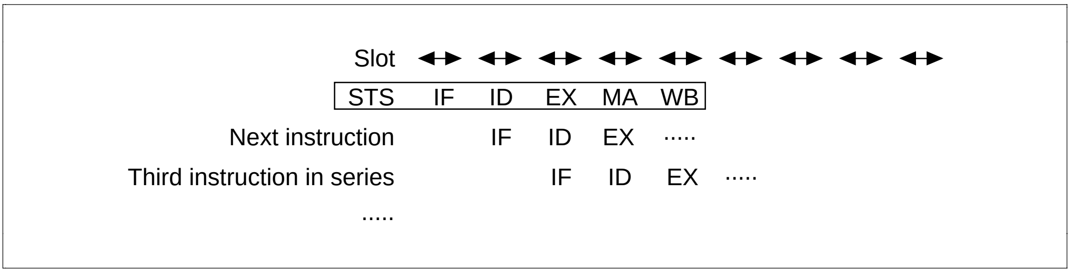


Figure 10-59 MAC → Register Transfer Instruction Pipeline

Operation Description:

The pipeline ends after five stages: IF, ID, EX, MA and WB. The MA stage is used to access the multiplier. This MA contends with IF (see section 10.2.1, Contention between Instruction Fetch (IF) and Memory Access (MA)). Also, if one of these instructions is followed by an instruction that uses the same destination register or an instruction that uses the multiplier, multiplier contention will result (see section 10.2.2, Effects of Memory Load Instructions on Pipelines, and section 10.2.4, Multiplier Access Contention).

(15) MAC → Memory Transfer Instructions

Instruction Types:

```
STS.L       MACH, @–Rn
STS.L       MACL, @–Rn
STS.L       DSR, @–Rn (SH3-DSP only)
STS.L       A0, @–Rn (SH3-DSP only)
STS.L       X0, @–Rn (SH3-DSP only)
STS.L       X1, @–Rn (SH3-DSP only)
STS.L       Y0, @–Rn (SH3-DSP only)
STS.L       Y1, @–Rn (SH3-DSP only)
```

Pipeline:

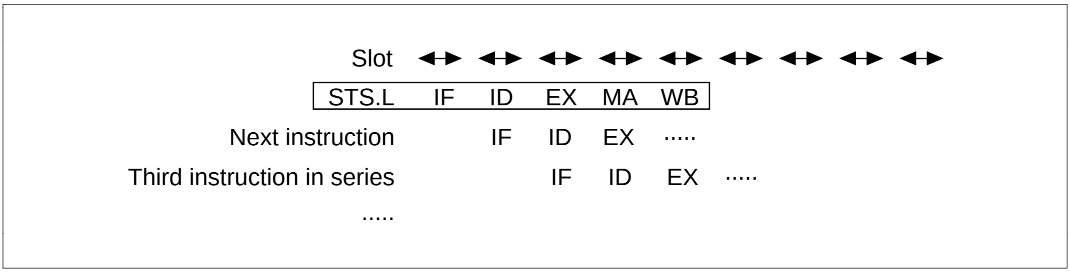


Figure 10-60 MAC → Memory Transfer Instruction Pipeline

Operation Description:

The pipeline ends after five stages: IF, ID, EX, MA and WB. The MA stage is used to access the multiplier. This MA contends with IF (see section 10.2.1, Contention between Instruction Fetch (IF) and Memory Access (MA)). Also, if one of these instructions is followed by an instruction that uses the multiplier, multiplier contention will result (see section 10.2.4, Multiplier Access Contention).

(16) RTE Instruction

Instruction Type:

RTE

Pipeline:

Slot	←→	←→	←→	←→	←→	←→	←→	←→	←→
RTE	IF	ID	EX	EX	EX				
Delay slot		IF	—	—	—	ID	EX	·····	
Branch destination			IF	—	—	—	(Fetch but then data is discarded)		
·····						IF	ID	EX	·····

Figure 10-61 RTE Instruction Pipeline

Operation Description:

The pipeline ends after five stages: IF, ID, EX, EX, and EX. RTE is a delayed branch instruction. The instruction following the RTE instruction, that is, the delay slot instruction, is fetched and executed. However, the instruction after that is fetched and discarded. The IF of the branch destination instruction starts from the slot after the slot that has the final EX stage of RTE.

(17) TRAP Instruction

Instruction Type:

```
TRAPA        #imm
```

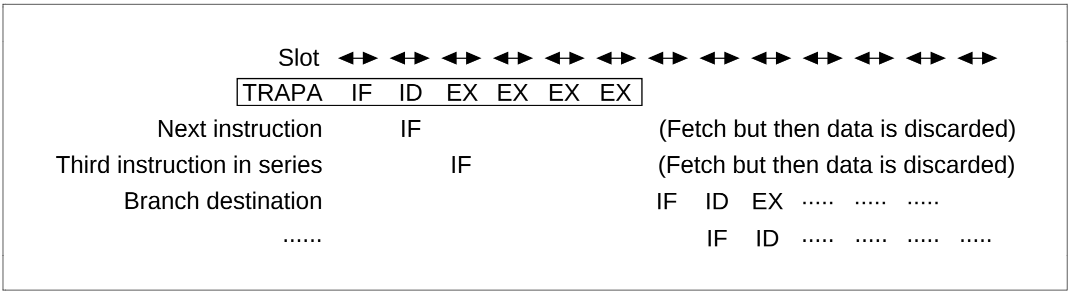


Figure 10-62 TRAP Instruction Pipeline

The pipeline has six stages*: IF, ID, EX, EX, EX, and EX (figure 10-62). TRAP is not a delayed branch instruction. The two instructions after the TRAP instruction are fetched, but they are discarded without being executed. The IF of the branch destination instruction starts from the next slot of the last EX of the TRAP instruction.

Note: * On the SH3-DSP there are eight stages: IF, ID, EX, EX, EX, EX, EX, and EX.

(18) SLEEP Instruction

Instruction Type:

SLEEP

Pipeline:

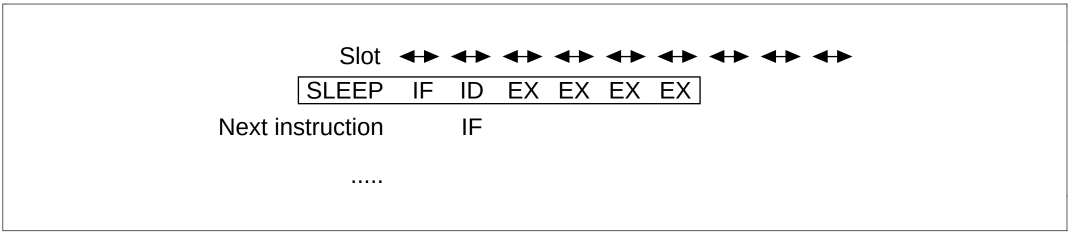


Figure 10-63 SLEEP Instruction Pipeline

Operation Description:

The pipeline has three stages: IF, ID, and EX (figure 10-63). It is issued until the IF of the next instruction. After the SLEEP instruction is executed, the CPU enters sleep mode or standby mode.

10.4.7 Exception Processing

(1) Interrupt Exception Processing

Instruction Type:

Interrupt exception processing

Pipeline:

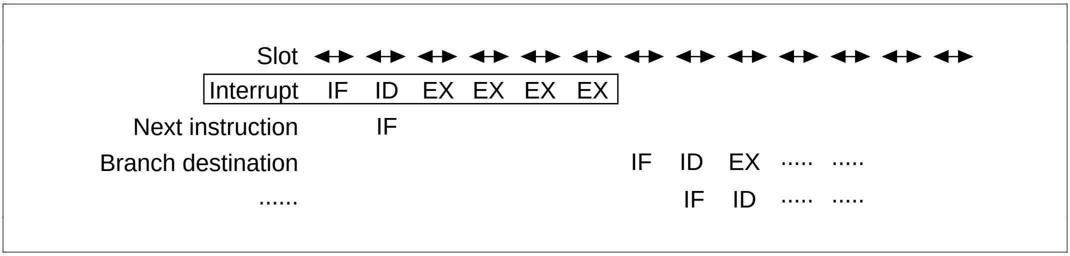


Figure 10-64 Interrupt Exception Processing Pipeline

Operation Description:

The interrupt is received during the ID stage of the instruction and everything after the ID stage is replaced by the interrupt exception processing sequence. The pipeline has six stages: IF, ID, EX, EX, EX, and EX (figure 10-64). Interrupt exception processing is not a delayed branch. In interrupt exception processing, an overrun fetch (IF) occurs. In branch destination instructions, the IF starts from the slot following the final EX in the interrupt exception processing.

Interrupt sources are NMI, IRL, and on-chip peripheral module interrupts. Refer to the *Hardware Manual* for details.

(2) Address Error Exception Processing

Instruction Type:

Address error exception processing

Pipeline:

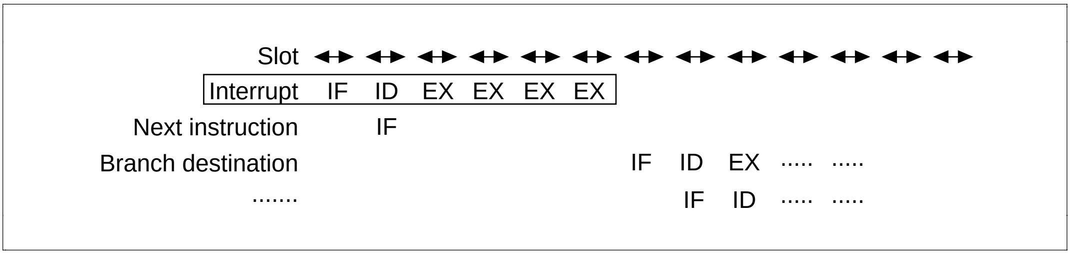


Figure 10-65 Address Error Exception Processing Pipeline

Operation Description:

The address error is received during the ID stage of the instruction and everything after the ID stage is replaced by the address error exception processing sequence. The pipeline has six stages: IF, ID, EX, EX, EX, and EX (figure 10-65). Address error exception processing is not a delayed branch. In address error exception processing, an overrun fetch (IF) occurs. In branch destination instructions, the IF starts from the slot following the final EX in the address error exception processing.

Address errors are caused by instruction fetches and by data reads or writes. Fetching an instruction from an odd address or fetching an instruction from an on-chip peripheral register causes an instruction fetch address error. Accessing word data from other than a word boundary, accessing longword data from other than a longword boundary, and accessing an on-chip peripheral register 8-bit space by longword cause a read or write address error. Refer to the *Hardware Manual* for details.

(3) TLB Related Exception Processing

Instruction Type:

TLB related exception processing

Pipeline:

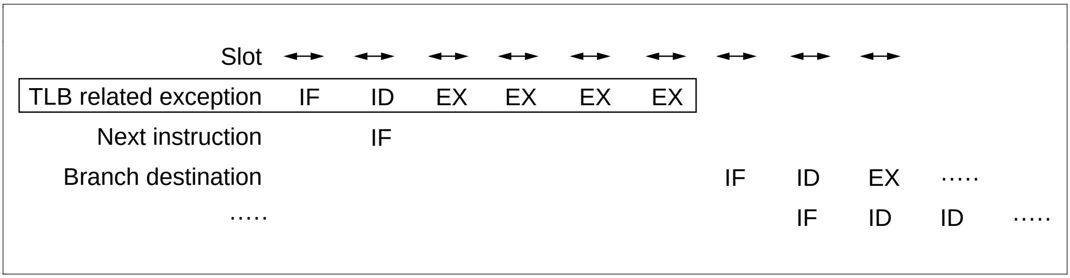


Figure 10-66 TLB Related Exception Processing Pipeline

Operation Description:

If a TLB related exception is received in the instruction's ID stage, the portion following the ID stage is replaced by the TLB related exception processing sequence.

The pipeline ends after six stages: IF, ID, EX, EX, EX, and EX. TLB related exception processing is not a delayed branch. In TLB related exception processing, an overrun fetch (IF) occurs. In branch destination instructions, the IF starts from the slot after the slot that has the final EX stage of the TLB related exception processing.

TLB related exceptions include TLB error, TLB invalid, TLB initial write, and TLB protection exceptions. Refer to the Hardware Manual for details.

(4) Illegal Instruction Exception Processing

Instruction Type:

Illegal instruction exception processing

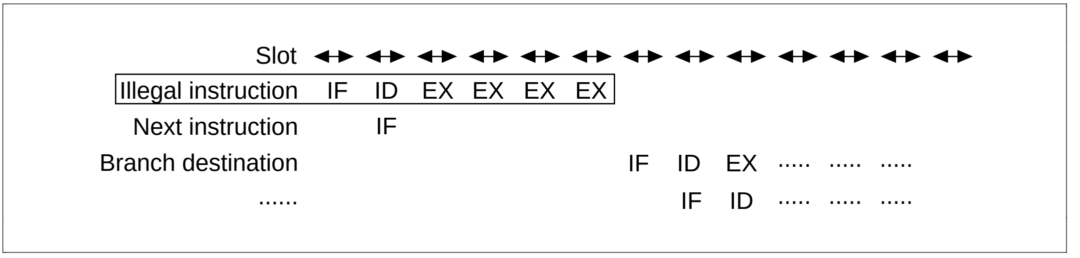


Figure 10-67 Illegal Instruction Exception Processing Pipeline

The illegal instruction is received during the ID stage of the instruction and everything after the ID stage is replaced by the illegal instruction exception processing sequence. The pipeline has six stages: IF, ID, EX, EX, EX, and EX (figure 10-67). Illegal instruction exception processing is not a delayed branch. In illegal instruction exception processing, an overrun fetch (IF) occurs. Whether there is an IF only in the next instruction or in the one after that as well depends on the instruction that was to be executed. In branch destination instructions, the IF starts from the slot following the final EX in the illegal instruction exception processing.

Illegal instruction exception processing is caused by ordinary illegal instructions and by instructions with illegal slots. When undefined code placed somewhere other than the slot directly after the delayed branch instruction (called the delay slot) is decoded, ordinary illegal instruction exception processing occurs. When undefined code placed in the delay slot is decoded or when an instruction placed in the delay slot to rewrite the program counter is decoded, an illegal slot instruction occurs. Refer to the *Hardware Manual* for details.

10.4.8 Pipeline for FPU Instructions (SH-3E Only)

Slot	◄►	◄►	◄►	◄►	◄►	◄►	◄►	◄►	◄►
Instruction A	IF	DF	E1	E2					
Next instruction		IF	ID	EX	.....				
Subsequent instruction			IF	ID	EX	.....			
.....									

Figure 10-68 FPU Pipeline During Data Transfer between Floating Point Register and Register

Slot	◄►	◄►	◄►	◄►	◄►	◄►	◄►	◄►	◄►
Instruction A	IF	DF	E1	E2	SF				
Next instruction		IF	ID	EX	.....				
Subsequent instruction			IF	ID	EX	.....			
.....									

Figure 10-69 FPU Pipeline During Floating Point Load

Slot	◄►	◄►	◄►	◄►	◄►	◄►	◄►	◄►	◄►
Instruction A	IF	DF	E1	E2					
Next instruction		IF	ID	EX	.....				
Subsequent instruction			IF	ID	EX	.....			
.....									

Figure 10-70 FPU Pipeline During Floating Point Store

Slot	◄►	◄►	◄►	◄►	◄►	◄►	◄►	◄►	◄►
Instruction A	IF	DF	E1						
Next instruction		IF	ID	EX	.....				
Subsequent instruction			IF	ID	EX	.....			
.....									

Figure 10-71 FPU Pipeline During Floating Point Compare

Slot	↔	↔	↔	↔	↔	↔	↔	↔	↔
Instruction A	IF	DF	E1	E2					
Next instruction		IF	ID	EX	.....				
Subsequent instruction			IF	ID	EX	.....			
.....									

Figure 10-72 FPU Pipeline During Floating Point Arithmetic Calculation Instruction (Excluding FDIV and FSQRT)

Case 1: Next Instruction is FPU Instruction

Slot	↔	↔	↔	↔	↔	↔	↔	↔	↔	↔	↔
Instruction A	IF	DF	E1	E1	E1	.....	E1	E2	SF		
Next instruction		IF	DF	—	—	—	—	E1	E2	SF	
Subsequent instruction							IF	DF	E1	E2	SF

Case 2: Next Instruction is CPU Instruction and Subsequent Instruction is FPU Instruction

Slot	↔	↔	↔	↔	↔	↔	↔	↔	↔	↔	↔
Instruction A	IF	DF	E1	E1	E1	.....	E1	E2	SF		
Next instruction		IF	DF								
Subsequent instruction			IF	DF	—	—	—	E1	E2	SF	

Notes: 1. FDIV and FSQRT require 13 cycles in the E1 stage.
2. The next instruction enters the CPU pipeline, it is deleted from the FPU pipeline after the DF stage.
3. Even if there are two to twelve CPU instructions between FDIV (or FSQRT) and the next FPU instructions, the situation is still interpreted in the same way as Case 2.

Figure 10-73 FPU Pipeline During FDIV and FSQRT Instructions

10.4.9 DSP Data Transfer Instructions (SH3-DSP Only)

(1) X Memory and Y Memory Load Instructions

Instruction Types:

```
NOPX
MOVX.W   @Ax,Dx
MOVX.W   @Ax+,Dx
MOVX.W   @Ax+Ix,Dx
```

Pipeline:

Slot	←→	←→	←→	←→	←→	←→	←→	←→
InstructionA	IF		EX	MA	WB/DSP			
Next instruction		ID	ID	EX	MA	WB/DSP		
Subsequent instruction		IF	IF	ID	EX	MA	WB/DSP	
. . . . . .								

Figure 10-74 X Memory and Y Memory Load Instruction Pipeline

Operation Description:

The pipeline has five stages: IF, IF, EX, MA, and WB/DSP. Data is transferred via the X bus, so there is no contention with the IF of other instructions.

(2) Y Memory Load Instructions

Instruction Types:

```
NOPY
MOVY.W   @Ay,Dy
MOVY.W   @Ay+,Dy
MOVY.W   @Ay+Iy,Dy
```

Pipeline:

Slot	←→	←→	←→	←→	←→	←→	←→	←→
Instruction A	IF	ID	EX	MA	WB/DSP			
Next instruction		IF	ID	EX	MA	WB/DSP		
Subsequent instruction			IF	ID	EX	MA	WB/DSP	
. . . . . .								

Figure 10-75 Y Memory Load Instruction Pipeline

Operation Description:

The pipeline has five stages: IF, ID, EX, MA, and WB/DSP. Data is transferred via the Y bus, so there is no contention with the IF of other instructions.

(3) X Memory Store Instructions

Instruction Types:

```
MOVX.W   Da,@Ax
MOVX.W   Da,@Ax+
MOVX.W   Da,@Ax+Ix
```

Pipeline:

Slot	←→	←→	←→	←→	←→	←→	←→	←→
Instruction A	IF	ID	EX	MA				
Next instruction		IF	ID	EX	MA	......		
Subsequent instruction			IF	ID	EX	MA	......	
. . . . . .								

Figure 10-76 X Memory Store Instruction Pipeline

Operation Description:

The pipeline has four stages: IF, ID, EX, and MA. If this instruction attempts to access the DSP operation result immediately after a DSP operation instruction, contention occurs (see section 10.2.6, Contention between DSP Data Operation Instructions and Store Instructions (SH3-DSP Only)).

(4) Y Memory Store Instructions

Instruction Types:

```
MOVY.W   Da,@Ay
MOVY.W   Da,@Ay+
MOVY.W   Da,@Ay+Iy
```

Pipeline:

Slot	←→	←→	←→	←→	←→	←→	←→	←→
Instruction A	IF	ID	EX	MA				
Next instruction		IF	ID	EX	MA	……		
Subsequent instruction			IF	ID	EX	MA	……	
. . . . . .								

Figure 10-77 Y Memory Store Instruction Pipeline

Operation Description:

The pipeline has four stages: IF, ID, EX, and MA. If this instruction attempts to access the DSP operation result immediately after a DSP operation instruction, contention occurs (see section 10.2.6, Contention between DSP Data Operation Instructions and Store Instructions (SH3-DSP Only)).

(5) Single Load Instructions

Instruction Types:

```
MOVS.W   @-As,Ds
MOVS.W   @As,Ds
MOVS.W   @As+,Ds
MOVS.W   @As+Is,Ds
MOVS.L   @-As,Ds
MOVS.L   @As,Ds
MOVS.L   @As+,Ds
MOVS.L   @As+Is,Ds
```

Pipeline:

```
Slot                     ←→    ←→    ←→    ←→    ←→    ←→    ←→    ←→
Instruction A            IF    ID    EX    MA    WB/DSP
Next instruction               IF    ID    EX    MA    WB/DSP
Subsequent instruction               IF    ID    EX    MA    WB/DSP
. . . . . .
```

Figure 10-78 Single Load Instruction Pipeline

Operation Description:

The pipeline has five stages: IF, ID, EX, MA, and WB/DSP. No contention occurs even if another instruction uses the destination register of this instruction.

(6) Single Store Instructions

Instruction Types:

```
MOVS.W   Ds,@-As
MOVS.W   Ds,@As
MOVS.W   Ds,@As+
MOVS.W   Ds,@As+Is
MOVS.L   Ds,@-As
MOVS.L   Ds,@As
MOVS.L   Ds,@As+
MOVS.L   Ds,@As+I
```

Pipeline:

Slot	←→	←→	←→	←→	←→	←→	←→	←→
Instruction A	IF	ID	EX	MA				
Next instruction		IF	ID	EX	MA	······		
Subsequent instruction			IF	ID	EX	MA	······	
· · · · · ·								

Figure 10-79 Single Store Instruction Pipeline

Operation Description:

The pipeline has four stages: IF, ID, EX, and MA. If this instruction attempts to store the DSP operation result immediately after a DSP operation instruction, contention occurs (see section 10.2.6, Contention between DSP Data Operation Instructions and Store Instructions (SH3-DSP Only)).

10.4.10 DSP Operation Instructions (SH3-DSP Only)

(1) ALU Arithmetic Operation Instructions

Instruction Types:

	PADD Sx, Sy,Dz(Du)		PNEG Sx,Dz
DCT	PADD Sx, Sy,Dz	DCT	PNEG Sx,Dz
DCF	PADD Sx, Sy,Dz	DCF	PNEG Sx,Dz
	PSUB Sx, Sy,Dz(Du)		PNEG Sy,Dz
DCT	PSUB Sx, Sy,Dz	DCT	PNEG Sy,Dz
DCF	PSUB Sx, Sy,Dz	DCF	PNEG Sy,Dz
	PCOPY Sx,Dz		PDEC Sx,Dz
DCT	PCOPY Sx,Dz	DCT	PDEC Sx,Dz
DCF	PCOPY Sx,Dz	DCF	PDEC Sx,Dz
	PCOPY Sy,Dz		PDEC Sy,Dz
DCT	PCOPY Sy,Dz	DCT	PDEC Sy,Dz
DCF	PCOPY Sy,Dz	DCF	PDEC Sy,Dz
	PDMSB Sx,Dz		PCLR Dz
DCT	PDMSB Sx,Dz	DCT	PCLR Dz
DCF	PDMSB Sx,Dz	DCF	PCLR Dz
	PDMSB Sy,Dz		PADDC Sx,Sy,Dz
DCT	PDMSB Sy,Dz		PSUBC Sx,Sy,Dz
DCF	PDMSB Sy,Dz		PCMP Sx,Sy
	PINC Sx,Dz		PABS Sx,Dz
DCT	PINC Sx,Dz		PABS Sy,Dz
DCF	PINC Sx,Dz		PRND Sx,Dz
	PINC Sy,Dz		PRND Sy,Dz
DCT	PINC Sy,Dz		
DCF	PINC Sy,Dz		

Pipeline:

Slot	←→	←→	←→	←→	←→	←→	←→	←→
Instruction A	IF	ID	EX	MA	WB/DSP			
Next instruction		IF	ID	EX	MA	WB/DSP		
Subsequent instruction			IF	ID	EX	MA	WB/DSP	
. . . . . .								

Figure 10-80 ALU Arithmetic Operation Instruction Pipeline

Operation Description:

The pipeline has five stages: IF, ID, EX, MA, and WB/DSP. If the condition of a conditional operation instruction is not satisfied, the WB/DSP stage is not executed (no operation), but the pipeline does not change.

(2) ALU Logical Operation Instructions

Instruction Types:

```
        POR  Sx,Sy,Dz
DCT     POR  Sx,Sy,Dz
DCF     POR  Sx,Sy,Dz
        PAND Sx,Sy,Dz
DCT     PAND Sx,Sy,Dz
DCF     PAND Sx,Sy,Dz
        PXOR Sx,Sy,Dz
DCT     PXOR Sx,Sy,Dz
DCF     PXOR Sx,Sy,Dz
```

Pipeline:

```
Slot                    <-->  <-->  <-->  <-->  <-->  <-->  <-->  <-->
Instruction A            IF    ID    EX    MA  WB/DSP
Next instruction               IF    ID    EX    MA  WB/DSP
Subsequent instruction               IF    ID    EX    MA  WB/DSP
. . . . . .
```

Figure 10-81 ALU Logical Operation Instruction Pipeline

Operation Description:

The pipeline has five stages: IF, ID, EX, MA, and WB/DSP. If the condition of a conditional operation instruction is not satisfied, the WB/DSP stage is not executed (no operation), but the pipeline does not change.

(3) ALU Logical Operation Instructions

Instruction Types:

```
        PSHA Sx,Sy,Dz
DCT     PSHA Sx,Sy,Dz
DCF     PSHA Sx,Sy,Dz
        PSHA #Imm,Dz
        PSHL Sx,Sy,Dz
DCT     PSHL Sx,Sy,Dz
DCF     PSHL Sx,Sy,Dz
        PSHL #Imm,Dz
```

Pipeline:

```
Slot                    ←→    ←→    ←→    ←→    ←→    ←→    ←→    ←→
Instruction A           IF    ID    EX    MA   WB/DSP
Next instruction              IF    ID    EX    MA   WB/DSP
Subsequent instruction              IF    ID    EX    MA   WB/DSP
. . . . . .
```

Figure 10-82 ALU Logical Operation Instruction Pipeline

Operation Description:

The pipeline has five stages: IF, ID, EX, MA, and WB/DSP. If the condition of a conditional operation instruction is not satisfied, the WB/DSP stage is not executed (no operation), but the pipeline does not change.

(4) Signed Multiplication Instruction

Instruction Types:

```
PMULS Se,Sf,Dg
```

Pipeline:

Slot	←→	←→	←→	←→	←→	←→	←→	←→
Instruction A	IF	ID	EX	MA	WB/DSP			
Next instruction		IF	ID	EX	MA	WB/DSP		
Subsequent instruction			IF	ID	EX	MA	WB/DSP	
. . . . . .								

Figure 10-83 Signed Multiplication Instruction Pipeline

Operation Description:

The pipeline has five stages: IF, ID, EX, MA, and WB/DSP.

(5) Register Transfer Instructions

Instruction Types:

```
        PSTS MACH,Dz
DCT     PSTS MACH,Dz
DCF     PSTS MACH,Dz
        PSTS MACL,Dz
DCT     PSTS MACL,Dz
DCF     PSTS MACL,Dz
        PLDS Dz,MACH
DCT     PLDS Dz,MACH
DCF     PLDS Dz,MACH
        PLDS Dz,MACL
DCT     PLDS Dz,MACL
DCF     PLDS Dz,MACL
```

Pipeline:

```
Slot                    ←→   ←→   ←→   ←→   ←→   ←→   ←→   ←→
Instruction A           IF   ID   EX   MA   WB/DSP
Next instruction             IF   ID   EX   MA   WB/DSP
Subsequent instruction            IF   ID   EX   MA   WB/DSP
. . . . . .
```

Figure 10-84 Register Transfer Instruction Pipeline

Operation Description:

The pipeline has five stages: IF, ID, EX, MA, and WB/DSP. If the condition of a conditional operation instruction is not satisfied, the WB/DSP stage is not executed (no operation), but the pipeline does not change. If a memory load is performed in parallel with this instruction using MOVX.W, MOVS.W, or MOVX.L, contention occurs. Contention also occurs if a memory store is performed immediately after this instruction using MOVX.W, MOVS.W, or MOVX.L (see section 10.2.7, Contention between DSP Register Transfer and Memory Load/Store Operations (SH3-DSP Only)).

Appendix A Instruction Code

A.1 Instruction Set by Addressing Mode

Table A-1 Instruction Set by Addressing Mode

Addressing Mode	Category	Sample Instruction		Types SH-3	SH-3E	SH3-DSP
No operand	—	NOP		11	11	11
Direct register addressing	Destination operand only	MOVT	Rn	18	23	18
	Source and destination operand	ADD	Rm,Rn	36	44	36
	Transfer to control register or system register	LDC	Rm,SR	16	19	26
	Transfer from control register or system register	STS	MACH,Rn	16	19	25
Indirect register addressing	Source operand only	JMP	@Rn	3	3	3
	Destination operand only	TAS.B	@Rn	1	1	1
	Data transfer direct from register	MOV.L	Rm,@Rn	6	8	6
Post-increment indirect register addressing	Multiply/accumulate operation	MAC.W	@Rm+,@Rn+	2	2	2
	Data transfer direct from register	MOV.L	@Rm+,Rn	3	4	3
	Load to control register or system register	LDC.L	@Rm+,SR	16	18	25
Pre-decrement indirect register addressing	Data transfer direct from register	MOV.L	Rm,@–Rn	3	4	3
	Store from control register or system register	STC.L	SR,@–Rn	16	18	25
Indirect register addressing with displacement	Data transfer direct to register	MOV.L	Rm, @(disp,Rn)	6	6	6
Indirect indexed register addressing	Data transfer direct to register	MOV.L	Rm,@(R0,Rn)	6	8	6
Indirect GBR addressing with displacement	Data transfer direct to register	MOV.L	R0, @(disp,GBR)	6	6	6
Indirect indexed GBR addressing	Immediate data transfer	AND.B	#imm, @(R0,GBR)	4	4	4
PC relative addressing with displacement	Data transfer direct to register	MOV.L	@(disp,PC), Rn	3	3	5
PC relative addressing with Rn	Branch instruction	BRAF	Rn	2	2	2
PC relative addressing	Branch instruction	BRA	disp	6	6	6
Immediate addressing	Load to register	FLDI0	FRn	0	2	0
	Arithmetic logical operations direct with register	ADD	#imm,Rn	7	7	7
	Specify exception processing vector	TRAPA	#imm	1	1	1
	Load to control register	SETRC	#imm	0	0	1
			Total:	189	220	227

A.1.1 No Operand

Table A-2 No Operand

Instruction	Operation	Code	Cycles	T Bit
CLRS	0 → S	0000000001001000	1	—
CLRT	0 → T	0000000000001000	1	0
CLRMAC	0 → MACH, MACL	0000000000101000	1	—
DIV0U	0 → M/Q/T	0000000000011001	1	0
LDTLB	PTEH/PTEL → TLB	0000000000111000	1	—
NOP	No operation	0000000000001001	1	—
RTE	Delayed branching, SSR/SPC → SR/PC	0000000000101011	4	—
RTS	Delayed branching, PR → PC	0000000000001011	2	—
SETS	1 → S	0000000001011000	1	—
SETT	1 → T	0000000000011000	1	1
SLEEP	Sleep	0000000000011011	4	—

A.1.2 Direct Register Addressing

Table A-3 Destination Operand Only

Instruction		Operation	Code	Cycles	T Bit
CMP/PL	Rn	Rn > 0, 1 → T	0100nnnn00010101	1	Comparison result
CMP/PZ	Rn	Rn ≥ 0, 1 → T	0100nnnn00010001	1	Comparison result
DT	Rn	Rn – 1 → Rn, when Rn is 0, 1 → T. When Rn is nonzero, 0 → T	0100nnnn00010000	1	Comparison result
FABS	FRn*	abs(FRn → FRn	1111nnnn01011101	1	—
FLOAT	FPUL, FRn*	(float)FPUL → FRn	1111nnnn00101101	1	—
FNEG	FRn*	–1.0 × FRn → FRn	1111nnnn01001101	1	—
FSQRT	FRn*	sqrt(FRn) → FRn	1111nnnn01101101	13	—
FTRC	FRm, FPUL*	(long)FRm → FPUL	1111mmmm00111101	1	—
MOVT	Rn	T → Rn	0000nnnn00101001	1	—
ROTL	Rn	T ← Rn ← MSB	0100nnnn00000100	1	MSB
ROTR	Rn	LSB → Rn → T	0100nnnn00000101	1	LSB
ROTCL	Rn	T ← Rn ← T	0100nnnn00100100	1	MSB
ROTCR	Rn	T → Rn → T	0100nnnn00100101	1	LSB
SHAL	Rn	T ← Rn ← 0	0100nnnn00100000	1	MSB
SHAR	Rn	MSB → Rn → T	0100nnnn00100001	1	LSB
SHLL	Rn	T ← Rn ← 0	0100nnnn00000000	1	MSB
SHLR	Rn	0 → Rn → T	0100nnnn00000001	1	LSB
SHLL2	Rn	Rn << 2 → Rn	0100nnnn00001000	1	—
SHLR2	Rn	Rn >> 2 → Rn	0100nnnn00001001	1	—
SHLL8	Rn	Rn << 8 → Rn	0100nnnn00011000	1	—
SHLR8	Rn	Rn >> 8 → Rn	0100nnnn00011001	1	—
SHLL16	Rn	Rn << 16 → Rn	0100nnnn00101000	1	—
SHLR16	Rn	Rn >> 16 → Rn	0100nnnn00101001	1	—

Note: * Floating point arithmetic calculation instruction or CPU instruction related to the FPU. These instructions are available only on the SH-3E.

Table A-4 Source and Destination Operand

Instruction		Operation	Code	Cycles	T Bit
ADD	Rm,Rn	Rn + Rm → Rn	0011nnnnmmmm1100	1	—
ADDC	Rm,Rn	Rn + Rm + T → Rn, carry → T	0011nnnnmmmm1110	1	Carry
ADDV	Rm,Rn	Rn + Rm → Rn, overflow → T	0011nnnnmmmm1111	1	Overflow
AND	Rm,Rn	Rn & Rm → Rn	0010nnnnmmmm1001	1	—
CMP/EQ	Rm,Rn	When Rn = Rm, 1 → T	0011nnnnmmmm0000	1	Comparison result
CMP/HS	Rm,Rn	When unsigned and Rn ≥ Rm, 1 → T	0011nnnnmmmm0010	1	Comparison result
CMP/GE	Rm,Rn	When signed and Rn ≥ Rm, 1 → T	0011nnnnmmmm0011	1	Comparison result
CMP/HI	Rm,Rn	When unsigned and Rn > Rm, 1 → T	0011nnnnmmmm0110	1	Comparison result
CMP/GT	Rm,Rn	When signed and Rn > Rm, 1 → T	0011nnnnmmmm0111	1	Comparison result
CMP/STR	Rm,Rn	When a byte in Rn equals a bytes in Rm, 1 → T	0010nnnnmmmm1100	1	Comparison result
DIV1	Rm,Rn	1 step division (Rn ÷ Rm)	0011nnnnmmmm0100	1	Calculation result
DIV0S	Rm,Rn	MSB of Rn → Q, MSB of Rm → M, M ^ Q → T	0010nnnnmmmm0111	1	Calculation result
DMULS.L	Rm,Rn	Signed operation of Rn x Rm → MACH, MACL	0011nnnnmmmm1101	2 (to 5)*[2]	—
DMULU.L	Rm,Rn	Unsigned operation of Rn × Rm → MACH, MACL	0011nnnnmmmm0101	2 (to 5)*[2]	—
EXTS.B	Rm,Rn	Sign – extend Rm from byte → Rn	0110nnnnmmmm1110	1	—
EXTS.W	Rm,Rn	Sign – extend Rm from word → Rn	0110nnnnmmmm1111	1	—
EXTU.B	Rm,Rn	Zero – extend Rm from byte → Rn	0110nnnnmmmm1100	1	—
EXTU.W	Rm,Rn	Zero – extend Rm from word → Rn	0110nnnnmmmm1101	1	—
FADD	FRm, FRn*[1]	FRm + FRn → FRn	1111nnnnmmmm0000	1	—

Table A-4 Source and Destination Operand (cont)

Instruction		Operation	Code	Cycles	T Bit
FCMP/EQ	FRm, FRn*[1]	FRn = FRm, 1 → T	1111nnnnmmmm0100	1	Comparison result
FCMP/GT	FRm, FRn*[1]	FRn > FRm, 1 → T	1111nnnnmmmm0101	1	Comparison result
FDIV	FRm, FRn*[1]	FRn/FRm → FRm	1111nnnnmmmm0011	13	—
FMAC	FR0,Frm FRn*[1]	(FR0 × FRm) + FRn → FRn	1111nnnnmmmm1110	1	—
FMOV	FRm, FRn*[1]	FRm → FRn	1111nnnnmmmm1100	1	—
FMUL	FRm, FRn*[1]	FRn × FRm → FRn	1111nnnnmmmm0010	1	—
FSUB	FRm, FRn*[1]	FRn – FRm → FRn	1111nnnnmmmm0001	1	—
MOV	Rm,Rn	Rm → Rn	0110nnnnmmmm0011	1	—
MUL.L	Rm,Rn	Rn × Rm → MAC	0000nnnnmmmm0111	2 (to 5)*[2]	—
MULS.W	Rm,Rn	With sign, Rn × Rm → MAC	0010nnnnmmmm1111	1 (to 3)*[2]	—
MULU.W	Rm,Rn	Unsigned, Rn × Rm → MAC	0010nnnnmmmm1110	1 (to 3)*[2]	—
NEG	Rm,Rn	0 – Rm → Rn	0110nnnnmmmm1011	1	—
NEGC	Rm,Rn	0 – Rm – T → Rn, Borrow → T	0110nnnnmmmm1010	1	Borrow
NOT	Rm,Rn	~Rm → Rn	0110nnnnmmmm0111	1	—
OR	Rm,Rn	Rn | Rm → Rn	0010nnnnmmmm1011	1	—
SHAD	Rm,Rn	Rn ≥ 0; Rn << Rm → Rn Rn < 0; Rn >> Rm → (MSB→)Rn	0100nnnnmmmm1100	1	—
SHLD	Rm,Rn	Rn ≥ 0; Rn << Rm → Rn Rn < 0; Rn >> Rm → (0→)Rn	0100nnnnmmmm1101	1	—
SUB	Rm,Rn	Rn – Rm → Rn	0011nnnnmmmm1000	1	—
SUBC	Rm,Rn	Rn – Rm – T → Rn, Borrow → T	0011nnnnmmmm1010	1	Borrow

Table A-4 Source and Destination Operand (cont)

Instruction		Operation	Code	Cycles	T Bit
SUBV	Rm,Rn	Rn – Rm → Rn, Underflow → T	0011nnnnmmmm1011	1	Underflow
SWAP.B	Rm,Rn	Rm → Swap upper and lower halves of lower 2 bytes → Rn	0110nnnnmmmm1000	1	—
SWAP.W	Rm,Rn	Rm → Swap upper and lower word → Rn	0110nnnnmmmm1001	1	—
TST	Rm,Rn	Rn & Rm, when result is 0, 1 → T	0010nnnnmmmm1000	1	Test results
XOR	Rm,Rn	Rn ^ Rm → Rn	0010nnnnmmmm1010	1	—
XTRCT	Rm,Rn	Rm: Center 32 bits of Rn → Rn	0010nnnnmmmm1101	1	—

Notes: 1. Floating point arithmetic calculation instruction or CPU instruction related to the FPU. These instructions are available only on the SH-3E.

2. Normal minimum number of execution states (the number in parentheses is the number of states when there is contention with preceding/following instructions).

Table A-5 Load and Store with Control Register or System Register

Instruction		Operation	Code	Cycles	T Bit
FLDS	FRm,FPUL*[1]	FRm → FPUL	1111mmmm00011101	1	—
LDC	Rm,SR	Rm → SR	0100mmmm00001110	5	LSB
LDC	Rm,GBR	Rm → GBR	0100mmmm00011110	1/3*[2]	—
LDC	Rm,VBR	Rm → VBR	0100mmmm00101110	1/3*[2]	—
LDC	Rm,SSR	Rm → SSR	0100mmmm00111110	1/3*[2]	—
LDC	Rm,SPC	Rm → SPC	0100mmmm01001110	1/3*[2]	—
LDC	Rm,MOD*[3]	Rm → MOD	0100mmmm01011110	3	—
LDC	Rm,RE*[3]	Rm → RE	0100mmmm01111110	3	—
LDC	Rm,RS*[3]	Rm → RS	0100mmmm01101110	3	—
LDC	Rm,R0_BANK	Rm → R0_BANK	0100mmmm10001110	1/3*[2]	—
LDC	Rm,R1_BANK	Rm → R1_BANK	0100mmmm10011110	1/3*[2]	—
LDC	Rm,R2_BANK	Rm → R2_BANK	0100mmmm10101110	1/3*[2]	—
LDC	Rm,R3_BANK	Rm → R3_BANK	0100mmmm10111110	1/3*[2]	—
LDC	Rm,R4_BANK	Rm → R4_BANK	0100mmmm11001110	1/3*[2]	—
LDC	Rm,R5_BANK	Rm → R5_BANK	0100mmmm11011110	1/3*[2]	—
LDC	Rm,R6_BANK	Rm → R6_BANK	0100mmmm11101110	1/3*[2]	—
LDC	Rm,R7_BANK	Rm → R7_BANK	0100mmmm11111110	1/3*[2]	—
LDS	Rm,FPSCR*[1]	Rm → FPSCR	0100mmmm01101010	1	—
LDS	Rm,FPUL*[1]	Rm → FPUL	0100mmmm01011010	1	—
LDS	Rm,MACH	Rm → MACH	0100mmmm00001010	1	—
LDS	Rm,MACL	Rm → MACL	0100mmmm00011010	1	—
LDS	Rm,PR	Rm → PR	0100mmmm00101010	1	—
LDS	Rm,DSR*[3]	Rm → DSR	0100mmmm01101010	1	—
LDS	Rm,A0*[3]	Rm → A0	0100mmmm01111010	1	—
LDS	Rm,X0*[3]	Rm → X0	0100mmmm10001010	1	—
LDS	Rm,X1*[3]	Rm → X1	0100mmmm10011010	1	—
LDS	Rm,Y0*[3]	Rm → Y0	0100mmmm10101010	1	—
LDS	Rm,Y1*[3]	Rm → Y1	0100mmmm10111010	1	—
SETRC	Rm*[3]	LSW of Rm → RC (MSW of SR), Repeat control flag → RF1, RF0	0100mmmm00010100	3	—

Notes: 1. Floating point arithmetic calculation instruction or CPU instruction related to the FPU. These instructions are available only on the SH-3E.
2. Three cycles on the SH3-DSP.
3. CPU instructions to provide support for DSP functions. These instructions can only be used with the SH3-DSP.

Table A-6 Load and Store from Control Register or System Register

Instruction		Operation	Code	Cycles	T Bit
FSTS	FPUL,FRn*[1]	FPUL → FRn	1111nnnn01011010	1	—
STC	SR,Rn	SR → Rn	0000nnnn00000010	1	—
STC	GBR,Rn	GBR → Rn	0000nnnn00010010	1	—
STC	VBR,Rn	VBR → Rn	0000nnnn00100010	1	—
STC	SSR,Rn	SSR → Rn	0000nnnn00110010	1	—
STC	SPC,Rn	SPC → Rn	0000nnnn01000010	1	—
STC	MOD,Rn*[2]	MOD → Rn	0000nnnn01010010	1	—
STC	RE,Rn*[2]	RE → Rn	0000nnnn01110010	1	—
STC	RS,Rn*[2]	RS → Rn	0000nnnn01100010	1	—
STC	R0_BANK,Rn	R0_BANK→ Rn	0000nnnn10000010	1	—
STC	R1_BANK,Rn	R1_BANK→ Rn	0000nnnn10010010	1	—
STC	R2_BANK,Rn	R2_BANK→ Rn	0000nnnn10100010	1	—
STC	R3_BANK,Rn	R3_BANK→ Rn	0000nnnn10110010	1	—
STC	R4_BANK,Rn	R4_BANK→ Rn	0000nnnn11000010	1	—
STC	R5_BANK,Rn	R5_BANK→ Rn	0000nnnn11010010	1	—
STC	R6_BANK,Rn	R6_BANK→ Rn	0000nnnn11100010	1	—
STC	R7_BANK,Rn	R7_BANK→ Rn	0000nnnn11110010	1	—
STS	FPSCR,Rn*[1]	FPSCR → Rn	1111nnnn01101010	1	—
STS	FPUL,Rn*[1]	FPUL → Rn	1111nnnn01011010	1	—
STS	MACH,Rn	MACH → Rn	0000nnnn00001010	1	—
STS	MACL,Rn	MACL → Rn	0000nnnn00011010	1	—
STS	PR,Rn	PR → Rn	0000nnnn00101010	1	—
STS	DSR,Rn*[2]	DSR → Rn	0000nnnn01101010	1	—
STS	A0,Rn*[2]	A0 → Rn	0000nnnn01111010	1	—
STS	X0,Rn*[2]	X0→Rn	0000nnnn10001010	1	—
STS	X1,Rn*[2]	X1→Rn	0000nnnn10011010	1	—
STS	Y0,Rn*[2]	Y0→Rn	0000nnnn10101010	1	—
STS	Y1,Rn*[2]	Y1→Rn	0000nnnn10111010	1	—

Notes: 1. Floating point arithmetic calculation instruction or CPU instruction related to the FPU. These instructions are available only on the SH-3E.

2. CPU instructions to provide support for DSP functions. These instructions can only be used with the SH3-DSP.

A.1.3 Indirect Register Addressing

Table A-7 Source Operand Only

Instruction		Operation	Code	Cycles	T Bit
JMP	@Rn	Delayed branching, Rn → PC	0100nnnn00101011	2	—
JSR	@Rn	Delayed branching, PC → Rn, Rn → PC	0100nnnn00001011	2	—
PREF	@Rn	(Rn) → cache	0000nnnn10000011	1	—

Note: * Two cycles on the SH3-DSP.

Table A-8 Destination Operand Only

Instruction		Operation	Code	Cycles	T Bit
TAS.B	@Rn	When (Rn) is 0, 1 → T, 1 → MSB of (Rn)	0100nnnn00011011	3	Test results

Note: * Four cycles on the SH3-DSP.

Table A-9 Data Transfer Direct to Register

Instruction		Operation	Code	Cycles	T Bit
FMOV.S	FRm,@Rn*	FRm → (FRn)	1111nnnnmmmm1010	1	—
FMOV.S	@Rm,FRn*	(Rm) → FRn	1111nnnnmmmm1000	1	—
MOV.B	Rm,@Rn	Rm → (Rn)	0010nnnnmmmm0000	1	—
MOV.W	Rm,@Rn	Rm → (Rn)	0010nnnnmmmm0001	1	—
MOV.L	Rm,@Rn	Rm → (Rn)	0010nnnnmmmm0010	1	—
MOV.B	@Rm,Rn	(Rm) → sign extension → Rn	0110nnnnmmmm0000	1	—
MOV.W	@Rm,Rn	(Rm) → sign extension → Rn	0110nnnnmmmm0001	1	—
MOV.L	@Rm,Rn	(Rm) → Rn	0110nnnnmmmm0010	1	—

Note: * Floating point arithmetic calculation instruction or CPU instruction related to the FPU. These instructions are available only on the SH-3E.

A.1.4 Post-Increment Indirect Register Addressing

Table A-10 Multiply/Accumulate Operation

Instruction	Operation	Code	Cycles	T Bit
MAC.L @Rm+,@Rn+	Signed operation of (Rn) × (Rm) + MAC → MAC	0000nnnnmmmm1111	2 (to 5)*	—
MAC.W @Rm+,@Rn+	Signed operation of (Rn) × (Rm) + MAC → MAC	0100nnnnmmmm1111	2 (to 5)*	—

Note: * Normal minimum number of execution states (the number in parenthesis is the number of states when there is contention with preceding/following instructions).

Table A-11 Data Transfer Direct from Register

Instruction	Operation	Code	Cycles	T Bit
FMOV.S @Rm+,FRn*	(Rm) → FRn, Rm + 4 → Rm	1111nnnnmmmm1001	1	—
MOV.B @Rm+,Rn	(Rm) → sign extension → Rn, Rm + 1 → Rm	0110nnnnmmmm0100	1	—
MOV.W @Rm+,Rn	(Rm) → sign extension → Rn, Rm + 2 → Rm	0110nnnnmmmm0101	1	—
MOV.L @Rm+,Rn	(Rm) → Rn, Rm + 4 → Rm	0110nnnnmmmm0110	1	—

Note: * Floating point arithmetic calculation instruction or CPU instruction related to the FPU. These instructions are available only on the SH-3E.

Table A-12 Load to Control Register or System Register

Instruction	Operation	Code	Cycles	T Bit
LDC.L @Rm+,SR	(Rm) → SR, Rm + 4 → Rm	0100mmmm00000111	7	LSB
LDC.L @Rm+,GBR	(Rm) → GBR, Rm + 4 → Rm	0100mmmm00010111	1/5*2	—
LDC.L @Rm+,VBR	(Rm) → VBR, Rm + 4 → Rm	0100mmmm00100111	1/5*2	—
LDC.L @Rm+,SSR	(Rm) → SSR, Rm + 4 → Rm	0100mmmm00110111	1/5*2	—
LDC.L @Rm+,SPC	(Rm) → SPC, Rm + 4 → Rm	0100mmmm01000111	1/5*2	—
LDC.L @Rm+,MOD*3	(Rm) → MOD, Rm + 4 → Rm	0100mmmm01010111	5	—
LDC.L @Rm+,RE*3	(Rm) → RE, Rm + 4 → Rm	0100mmmm01110111	5	—
LDC.L @Rm+,RS*3	(Rm) → RS, Rm + 4 → Rm	0100mmmm01100111	5	—
LDC.L @Rm+,R0_BANK	(Rm) → R0_BANK, Rm + 4 → Rm	0100mmmm10000111	1/5*2	—

Table A-12 Load to Control Register or System Register (cont)

Instruction	Operation	Code	Cycles	T Bit
LDC.L @Rm+,R1_BANK	(Rm) → R1_BANK, Rm + 4 → Rm	0100mmmm10010111	1/5*2	—
LDC.L @Rm+,R2_BANK	(Rm) → R2_BANK, Rm + 4 → Rm	0100mmmm10100111	1/5*2	—
LDC.L @Rm+,R3_BANK	(Rm) → R3_BANK, Rm + 4 → Rm	0100mmmm10110111	1/5*2	—
LDC.L @Rm+,R4_BANK	(Rm) → R4_BANK, Rm + 4 → Rm	0100mmmm11000111	1/5*2	—
LDC.L @Rm+,R5_BANK	(Rm) → R5_BANK, Rm + 4 → Rm	0100mmmm11010111	1/5*2	—
LDC.L @Rm+,R6_BANK	(Rm) → R6_BANK, Rm + 4 → Rm	0100mmmm11100111	1/5*2	—
LDC.L @Rm+,R7_BANK	(Rm) → R7_BANK, Rm + 4 → Rm	0100mmmm11110111	1/5*2	—
LDS.L @Rm+,FPSCR*1	(Rm) → FPSCR, Rm + 4 → Rm	0100mmmm01100110	1	—
LDS.L @Rm+,FPUL*1	(Rm) → FPUL, Rm + 4 → Rm	0100mmmm01010110	1	—
LDS.L @Rm+,MACH	(Rm) → MACH, @Rm + 4 → Rm	0100mmmm00000110	1	—
LDS.L @Rm+,MACL	(Rm) → MACL, @Rm + 4 → Rm	0100mmmm00010110	1	—
LDS.L @Rm+,PR	(Rm) → PR, @Rm + 4 → Rm	0100mmmm00100110	1	—
LDS.L @Rm+,DSR*3	(Rm) → DSR, Rm + 4 → Rm	0100mmmm01100110	1	—
LDS.L @Rm+,A0*3	(Rm) → A0, Rm + 4 → Rm	0100mmmm01110110	1	—
LDS.L @Rm+,X0*3	(Rm) → X0,Rm+4 → Rm	0100nnnn10000110	1	—
LDS.L @Rm+,X1*3	(Rm) → X1,Rm+4 → Rm	0100nnnn10010110	1	—
LDS.L @Rm+,Y0*3	(Rm) → Y0,Rm+4 → Rm	0100nnnn10100110	1	—
LDS.L @Rm+,Y1*3	(Rm) → Y1,Rm+4 → Rm	0100nnnn10110110	1	—

Notes: 1. Floating point arithmetic calculation instruction or CPU instruction related to the FPU. These instructions are available only on the SH-3E.
2. Five cycles on the SH3-DSP.
3. CPU instructions to provide support for DSP functions. These instructions can only be used with the SH3-DSP.

A.1.5 Pre-Decrement Indirect Register Addressing

Table A-13 Data Transfer Direct from Register

Instruction	Operation	Code	Cycles	T Bit
FMOV.S FRm,@–Rn*	Rn – 4 → Rn, FRm → (Rn)	1111nnnnmmmm1011	1	—
MOV.B Rm,@–Rn	Rn – 1 → Rn, Rm → (Rn)	0010nnnnmmmm0100	1	—
MOV.W Rm,@–Rn	Rn – 2 → Rn, Rm → (Rn)	0010nnnnmmmm0101	1	—
MOV.L Rm,@–Rn	Rn – 4 → Rn, Rm → (Rn)	0010nnnnmmmm0110	1	—

Note: * Floating point arithmetic calculation instruction or CPU instruction related to the FPU. These instructions are available only on the SH-3E.

Table A-14 Store from Control Register or System Register

Instruction	Operation	Code	Cycles	T Bit
STC.L SR,@-Rn	Rn – 4 → Rn, SR → (Rn)	0100nnnn00000011	1/2*1	—
STC.L GBR,@-Rn	Rn – 4 → Rn, GBR → (Rn)	0100nnnn00010011	1/2*1	—
STC.L VBR,@-Rn	Rn – 4 → Rn, VBR → (Rn)	0100nnnn00100011	1/2*1	—
STC.L SSR,@–Rn	Rn–4 → Rn, SSR → (Rn)	0100nnnn00110011	1/2*1	—
STC.L SPC,@–Rn	Rn–4 → Rn, SPC → (Rn)	0100nnnn01000011	1/2*1	—
STC.L MOD,@-Rn*3	Rn – 4 → Rn, MOD → (Rn)	0100nnnn01010011	2	—
STC.L RE,@-Rn*3	Rn – 4 → Rn, RE → (Rn)	0100nnnn01110011	2	—
STC.L RS,@-Rn*3	Rn – 4 → Rn, RS → (Rn)	0100nnnn01100011	2	—
STC.L R0_BANK, @–Rn	Rn–4 → Rn, R0_BANK → (Rn)	0100nnnn10000011	2	—
STC.L R1_BANK, @–Rn	Rn–4 → Rn, R1_BANK → (Rn)	0100nnnn10010011	2	—
STC.L R2_BANK, @–Rn	Rn–4 → Rn, R2_BANK → (Rn)	0100nnnn10100011	2	—
STC.L R3_BANK, @–Rn	Rn–4 → Rn, R3_BANK → (Rn)	0100nnnn10110011	2	—
STC.L R4_BANK, @–Rn	Rn–4 → Rn, R4_BANK → (Rn)	0100nnnn11000011	2	—
STC.L R5_BANK, @–Rn	Rn–4 → Rn, R5_BANK → (Rn)	0100nnnn11010011	2	—
STC.L R6_BANK, @–Rn	Rn–4 → Rn, R6_BANK → (Rn)	0100nnnn11100011	2	—
STC.L R7_BANK, @–Rn	Rn–4 → Rn, R7_BANK → (Rn)	0100nnnn11110011	2	—

Table A-14 Store from Control Register or System Register (cont)

Instruction		Operation	Code	Cycles	T Bit
STS.L	FPSCR,@–Rn*	Rn – 4 → Rn, FPSCR → (Rn)	0100nnnn01100010	1	—
STS.L	FPUL,@–Rn*	Rn – 4 → Rn, FPUL → (Rn)	0100nnnn01010010	1	—
STS.L	MACH,@–Rn	Rn – 4 → Rn, MACH → (Rn)	0100nnnn00000010	1	—
STS.L	MACL,@–Rn	Rn – 4 → Rn, MACL → (Rn)	0100nnnn00010010	1	—
STS.L	PR,@–Rn	Rn – 4 → Rn, PR → (Rn)	0100nnnn00100010	1	—
STS.L	DSR,@–Rn*3	Rn – 4 → Rn, DSR → (Rn)	0100nnnn01100010	1	—
STS.L	A0,@–Rn*3	Rn – 4 → Rn, A0 → (Rn)	0100nnnn01100010	1	—
STS.L	X0,@-Rn*3	Rn–4→Rn,X0→(Rn)	0100nnnn10000010	1	—
STS.L	X1,@-Rn*3	Rn–4→Rn,X1→(Rn)	0100nnnn10010010	1	—
STS.L	Y0,@-Rn*3	Rn–4→Rn,Y0→(Rn)	0100nnnn10100010	1	—
STS.L	Y1,@-Rn*3	Rn–4→Rn,Y1→(Rn)	0100nnnn10110010	1	—

Notes: 1. Floating point arithmetic calculation instruction or CPU instruction related to the FPU. These instructions are available only on the SH-3E.

2. Two cycles on the SH3-DSP.

3. CPU instructions to provide support for DSP functions. These instructions can only be used with the SH3-DSP.

A.1.6 Indirect Register Addressing with Displacement

Table A-15 Indirect Register Addressing with Displacement

Instruction		Operation	Code	Cycles	T Bit
MOV.B	R0,@(disp,Rn)	R0 → (disp + Rn)	10000000nnnndddd	1	—
MOV.W	R0,@(disp,Rn)	R0 → (disp + Rn)	10000001nnnndddd	1	—
MOV.L	Rm,@(disp,Rn)	Rm → (disp + Rn)	0001nnnnmmmmdddd	1	—
MOV.B	@(disp,Rm),R0	(disp + Rm) → sign extension → R0	10000100mmmmdddd	1	—
MOV.W	@(disp,Rm),R0	(disp + Rm) → sign extension → R0	10000101mmmmdddd	1	—
MOV.L	@(disp,Rm),Rn	(disp + Rm) → Rn	0101nnnnmmmmdddd	1	—

A.1.7 Indirect Indexed Register Addressing

Table A-16 Indirect Indexed Register Addressing

Instruction	Operation	Code	Cycles	T Bit
MOV.B Rm,@(R0,Rn)	Rm → (R0 + Rn)	0000nnnnmmmm0100	1	—
MOV.W Rm,@(R0,Rn)	Rm → (R0 + Rn)	0000nnnnmmmm0101	1	—
MOV.L Rm,@(R0,Rn)	Rm → (R0 + Rn)	0000nnnnmmmm0110	1	—
FMOV.S FRm,@(R0,Rn)*	FRm → (R0 + Rn)	1111nnnnmmmm0111	1	—
MOV.B @(R0,Rm),Rn	(R0 + Rm) → sign extension → Rn	0000nnnnmmmm1100	1	—
MOV.W @(R0,Rm),Rn	(R0 + Rm) → sign extension → Rn	0000nnnnmmmm1101	1	—
MOV.L @(R0,Rm),Rn	(R0 + Rm) → Rn	0000nnnnmmmm1110	1	—
FMOV.S @(R0,FRm),FRm*	(R0 + Rn) → FRn	1111nnnnmmmm0110	1	—

Note: * Floating point arithmetic calculation instruction or CPU instruction related to the FPU. These instructions are available only on the SH-3E.

A.1.8 Indirect GBR Addressing with Displacement

Table A-17 Indirect GBR Addressing with Displacement

Instruction	Operation	Code	Cycles	T Bit
MOV.B R0,@(disp,GBR)	R0 → (disp + GBR)	11000000dddddddd	1	—
MOV.W R0,@(disp,GBR)	R0 → (disp + GBR)	11000001dddddddd	1	—
MOV.L R0,@(disp,GBR)	R0 → (disp + GBR)	11000010dddddddd	1	—
MOV.B @(disp,GBR),R0	(disp + GBR) → sign extension → R0	11000100dddddddd	1	—
MOV.W @(disp,GBR),R0	(disp × 2 + GBR) → sign extension → R0	11000101dddddddd	1	—
MOV.L @(disp,GBR),R0	(disp × 4 + GBR) → R0	11000110dddddddd	1	—

A.1.9 Indirect Indexed GBR Addressing

Table A-18 Indirect Indexed GBR Addressing

Instruction	Operation	Code	Cycles	T Bit
AND.B #imm,@(R0,GBR)	(R0 + GBR) & imm → (R0 + GBR)	11001101iiiiiiii	3	—
OR.B #imm,@(R0,GBR)	(R0 + GBR) | imm → (R0 + GBR)	11001111iiiiiiii	3	—
TST.B #imm,@(R0,GBR)	(R0 + GBR) & imm, when result is 0, 1 → T	11001100iiiiiiii	3	Test results
XOR.B #imm,@(R0,GBR)	(R0 + GBR) ^ imm → (R0 + GBR)	11001110iiiiiiii	3	—

A.1.10 PC Relative Addressing with Displacement

Table A-19 PC Relative Addressing with Displacement

Instruction	Operation	Code	Cycles	T Bit
MOV.W @(disp,PC),Rn	(disp × 2 + PC) → sign extension → Rn	1001nnnndddddddd	1	—
MOV.L @(disp,PC),Rn	(disp × 4 + PC) → Rn	1101nnnndddddddd	1	—
MOVA @(disp,PC),R0	disp × 4 + PC → R0	11000111dddddddd	1	—
LDRS @(disp,pc)*	disp × 2+PC→RS	10001100dddddddd	3	—
LDRE @(disp,pc)*	disp × 2+PC→RE	10001110dddddddd	3	—

Note: * SH3-DSP instructions.

A.1.11 PC Relative Addressing

Table A-20 PC Relative Addressing with Rm

Instruction	Operation	Code	Cycles	T Bit
BRAF Rm	Delayed branch, Rm + PC → PC	0000mmmm00100011	2	—
BSRF Rm	Delayed branch, PC → PR, Rm + PC → PC	0000mmmm00000011	2	—

Table A-21 PC Relative Addressing

Instruction		Operation	Code	Cycles	T Bit
BF	label	When T = 0, disp × 2 + PC → PC; when T = 1, nop	10001011dddddddd	3/1	—
BF/S	label	If T = 0, disp × 2 + PC → PC; if T = 1, nop	10001111dddddddd	2/1*	—
BT	label	When T = 1, disp × 2 + PC → PC; when T = 1, nop	10001001dddddddd	3/1	—
BT/S	label	If T = 1, disp × 2 + PC → PC; if T = 0, nop	10001101dddddddd	2/1*	—
BRA	label	Delayed branching, disp × 2 + PC → PC	1010dddddddddddd	2	—
BSR	label	Delayed branching, PC → PR, disp × 2 + PC → PC	1011dddddddddddd	2	—

Note: * One state when it does not branch.

A.1.12 Immediate

Table A-22 Load to Register

Instruction		Operation	Code	Cycles	T Bit
FLDI0	FRn*	0.0 → FRn	1111nnnn10001101	1	—
FLDI1	FRn*	1.0 → FRn	1111nnnn10011101	1	—

Note: * Floating point arithmetic calculation instruction or CPU instruction related to the FPU. These instructions are available only on the SH-3E.

Table A-23 Arithmetic Logical Operations Direct with Register

Instruction		Operation	Code	Cycles	T Bit
ADD	#imm,Rn	Rn + #imm → Rn	0111nnnniiiiiiii	1	—
AND	#imm,R0	R0 & imm → R0	11001001iiiiiiii	1	—
CMP/EQ	#imm,R0	When R0 = imm, 1 → T	10001000iiiiiiii	1	Comparison result
MOV	#imm,Rn	#imm → sign extension → Rn	1110nnnniiiiiiii	1	—
OR	#imm,R0	R0 | imm → R0	11001011iiiiiiii	1	—
TST	#imm,R0	R0 & imm, when result is 0, 1 → T	11001000iiiiiiii	1	Test results
XOR	#imm,R0	R0 ^ imm → R0	11001010iiiiiiii	1	—

Table A-24 Specify Exception Processing Vector

Instruction		Operation	Code	Cycles	T Bit
TRAPA	#imm	imm → TRA, PC → SPC, SR → SSR, 1 → SR.MD/BL/RB, 0x160 → EXPEVT VBR + H'00000100 → PC	11000011iiiiiiii	6/8*	—

Note: * Eight cycles on the SH3-DSP.

Table A-25 Load to Control Register

Instruction	Operation	Code	Cycles	T Bit
SETRC #imm	imm→RC(SR[23:16]), zeros→SR[27:24]	10000010iiiiiiii	3	—

Note: * SH3-DSP instruction.

A.2 Instruction Sets by Instruction Format

Tables A-26 to A-57 list instruction codes and execution cycles by instruction formats.

Table A-26 Instruction Sets by Format

Format	Category	Sample Instruction		Types		
				SH-3	SH-3E	SH3-DSP
0	—	NOP		11	11	11
n	Direct register addressing	MOVT	Rn	18	18	18
	Direct register addressing (store with control or system registers)	STS	MACH,Rn	18	18	25
	Indirect register addressing	TAS	@Rn	1	1	1
	Pre-decrement indirect register addressing	STC.L	SR,@–Rn	16	18	25
	Floating point instruction	FABS	FRn	—	7	—
m	Direct register addressing (load with control or system registers)	LDC	Rm,SR	16	18	26
	PC relative addressing with Rm	BRAF	Rm	2	2	2
	Indirect register addressing	JMP	@Rm	2	2	2
	Post-increment indirect register addressing	LDC.L	@Rm+,SR	16	18	25
	Floating point instruction	FLDS	FRm,FPUL	—	2	—
nm	Direct register addressing	ADD	Rm,Rn	36	36	36
	Indirect register addressing	MOV.L	Rm,@Rn	6	6	6
	Post-increment indirect register addressing (multiply/accumulate operation)	MAC.W	@Rm+,@Rn+	2	2	2
	Post-increment indirect register addressing	MOV.L	@Rm+,Rn	3	3	3
	Pre-decrement indirect register addressing	MOV.L	Rm,@–Rn	3	3	3
	Indirect indexed register addressing	MOV.L	Rm,@(R0,Rn)	6	6	6
	Floating point instruction	FADD	FRm,FRn	—	14	—
md	Indirect register addressing with displacement	MOV.B	@(disp,Rm),R0	2	2	2
nd4	Indirect register addressing with displacement	MOV.B	R0,@(disp,Rn)	2	2	2
nmd	Indirect register addressing with displacement	MOV.L	Rm,@(disp,Rn)	2	2	2
d	Indirect GBR addressing with displacement	MOV.L	R0,@(disp,GBR)	6	6	6
	Indirect PC addressing with displacement	MOVA	@(disp,PC),R0	1	1	3
	PC relative addressing	BF	label	4	4	4
d12	PC relative addressing	BRA	label	2	2	2

Note: * The figures in parentheses ( ) are the totals excluding the SH-3E instructions.

Table A-26 Instruction Sets by Format (cont)

Format	Category	Sample Instruction		Types SH-3	SH-3E	SH3-DSP
nd8	PC relative addressing with displacement	MOV.L	@(disp,PC),Rn	2	2	2
i	Indirect indexed GBR addressing	AND.B	#imm,@(R0,GBR)	4	4	4
	Immediate addressing (arithmetic and logical operations direct with register)	AND	#imm,R0	5	5	5
	Immediate addressing (specify exception processing vector)	TRAPA	#imm	1	1	1
	Load to control register (SH3-DSP only)	SETRC	#imm	—	—	1
ni	Immediate addressing (direct register arithmetic operations and data transfers)	ADD	#imm,Rn	2	2	2
			Total:	189	220	227

A.2.1 0 Format

Table A-27 0 Format

Instruction	Operation	Code	Cycles	T Bit
CLRS	0 → S	0000000001001000	1	—
CLRT	0 → T	0000000000001000	1	0
CLRMAC	0 → MACH, MACL	0000000000101000	1	—
DIV0U	0 → M/Q/T	0000000000011001	1	0
LDTLB	PTEH/PTEL → TLB	0000000000111000	1	—
NOP	No operation	0000000000001001	1	—
RTE	Delayed branch, SSR/SPC → SR/PC	0000000000101011	4	—
RTS	Delayed branching, PR → PC	0000000000001011	2	—
SETS	1 → S	0000000001011000	1	—
SETT	1 → T	0000000000011000	1	1
SLEEP	Sleep	0000000000011011	4*	—

Note: * This is number of states until a transition is made to the Sleep state.

A.2.2 n Format

Table A-28 Direct Register

Instruction		Operation	Code	Cycles	T Bit
CMP/PL	Rn	Rn > 0, 1 → T	0100nnnn00010101	1	Comparison result
CMP/PZ	Rn	Rn ≥ 0, 1 → T	0100nnnn00010001	1	Comparison result
DT	Rn	Rn – 1 → Rn, when Rn is 0, 1 → T. When Rn is nonzero, 0 → T	0100nnnn00010000	1	Comparison result
MOVT	Rn	T → Rn	0000nnnn00101001	1	—
ROTL	Rn	T ← Rn ← MSB	0100nnnn00000100	1	MSB
ROTR	Rn	LSB → Rn → T	0100nnnn00000101	1	LSB
ROTCL	Rn	T ← Rn ← T	0100nnnn00100100	1	MSB
ROTCR	Rn	T → Rn → T	0100nnnn00100101	1	LSB
SHAL	Rn	T ← Rn ← 0	0100nnnn00100000	1	MSB
SHAR	Rn	MSB → Rn → T	0100nnnn00100001	1	LSB
SHLL	Rn	T ← Rn ← 0	0100nnnn00000000	1	MSB
SHLR	Rn	0 → Rn → T	0100nnnn00000001	1	LSB
SHLL2	Rn	Rn << 2 → Rn	0100nnnn00001000	1	—
SHLR2	Rn	Rn >> 2 → Rn	0100nnnn00001001	1	—
SHLL8	Rn	Rn << 8 → Rn	0100nnnn00011000	1	—
SHLR8	Rn	Rn >> 8 → Rn	0100nnnn00011001	1	—
SHLL16	Rn	Rn << 16 → Rn	0100nnnn00101000	1	—
SHLR16	Rn	Rn >> 16 → Rn	0100nnnn00101001	1	—

Table A-29 Direct Register (Store with Control and System Registers)

Instruction		Operation	Code	Cycles	T Bit
STC	SR,Rn	SR → Rn	0000nnnn00000010	1	—
STC	GBR,Rn	GBR → Rn	0000nnnn00010010	1	—
STC	VBR,Rn	VBR → Rn	0000nnnn00100010	1	—
STC	SSR,Rn	SSR → Rn	0000nnnn00110010	1	—
STC	SPC,Rn	SPC → Rn	0000nnnn01000010	1	—
STC	MOD,Rn*2	MOD → Rn	0000nnnn01010010	1	—
STC	RE,Rn*2	RE → Rn	0000nnnn01110010	1	—
STC	RS,Rn*2	RS → Rn	0000nnnn01100010	1	—
STC	R0_BANK,Rn	R0_BANK→ Rn	0000nnnn10000010	1	—
STC	R1_BANK,Rn	R1_BANK→ Rn	0000nnnn10010010	1	—
STC	R2_BANK,Rn	R2_BANK→ Rn	0000nnnn10100010	1	—
STC	R3_BANK,Rn	R3_BANK→ Rn	0000nnnn10110010	1	—
STC	R4_BANK,Rn	R4_BANK→ Rn	0000nnnn11000010	1	—
STC	R5_BANK,Rn	R5_BANK→ Rn	0000nnnn11010010	1	—
STC	R6_BANK,Rn	R6_BANK→ Rn	0000nnnn11100010	1	—
STC	R7_BANK,Rn	R7_BANK→ Rn	0000nnnn11110010	1	—
STS	FPSCR,Rn*1	FPSCR→ Rn	0000nnnn01101010	1	—
STS	FPUL,Rn*1	FPUL→ Rn	0000nnnn01011010	1	—
STS	MACH,Rn	MACH → Rn	0000nnnn00001010	1	—
STS	MACL,Rn	MACL → Rn	0000nnnn00011010	1	—
STS	PR,Rn	PR → Rn	0000nnnn00101010	1	—
STS	DSR,Rn*2	DSR → Rn	0000nnnn01101010	1	—
STS	A0,Rn*2	A0 → Rn	0000nnnn01111010	1	—
STS	X0,Rn*2	X0→Rn	0000nnnn10001010	1	—
STS	X1,Rn*2	X1→Rn	0000nnnn10011010	1	—
STS	Y0,Rn*2	Y0→Rn	0000nnnn10101010	1	—
STS	Y1,Rn*2	Y1→Rn	0000nnnn10111010	1	—

Notes: 1. SH-3E instructions.
2. SH3-DSP instructions.

Table A-30 Indirect Register

Instruction		Operation	Code	Cycles	T Bit
TAS.B	@Rn	When (Rn) is 0, 1 → T, 1 → MSB of (Rn)	0100nnnn00011011	3/4*	Test results

Note: * Four cycles on the SH3-DSP.

Table A-31 Indirect Pre-Decrement Register

Instruction	Operation	Code	Cycles	T Bit
STC.L SR,@-Rn	Rn – 4 → Rn, SR → (Rn)	0100nnnn00000011	1/2*2	—
STC.L GBR,@-Rn	Rn – 4 → Rn, GBR → (Rn)	0100nnnn00010011	1/2*2	—
STC.L VBR,@-Rn	Rn – 4 → Rn, VBR → (Rn)	0100nnnn00100011	1/2*2	—
STC.L SSR,@–Rn	Rn–4 → Rn, SSR → (Rn)	0100nnnn00110011	1/2*2	—
STC.L SPC,@–Rn	Rn–4 → Rn, SPC → (Rn)	0100nnnn01000011	1/2*2	—
STC.L MOD,@-Rn*3	Rn – 4 → Rn, MOD → (Rn)	0100nnnn01010011	2	—
STC.L RE,@-Rn*3	Rn – 4 → Rn, RE → (Rn)	0100nnnn01110011	2	—
STC.L RS,@-Rn*3	Rn – 4 → Rn, RS → (Rn)	0100nnnn01100011	2	—
STC.L R0_BANK,@–Rn	Rn–4 → Rn, R0_BANK → (Rn)	0100nnnn10000011	2	—
STC.L R1_BANK,@–Rn	Rn–4 → Rn, R1_BANK → (Rn)	0100nnnn10010011	2	—
STC.L R2_BANK,@–Rn	Rn–4 → Rn, R2_BANK → (Rn)	0100nnnn10100011	2	—
STC.L R3_BANK,@–Rn	Rn–4 → Rn, R3_BANK → (Rn)	0100nnnn10110011	2	—
STC.L R4_BANK,@–Rn	Rn–4 → Rn, R4_BANK → (Rn)	0100nnnn11000011	2	—
STC.L R5_BANK,@–Rn	Rn–4 → Rn, R5_BANK → (Rn)	0100nnnn11010011	2	—
STC.L R6_BANK,@–Rn	Rn–4 → Rn, R6_BANK → (Rn)	0100nnnn11100011	2	—
STC.L R7_BANK,@–Rn	Rn–4 → Rn, R7_BANK → (Rn)	0100nnnn11110011	2	—
STS.L FPSCR,@-Rn*1	Rn–4 → Rn, FPSCR → @Rn	0100nnnn01100010	1	—
STS.L FPUL,@-Rn*1	Rn–4 → Rn, FPUL → @Rn	0100nnnn01010010	1	—
STS.L MACH,@–Rn	Rn – 4 → Rn, MACH → (Rn)	0100nnnn00000010	1	—
STS.L MACL,@–Rn	Rn – 4 → Rn, MACL → (Rn)	0100nnnn00010010	1	—
STS.L PR,@–Rn	Rn – 4 → Rn, PR → (Rn)	0100nnnn00100010	1	—
STS.L DSR,@–Rn*3	Rn – 4 → Rn, DSR → (Rn)	0100nnnn01100010	1	—
STS.L A0,@–Rn*3	Rn – 4 → Rn, A0 → (Rn)	0100nnnn01100010	1	—
STS.L X0,@-Rn*3	Rn–4→Rn,X0→(Rn)	0100nnnn10000010	1	—
STS.L X1,@-Rn*3	Rn–4→Rn,X1→(Rn)	0100nnnn10010010	1	—
STS.L Y0,@-Rn*3	Rn–4→Rn,Y0→(Rn)	0100nnnn10100010	1	—
STS.L Y1,@-Rn*3	Rn–4→Rn,Y1→(Rn)	0100nnnn10110010	1	—

Notes: 1. SH-3E instructions.
2. Two cycles on the SH3-DSP.
3. SH3-DSP instructions.

Table A-32 Floating Point Instructions (SH-3E Only)

Instruction		Operation	Code	Cycles	T Bit
FABS	FRn	|FRn| → FRn	1111nnnn01011101	1	—
FLDI0	FRn	H'00000000 → FRn	1111nnnn10001101	1	—
FLDI1	FRn	H'3F800000 → FRn	1111nnnn10011101	1	—
FLOAT	FPUL,FRn	(float)FPUL → FRn	1111nnnn00101101	1	—
FNEG	FRn	—FRn → FRn	1111nnnn01001101	1	—
FSQRT	FRn	√ FRn → FRn	1111nnnn01101101	13	—
FSTS	FPUL,FRn	FPUL → FRn	1111nnnn00001101	1	—

A.2.3 m Format

Table A-33 Direct Register (Load from Control and System Registers)

Instruction		Operation	Code	Cycles	T Bit
LDC	Rm,SR	Rm → SR	0100mmmm00001110	5	LSB
LDC	Rm,GBR	Rm → GBR	0100mmmm00011110	1/3*[2]	—
LDC	Rm,VBR	Rm → VBR	0100mmmm00101110	1/3*[2]	—
LDC	Rm,SSR	Rm → SSR	0100mmmm00111110	1/3*[2]	—
LDC	Rm,SPC	Rm → SPC	0100mmmm01001110	1/3*[2]	—
LDC	Rm,MOD*[3]	Rm → MOD	0100mmmm01011110	3	—
LDC	Rm,RE*[3]	Rm → RE	0100mmmm01111110	3	—
LDC	Rm,RS*[3]	Rm → RS	0100mmmm01101110	3	—
LDC	Rm,R0_BANK	Rm → R0_BANK	0100mmmm10001110	1/3*[2]	—
LDC	Rm,R1_BANK	Rm → R1_BANK	0100mmmm10011110	1/3*[2]	—
LDC	Rm,R2_BANK	Rm → R2_BANK	0100mmmm10101110	1/3*[2]	—
LDC	Rm,R3_BANK	Rm → R3_BANK	0100mmmm10111110	1/3*[2]	—
LDC	Rm,R4_BANK	Rm → R4_BANK	0100mmmm11001110	1/3*[2]	—
LDC	Rm,R5_BANK	Rm → R5_BANK	0100mmmm11011110	1/3*[2]	—
LDC	Rm,R6_BANK	Rm → R6_BANK	0100mmmm11101110	1/3*[2]	—
LDC	Rm,R7_BANK	Rm → R7_BANK	0100mmmm11111110	1/3*[2]	—
LDS	Rm,FPSCR*[1]	Rm → FPSCR	0100nnnn01101010	1	—
LDS	Rm,FPUL*[1]	Rm → FPUL	0100nnnn01011010	1	—
LDS	Rm,MACH	Rm → MACH	0100mmmm00001010	1	—
LDS	Rm,MACL	Rm → MACL	0100mmmm00011010	1	—

Table A-33 Direct Register (Load from Control and System Registers) (cont)

Instruction	Operation	Code	Cycles	T Bit
LDS Rm,PR	Rm → PR	0100mmmm00101010	1	—
LDS Rm,DSR*3	Rm → DSR	0100mmmm01101010	1	—
LDS Rm,A0*3	Rm → A0	0100mmmm01111010	1	—
LDS Rm,X0*3	Rm → X0	0100mmmm10001010	1	—
LDS Rm,X1*3	Rm → X1	0100mmmm10011010	1	—
LDS Rm,Y0*3	Rm → Y0	0100mmmm10101010	1	—
LDS Rm,Y1*3	Rm → Y1	0100mmmm10111010	1	—
SETRC #imm*3	imm→RC(SR[23:16]), zeros→SR[27:24]	10000010iiiiiiii	3	—

Notes: 1. SH-3E instructions.
2. Three cycles on the SH3-DSP.
3. SH3-DSP instructions.

Table A-34 PC Relative Addressing with Rm

Instruction	Operation	Code	Cycles	T Bit
BRAF Rm	Delayed branch, Rm + PC → PC	0000mmmm00100011	2	—
BSRF Rm	Delayed branch, PC → PR, Rm + PC → PC	0000mmmm00000011	2	—

Table A-35 Indirect Register

Instruction	Operation	Code	Cycles	T Bit
JMP @Rm	Delayed branch, Rm → PC	0100mmmm00101011	2	—
JSR @Rm	Delayed branch, PC → PR, Rm → PC	0100mmmm00001011	2	—

Table A-36 Indirect Post-Increment Register

Instruction	Operation	Code	Cycles	T Bit
LDC.L @Rm+,SR	(Rm) → SR, Rm + 4 → Rm	0100mmmm00000111	7	LSB
LDC.L @Rm+,GBR	(Rm) → GBR, Rm + 4 → Rm	0100mmmm00010111	1/5*2	—
LDC.L @Rm+,VBR	(Rm) → VBR, Rm + 4 → Rm	0100mmmm00100111	1/5*2	—
LDC.L @Rm+,SSR	(Rm) → SSR, Rm + 4 → Rm	0100mmmm00110111	1/5*2	—
LDC.L @Rm+,SPC	(Rm) → SPC, Rm + 4 → Rm	0100mmmm01000111	1/5*2	—

Table A-36 Indirect Post-Increment Register (cont)

Instruction	Operation	Code	Cycles	T Bit
LDC.L @Rm+,MOD*[3]	(Rm) → MOD, Rm + 4 → Rm	0100mmmm01010111	5	—
LDC.L @Rm+,RE*[3]	(Rm) → RE, Rm + 4 → Rm	0100mmmm01110111	5	—
LDC.L @Rm+,RS*[3]	(Rm) → RS, Rm + 4 → Rm	0100mmmm01100111	5	—
LDC.L @Rm+,R0_BANK	(Rm) → R0_BANK, Rm + 4 → Rm	0100mmmm10000111	1/5*[2]	—
LDC.L @Rm+,R1_BANK	(Rm) → R1_BANK, Rm + 4 → Rm	0100mmmm10010111	1/5*[2]	—
LDC.L @Rm+,R2_BANK	(Rm) → R2_BANK, Rm + 4 → Rm	0100mmmm10100111	1/5*[2]	—
LDC.L @Rm+,R3_BANK	(Rm) → R3_BANK, Rm + 4 → Rm	0100mmmm10110111	1/5*[2]	—
LDC.L @Rm+,R4_BANK	(Rm) → R4_BANK, Rm + 4 → Rm	0100mmmm11000111	1/5*[2]	—
LDC.L @Rm+,R5_BANK	(Rm) → R5_BANK, Rm + 4 → Rm	0100mmmm11010111	1/5*[2]	—
LDC.L @Rm+,R6_BANK	(Rm) → R6_BANK, Rm + 4 → Rm	0100mmmm11100111	1/5*[2]	—
LDC.L @Rm+,R7_BANK	(Rm) → R7_BANK, Rm + 4 → Rm	0100mmmm11110111	1/5*[2]	—
LDS.L @Rm+,FPSCR*[1]	@Rm → FPSCR, Rm + 4 → Rm	0100nnnn01100110	1	—
LDS.L @Rm+,FPUL*[1]	@Rm → FPUL, Rm + 4 → Rm	0100nnnn01010110	1	—
LDS.L @Rm+,MACH	(Rm) → MACH, Rm + 4 → Rm	0100mmmm00000110	1	—
LDS.L @Rm+,MACL	(Rm) → MACL, Rm + 4 → Rm	0100mmmm00010110	1	—
LDS.L @Rm+,PR	(Rm) → PR, Rm + 4 → Rm	0100mmmm00100110	1	—
LDS.L @Rm+,DSR*[3]	(Rm) → DSR, Rm + 4 → Rm	0100mmmm01100110	1	—
LDS.L @Rm+,A0*[3]	(Rm) → A0, Rm + 4 → Rm	0100mmmm01110110	1	—
LDS.L@Rm+,X0*[3]	(Rm) → X0,Rm+4 → Rm	0100nnnn10000110	1	—
LDS.L@Rm+,X1*[3]	(Rm) → X1,Rm+4 → Rm	0100nnnn10010110	1	—
LDS.L@Rm+,Y0*[3]	(Rm) → Y0,Rm+4 → Rm	0100nnnn10100110	1	—
LDS.L@Rm+,Y1*[3]	(Rm) → Y1,Rm+4 → Rm	0100nnnn10110110	1	—

Notes: 1. SH-3E instructions.
2. Five cycles on the SH3-DSP.
3. The instruction of SH3-DSP.

Table A-37 Floating Point Instructions (SH-3E Only)

Instruction		Operation	Code	Cycles	T Bit
FLDS	FRm,FPUL	FRm → FPUL	1111nnnn00011101	1	—
FTRC	FRm,FPUL	(long)FRm → FPUL	1111nnnn00111101	1	—

A.2.4 nm Format

Table A-38 Direct Register

Instruction		Operation	Code	Cycles	T Bit
ADD	Rm,Rn	Rm + Rn → Rn	0011nnnnmmmm1100	1	—
ADDC	Rm,Rn	Rn + Rm + T → Rn, carry → T	0011nnnnmmmm1110	1	Carry
ADDV	Rm,Rn	Rn + Rm → Rn, overflow → T	0011nnnnmmmm1111	1	Overflow
AND	Rm,Rn	Rn & Rm → Rn	0010nnnnmmmm1001	1	—
CMP/EQ	Rm,Rn	When Rn = Rm, 1 → T	0011nnnnmmmm0000	1	Comparison result
CMP/HS	Rm,Rn	When unsigned and Rn ≥ Rm, 1 → T	0011nnnnmmmm0010	1	Comparison result
CMP/GE	Rm,Rn	When signed and Rn ≥ Rm, 1 → T	0011nnnnmmmm0011	1	Comparison result
CMP/HI	Rm,Rn	When unsigned and Rn > Rm, 1 → T	0011nnnnmmmm0110	1	Comparison result
CMP/GT	Rm,Rn	When signed and Rn > Rm, 1 → T	0011nnnnmmmm0111	1	Comparison result
CMP/STR	Rm,Rn	When a byte in Rn equals a byte in Rm, 1 → T	0010nnnnmmmm1100	1	Comparison result
DIV1	Rm,Rn	1 step division (Rn ÷ Rm)	0011nnnnmmmm0100	1	Calculation result
DIV0S	Rm,Rn	MSB of Rn → Q, MSB of Rm → M, M ^ Q → T	0010nnnnmmmm0111	1	Calculation result
DMULS.L	Rm,Rn	Signed operation of Rn × Rm → MACH, MACL	0011nnnnmmmm1101	2 (to 5)*	—
DMULU.L	Rm,Rn	Unsigned operation of Rn × Rm → MACH, MACL	0011nnnnmmmm0101	2 (to 5)*	—
EXTS.B	Rm,Rn	Sign-extend Rm from byte → Rn	0110nnnnmmmm1110	1	—

Table A-38 Direct Register (cont)

Instruction		Operation	Code	Cycles	T Bit
EXTS.W	Rm,Rn	Sign-extend Rm from word → Rn	0110nnnnmmmm1111	1	—
EXTU.B	Rm,Rn	Zero-extend Rm from byte → Rn	0110nnnnmmmm1100	1	—
EXTU.W	Rm,Rn	Zero-extend Rm from word → Rn	0110nnnnmmmm1101	1	—
MOV	Rm,Rn	Rm → Rn	0110nnnnmmmm0011	1	—
MUL.L	Rm,Rn	Rn × Rm → MAC	0000nnnnmmmm0111	2 (to 5)*	—
MULS	Rm,Rn	With sign, Rn × Rm → MAC	0010nnnnmmmm1111	1 (to 3)*	—
MULU	Rm,Rn	Unsigned, Rn × Rm → MAC	0010nnnnmmmm1110	1 (to 3)*	—
NEG	Rm,Rn	0 – Rm → Rn	0110nnnnmmmm1011	1	—
NEGC	Rm,Rn	0 – Rm – T → Rn, Borrow → T	0110nnnnmmmm1010	1	Borrow
NOT	Rm,Rn	~Rm → Rn	0110nnnnmmmm0111	1	—
OR	Rm,Rn	Rn | Rm → Rn	0010nnnnmmmm1011	1	—
SHAD	Rm,Rn	Rn ≥ 0; Rn << Rm → Rn Rn < 0; Rn >> Rm → [MSB→Rn]	0100nnnnmmmm1100	1	—
SHLD	Rm,Rn	Rn ≥ 0; Rn << Rm → Rn Rn < 0; Rn >> Rm → [0→Rn]	0100nnnnmmmm1101	1	—
SUB	Rm,Rn	Rn – Rm → Rn	0011nnnnmmmm1000	1	—
SUBC	Rm,Rn	Rn – Rm – T → Rn, Borrow → T	0011nnnnmmmm1010	1	Borrow
SUBV	Rm,Rn	Rn – Rm → Rn, Underflow → T	0011nnnnmmmm1011	1	Under-flow
SWAP.B	Rm,Rn	Rm → Swap upper and lower halves of lower 2 bytes → Rn	0110nnnnmmmm1000	1	—
SWAP.W	Rm,Rn	Rm → Swap upper and lower word → Rn	0110nnnnmmmm1001	1	—
TST	Rm,Rn	Rn & Rm, when result is 0, 1 → T	0010nnnnmmmm1000	1	Test results
XOR	Rm,Rn	Rn ^ Rm → Rn	0010nnnnmmmm1010	1	—
XTRCT	Rm,Rn	Rm: Center 32 bits of Rn → Rn	0010nnnnmmmm1101	1	—

Note: * Normal minimum number of execution states (the number in parentheses is the number of states when there is contention with preceding/following instructions).

Table A-39 Indirect Register

Instruction	Operation	Code	Cycles	T Bit
MOV.B Rm,@Rn	Rm → (Rn)	0010nnnnmmmm0000	1	—
MOV.W Rm,@Rn	Rm → (Rn)	0010nnnnmmmm0001	1	—
MOV.L Rm,@Rn	Rm → (Rn)	0010nnnnmmmm0010	1	—
MOV.B @Rm,Rn	(Rm) → sign extension → Rn	0110nnnnmmmm0000	1	—
MOV.W @Rm,Rn	(Rm) → sign extension → Rn	0110nnnnmmmm0001	1	—
MOV.L @Rm,Rn	(Rm) → Rn	0110nnnnmmmm0010	1	—

Table A-40 Indirect Post-Increment Register (Multiply/Accumulate Operation)

Instruction	Operation	Code	Cycles	T Bit
MAC.L @Rm+,@Rn+	Signed operation of (Rn) × (Rm) + MAC → MAC, Rn + 4 → Rn, Rm + 4 → Rm	0000nnnnmmmm1111	2 (to 5)*	—
MAC.W @Rm+,@Rn+	Signed operation of (Rn) × (Rm) + MAC → MAC, Rn + 2 → Rn, Rm + 2 → Rm	0100nnnnmmmm1111	2 (to 5)*	—

Note: * Normal minimum number of execution states (the number in parentheses is the number of states when there is contention with preceding/following instructions).

Table A-41 Indirect Post-Increment Register

Instruction	Operation	Code	Cycles	T Bit
MOV.B @Rm+,Rn	(Rm) → sign extension → Rn, Rm + 1 → Rm	0110nnnnmmmm0100	1	—
MOV.W @Rm+,Rn	(Rm) → sign extension → Rn, Rm + 2 → Rm	0110nnnnmmmm0101	1	—
MOV.L @Rm+,Rn	(Rm) → Rn, Rm + 4 → Rm	0110nnnnmmmm0110	1	—

Table A-42 Indirect Pre-Decrement Register

Instruction	Operation	Code	Cycles	T Bit
MOV.B Rm,@–Rn	Rn – 1 → Rn, Rm → (Rn)	0010nnnnmmmm0100	1	—
MOV.W Rm,@–Rn	Rn – 2 → Rn, Rm → (Rn)	0010nnnnmmmm0101	1	—
MOV.L Rm,@–Rn	Rn – 4 → Rn, Rm → (Rn)	0010nnnnmmmm0110	1	—

Table A-43 Indirect Indexed Register

Instruction	Operation	Code	Cycles	T Bit
MOV.B Rm,@(R0,Rn)	Rm → (R0 + Rn)	0000nnnnmmmm0100	1	—
MOV.W Rm,@(R0,Rn)	Rm → (R0 + Rn)	0000nnnnmmmm0101	1	—
MOV.L Rm,@(R0,Rn)	Rm → (R0 + Rn)	0000nnnnmmmm0110	1	—
MOV.B @(R0,Rm),Rn	(R0 + Rm) → sign extension → Rn	0000nnnnmmmm1100	1	—
MOV.W @(R0,Rm),Rn	(R0 + Rm) → sign extension → Rn	0000nnnnmmmm1101	1	—
MOV.L @(R0,Rm),Rn	(R0 + Rm) → Rn	0000nnnnmmmm1110	1	—

Table A-44 Floating Point Instructions (SH-3E Only)

Instruction		Operation	Code	Cycles	T Bit
FADD	FRm,FRn	FRn+FRm → FRn	1111nnnnmmmm0000	1	—
FCMP/EQ	FRm,FRn	(FRn=FRm)? 1:0 → T	1111nnnnmmmm0100	1	Comparison result
FCMP/GT	FRm,FRn	(FRn>FRm)? 1:0 → T	1111nnnnmmmm0101	1	Comparison result
FDIV	FRm,FRn	FRn/FRm → FRn	1111nnnnmmmm0011	13	—
FMAC	FR0,FRm,FRn	FR0×FRm+FRn → FRn	1111nnnnmmmm1110	1	—
FMOV	FRm,FRn	FRm → FRn	1111nnnnmmmm1100	1	—
FMOV.S	@(R0,Rm),FRn	(R0+Rm) → FRn	1111nnnnmmmm0110	1	—
FMOV.S	@Rm+,FRn	(Rm) → FRn,Rm+4 → Rm	1111nnnnmmmm1001	1	—
FMOV.S	@Rm,FRn	(Rm) → FRn	1111nnnnmmmm1000	1	—
FMOV.S	Frm,@(R0,Rn)	(FRm) → (R0+Rn)	1111nnnnmmmm0111	1	—
FMOV.S	FRm,@-Rn	Rn-4 → Rn, FRm → (Rn)	1111nnnnmmmm1011	1	—
FMOV.S	FRm,@Rn	FRm → (Rn)	1111nnnnmmmm1010	1	—
FMUL	FRm,FRn	FRn × FRm → FRn	1111nnnnmmmm0010	1	—
FSUB	FRm,FRn	FRn-FRm → FRn	1111nnnnmmmm0001	1	—

A.2.5 md Format

Table A-45 md Format

Instruction		Operation	Code	Cycles	T Bit
MOV.B	@(disp,Rm),R0	(disp + Rm) → sign extension → R0	10000100mmmmdddd	1	—
MOV.W	@(disp,Rm),R0	(disp × 2+ Rm) → sign extension → R0	10000101mmmmdddd	1	—

A.2.6 nd4 Format

Table A-46 nd4 Format

Instruction	Operation	Code	Cycles	T Bit
MOV.B R0,@(disp,Rn)	R0 → (disp + Rn)	10000000nnnndddd	1	—
MOV.W R0,@(disp,Rn)	R0 → (disp × 2 + Rn)	10000001nnnndddd	1	—

A.2.7 nmd Format

Table A-47 nmd Format

Instruction	Operation	Code	Cycles	T Bit
MOV.L Rm,@(disp,Rn)	Rm → (disp × 4 + Rn)	0001nnnnmmmmdddd	1	—
MOV.L @(disp,Rm),Rn	(disp × 4 + Rm) → Rn	0101nnnnmmmmdddd	1	—

A.2.8 d Format

Table A-48 Indirect GBR with Displacement

Instruction	Operation	Code	Cycles	T Bit
MOV.B R0,@(disp,GBR)	R0 → (disp + GBR)	11000000dddddddd	1	—
MOV.W R0,@(disp,GBR)	R0 → (disp × 2 + GBR)	11000001dddddddd	1	—
MOV.L R0,@(disp,GBR)	R0 → (disp × 4 + GBR)	11000010dddddddd	1	—
MOV.B @(disp,GBR),R0	(disp + GBR) → sign extension → R0	11000100dddddddd	1	—
MOV.W @(disp,GBR),R0	(disp × 2 + GBR) → sign extension → R0	11000101dddddddd	1	—
MOV.L @(disp,GBR),R0	(disp × 4 + GBR) → R0	11000110dddddddd	1	—

Table A-49 PC Relative with Displacement

Instruction	Operation	Code	Cycles	T Bit
MOVA @(disp,PC),R0	disp × 4 + PC → R0	11000111dddddddd	1	—
LDRS @(disp,pc)*	disp × 2+PC→RS	10001100dddddddd	3	—
LDRE @(disp,pc)*	disp × 2+PC→RE	10001110dddddddd	3	—

Note: * SH3-DSP instructions.

Table A-50 PC Relative

Instruction	Operation	Code	Cycles	T Bit
BF label	When T = 0, disp × 2 + PC → PC; when T = 1, nop	10001011dddddddd	3/1	—
BF/S label	If T = 0, disp × 2 + PC → PC; if T = 1, nop	10001111dddddddd	2/1*	—
BT label	When T = 1, disp × 2 + PC → PC; when T = 0, nop	10001001dddddddd	3/1	—
BT/S label	If T = 1, disp × 2 + PC → PC; if T = 0, nop	10001101dddddddd	2/1*	

Note: * One state when it does not branch.

A.2.9 d12 Format

Table A-51 d12 Format

Instruction	Operation	Code	Cycles	T Bit
BRA label	Delayed branching, disp × 2 + PC → PC	1010dddddddddddd	2	—
BSR label	Delayed branching, PC → PR, disp × 2 + PC → PC	1011dddddddddddd	2	—

A.2.10 nd8 Format

Table A-52 nd8 Format

Instruction	Operation	Code	Cycles	T Bit
MOV.W @(disp,PC),Rn	(disp × 2 + PC) → sign extension → Rn	1001nnnndddddddd	1	—
MOV.L @(disp,PC),Rn	(disp × 4 + PC) → Rn	1101nnnndddddddd	1	—

A.2.11 i Format

Table A-53 Indirect Indexed GBR

Instruction	Operation	Code	Cycles	T Bit
AND.B #imm,@(R0,GBR)	(R0 + GBR) & imm → (R0 + GBR)	11001101iiiiiiii	3	—
OR.B #imm,@(R0,GBR)	(R0 + GBR) | imm → (R0 + GBR)	11001111iiiiiiii	3	—
TST.B #imm,@(R0,GBR)	(R0 + GBR) & imm, when result is 0, 1 → T	11001100iiiiiiii	3	Test results
XOR.B #imm,@(R0,GBR)	(R0 + GBR) ^ imm → (R0 + GBR)	11001110iiiiiiii	3	—

Table A-54 Immediate (Arithmetic Logical Operation with Direct Register)

Instruction		Operation	Code	Cycles	T Bit
AND	#imm,R0	R0 & imm → R0	11001001iiiiiiii	1	—
CMP/EQ	#imm,R0	When R0 = imm, 1 → T	10001000iiiiiiii	1	Comparison results
OR	#imm,R0	R0 | imm → R0	11001011iiiiiiii	1	—
TST	#imm,R0	R0 & imm, when result is 0, 1 → T	11001000iiiiiiii	1	Test results
XOR	#imm,R0	R0 ^ imm → R0	11001010iiiiiiii	1	—

Table A-55 Immediate (Specify Exception Processing Vector)

Instruction		Operation	Code	Cycles	T Bit
TRAPA	#imm	imm → TRA, PC → SPC, SR → SSR, 1 → SR.MD/BL/RB, 0x160 → EXPEVT, VBR + H'00000100 → PC	11000011iiiiiiii	6/8*	—

Note: * Eight cycles on the SH3-DSP.

Table A-56 Load to Control Register (SH3-DSP Only)

Instruction	Operation	Code	Cycles	T Bit
SETRC #imm	imm→RC(SR[23:16]), zeros→SR[27:24]	10000010iiiiiiii	3	—

A.2.12 ni Format

Table A-57 ni Format

Instruction		Operation	Code	Cycles	T Bit
ADD	#imm,Rn	Rn + #imm → Rn	0111nnnniiiiiiii	1	—
MOV	#imm,Rn	#imm → sign extension → Rn	1110nnnniiiiiiii	1	—

A.3 Operation Code Map

Table A-58 Operation Code Map

Instruction Code				Fx: 0000	Fx: 0001	Fx: 0010	Fx: 0011–1111
MSB			LSB	MD: 00	MD: 01	MD: 10	MD: 11
0000	Rn	Fx	0000				
0000	Rn	Fx	0001				
0000	Rn	00MD	0010	STC SR,Rn	STC GBR,Rn	STC VBR,Rn	STC SSR,Rn
0000	Rn	01MD	0010	STC SPC,Rn	STC MOD,Rn*2	STC RS,Rn*2	STC RE,Rn*2
0000	Rn	10MD	0010	STC R0_BANK,Rn	STC R1_BANK,Rn	STC R2_BANK,Rn	STC R3_BANK,Rn
0000	Rn	11MD	0010	STC R4_BANK,Rn	STC R5_BANK,Rn	STC R6_BANK,Rn	STC R7_BANK,Rn
0000	Rn	00MD	0011	BSRF Rm		BRAF Rm	
0000	Rm	10MD	0011	PREF @Rm			
0000	Rn	Rm	01MD	MOV.B Rm, @(R0,Rn)	MOV.W Rm, @(R0,Rn)	MOV.L Rm, @(R0,Rn)	MUL.L Rm,Rn
0000	0000	00MD	1000	CLRT	SETT	CLRMAC	LDTLB
0000	0000	01MD	1000	CLRS	SETS		
0000	0000	Fx	1001	NOP	DIVOU		
0000	0000	Fx	1010				
0000	0000	Fx	1011	RTS	SLEEP	RTE	
0000	Rn	Fx	1000				
0000	Rn	Fx	1001			MOVT Rn	
0000	Rn	00MD	1010	STS MACH,Rn	STS MACL,Rn	STS PR,Rn	
0000	Rn	01MD	1010		STS FPUL,Rn*1	STS FPSCR,Rn*1 STS DSR,Rn*2	STS A0,Rn*2
0000	Rn	10MD	1010	STS X0,Rn*2	STS X1,Rn*2	STS Y0,Rn*2	STS Y1,Rn*2
0000	Rn	Fx	1011				
0000	Rn	Rm	11MD	MOV.B @(R0,Rm),Rn	MOV.W @(R0,Rm),Rn	MOV.L @(R0,Rm),Rn	MAC.L @Rm+,@Rn+
0001	Rn	Rm	disp	MOV.L Rm,@(disp:4,Rn)			
0010	Rn	Rm	00MD	MOV.B Rm,@Rn	MOV.W Rm,@Rn	MOV.L Rm,@Rn	
0010	Rn	Rm	01MD	MOV.B Rm,@-Rn	MOV.W Rm,@–Rn	MOV.L Rm,@–Rn	DIV0S Rm,Rn
0010	Rn	Rm	10MD	TST Rm,Rn	AND Rm,Rn	XOR Rm,Rn	OR Rm,Rn
0010	Rn	Rm	11MD	CMP/STR Rm,Rn	XTRCT Rm,Rn	MULU.W Rm,Rn	MULS.W Rm,Rn
0011	Rn	Rm	00MD	CMP/EQ Rm,Rn		CMP/HS Rm,Rn	CMP/GE Rm,Rn

Table A-58 Operation Code Map (cont)

Instruction Code				Fx: 0000	Fx: 0001	Fx: 0010	Fx: 0011–1111
MSB			LSB	MD: 00	MD: 01	MD: 10	MD: 11
0011	Rn	Rm	01MD	DIV1 Rm,Rn	DMULU.L Rm,Rn	CMP/HI Rm,Rn	CMP/GT Rm,Rn
0011	Rn	Rm	10MD	SUB Rm,Rn		SUBC Rm,Rn	SUBV Rm,Rn
0011	Rn	Rm	11MD	ADD Rm,Rn	DMULU.L Rm,Rn	ADDC Rm,Rn	ADDV Rm,Rn
0100	Rn	Fx	0000	SHLL Rn	DT Rn	SHAL Rn	
0100	Rn	Fx	0001	SHLR Rn	CMP/PZ Rn	SHAR Rn	
0100	Rn	00MD	0010	STS.L MACH, @–Rn	STS.L MACL, @–Rn	STS.L PR, @–Rn	
0100	Rn	01MD	0010		STS.L FPUL, @–Rn*1	STS.L DSR, @–Rn*2 STS.L FPSCR, @–Rn*1	STS.L A0, @–Rn*2
0100	Rn	10MD	0010	STS.L X0, @–Rn*2	STS.L X1, @–Rn*2	STS.L Y0, @–Rn*2	STS.L Y1, @–Rn*2
0100	Rn	00MD	0011	STC.L SR,@–Rn	STC.L GBR,@–Rn	STC.L VBR,@–Rn	STC.L SSR,A-Rn
0100	Rn	01MD	0011	STC.L SPC,@-Rn	STS.L MOD, @–Rn*2	STS.L RS, @–Rn*2	STS.L RE, @–Rn*2
0100	Rn	10MD	0011	STC.L R0_BANK,@-Rn	STC.L R1_BANK,@-Rn	STC.L R2_BANK,@-Rn	STC.L R3_BANK,@-Rn
0100	Rn	11MD	0011	STC.L R4_BANK,@-Rn	STC.L R5_BANK,@-Rn	STC.L R6_BANK,@-Rn	STC.L R7_BANK,@-Rn
0100	Rm/ Rn	Fx	0100	ROTL Rn	SETRC Rm	ROTCL Rn	
0100	Rn	Fx	0101	ROTR Rn	CMP/PL Rn	ROTCR Rn	
0100	Rm	00MD	0110	LDS.L @Rm+,MACH	LDS.L @Rm+,MACL	LDS.L @Rm+,PR	
0100	Rm	01MD	0110		LDS.L @Rm+,FPUL*1	LDS.L @Rm+,DSR*2 LDS.L @Rm+,FPSCR*1	LDS.L @Rm+,A0*2
0100	Rm	10MD	0110	LDS.L @Rm+,X0*2	LDS.L @Rm+,X1*2	LDS.L @Rm+,Y0*2	LDS.L @Rm+,Y1*2
0100	Rm	00MD	0111	LDC.L @Rm+,SR	LDC.L @Rm+,GBR	LDC.L @Rm+,VBR	LDC.L @Rm+,SSR

Table A-58 Operation Code Map (cont)

Instruction Code				Fx: 0000	Fx: 0001	Fx: 0010	Fx: 0011–1111
MSB			LSB	MD: 00	MD: 01	MD: 10	MD: 11
0100	Rm	01MD	0111	LDC.L @Rm+,SPC	LDC.L @Rm+,MOD*[2]	LDC.L @Rm+,RS*[2]	LDC.L @Rm+,RE*[2]
0100	Rm	10MD	0111	LDC.L @Rm+,R0_BANK	LDC.L @Rm+,R1_BANK	LDC.L @Rm+,R2_BANK	LDC.L @Rm+,R3_BANK
0100	Rm	11MD	0111	LDC.L @Rm+,R4_BANK	LDC.L @Rm+,R5_BANK	LDC.L @Rm+,R6_BANK	LDC.L @Rm+,R7_BANK
0100	Rn	Fx	1000	SHLL2 Rn	SHLL8 Rn	SHLL16 Rn	
0100	Rn	Fx	1001	SHLR2 Rn	SHLR8 Rn	SHLR16 Rn	
0100	Rm	00MD	1010	LDS Rm,MACH	LDS Rm,MACL	LDS Rm,PR	
0100	Rm	01MD	1010		LDS Rm,FPUL*[1]	LDS Rm,DSR*[2] LDS Rm,FPSCR*[1]	LDS Rm,A0*[2]
0100	Rm	10MD	1010	LDS Rm,X0*[2]	LDS Rm,X1*[2]	LDS Rm,Y0*[2]	LDS Rm,Y1*[2]
0100	Rn	Fx	1011	JSR @Rm	TAS.B @Rn	JMP @Rm	
0100	Rm	Rm	1100	SHAD Rm,Rn			
0100	Rm	Rm	1101	SHLD Rm,Rn			
0100	Rm	00MD	1110	LDC Rm,Sr	LDC Rm,GBR	LDC Rm,VBR	LDC Rm,SSR
0100	Rm	01MD	1110	LDC Rm,SPC	LDC Rm,MOD*[2]	LDC Rm,RS*[2]	LDC Rm,RE*[2]
0100	Rm	10MD	1110	LDC Rm,R0_BANK	LDC Rm,R1_BANK	LDC Rm,R2_BANK	LDC Rm,R3_BANK
0100	Rm	11MD	1110	LDC Rm,R4_BANK	LDC Rm,R5_BANK	LDC Rm,R6_BANK	LDC Rm,R7_BANK
0100	Rn	Rm	1111	MAC.W @Rm+,@Rn+			
0101	Rn	Rm	disp	MOV.L @(disp:4,Rm),Rn			
0110	Rn	Rm	00MD	MOV.B @Rm,Rn	MOV.W @Rm,Rn	MOV.L @Rm,Rn	MOV Rm,Rn
0110	Rn	Rm	01MD	MOV.B @Rm+,Rn	MOV.W @Rm+,Rn	MOV.L @Rm+,Rn	NOT Rm,Rn
0110	Rn	Rm	10MD	SWAP.B Rm,Rn	SWAP.W Rm,Rn	NEGC Rm,Rn	NEG Rm,Rn
0110	Rn	Rm	11MD	EXTU.B Rm,Rn	EXTU.W Rm,Rn	EXTS.B Rm,Rn	EXTS.W Rm,Rn
0111	Rn	imm		ADD #imm:8,Rn			
1000	00MD	Rn	disp	MOV.B R0,@(disp:4,Rn)	MOV.W R0,@(disp:4,Rn)	SETRC #imm*[2]	
		imm					
1000	01MD	Rm	disp	MOV.B @(disp:4,Rm),R0	MOV.W @(disp:4,Rm),R0		

Table A-58 Operation Code Map (cont)

Instruction Code				Fx: 0000	Fx: 0001	Fx: 0010	Fx: 0011–1111
MSB			LSB	MD: 00	MD: 01	MD: 10	MD: 11
1000	10MD	imm/disp		CMP/EQ #imm:8,R0	BT disp:8		BF label:8
1000	10MD	imm/disp		LDRS @(disp,PC)*2	BT/S disp:8	LDRE @(disp,PC)*2	BF/S label:8
1001	Rn	disp		MOV.W @(disp:8,PC),Rn			
1010	disp			BRA label:12			
1011	disp			BSR label:12			
1100	00MD	imm/disp		MOV.B R0, @(disp:8, GBR)	MOV.W R0, @(disp:8, GBR)	MOV.L R0, @(disp:8, GBR)	TRAPA #imm:8
1100	01MD	disp		MOV.B @(disp:8, GBR),R0	MOV.W @(disp:8, GBR),R0	MOV.L @(disp:8, GBR),R0	MOVA @(disp:8, PC),R0
1100	10MD	imm		TST #imm:8,R0	AND #imm:8,R0	XOR #imm:8,R0	OR #imm:8,R0
1100	11MD	imm		TST.B #imm:8, @(R0,GBR)	AND.B #imm:8, @(R0,GBR)	XOR.B #imm:8, @(R0,GBR)	OR.B #imm:8, @(R0,GBR)
1101	Rn	disp		MOV.L @(disp:8,PC),Rn			
1110	Rn	imm		MOV #imm:8,Rn			
1111	Rn	Rm	00MD	FADD FRm,FRn*1	FSUB FRm,FRn*1	FMUL FRm,FRn*1	FDIV FRm,FRn*1
1111	Rn	Rm	01MD	FCMP/EQ FRm,FRn*1	FCMP/GT FRm,FRn*1	FMOV.S @(R0,Rm),FRm*1	FMOV.S FRm,@(R0,Rn) *1
1111	Rn	Rm	10MD	FMOV.S @Rm,FRn*1	FMOV.S @Rm+,FRm*1	FMOV.S FRm,@Rn*1	FMOV.S FRm,@-Rn*1
1111	Rn	Rm	1100	FMOV FRm,FRn*1			
1111	Rn	00MD	1101	FSTS FPUL,FRn*1	FLDS FRn,FPUL*1	FLOAT FPUL,FRn*1	FTRC FRn, FPUL*1
1111	Rn	01MD	1101	FNEG FRn*1	FABS FRn*1	FSQRT FRn*1	
1111	Rn	10MD	1101	FLDI0 FRn*1	FLDI1 FRn*1		
1111	Rn	Rm	1110	FMAC FR0,FRm,FRn*1			
1111	00**	****		(MOVX.W, MOVY.W, DPS double data transfer instructions) (SH3-DSP)			

Table A-58 Operation Code Map (cont)

Instruction Code			Fx: 0000	Fx: 0001	Fx: 0010	Fx: 0011–1111
MSB		**LSB**	**MD: 00**	**MD: 01**	**MD: 10**	**MD: 11**
1111	01**	****	(MOVS.W, MOVS.L, DPS single data transfer instructions) (SH3-DSP)			
1111	10**	****	(DPS parallel processing instructions, field A: MOVX.W, MOVY.W, DPS double data transfer instructions, field B: PSHL to PLDS, DPS operation instructions) (SH3-DSP)			
1111	11**	****				

Notes: 1. Floating point arithmetic calculation instruction or CPU instruction related to the FPU. These instruction are available only on the SH-3E

2. CPU instructions to provide support for DSP functions. These instructions can only be used with the SH3-DSP.

Table A-59 Operation Code Map for DSP Operation Instructions (B Field)

Instruction Code				Fx: 0000	Fx: 0001	Fx: 0010	Fx: 0011–1111
MSB			**LSB**	**cc:00**	**cc:01***	**cc:10 (DCT)**	**cc:11 (DCF)**
0000	imm		zzzz	PSHL #imm, Dz			
0000	1***	****	****				
0001	imm		zzzz	PSHA #imm, Dz			
0001	1***	****	****				
001*	****	****	****				
0100	eeff	xxyy	gguu	PMULS Se, Sf, Dg			
0101	****	****	****				
0110	eeff	xxyy	gguu	PSUB Sx, Sy, Du PMULS Se, Sf, Dg			
0111	eeff	xxyy	gguu	PADD Sx, Sy, Du PMULS Se, Sf, Dg			
1000	00cc	xxyy	zzzz		[if cc] PSHL Sx, Sy, Dz		
1000	01cc	xxyy	zzzz	PCMP Sx, Sy			
1000	10cc	xxyy	zzzz	PABS Sx, Dz	[if cc] PDEC Sx, Dz		
1000	11cc	xxyy	zzzz		[if cc] PCLR DZ		
1001	00cc	xxyy	zzzz		[if cc] PSHA Sx, Sy, Dz		
1001	01cc	xxyy	zzzz		[if cc] PAND Sx, SY, Dz		
1001	10cc	xxyy	zzzz	PRND Sx, Dz	[if cc]PINC Sx, Dz		

Table A-59 Operation Code Map for DSP Operation Instructions (B Field) (cont)

Instruction Code				Fx: 0000	Fx: 0001	Fx: 0010	Fx: 0011–1111
MSB			**LSB**	**cc:00**	**cc:01***	**cc:10 (DCT)**	**cc:11 (DCF)**
1001	11cc	xxyy	zzzz		[if cc] PDMSB Sy Dz		
1010	00cc	xxyy	zzzz	PSUBC Sx, Sy, Dz	[if cc] PSUB Sx, Sy, Dz		
1010	01cc	xxyy	zzzz		[if cc] PXOR Sx, Sy, Dz		
1010	10cc	xxyy	zzzz	PABS Sy, Dz	[if cc] PDEC Sy, Dz		
1010	11cc	xxyy	zzzz				
1011	00cc	xxyy	zzzz	PADDC Sx, Sy, Dz	[if cc] PADD Sx, Sy, Dz		
1011	01cc	xxyy	zzzz		[if cc] POR Sx, Sy, Dz		
1011	10cc	xxyy	zzzz	PRND Sy, Dz	[if cc] PINC Sy, Dz		
1011	11cc	xxyy	zzzz		[if cc] PDMSB Sy, Dz		
1100	0***	****	****				
1100	10cc	xxyy	zzzz		[if cc] PNEG Sx, Dz		
1100	11cc	xxyy	zzzz		[if cc] PSTS MACH, Dz		
1101	0***	****	****				
1101	10cc	xxyy	zzzz		[if cc] PCOPY Sx, Dz		
1101	11cc	xxyy	zzzz		[if cc] PSTS MACL, Dz		
1110	0****	****	****				
1110	10cc	xxyy	zzzz		[if cc] PNEG Sy, Dz		
1110	11cc	xxyy	zzzz		[if cc] PLDS Dz, MACH		
1111	0***	****	****				
1111	10cc	xxyy	zzzz		[if cc] PCOPY Sy, Dz		
1111	11cc	xxyy	zzzz		[if cc] PLDS DZ, MACL		

Note: * Unconditional

Appendix B Pipeline Operation and Contention

The SH-3/SH-3E/DSP series is designed so that basic instructions are executed in one cycle. Two or more cycles are required for instructions when, for example, the branch destination address is changed by a branch instruction or when the number of cycles is increased by contention between MA and IF. Table B-1 gives the number of execution cycles and stages for different types of contention and their instructions. Instructions without contention and instructions that require 2 or more cycles even without contention are also shown.

Instructions contend in the following ways:

- Operations and transfers between registers are executed in one cycle with no contention.
- No contention occurs, but the instruction still requires 2 or more cycles.
- Contention occurs, increasing the number of execution cycles. Contention combinations are:
  - MA contends with IF
  - MA contends with IF and sometimes with memory loads as well
  - MA contends with IF and sometimes with the multiplier as well
  - MA contends with IF and sometimes with memory loads and sometimes with the multiplier

Table B-1 Instructions and Their Contention Patterns

Contention	Cycles	Stages	Instructions
None	1	3	• Transfers between registers • Operations between registers (except when a multiplier is involved) • Logical operations between registers • Shift and dynamic shift instructions • System control ALU instructions
	2	3	Unconditional branches
	3/1	3	Conditional branches
	2/1	3	Delayed conditional branch instructions
	4	3	SLEEP instruction
	4	5	RTE instruction
	5	5	LDC instruction (SR), register to SR
	6/8*2	9	TRAP instruction
• MA contends with IF	1	4	• Memory store instructions • STS.L instruction (PR) • Cache instruction
	1/2*2	4	• Bank register other than STC.L instruction
	2	5	STC.L instruction (bank register)
	3	6	• Memory logic operations
	3/4*2	6	• TAS instruction
	7	7	LDC.L instruction (SR), memory to SR
• MA contends with IF. • Causes memory load contention.	1	5	• Memory load instructions • LDS.L instruction (PR)
	1/5*2	5	• LDC.L instruction

Table B-1 Instructions and Their Contention Patterns (cont)

Contention	Cycles	Stages	Instructions
• MA contends with IF. • Causes multiplier contention.	1	4	• Register to MAC transfer instructions • Memory to MAC transfer instructions • MAC to memory transfer instructions
	1 (to 3)*[1]	6	Multiplication instructions (excluding PMULS)
	2 (to 5)*[1]	7	Multiply/accumulate instructions
	2 (to 5)*[1]	9	Double length multiply/accumulate instructions
	2 (to 5)*[1]	9	Double length multiplication instructions
• MA contends with IF. • Causes memory load, contention. • Causes multiplier contention. • Causes DSP operation contention	1	5	MAC/DSP to register transfer instructions

Notes: 1. The normal minimum number of execution states. (The number in parentheses is the number in contention with the preceding/following instructions.)

2. In the case of the SH3-DSP, the figures on the right indicate the number of cycles and stages.

SH-3/SH-3E/SH-DSP Programming Manual

Publication Date: 1st Edition, September 1995
3rd Edition, September 2000
Published by: Electronic Devices Sales & Marketing Group
Semiconductor & Integrated Circuits
Hitachi, Ltd.
Edited by: Technical Documentation Group
Hitachi Kodaira Semiconductor Co., Ltd.