REJ09B0171-0500O

Everywhere you imagine. RENESAS

The revision list can be viewed directly by clicking the title page.

The revision list summarizes the locations of revisions and additions. Details should always be checked by referring to the relevant text.

32

SH-1/SH-2/SH-DSP

Software Manual

Renesas 32-Bit RISC Microcomputer
SuperH™ RISC engine Family

Software Manual

Rev. 5.00
Revision Date: Jun 30, 2004

Renesas Technology
www.renesas.com

Introduction

The SH-1 and SH-2 incorporates a RISC (Reduced Instruction Set Computer) type CPU. A basic instruction can be executed in one clock cycle, realizing high performance operation. A built-in multiplier can execute multiplication and addition as quickly as DSP.

The SH-DSP is a 32 bit microcontroller based on Renesas SuperH^{TM} RISC engine that realizes the same signal processing capability as a general usage DSP (Digital Signal Processor). The SH-DSP offers an improvement on the DSP functions of multiplication and multiply and accumulate in SuperH microprocessors by using a DSP style data path function. It maintains upward compatibility at the object code level with the SH-1 and SH-2 microprocessors and has the many functions, low power usage, and low price of other SuperH microprocessors.

The SH-DSP achieves high performance in processing operations by using a RISC CPU core and a DSP unit with DSP functions. This new type of single chip RISC-DSP simultaneously integrates the peripheral functions needed to build systems into the SH-DSP and provides the lower-power consumption vital to microprocessor applications.

This Software Manual describes in detail the basic architecture and instructions for the SH-1, SH2, and SH-DSP and is intended as a reference on instruction operation and architecture. It also covers the operation of pipelines, which are a feature of the SuperH microprocessor.

For software development environment system, contact your Renesas Technology Corp. sales office.

Note: SuperH^{TM} is a trademark of Renesas Technology Corp.

Main Revisions for this Edition

Item	Page	Revision (See Manual for Details)
All	—	All references to Hitachi, Hitachi, Ltd., Hitachi Semiconductors, and other Hitachi brand names changed to Renesas Technology Corp.
		Designation for categories changed from “series” to “group”

Contents

Section 1 Features

1.1 SH-1 and SH-2 Features

The SH-1 and SH-2 CPU have 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-1 and SH-2 CPU features.

Table 1.1 SH-1 and SH-2 CPU Features

Item	Feature
Architecture	• Original Renesas Technology architecture • 32-bit internal data bus
General-register machine	• Sixteen 32-bit general registers • Three 32-bit control registers • Four 32-bit system registers
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 (SH-1 CPU)	• Multiplication operations (16 bits $\times$ 16 bits $\rightarrow$ 32 bits) executed in 1 to 3 cycles, and multiplication/accumulation operations (16 bits $\times$ 16 bits + 42 bits $\rightarrow$ 42 bits) executed in 3/(2)* cycles
On-chip multiplier (SH-2 CPU)	• Multiplication operations executed in 1 to 2 cycles (16 bits $\times$ 16 bits $\rightarrow$ 32 bits) or 2 to 4 cycles (32 bits $\times$ 32 bits $\rightarrow$ 64 bits), and multiplication/accumulation operations executed in 3/(2)* cycles (16 bits $\times$ 16 bits + 64 bits $\rightarrow$ 64 bits) or 3/(2 to 4)* cycles (32 bits $\times$ 32 bits + 64 bits $\rightarrow$ 64 bits)
Pipeline	• Five-stage pipeline

Item	Feature
Processing states	• Reset state • Exception processing state • Program execution state • Power-down state • Bus release state
Power-down states	• Sleep mode • Standby mode

Note: * The normal minimum number of execution cycles (The number in parentheses in the number in contention with preceding/following instructions).

1.2 SH-DSP Features

The SH-DSP is a 32-bit microcontroller based on the Renesas SuperH RISC engine (abbreviated below as “SuperH”) and incorporating the signal processing performance of a general-use digital signal processor (DSP). The SuperH already supported some DSP type instructions, such as multiply and accumulate. In the SH-DSP, the DSP functions have been enhanced, and full DSP data bus have been implemented. The SH-DSP is backward compatible at the object code level with the SH-1 and SH-2 CPUs.

The SuperH only has 16-bit instructions. The SH-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 SuperH uses a standard Neumann architecture, but the SH-DSP has the DSP data bus of the expanded Harvard architecture.

Table 1.2 lists the added features of the SH-DSP.

Table 1.2 Features of SH-DSP Series Microprocessor CPUs

Feature	Description
DSP unit	• 1 cycle multiplier • 16 bits $\times$ 16 bits $\rightarrow$ 32 bits (fixed decimal point) • Arithmetic logic unit (ALU) • Barrel shifter • DSP registers • MSB detection
DSP registers	• Two 40-bit data registers • Six 32-bit data registers • DSP status register (DSR) • Modulo register (MOD, 32 bits) added to control registers • Repeat counter (RC) added to status registers (SR) • Repeat start register (RS) and repeat end register (RE) added to control registers
DSP data b1us	• Expanded Harvard architecture • Simultaneous access of two data bus and one instruction bus
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 both the WB stage and the DSP stage.

Section 2 Register Configuration

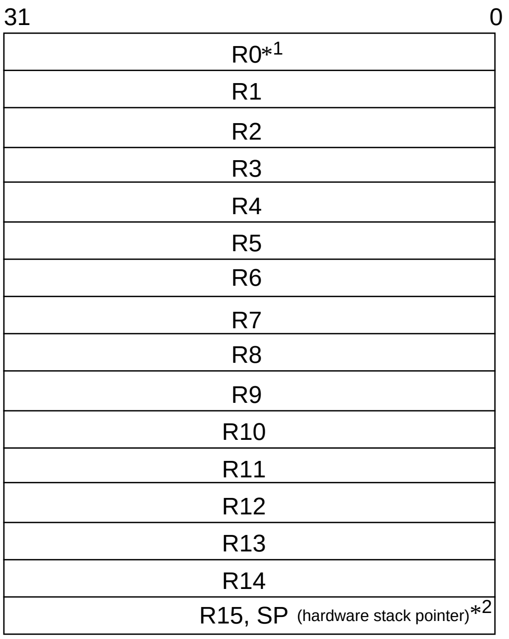
The register set of the SH-1 and SH-2 consists of sixteen 32-bit general registers, three 32-bit control registers and four 32-bit system registers.

The SH-DSP maintains upward compatibility with the SH-1 and SH-2 microprocessors on the object code level. To this end, it has the same registers as the SuperH microprocessors, with the addition of several other registers. Three control registers have been added: the repeat start register (RS), the repeat end register (RE), and the modulo register (MOD). Six other registers have also been added: the DSP status register (DSR), which is a system register, and eight DSP data registers (A0, A1, X0, X1, Y0, Y1, M0, and M1).

The general registers are used the same as in the SH-1 and SH-2 when SuperH type instructions are involved. With DSP type instructions, however, they are used as address registers and index registers for accessing memory.

2.1 General Registers

There are 16 general registers (Rn) numbered R0–R15, which are 32 bits in length (figure 2.1). General registers are used for data processing and address calculation. R0 is also used as an index register. Several instructions use R0 as a fixed source or destination register. R15 is used as the hardware stack pointer (SP). Saving and recovering the status register (SR) and program counter (PC) in exception processing is accomplished by referencing the stack using R15.



Notes: 1. R0 functions as an index register in the indirect indexed register addressing mode and indirect indexed GBR addressing mode. In some instructions, R0 functions as a fixed source register or destination register.
2. R15 functions as a hardware stack pointer (SP) during exception processing.

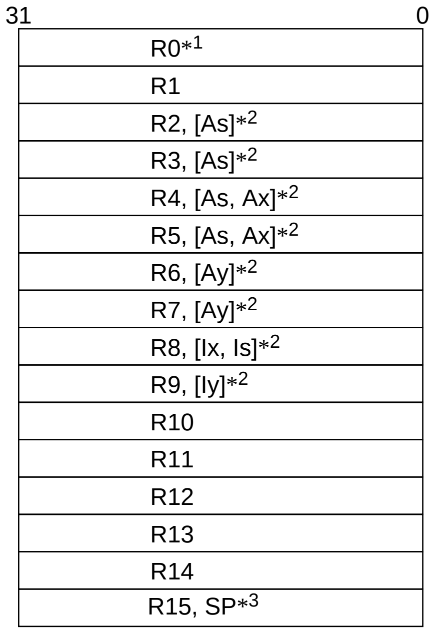
Figure 2.1 General Registers (SH-1 and SH-2)

With DSP type instructions, eight of the 16 general registers are used in addressing the X and Y data memory and the data memory that uses the I bus (single data).

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 I 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.

Figure 2.2 shows the general registers.



Notes: 1. R0 functions as an index register in the indirect indexed register addressing mode and indirect indexed GBR addressing mode. In some instructions, R0 functions as a source register or destination register.
2. Used as memory address register and memory index register with DSP instructions.
3. R15 functions as a hardware stack pointer (SP) during exception processing.

Figure 2.2 Organization of General Registers (SH-DSP)

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.

2.2 Control Registers

The 32-bit control registers consist of the 32-bit status register (SR), global base register (GBR), and vector base register (VBR) (figure 2.3). The status register indicates processing states. The global base register functions as a base address for the indirect GBR addressing mode to transfer data to the registers of on-chip peripheral modules. The vector base register functions as the base address of the exception processing vector area (including interrupts).

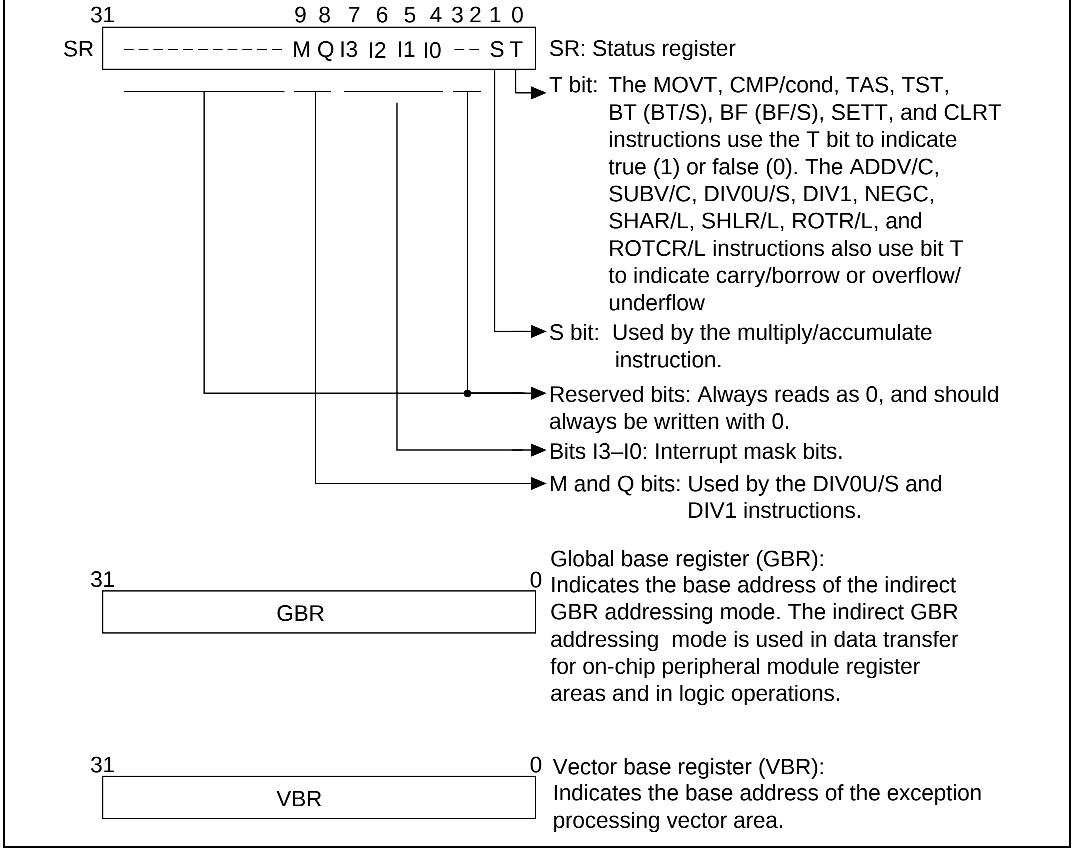

31
9 8 7 6 5 4 3 2 1 0
SR
– – – – – – – – – – – M Q I3 I2 I1 I0 – – S T
SR: Status register
T bit: The MOVT, CMP/cond, TAS, TST, BT (BT/S), BF (BF/S), SETT, and CLRT instructions use the T bit to indicate true (1) or false (0). The ADDV/C, SUBV/C, DIV0U/S, DIV1, NEGC, SHAR/L, SHLR/L, ROTR/L, and ROTCR/L instructions also use bit T to indicate carry/borrow or overflow/ underflow
S bit: Used by the multiply/accumulate instruction.
Reserved bits: Always reads as 0, and should always be written with 0.
Bits I3–I0: Interrupt mask bits.
M and Q bits: Used by the DIV0U/S and DIV1 instructions.
31
0
GBR
Global base register (GBR): Indicates the base address of the indirect GBR addressing mode. The indirect GBR addressing mode is used in data transfer for on-chip peripheral module register areas and in logic operations.
31
0
VBR
Vector base register (VBR): Indicates the base address of the exception processing vector area.

Figure 2.3 Control Registers (SH-1 and SH-2)

The SH-SDP additionally has a repeat start (RS) register, a repeat end (RE) register, and a 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, 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 2.4 shows the control registers. Table 2.1 shows the bits of the SR register.

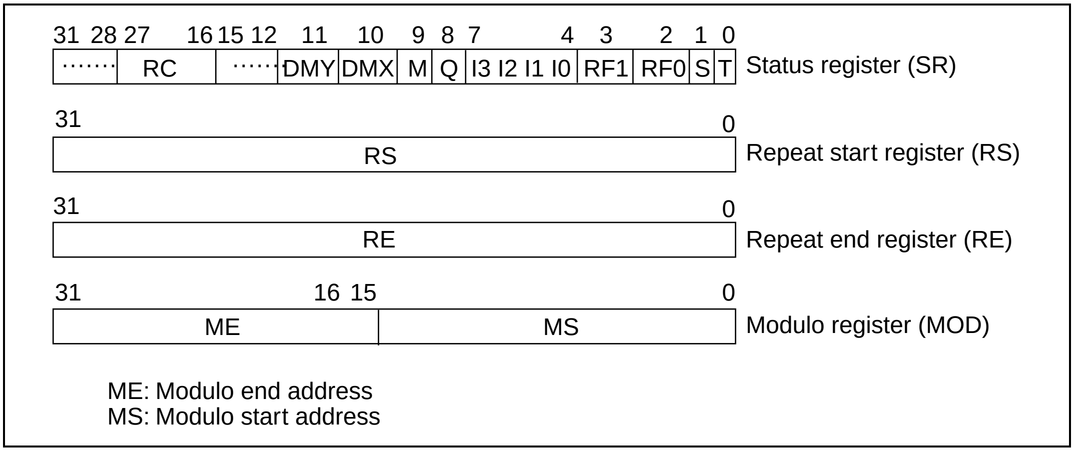


Figure 2.4 Organization of the Control Registers (SH-DSP)

Table 2.1 SR Register Bits

Bits	Name	Function
27–16	Repeat counter (RC)	Specifies the number of iterations for repeat (loop) control (2 to 4095)
11	Specification of modulo addressing for Y pointer (DMY)	1: Modulo addressing mode becomes valid for the Y memory address register Ay (R6, R7)
10	Specification of modulo addressing for X pointer (DMX)	1: Modulo addressing mode becomes valid for the X memory address register Ax (R4, R5)
9	Bit M	Used by the DIV0S/U and DIV1 instructions
8	Bit Q	
7–4	Interrupt request mask (IMASK)	Indicate the level of interrupt request accepted (0-15)
3–2	Repeat flag (RF1, RF0)	Used to control zero-overhead repeating (loop) 00: 1 step repeat 01: 2 step repeat 11: 3 step repeat 10: Repeat of 4 or more steps
1	Saturation operation bit (S)	Used by MAC and DSP instructions 1: Specifies saturation operation (prevents overflows)
0	Bit T	For MOVT, CMP/cond, TAS, TST, BT, BF, SETT, CLRT, and DT instructions: 0: FALSE 1: TRUE For ADDV/C, SUBV/C, DIV0U/S, DIV1, NEGC, SHAR/L, SHLR/L, ROTR/L and ROTCR/L instructions: 1: Indicates a carry, borrow, overflow or underflow
31–28, 15–12	Reserved	0: Always reads 0; Always write 0.

Dedicated load and store instructions are used to access the RS, RE, and MOD registers. For example, to access the RS register, do the following:

```
LDC    Rm,          RS; Rm → RS
LDC.L  @Rm+, RS;    (Rm) → RS, Rm+4 → Rm
STC    RS, Rn;      RS → Rn
STC.L  RS, @-Rn;    Rn-4 → Rn, RS → (Rn)
```

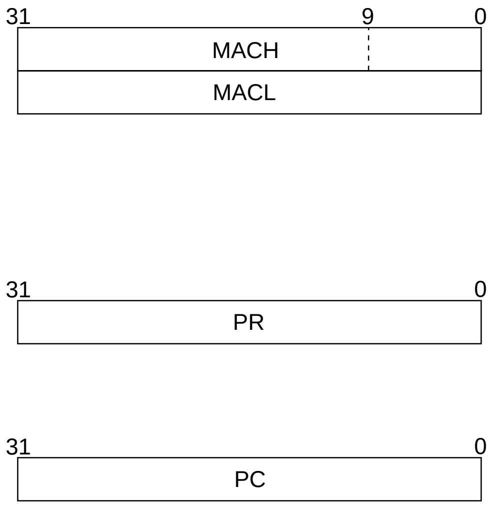
The following instructions set addresses in the RS, RE registers for zero overhead repeat control:

```
LDRS   @(disp, PC);  disp × 2 + PC → RS
LDRE   @(disp, PC);  disp × 2 + PC → RE
```

The GBR and VBR registers are the same as the previous SuperH registers. Four control bits (DMX, DMY, RF1, and RF0 bits) and an RC counter have been added to the SR register. The RS, RE, and MOD registers are new registers.

2.3 System Registers

System registers consist of four 32-bit registers: high and low multiply and accumulate registers (MACH and MACL), the procedure register (PR), and the program counter (PC). The multiply and accumulate registers store the results of multiply and multiply and accumulate operations. The procedure register stores the return address from the subroutine procedure. The program counter indicates the address of the program executing and controls the flow of the processing. The PC counter points to four bytes ahead of the instruction currently executing. These registers are the same as the SuperH microprocessor registers.



Multiply and accumulate register high (MACH) Multiply and accumulate register low (MACL)
These are the registers for storing the results of multiply and accumulate operations. On the SH-2 CPU, MACH has 32 valid bits. On the SH-1 CPU, only the lower 10 bits of MACH are valid, and data is sign extended to 32 bits when read.

Procedure register (PR)
This register is used to store the return destination addresses for subroutine procedures.

Program counter (PC)
The PC indicates the next four bytes (two instructions) following the instruction currently being executed.

Note: These are used only when executing an instruction that was supported by SH-1 and SH-2. They are not used for multiplication instructions newly added for the SH-DSP (PMULS).

Figure 2.5 Organization of the System Registers

In addition, the SH-DSP also uses as its system registers the DSP status register (DSR) and five of the eight data registers (A0, X0, X1, Y0, Y1), which are all registers of the DSP unit and will be described later (DSP registers). The A0 register is a 40-bit register, but the guard bit section (A0G) is ignored in data read from A0. When data is input to the A0 register, the MSB of the data is copied to the guard bit section (A0G).

2.4 DSP Registers

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 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.

Note that five registers, A0, X0, X1, Y0, and Y1, can also be used as system registers.

Figure 2.6 shows the DSP registers. Table 2.2 lists the DSR register bit functions.

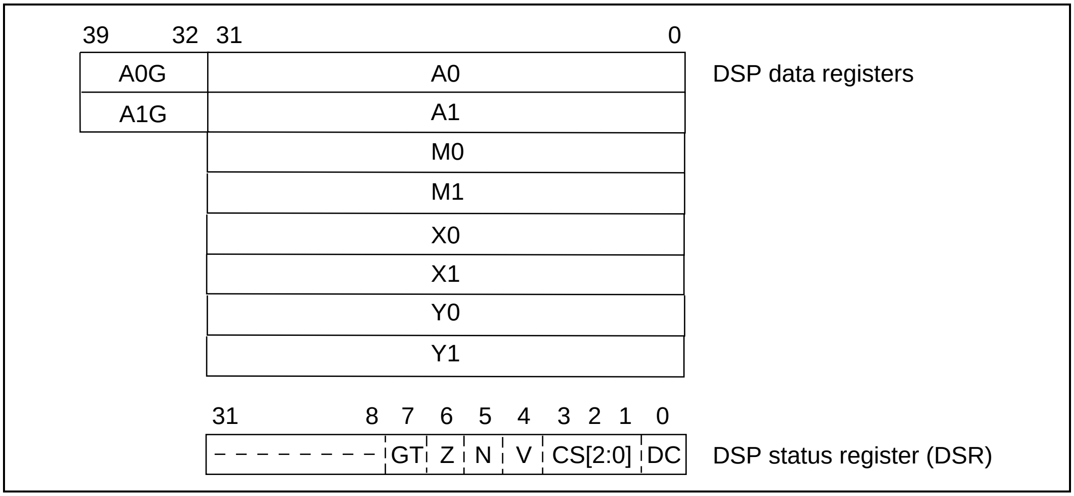


Figure 2.6 Organization of the DSP Registers

Table 2.2 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 A0, X0, X1, Y0, Y1, and DSR registers as a system registers.

2.5 Precautions for Handling of Guard Bit and Overflow

Data operation in the DSP unit is basically executed in 32 bits. Actual operation, however, is made in 40-bit length including 8 guard bits. When the guard bits are inconsistent with the value of MSB of 32 bits, the operation result is handled as overflow. In this case, the N bit indicates the correct condition of the operation result whether overflow has occurred or not. This is also the same when the destination operand is a register of 32 bits in length. Each status flag is updated always assuming guard bits of 8 bits.

If line overflow occurs so that the result is not correctly indicated even though the guard bits are used, the N flag cannot show the correct condition. Refer to section 8.1, ALU Fixed Decimal Point Operation, DC Bit, for details.

2.6 Initial Values of Registers

Table 2.3 lists the values of the registers after reset.

Table 2.3 Initial Values of Registers

Classification	Register	Initial Value
General registers	R0–R14	Undefined
	R15 (SP)	Value of the stack pointer in the vector address table
Control registers	SR	• Bits I3 to I0 are 1111(H'F), reserved bits are 0, and other bits are undefined RC, DMY, DMX, RF1, and RF0 are 0 (additional bits on SH-DSP)
	RS RE	Undefined
	GBR	Undefined
	VBR	H'00000000
	MOD	Undefined
System registers	MACH, MACL, PR	Undefined
	PC	Value of the program counter in the vector address table
DSP registers	A0, A0G, A1, A1G, M0, M1, X0, X1, Y0, Y1	Undefined
	DSR	H'00000000

Section 3 Data Formats

3.1 Data Format in Registers

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

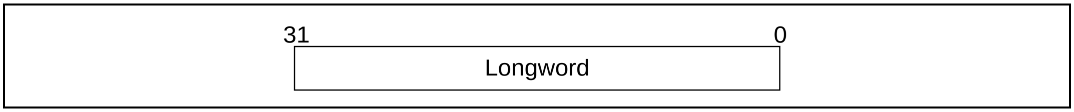


Figure 3.1 Data Format in Registers

3.2 Data Format in Memory

Memory data formats are classified into bytes, words, and longwords. Byte data can be accessed from any address, but an address error will occur if you try to access word data starting from an address other than 2n or longword data starting from an address other than 4n. In such cases, the data accessed cannot be guaranteed. The hardware stack area, which is referred to by the hardware stack pointer (SP, R15), uses only longword data starting from address 4n because this area stores the program counter (PC) and status register (SR). See the hardware manual for more information on address errors.

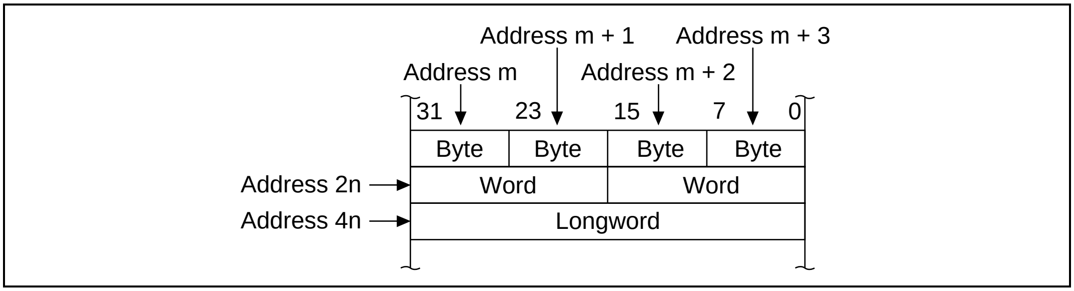


Figure 3.2 Data Format in Memory (Big Endian)

Byte data is arranged as shown below for products with a built-in little endian function. To determine whether a specific product supports little endian operation, refer to the corresponding hardware manual.

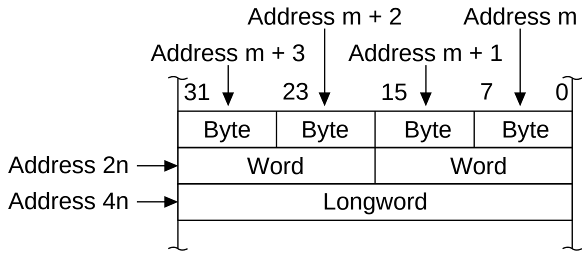


Figure 3.3 Data Format in Memory (Little Endian)

3.3 Immediate Data Format

Byte immediate data is located in an instruction code. Immediate data accessed by the MOV, ADD, and CMP/EQ instructions is sign-extended and is handled in registers as longword data. Immediate data accessed by the TST, AND, OR, and XOR instructions is zero-extended and is handled as longword data. Consequently, AND instructions with immediate data always clear the upper 24 bits of the destination register.

Word or longword immediate data is not located in the instruction code but rather is stored in a memory table. The memory table is accessed by a immediate data transfer instruction (MOV) using the PC relative addressing mode with displacement. Specific examples are given in section 7, CPU Core Instruction Features, instruction 8, and table 7.4.

3.4 DSP Type Data Formats

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

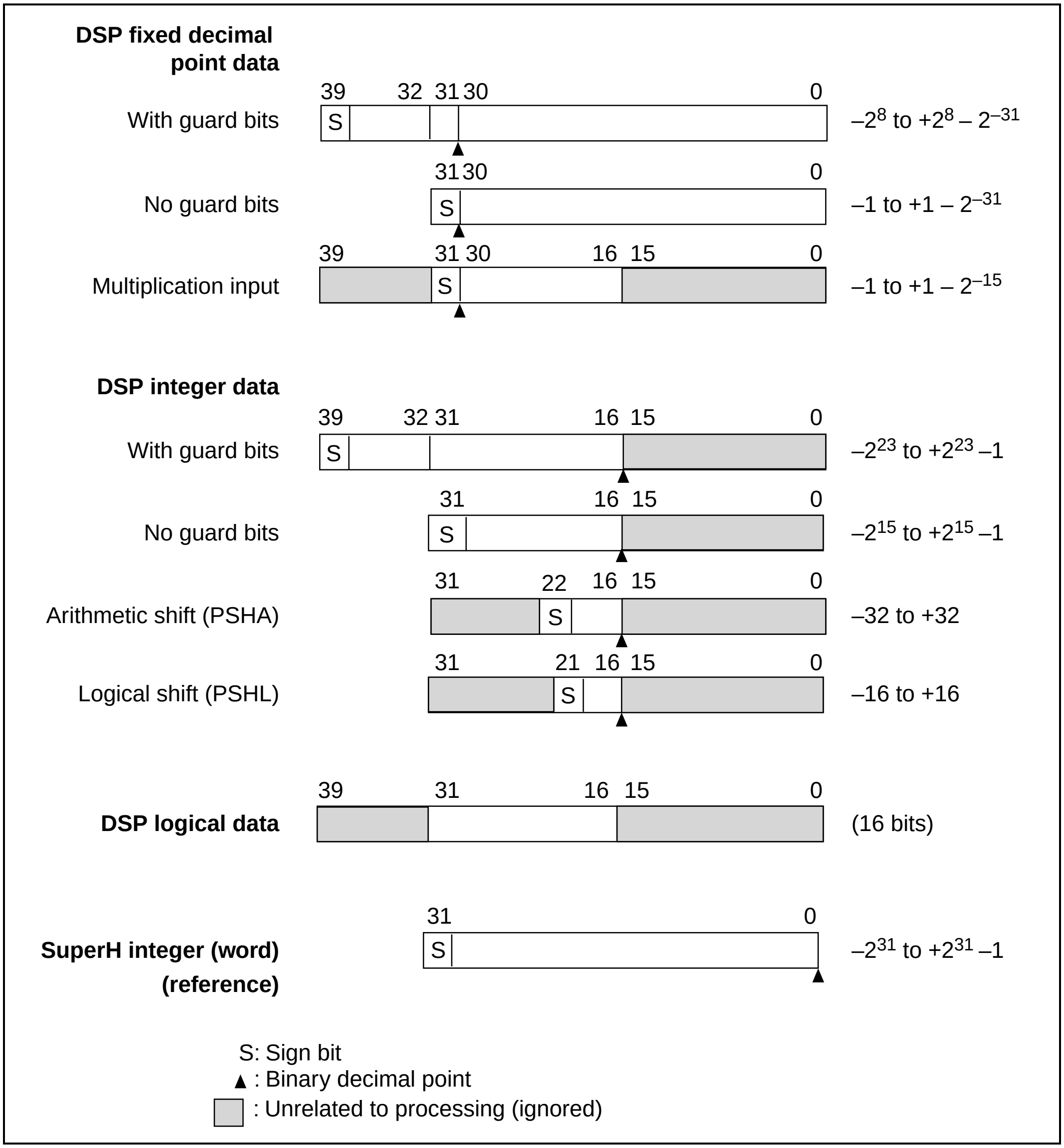


Figure 3.4 DSP Data Formats

3.5 DSP Instructions and Data Formats

The data format and valid data length varies with the instruction and DSP register. Instructions that access the DSP data register fall into three categories: DSP data processing, X and Y data transfer processing, and single data transfer processing.

3.5.1 DSP Data Processing

When the A0 or A1 register is used as the source register in DSP fixed decimal point data processing, the guard bits (32–39) are enabled. When any other register is used as the source register (M0, M1, X0, X1, Y0, or Y1), the register data's sign-extended portion goes to bits 32–39. When the A0 or A1 register is used as the destination register, the guard bits (32–39) are enabled. When any other register is used as the destination register, the resulting data's bits 32–39 are ignored.

DSP integer data processing is the same as DSP fixed decimal point data processing. The bottom word (the bottom 16 bits, or bits 0–15) of the source register, however, is ignored. The bottom word of the destination register is cleared with zeroes.

The top word (top 16 bits, or bits 16–31) of the source register for DSP logical data processing is enabled. The bottom word and the guard bits of registers A0 and A1 are ignored. The top word of the destination register is enabled. The bottom word and the guard bits of registers A0 and A1 are cleared with zeroes.

3.5.2 X and Y Data Transfers

The MOVX.W and MOVY.W instructions access the X and Y memory through the 16-bit X and Y data buses. The part of data loaded to a register or stored from a register is the top word (bits 16–31). The bottom word is cleared with zeroes.

3.5.3 Single Data Transfers

The MOVS.W and MOVS.L instructions can access any memory through the instruction data bus (IDB). All DSP registers are connected to the IDB bus, which can serve as either the source and destination register during a data transfer. There are two data transfer modes: word and longword. In word mode, data is loaded to the top word of the DSP register or stored from the top word, except for the A0G and A1G registers. In longword mode, data is loaded to the 32 bits of the DSP register or stored from the 32 bits, except for the A0G and A1G registers.

In single data transfers, the A0G and A1G registers can be handled as independent registers. Eight bits of data can be loaded to or stored from the A0G and A1G registers.

When the A0G or A1G register is the source register, only eight bits are stored from the register. The top bits are sign extended.

When the A0G or A1G register is the destination register, the bottom eight bits are loaded to the register. The A0 and A1 registers are not cleared with zeros, so the values are preserved.

Tables 3.1 and 3.2 list the data formats on the register with the DSP instructions. With some instructions, not all registers can be accessed. For example, the PMULS instruction can specified the A1 register as the source register, but not the A0 register. For more information, see the description of the instruction.

Figure 3.5 shows the relationship between the DSP registers and buses during data transfers.

Table 3.1 Data Format of DSP Instruction Source Register

Register	Instruction		Guard Bits 39–32	Register Bits 31–16	Register Bits 15–0
A0, A1	DSP operation	Fixed decimal, PDMSB, PSHA		40 bit data	
		Integer	24 bit data		—
		Logic, PSHL, PMULS	—	16 bit data	
	Data transfer	MOVX.W, MOVY.W, MOVS.W		16 bit data	
		MOVS.L		32 bit data	
A0G, A1G	Data transfer	MOVS.W	Data	—	—
		MOVS.L	Data		
X0, X1, Y0, Y1, M0, M1	DSP operation	Fixed decimal, PDMSB, PSHA	Sign*	32 bit data	
		Integer		16 bit data	—
		Logic, PSHL, PMULS	—	16 bit data	—
	Data transfer	MOVS.W		16 bit data	
		MOVS.L		32 bit data	

Note: * The sign is extended and stored in the ALU's guard bits.

Table 3.2 Data Format of DSP Instruction Destination Register

Register	Instruction		Guard Bits	Register Bits	
			39–32	31–16	15–0
A0, A1	DSP operation	Fixed decimal, PSHA, PMULS	(Sign extend)	40 bit result	
		Integer, PDMSB	(Sign extend)	24 bit result	Clear to 0
		Logic, PSHL	Clear to 0	16 bit result	Clear to 0
	Data transfer	MOVS.W	Sign extend	16 bit data	Clear to 0
		MOVS.L	Sign extend	32 bit data	
A0G, A1G	Data transfer	MOVS.W	Data	Not updated	
		MOVS.L	Data	Not updated	
X0, X1, Y0, Y1, M0, M1	DSP operation	Fixed decimal, PSHA, PMULS	—	32 bit result	
		Integer, logic, PDMSB, PSHL		16 bit result	Clear to 0
	Data transfer	MOVX.W, MOVY.W, MOVS.W		16 bit data	Clear to 0
		MOVS.L		32 bit data	

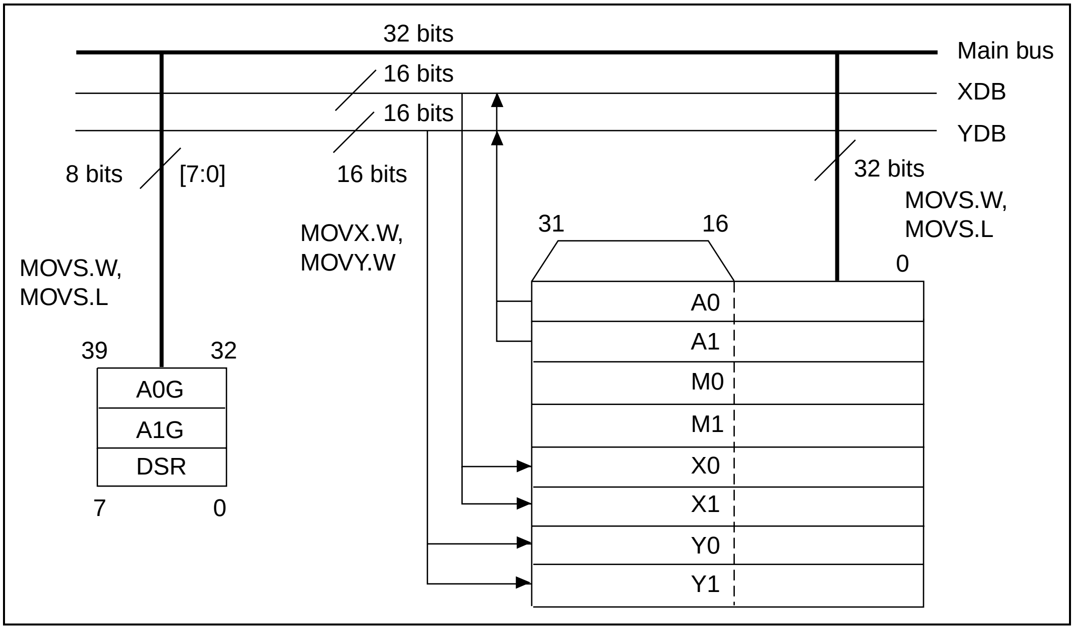


Figure 3.5 Relationship between DSP Registers and Buses during Data Transfer

Section 4 Instruction Features

4.1 RISC-Type Instruction Set

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

16-Bit Fixed Length: All instructions are 16 bits long, increasing program coding efficiency.

One Instruction/Cycle: Basic instructions can be executed in one cycle using the pipeline system. Instructions are executed in 50 ns at 20 MHz, in 35 ns at 28.7MHz.

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 calculated with longword data (table 4.1). Immediate data is sign-extended for arithmetic operations or zero-extended for logic operations. It also is calculated with longword data.

Table 4.1 Sign Extension of Word Data

SH-1/SH-2/SH-DSP CPU	Description	Example for Other CPU

`MOV.W    @(disp,PC),R1`
`ADD      R1,R0`
`    .........`
`.DATA.W  H'1234`
	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`

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

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.

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 4.2). With delayed branching, branching occurs after execution of the slot instruction. However, instructions such as register changes etc. are executed in the order of delayed branch instruction, then delay slot instruction. For example, even if the register in which the branch destination address has been loaded is changed by the delay slot instruction, the branch will still be made using the value of the register prior to the change as the branch destination address.

Table 4.2 Delayed Branch Instructions

SH-1/SH-2/SH-DSP CPU		Description	Example for Other CPU	
`BRA`	`TRGET`	Executes an ADD before branching to TRGET.	`ADD.W`	`R1,R0`
`ADD`	`R1,R0`		`BRA`	`TRGET`

Multiplication/Accumulation Operation:

SH-1 CPU: 16bit × 16bit → 32-bit multiplication operations are executed in one to three cycles. 16bit × 16bit + 42bit → 42-bit multiplication/accumulation operations are executed in two to three cycles.

SH-2/SH-DSP CPU: 16bit × 16bit → 32-bit multiplication operations are executed in one to two cycles. 16bit × 16bit + 64bit → 64-bit multiplication/accumulation operations are executed in two to three cycles. 32bit × 32bit → 64-bit multiplication and 32bit × 32bit + 64bit → 64-bit multiplication/accumulation operations are executed in two to four cycles.

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 4.3). The number of instructions after T bit in the status register is kept to a minimum to improve the processing speed.

Table 4.3 T Bit

SH-1/SH-2/SH-DSP CPU		Description	Example for Other CPU	
`CMP/GE`	`R1,R0`	T bit is set when R0 ≥ R1. The program branches to TRGET0.	`CMP.W`	`R1,R0`
`BT`	`TRGET0`	When R0 ≥ R1 and to TRGET1.	`BGE`	`TRGET0`
`BF`	`TRGET1`	When R0 < R1.	`BLT`	`TRGET1`
`ADD`	`#–1,R0`	T bit is not changed by ADD.	`SUB.W`	`#1,R0`
`CMP/EQ`	`#0,R0`	T bit is set when R0 = 0.	`BEQ`	`TRGET`
`BT`	`TRGET`	The program branches if R0 = 0.		

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 4.4).

Table 4.4 Immediate Data Accessing

Classification	SH-1/SH-2/SH-DSP CPU	Example for Other 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).

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 4.5 Absolute Address

Classification	SH-1/SH-2/SH-DSP CPU	Example for Other CPU

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

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 4.6 Displacement Accessing

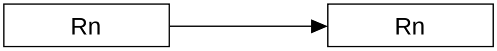
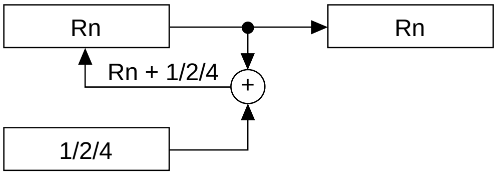
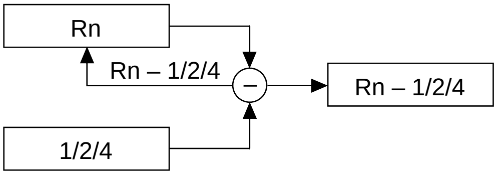
Classification	SH-1/SH-2/SH-DSP CPU	Example for Other CPU

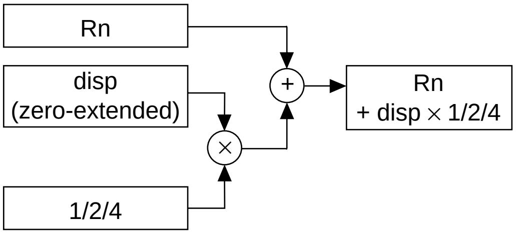
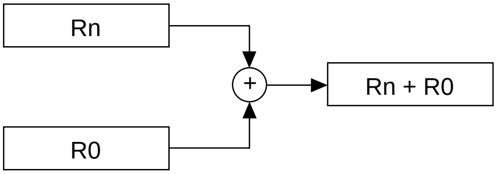
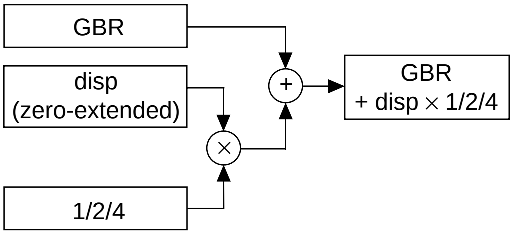
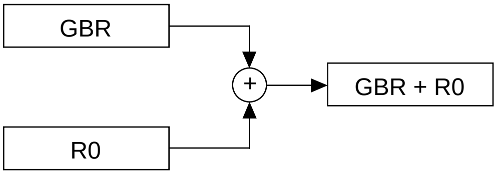
16-bit displacement	`MOV.W @(disp,PC),R0`
	`MOV.W @(R0,R1),R2`
	`..................`
	`.DATA.W H'1234`
		`MOV.W @(H'1234,R1),R2`

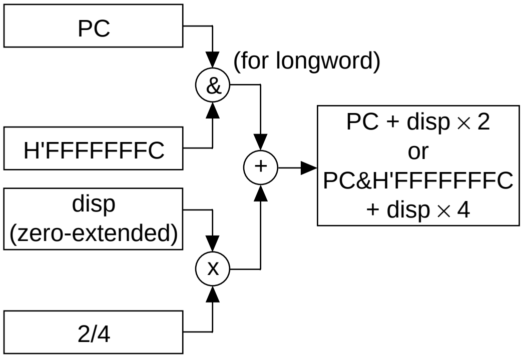
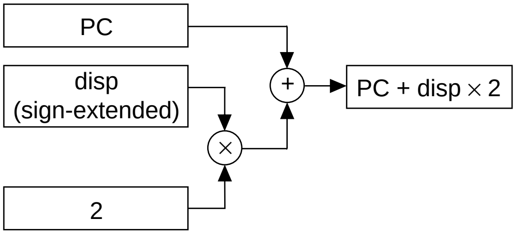
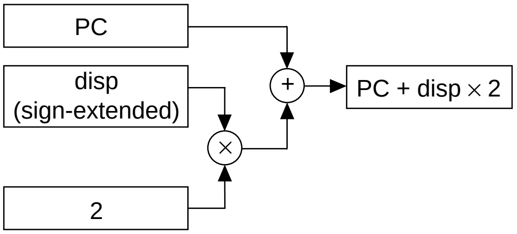
4.2 Addressing Modes

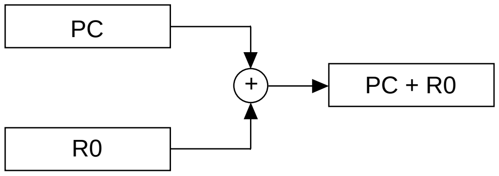
Addressing modes effective address calculation by the CPU core are described below.

Table 4.7 Addressing Modes and Effective Addresses

Addressing Mode	Instruction Format	Effective Addresses Calculation	Formula
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, or 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, or 4 for a longword operation. 	Byte: Rn – 1 → Rn Word: Rn – 2 → Rn Longword: Rn – 4 → Rn (Instruction executed with Rn after calculation)

Addressing Mode	Instruction Format	Effective Addresses Calculation	Formula
Indirect register addressing with displacement	@(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, or is quadrupled for a longword operation.  Rn disp (zero-extended) 1/2/4 + × Rn + disp × 1/2/4	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 displacement	@(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, or is quadrupled for a longword operation.  GBR disp (zero-extended) 1/2/4 + × GBR + disp × 1/2/4	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 R0. 	GBR + R0

Addressing Mode	Instruction Format	Effective Addresses Calculation	Formula
PC relative 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 disp is doubled for a word operation, or 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

Addressing Mode	Instruction Format	Effective Addresses Calculation	Formula
PC relative addressing (cont)	Rn*	The effective address is the register PC plus Rn. 	PC + Rn
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.	—

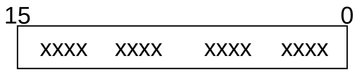
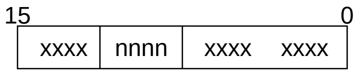
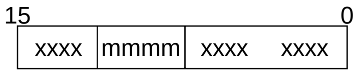
Note: * Applies to the SH-2 and SH-DSP. This addressing mode is not supported by the SH-1.

4.3 Instruction Format

The instruction format table, table 4.8, 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 4.8 Instruction Formats

Instruction Formats	Source Operand	Destination Operand	Example
0 format 	—	—	`NOP`
n format 	—	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 	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`

Instruction Formats	Source Operand	Destination Operand	Example
nm format 15 0 xxxx | nnnn | mmmm | xxxx	mmmm: Direct register	nnnn: Direct register	`ADD    Rm,Rn`
	mmmm: Direct register	nnnn: Indirect 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 0 xxxx xxxx | mmmm | dddd	mmmmdddd: indirect register with displacement	R0 (Direct register)	`MOV.B @(disp,Rm),R0`
nd4 format 15 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.

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`

Note: Applies to the SH-2 and SH-DSP. The BRAF instruction is not supported by the SH-1.

4.4 DSP

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.

4.5 DSP Data Addressing

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 4.9 summarizes the data transfer instructions.

Table 4.9 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-increment	Nop/Inc(+2, +4)/Index addition: Post-increment
	—	Dec(–2, –4): Pre-decrement
Modulo addressing	Available	Not available
Data buses	XDB, YDB	IDB
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	Da: 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

4.5.1 X and Y Data Addressing

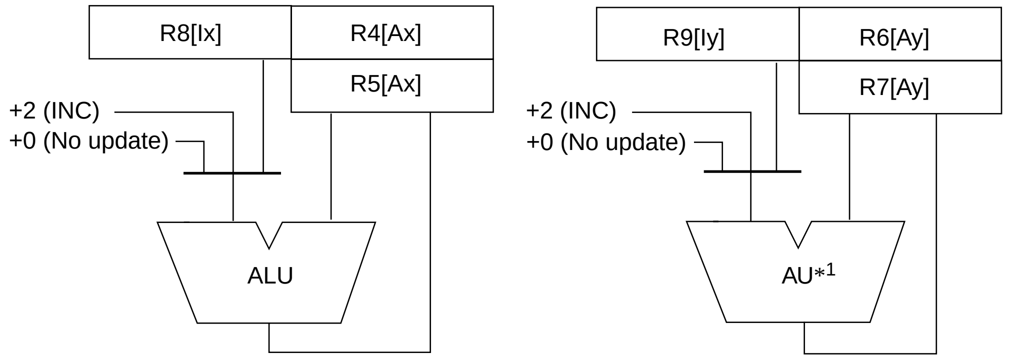
The DSP command allows X and Y data memories to be accessed simultaneously using the MOVX.W and MOVY.W instructions. DSP instructions have two pointers so they can access the X and Y data memories simultaneously. DSP instructions have only pointer addressing; immediate addressing is not available. Address 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-increment).
- Increment address register addressing: The Ax and Ay registers are address pointers. +2 is added to them after data transfer (post-increment).

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. For X and Y data addressing, only bits 1 to 15 of the address pointer are valid. When performing X and Y data addressing, make sure to write 0 to bit 0 of the address pointer and index register.

Figure 4.1 shows the X and Y data transfer addressing. With using the X or Y bus to access X memory or Y memory, Ax (R4 or R5) and Ay (R6 or R7) upper reads [?? words] are ignored. Also, the results of XX AY+, XX Ay + Iv are stored in the lower word of Ay, and the previous value of the upper word is retained.



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

Figure 4.1   X and Y Data Transfer Addressing

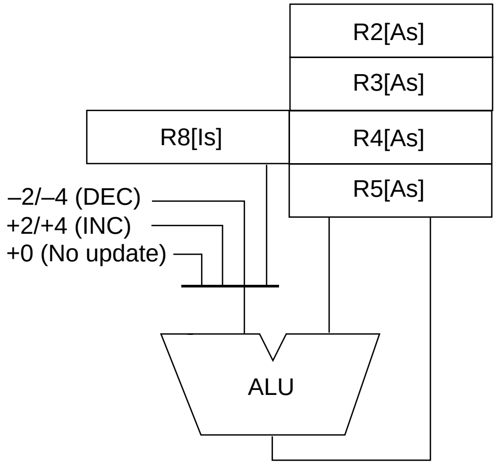
4.5.2 Single Data Addressing

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-increment).
- Increment address register addressing: The As register is the address pointer. +2 or +4 is added to it after data transfer (post-increment).
- 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-decrement).

The address pointer uses the R8 register as its index register (Is). Figure 4.2 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-increment methods. Decrement is a pre-decrement method.

Figure 4.2 Single Data Transfer Addressing

4.5.3 Modulo Addressing

Like other DSPs, the SH-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.W   mEnd;              Lower 16bit of ModEnd
          .DATA.W   mStart;            Lower 16bit 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 4.3 shows a block diagram of modulo addressing.

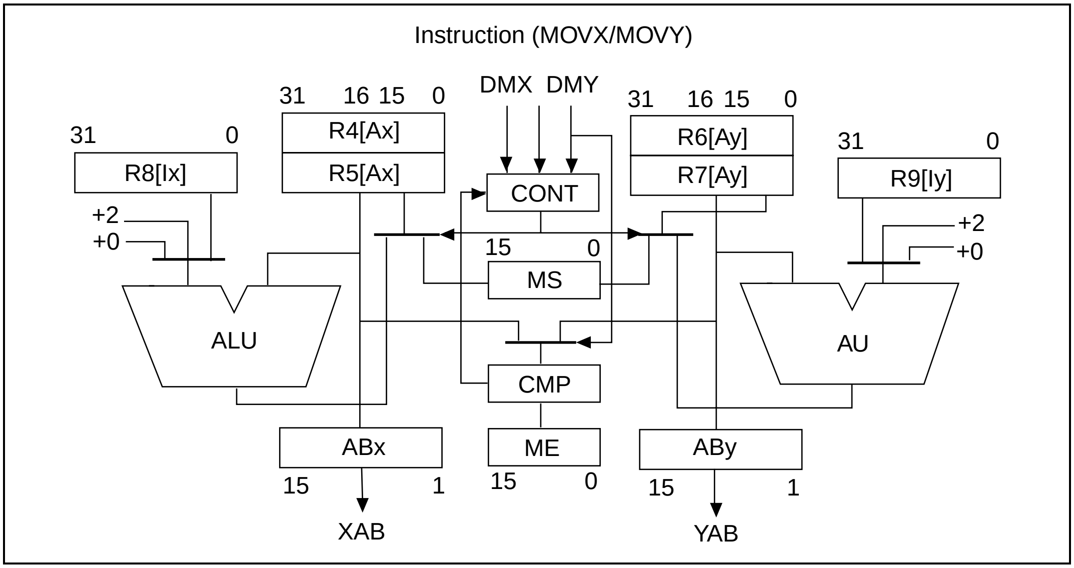


Figure 4.3 Modulo Addressing

The following is an example of modulo addressing.

```
MS=H'C008; ME=H'C00C; 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.

4.5.4 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;
}
```

4.6 Instruction Formats for DSP Instructions

New instructions have been added to the SH-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 4.4 shows their instruction formats.

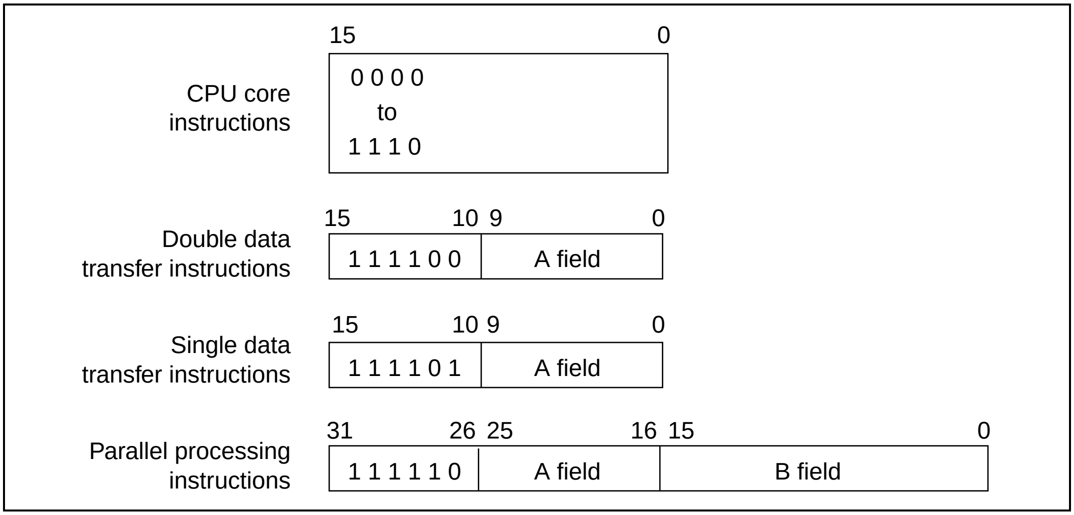


Figure 4.4 Instruction Formats of DSP Instructions

4.6.1 Double and Single Data Transfer Instructions

Table 4.10 shows the instruction formats for double data transfer instructions. Table 4.11 shows the instruction formats for single data transfer instructions

Table 4.10 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`
	`MOVX.W  @Ax+,Dx`
	`MOVX.W  @Ax+Ix,Dx`
								Ax	

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

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

Category	Mnemonic	7	6	5	4	3	2	1	0
X memory data transfers	`NOPX`	0		0		0	0		

	`MOVX.W  @Ax,Dx`
	`MOVX.W  @Ax+,Dx`
	`MOVX.W  @Ax+Ix,Dx`
		Dx		0		0
						1
						1
							1
							0
							1		

	`MOVX.W  Da,@Ax`
	`MOVX.W  Da,@Ax+`
	`MOVX.W  Da,@Ax+Ix`
		Da		1		0
						1
						1
							1
							0
							1		

Y memory data transfers	`NOPY`		0		0			0	0

	`MOVY.W  @Ay,Dy`
	`MOVY.W  @Ay+,Dy`
	`MOVY.W  @Ay+Iy,Dy`
			Dy		0			0
								1
								1
									1
									0
									1

	`MOVY.W  Da,@Ay`
	`MOVY.W  Da,@Ay+`
	`MOVY.W  Da,@Ay+Iy`
			Da		1			0
								1
								1
									1
									0
									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 4.11 Instruction Formats for Single Data Transfers

Category	Mnemonic	15	14	13	12	11	10	9	8

Single data transfer	`MOVS.W @–As,Ds`
	`MOVS.W @As,Ds`
	`MOVS.W @As+,Ds`
	`MOVS.W @As+Is,Ds`
		1	1	1	1	0	1	As
								0: R4
								1: R5
								2: R2
								3: R3	

	`MOVS.W Ds,@A–s`
	`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`								

Category	Mnemonic	7	6	5	4	3	2	1	0

Single data transfer	`MOVS.W @–As,Ds`
	`MOVS.W @As,Ds`
	`MOVS.W @As+,Ds`
	`MOVS.W @As+Is,Ds`
		Ds		0: (*)
				1: (*)
				2: (*)
				3: (*)
						0
						0
						1
						1
							0
							1
							0
							1
								0	0

	`MOVS.W Ds,@A–s`
	`MOVS.W Ds,@As`
	`MOVS.W Ds,@As+`
	`MOVS.W Ds,@As+Is`
				4: (*)
				5: A1
				6: (*)
				7: A0
						0
						0
						1
						1
							0
							1
							0
							1
								0	1

	`MOVS.L @–As,Ds`
	`MOVS.L @As,Ds`
	`MOVS.L @As+,Ds`
	`MOVS.L @As+Is,Ds`
				8: X0
				9: X1
				A: Y0
				B: Y1
						0
						0
						1
						1
							0
							1
							0
							1
								1	0

	`MOVS.L Ds,@A–s`
	`MOVS.L Ds,@As`
	`MOVS.L Ds,@As+`
	`MOVS.L Ds,@As+Is`
				C: M0
				D: A1G
				E:M1
				F:A0G
						0
						0
						1
						1
							0
							1
							0
							1
								1	1

Note: * System reserved code

4.6.2 Parallel Processing Instructions

Parallel processing instructions are used by the SH-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 4.12 lists the field A parallel data transfer instructions; figure 4.14 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 4.10.

Table 4.12 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	

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`
	`MOVX.W  @Ax+,Dx`
	`MOVX.W  @Ax+Ix,Dx`
			0		0
					1
					1
						1
						0
						1			

	`MOVX.W  Da,@Ax`
	`MOVX.W  Da,@Ax+`
	`MOVX.W  Da,@Ax+Ix`
			1		0
					1
					1
						1
						0
						1			

Y memory data transfers	`NOPY`	0		0			0	0	

	`MOVY.W  @Ay,Dy`
	`MOVY.W  @Ay+,Dy`
	`MOVY.W  @Ay+Iy,Dy`
		Dy		0			0
							1
							1
								1
								0
								1	

	`MOVY.W  Da,@Ay`
	`MOVY.W  Da,@Ay+`
	`MOVY.W  Da,@Ay+Iy`
		Da		1			0
							1
							1
								1
								0
								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

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										
	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		0:Y0 1:Y1		0:X0 1:X1		0:Y0 1:Y1		0:M0 1:M1		0:X0 1:Y0	
	PSUB Sx, Sy, Du PMULS Se, Sf, Dg				0	1	1	0	2:Y0 3:A1		2:X0 3:A1		2:A0 3:A1		2:M0 3:M1		2:A0 3:A1		2:A0 3:A1	
	PADD Sx, Sy, Du PMULS Se, Sf, Dg				0	1	1	1												
Three operand instructions	Reserved				1	0	0 0	0 1	0	0	0	0					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)			
	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 0 1	1	1										
Ⓐ	Ⓑ		Ⓒ					Ⓓ							Ⓔ					

Figure 4.5 Field B ALU Operation Instructions and Multiplication Instructions

A	B		C			D		E			
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

Figure 4.5 Field B ALU Operation Instructions and Multiplication Instructions (cont)

4.7 ALU Fixed Decimal Point Operations

4.7.1 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 4.6 shows the ALU fixed decimal point operation flowchart.

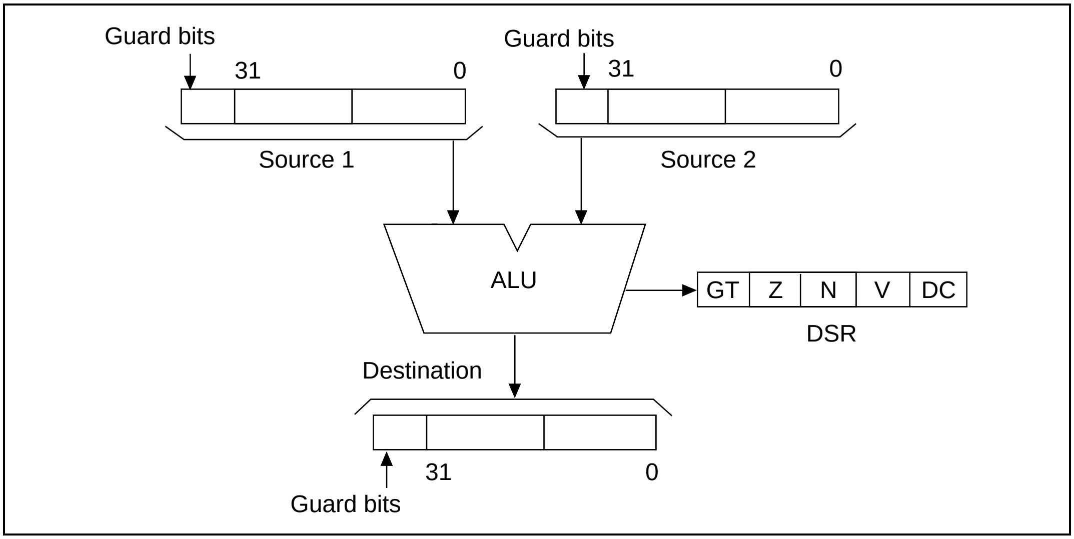


Figure 4.6 ALU Fixed Decimal Point Operation Flowchart

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 4.7 is a flowchart of the operation.

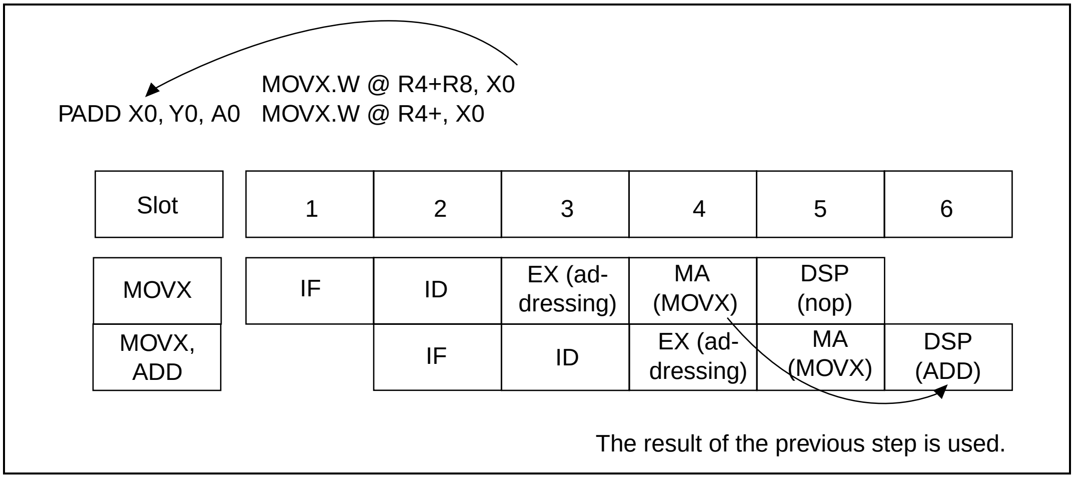


Figure 4.7 Sample Processing Flowchart

4.7.2 Instructions and Operands

Table 4.13 shows the types of ALU fixed decimal point arithmetic operations. Table 4.14 shows the correspondence between the operands and registers.

Table 4.13 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 4.14 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.

4.7.3 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 4.8 shows examples when carries and borrows occur.

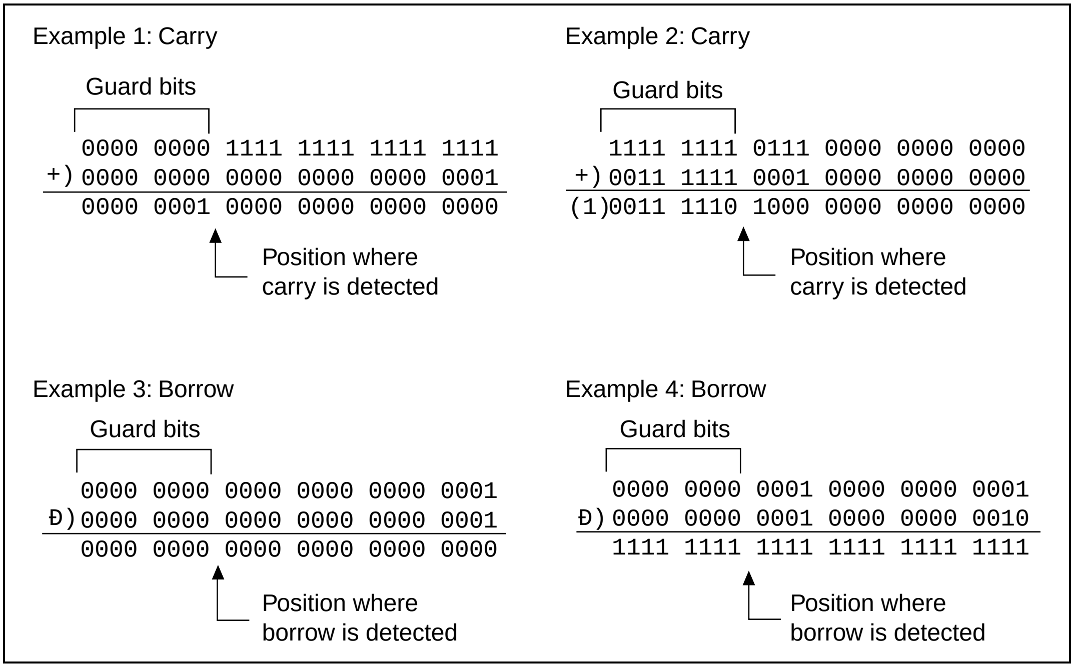


Figure 4.8 Examples of Carries and Borrows

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 4.9 shows an example of distinguishing negative from positive. In this mode, the DC bit has the same value as the condition bit N.

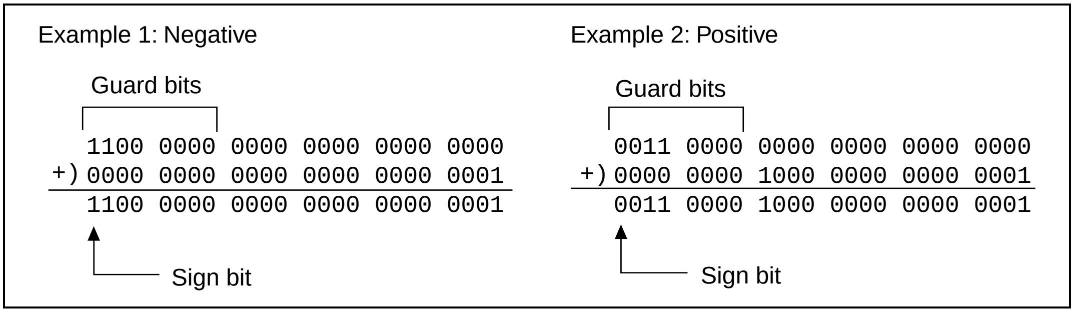


Figure 4.9 Distinguishing Negative and Positive

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 4.10 shows an example of distinguishing overflows.

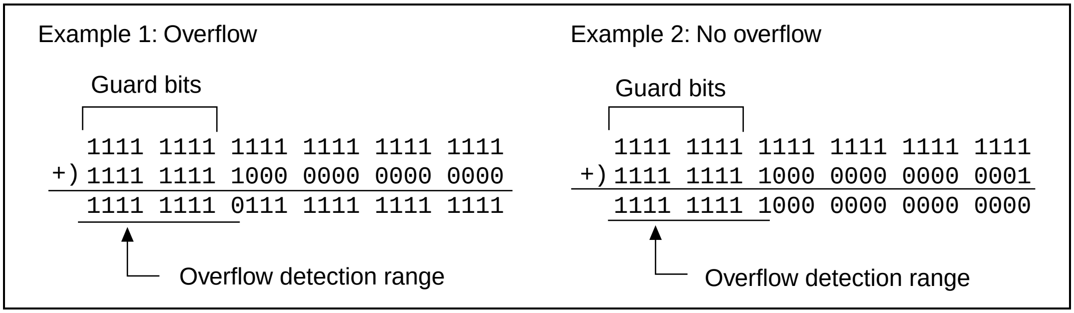


Figure 4.10 Distinguishing Overflows

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.

4.7.4 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.

4.7.5 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.

4.8 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 4.11 shows the ALU integer operation flowchart.

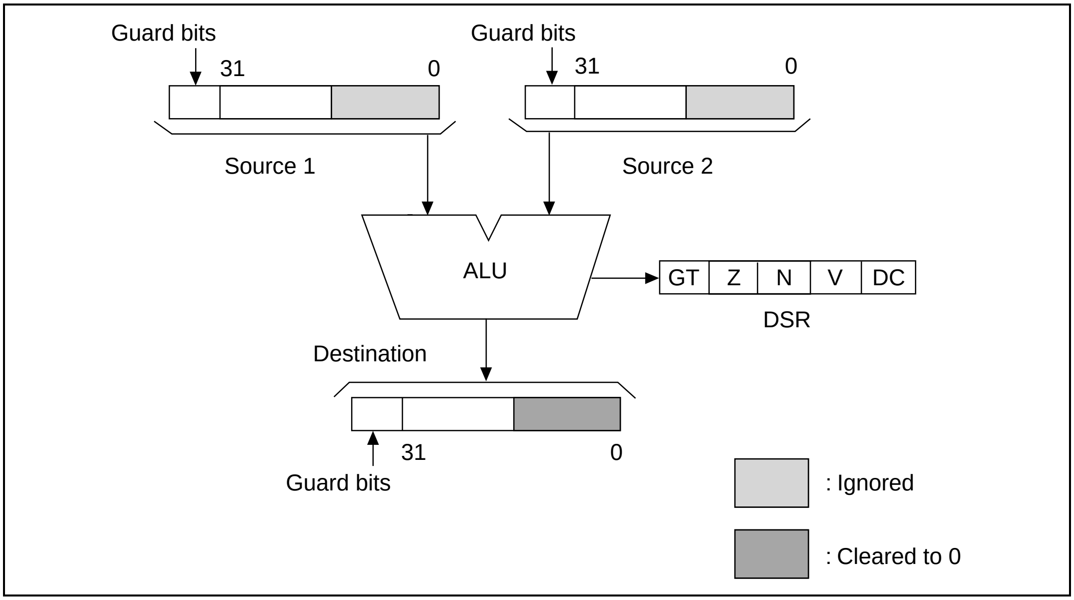


Figure 4.11 ALU Integer Operation Flowchart

Table 4.15 lists the types of ALU integer operations. Table 4.16 shows the correspondence between the operands and registers.

Table 4.15 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 4.16 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.

4.9 ALU Logical Operations

4.9.1 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 4.12 shows the ALU logical operation flowchart.

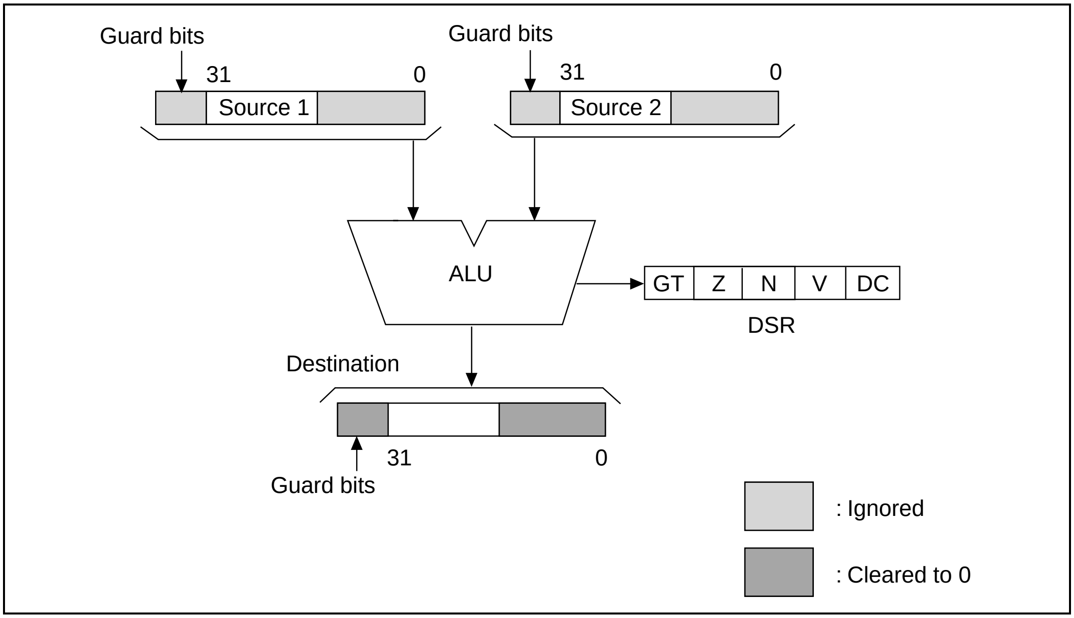


Figure 4.12 ALU Logical Operation Flowchart

4.9.2 Instructions and Operands

Table 4.17 lists the types of ALU logical arithmetic operations. Table 4.18 shows the correspondence between the operands and registers, which is the same as for ALU fixed decimal point operations.

Table 4.17 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 4.18 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.

4.9.3 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.

4.9.4 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.

4.10 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 4.13 shows a flowchart of fixed decimal point multiplication.

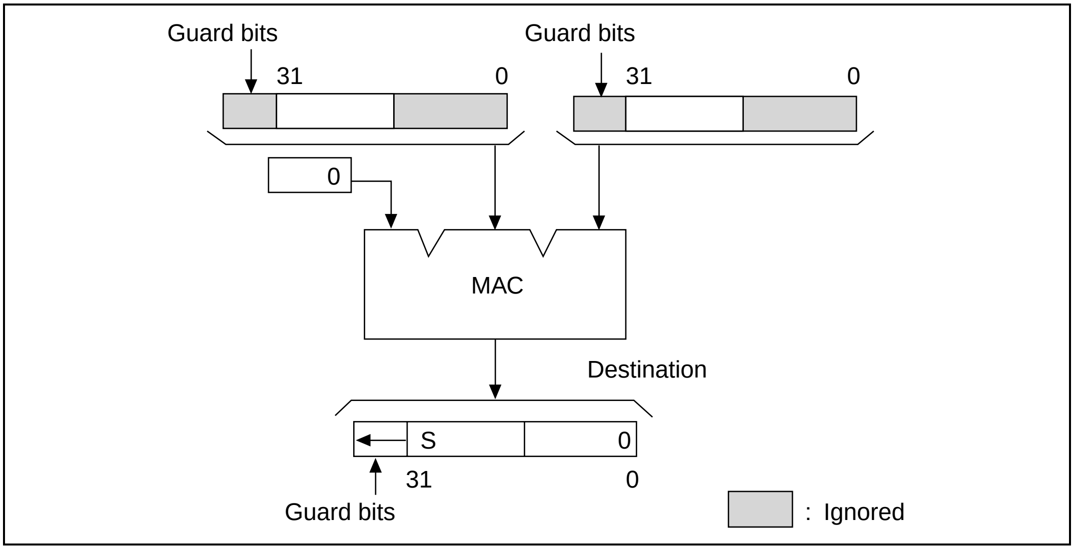


Figure 4.13 Fixed Decimal Point Multiplication Flowchart

Table 4.19 shows the fixed decimal point multiplication instruction. Table 4.20 shows the correspondence between the operands and registers.

Table 4.19 Fixed Decimal Point Multiplication

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

Table 4.20 Correspondence between Operands and Registers for Fixed Decimal Point Multiplication

Operand	X0	X1	Y0	Y1	M0	M1	A0	A1
Se	Yes	Yes	Yes					Yes
Sf	Yes		Yes	Yes				Yes
Dg					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.

4.11 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 4.21 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 4.22.

Table 4.21 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 4.22 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.

4.11.1 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 4.14 shows the arithmetic shift operation flowchart.

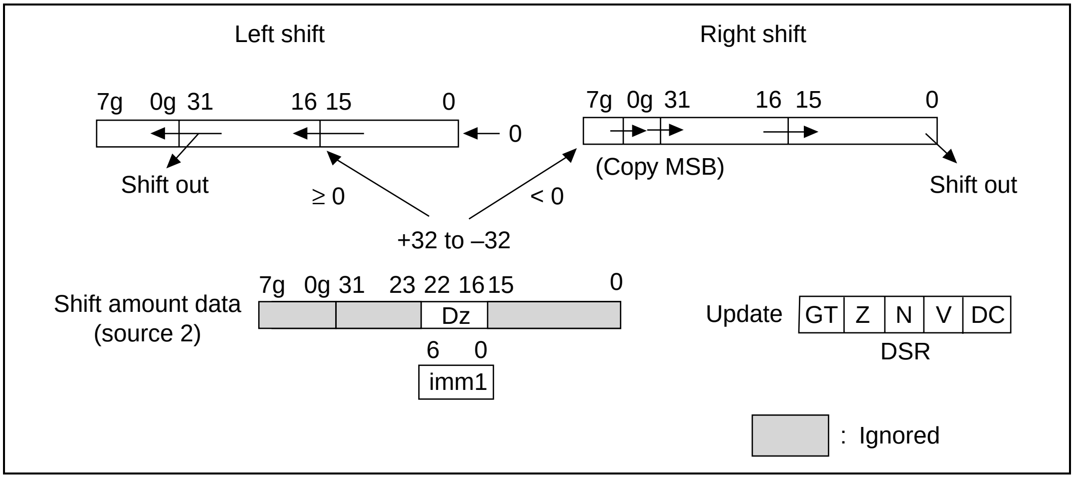


Figure 4.14 Arithmetic Shift Operation Flowchart

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.

4.11.2 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 4.15 shows the logical shift operation flowchart.

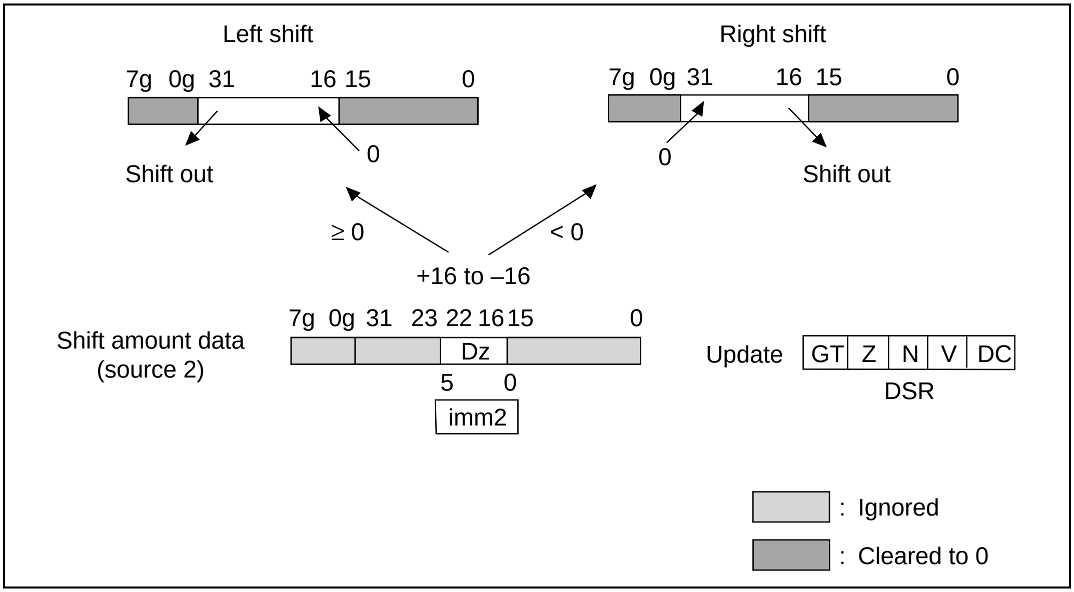


Figure 4.15 Logical Shift Operation Flowchart

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.

4.12 The MSB Detection Instruction

4.12.1 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 4.16 shows the MSB detection instruction flowchart. Table 4.24 shows the relationship between source data and destination data.

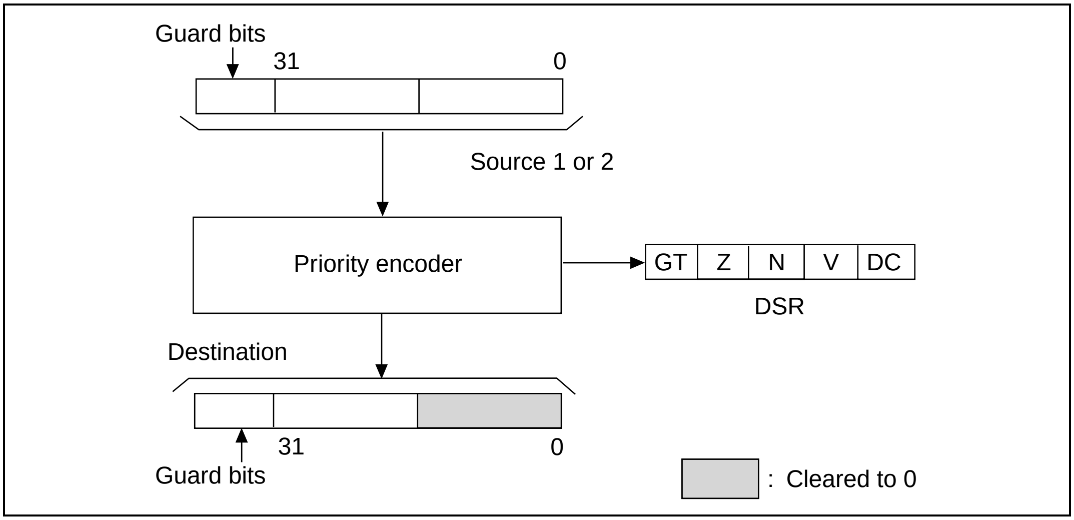


Figure 4.16   MSB Detection Flowchart

Table 4.23 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

	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.

4.12.2 Instructions and Operands

Table 4.24 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 4.25.

Table 4.24 MSB Detection Instruction

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

Table 4.25 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.

4.12.3 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.

4.12.4 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.

4.13 Rounding

4.13.1 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 4.17 shows the rounding flowchart. Figure 4.18 shows the rounding process definitions.

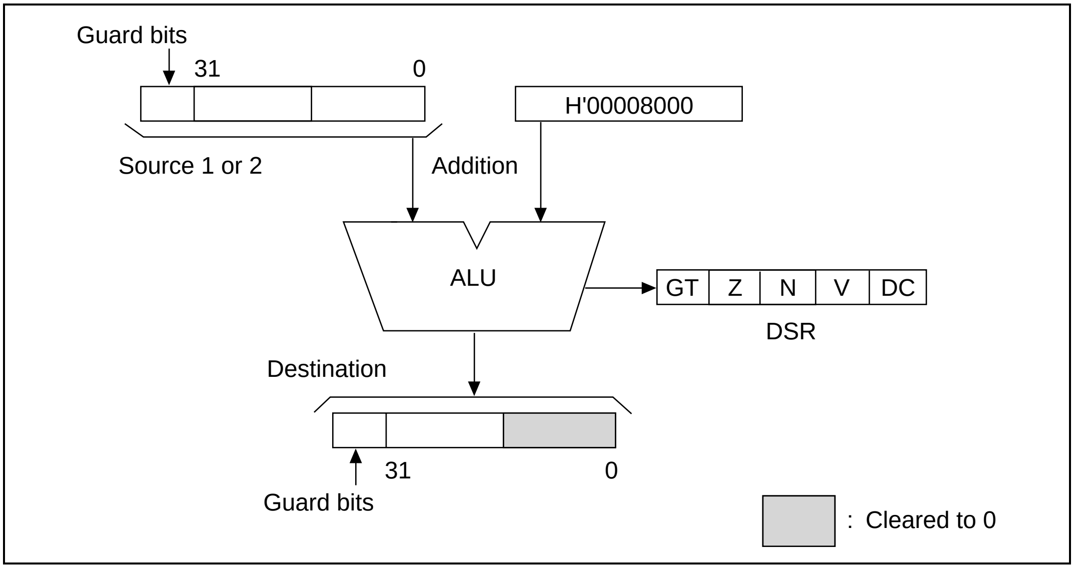


Figure 4.17 Rounding Flowchart

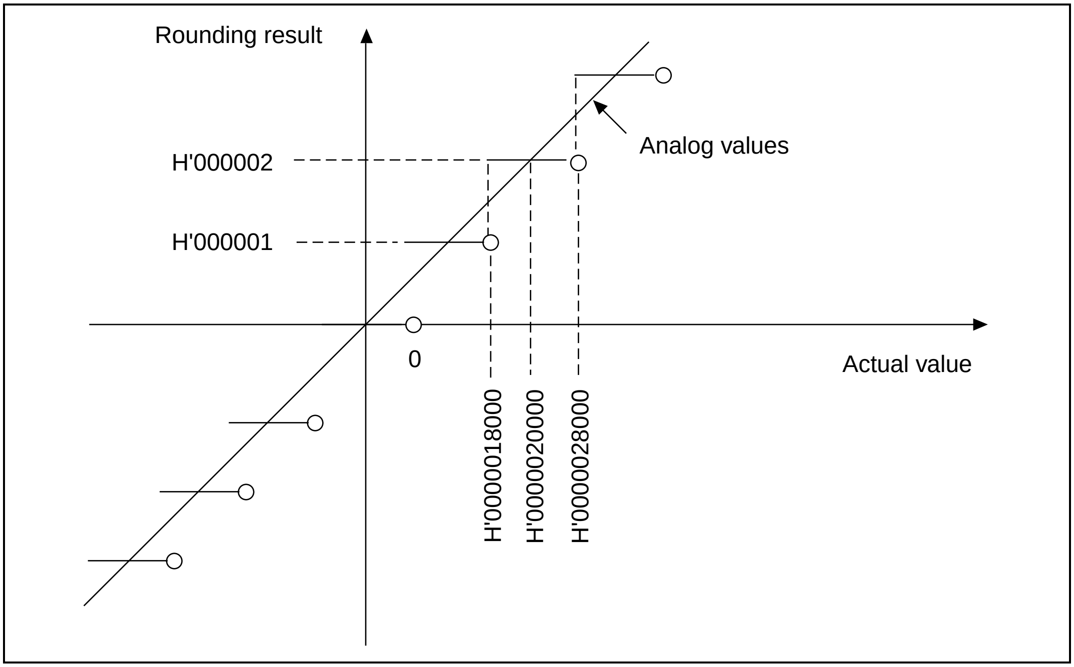


Figure 4.18 Rounding Process Definitions

4.13.2 Instructions and Operands

Table 4.26 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 4.27.

Table 4.26 Rounding Instruction

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

Table 4.27 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.

4.13.3 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.

4.13.4 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.

4.13.5 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.

4.14 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 4.28 shows the relationship between the CS bits and the DC bit.

Table 4.28 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		

4.15 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 existing SH-1 and SH-2. 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 4.29 shows the overflow definitions for fixed decimal point arithmetic operations. Table 4.30 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 4.29 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 4.30 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.

4.16 Data Transfers

The SH-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 IDB bus

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

Table 4.31 shows the various functions.

Table 4.31 Data Transfer Functions

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

4.16.1 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.

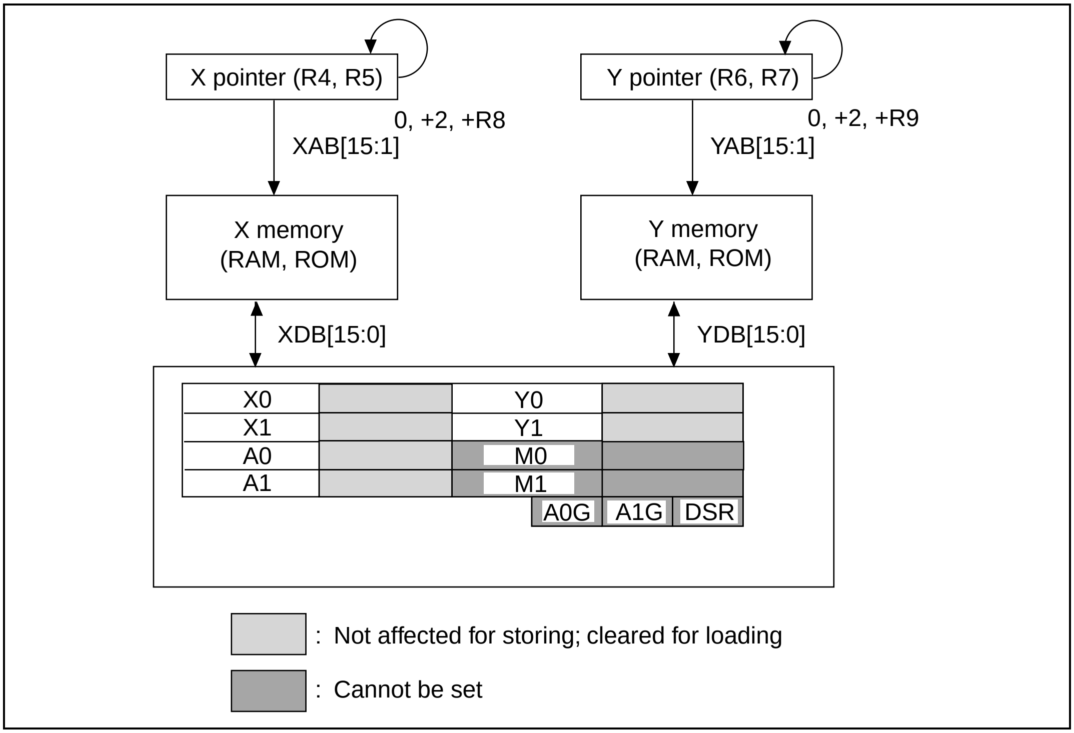


Figure 4.19 Flowchart of X and Y Memory Data Transfers

4.16.2 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 IAB and IDB buses in place of the X bus and Y bus, so contention occurs on the IDB 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 IDB 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.

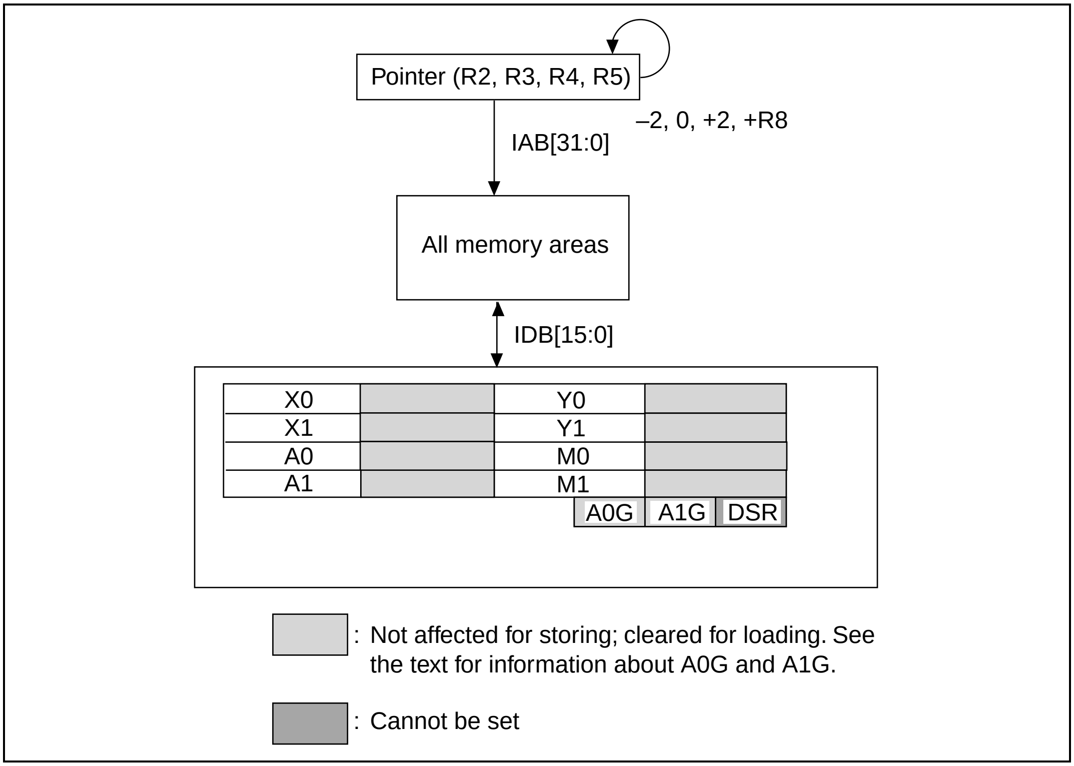


Figure 4.20 Single Data Transfer Flowchart (Word)

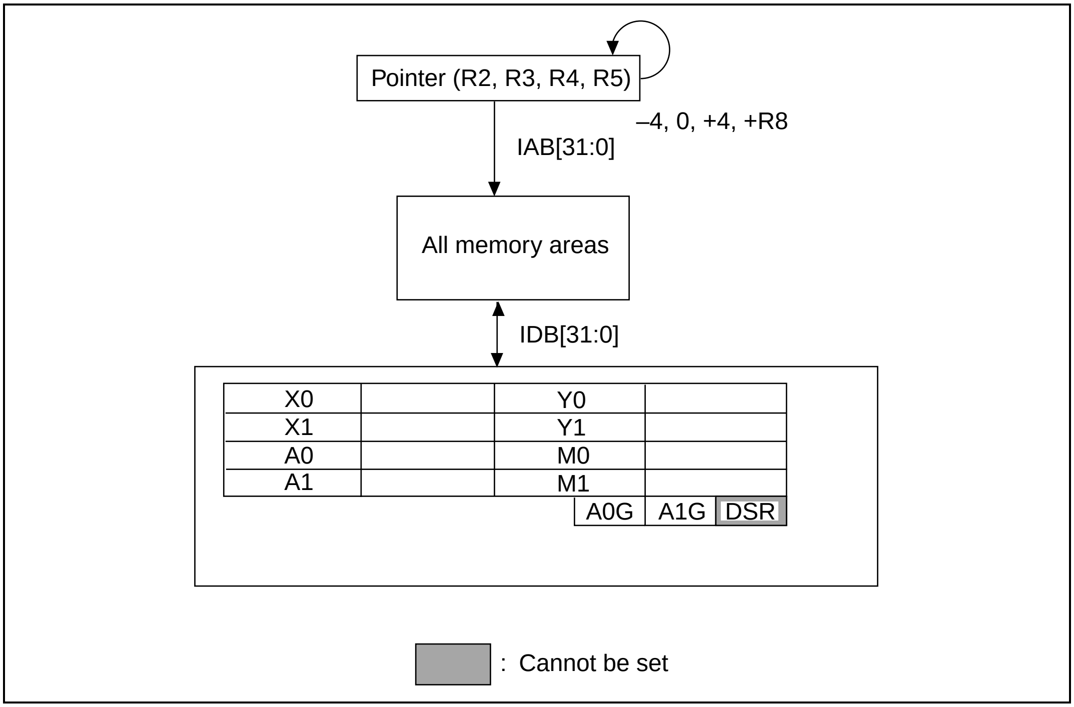


Figure 4.21 Single Data Transfer Flowchart (Longword)

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 4.22 shows an example.

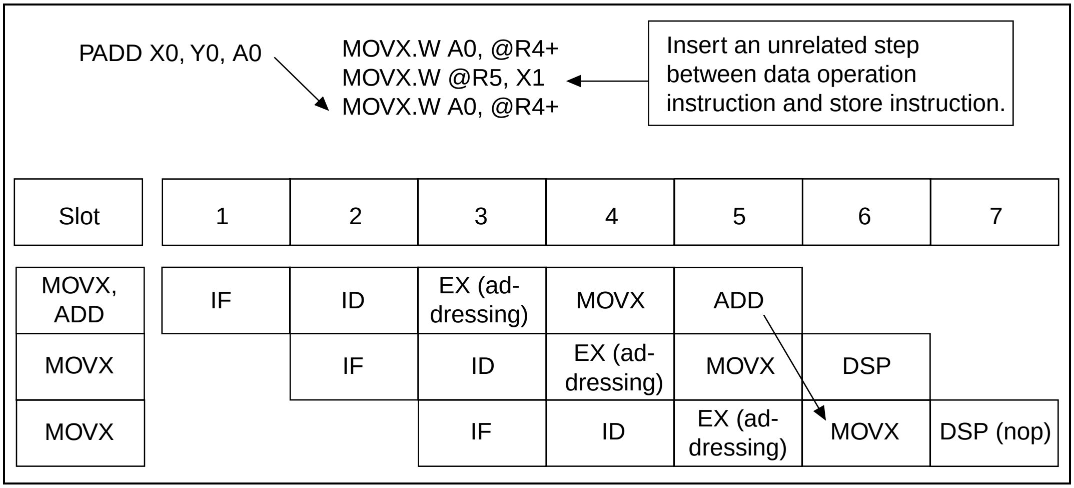


Figure 4.22 Example of the Execution of Operation and Data Store Instructions

4.17 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 4.32 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 4.32 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

4.18 DSP Repeat (Loop) Control

The SH-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. See 4.19.1 Notes.) 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 they are described, error exemption processing is started and the address values shown in table 4.33 are pushed out to the stack area pointed by R15.

Table 4.33 PC Values Pushed Out (1)

Conditions	Position	Address Pushed Out
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 they are described, error exception processing is started and the address values shown in table 4.34 are pushed out to the stack area pointed by R15. 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 secured.

Table 4.34 PC Values Pushed Out (2)

Conditions	Position	Address Pushed Out
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 SH-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 the branch destination is “RptStart” or any address ahead of it, content of RC in the SR register is not updated.
10. While the repeat is being executed, interruption is restricted. Figure 4.23 shows the flow for each stage of EX. The initial EX stage of interruption or the bus error exception is usually started immediately after the EX stage of the instruction is completed (indicated by “A”). However, in the EX stage of the next instr0, only the bus error exception can be designated by “B” to continue. At the EX stage of instr1, neither interruption nor bus exception can be continued by “C”. Only the EX stage of instr2 can be continued.

```
      A:  All interruption and bus error exceptions are accepted.
      B:  Only the bus error exception is accepted.
      C:  No interruption and bus error exceptions are 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  ← C          Start:         instr1  ← C          Start:          instr1  ← C
                 instr2  ← A          End:           instr2  ← C                          instr2  ← C
                                                     instr3  ← A          End:            instr3  ← C
                                                                                          instr4  ← A

   More than 4 steps repeat

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


   When RC=0:  All interruptions and bus errors are accepted.
```

Figure 4.23 Restriction on Acceptance of Interruption by Repeat Module

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

Table 4.35 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;(RptStart)
          LDRE   RptStart0+4;(RptStart)
          SETRC  RptCount;
             ----
RptStart0:instr0;
RtpStart: instr1;   Repeat instruction
          instr2;
```

Case 2: Two repeat instructions

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

Case 3: Three repeat instructions

```
           LDRS   RptStart0+4;(RptStart)
           LDRE   RptStart0+4;(RptEnd)
           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;(RptEnd)
           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 delicate handling of the labeling and offset described in table 4.34 and Note 1. 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, RptStart, 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, RptStart, RptCount
             ----
          instr0;
RtpStart: instr1;   Repeat instruction 1
          instr2;   Repeat instruction 2
          instr3;   Repeat instruction 3
-----------------------------------------
          instrN-3; Repeat instruction N-3
          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.

4.19 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'00008233, R1=H'00000004
                   (R4)=H'1111, (R6)=H'2222
After execution:   X0=H'11110000, Y0= H'55555555, A0=H'00888888,
                   R4=H'00008002, R6=H'00008237, 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'00008233, 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'00008237, R1=H'00000004
                   (R4)=H'1111, (R6)=H'1234
```

Section 5 Instruction Set

The SH-DSP instructions are divided into three groups. CPU instructions are executed by the CPU core, and DSP data transfer instructions and DSP operation instructions are executed by the DSP unit. Some CPU instructions support DSP functions. The description of the instruction set is divided into these three groups.

5.1 Instruction Set for CPU Instructions

Table 5.1 lists instructions by classification.

Table 5.1 Classification of CPU Instructions

				Applicable Instructions			
Classification	**Types**	**Operation Code**	**Function**	**SH-1**	**SH-2**	**SH-DSP**	**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	○	○	○	
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	○	○	○	

Classification	Types	Operation Code	Function	Applicable Instructions			No. of Instructions
				SH-1	SH-2	SH-DSP	
Arithmetic operations (cont)		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	○	○	○	
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	10	ROTCL	One-bit left rotation with T bit	○	○	○	14
		ROTCR	One-bit right rotation with T bit	○	○	○	
		ROTL	One-bit left rotation	○	○	○	
		ROTR	One-bit right rotation	○	○	○	
		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	○	○	○	

				Applicable Instructions			
Classification	**Types**	**Operation Code**	**Function**	**SH-1**	**SH-2**	**SH-DSP**	**No. of Instructions**
Branch	9	BF	Conditional branch (T = 0)	○	○	○	11
			Conditional branch with delay	—	○	○	
		BT	Conditional branch (T = 1)	○	○	○	
			Conditional branch with delay	—	○	○	
		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	○	○	○	
System control	14	CLRMAC	MAC register clear	○	○	○	71
		CLRT	T bit clear	○	○	○	
		LDC	Load to control register	○	○	○	
		LDRE	Load to repeat end register	—	—	○	
		LDRS	Load to repeat start register	—	—	○	
		LDS	Load to system register	○	○	○	
		NOP	No operation	○	○	○	
		RTE	Return from exception processing	○	○	○	
		SETRC	Set number of repeats	—	—	○	
		SETT	T bit set	○	○	○	
		SLEEP	Shift into power-down state	○	○	○	
		STC	Storing control register data	○	○	○	
		STS	Storing system register data	○	○	○	
		TRAPA	Trap exception handling	○	○	○	
Total:	65						182

Instruction codes, operation, and execution cycles are listed as shown in table 10.2 by classification.

Table 5.2 Instruction Code Format

Item	Format	Explanation
Instruction mnemonic	OP.Sz SRC,DEST	OP: Operation code Sz: Size SRC: Source DEST: Destination Rm: Source register Rn: Destination register imm: Immediate data disp: Displacement*[1]
Instruction code	MSB ↔ LSB	mmmm: Source register nnnn: Destination register 0000: R0 0001: R1 .... 1111: R15 iiii: Immediate data dddd: Displacement
Operation summary	→, ← (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
Execution cycles		Value when no wait states are inserted*[2]
Instruction execution 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	—: No change	Value of T bit after instruction is executed

Notes: 1. Scaled (×1, ×2, or ×4) according to the size of the instruction's operand. For more information, see section 12, Instruction Descriptions.

2. Instruction execution cycles: The executions cycles shown in the table are minimums. The actual number of cycles may be increased when (1) 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.

5.1.1 Data Transfer Instructions

Table 5.3 Data Transfer Instructions

Instruction		Operation	Cycles	T Bit	Applicable Instructions		
					SH-1	SH-2	SH-DSP
`MOV`	`#imm,Rn`	imm → Sign extension → Rn	1	—	○	○	○
`MOV.W`	`@(disp,PC),Rn`	(disp × 2 + PC) → Sign extension → Rn	1	—	○	○	○
`MOV.L`	`@(disp,PC),Rn`	(disp × 4 + PC) → Rn	1	—	○	○	○
`MOV`	`Rm,Rn`	Rm → Rn	1	—	○	○	○
`MOV.B`	`Rm,@Rn`	Rm → (Rn)	1	—	○	○	○
`MOV.W`	`Rm,@Rn`	Rm → (Rn)	1	—	○	○	○
`MOV.L`	`Rm,@Rn`	Rm → (Rn)	1	—	○	○	○
`MOV.B`	`@Rm,Rn`	(Rm) → Sign extension → Rn	1	—	○	○	○
`MOV.W`	`@Rm,Rn`	(Rm) → Sign extension → Rn	1	—	○	○	○
`MOV.L`	`@Rm,Rn`	(Rm) → Rn	1	—	○	○	○
`MOV.B`	`Rm,@–Rn`	Rn–1 → Rn, Rm → (Rn)	1	—	○	○	○
`MOV.W`	`Rm,@–Rn`	Rn–2 → Rn, Rm → (Rn)	1	—	○	○	○
`MOV.L`	`Rm,@–Rn`	Rn–4 → Rn, Rm → (Rn)	1	—	○	○	○
`MOV.B`	`@Rm+,Rn`	(Rm) → Sign extension → Rn, Rm + 1 → Rm	1	—	○	○	○
`MOV.W`	`@Rm+,Rn`	(Rm) → Sign extension → Rn, Rm + 2 → Rm	1	—	○	○	○
`MOV.L`	`@Rm+,Rn`	(Rm) → Rn, Rm + 4 → Rm	1	—	○	○	○
`MOV.B`	`R0,@(disp,Rn)`	R0 → (disp + Rn)	1	—	○	○	○
`MOV.W`	`R0,@(disp,Rn)`	R0 → (disp × 2 + Rn)	1	—	○	○	○
`MOV.L`	`Rm,@(disp,Rn)`	Rm → (disp × 4 + Rn)	1	—	○	○	○
`MOV.B`	`@(disp,Rm),R0`	(disp + Rm) → Sign extension → R0	1	—	○	○	○
`MOV.W`	`@(disp,Rm),R0`	(disp × 2 + Rm) → Sign extension → R0	1	—	○	○	○
`MOV.L`	`@(disp,Rm),Rn`	(disp × 4 + Rm) → Rn	1	—	○	○	○

Instruction	Operation	Cycles	T Bit	Applicable Instructions SH-1	SH-2	SH-DSP
`MOV.B Rm,@(R0,Rn)`	Rm → (R0 + Rn)	1	—	○	○	○
`MOV.W Rm,@(R0,Rn)`	Rm → (R0 + Rn)	1	—	○	○	○
`MOV.L Rm,@(R0,Rn)`	Rm → (R0 + Rn)	1	—	○	○	○
`MOV.B @(R0,Rm),Rn`	(R0 + Rm) → Sign extension → Rn	1	—	○	○	○
`MOV.W @(R0,Rm),Rn`	(R0 + Rm) → Sign extension → Rn	1	—	○	○	○
`MOV.L @(R0,Rm),Rn`	(R0 + Rm) → Rn	1	—	○	○	○
`MOV.B R0,@(disp,GBR)`	R0 → (disp + GBR)	1	—	○	○	○
`MOV.W R0,@(disp,GBR)`	R0 → (disp × 2 + GBR)	1	—	○	○	○
`MOV.L R0,@(disp,GBR)`	R0 → (disp × 4 + GBR)	1	—	○	○	○
`MOV.B @(disp,GBR),R0`	(disp + GBR) → Sign extension → R0	1	—	○	○	○
`MOV.W @(disp,GBR),R0`	(disp × 2 + GBR) → Sign extension → R0	1	—	○	○	○
`MOV.L @(disp,GBR),R0`	(disp × 4 + GBR) → R0	1	—	○	○	○
`MOVA @(disp,PC),R0`	disp × 4 + PC → R0	1	—	○	○	○
`MOVT Rn`	T → Rn	1	—	○	○	○
`SWAP.B Rm,Rn`	Rm → Swap the bottom two bytes → REG	1	—	○	○	○
`SWAP.W Rm,Rn`	Rm → Swap two consecutive words → Rn	1	—	○	○	○
`XTRCT Rm,Rn`	Rm: Middle 32 bits of Rn → Rn	1	—	○	○	○

5.1.2 Arithmetic Instructions

Table 5.4 Arithmetic Instructions

Instruction		Operation	Cycles	T Bit	Applicable Instructions		
					SH-1	SH-2	SH-DSP
`ADD`	`Rm,Rn`	Rn + Rm → Rn	1	—	○	○	○
`ADD`	`#imm,Rn`	Rn + imm → Rn	1	—	○	○	○
`ADDC`	`Rm,Rn`	Rn + Rm + T → Rn, Carry → T	1	Carry	○	○	○
`ADDV`	`Rm,Rn`	Rn + Rm → Rn, Overflow → T	1	Overflow	○	○	○
`CMP/EQ`	`#imm,R0`	If R0 = imm, 1 → T, If R0 ≠ imm, 0 → T	1	Comparison result	○	○	○
`CMP/EQ`	`Rm,Rn`	If Rn = Rm, 1 → T, If Rn ≠ Rm, 0 → T	1	Comparison result	○	○	○
`CMP/HS`	`Rm,Rn`	If Rn ≥ Rm with unsigned data, 1 → T, If Rn < Rm, 0 → T	1	Comparison result	○	○	○
`CMP/GE`	`Rm,Rn`	If Rn ≥ Rm with signed data, 1 → T, If Rn < Rm, 0 → T	1	Comparison result	○	○	○
`CMP/HI`	`Rm,Rn`	If Rn > Rm with unsigned data, 1 → T, If Rn ≤ Rm, 0 → T	1	Comparison result	○	○	○
`CMP/GT`	`Rm,Rn`	If Rn > Rm with signed data, 1 → T, If Rn ≤ Rm, 0 → T	1	Comparison result	○	○	○
`CMP/PL`	`Rn`	If Rn > 0, 1 → T, If Rn ≤ 0, 0 → T	1	Comparison result	○	○	○
`CMP/PZ`	`Rn`	If Rn ≥ 0, 1 → T, If Rn < 0, 0 → T	1	Comparison result	○	○	○
`CMP/STR`	`Rm,Rn`	If Rn and Rm have an equivalent byte, 1 → T, If not equivalent byte, 0 → T	1	Comparison result	○	○	○
`DIV1`	`Rm,Rn`	Single-step division (Rn/Rm)	1	Calculation result	○	○	○

Instruction		Operation	Cycles	T Bit	Applicable Instructions		
					SH-1	SH-2	SH-DSP
`DIV0S`	`Rm,Rn`	MSB of Rn → Q, MSB of Rm → M, M ^ Q → T	1	Calculation result	○	○	○
`DIV0U`		0 → M/Q/T	1	0	○	○	○
`DMULS.L`	`Rm,Rn`	Signed operation of Rn × Rm → MACH, MACL 32 × 32 → 64 bits	2–4*	—	—	○	○
`DMULU.L`	`Rm,Rn`	Unsigned operation of Rn × Rm → MACH, MACL 32 × 32 → 64 bits	2–4*	—	—	○	○
`DT`	`Rn`	Rn – 1 → Rn, if Rn = 0, 1 → T, else 0 → T	1	Comparison result	—	○	○
`EXTS.B`	`Rm,Rn`	A byte in Rm is sign-extended → Rn	1	—	○	○	○
`EXTS.W`	`Rm,Rn`	A word in Rm is sign-extended → Rn	1	—	○	○	○
`EXTU.B`	`Rm,Rn`	A byte in Rm is zero-extended → Rn	1	—	○	○	○
`EXTU.W`	`Rm,Rn`	A word in Rm is zero-extended → Rn	1	—	○	○	○
`MAC.L`	`@Rm+,@Rn+`	Signed operation of (Rn) × (Rm) + MAC → MAC	3/(2–4)*	—	—	○	○
`MAC.W`	`@Rm+,@Rn+`	Signed operation of (Rn) × (Rm) + MAC → MAC (SH-2) 16 × 16 + 64 → 64 bits (SH-1) 16 × 16 + 42 → 42 bits	3/(2)*	—	○	○	○
`MUL.L`	`Rm,Rn`	Rn × Rm → MACL 32 × 32 → 32 bits	2–4*	—	—	○	○
`MULS.W`	`Rm,Rn`	Signed operation of Rn × Rm → MAC 16 × 16 → 32 bits	1–3*	—	○	○	○
`MULU.W`	`Rm,Rn`	Unsigned operation of Rn × Rm → MAC 16 × 16 → 32 bits	1–3*	—	○	○	○

Instruction		Operation	Cycles	T Bit	Applicable Instructions		
					SH-1	SH-2	SH-DSP
`NEG`	`Rm,Rn`	0–Rm → Rn	1	—	○	○	○
`NEGC`	`Rm,Rn`	0–Rm–T → Rn, Borrow → T	1	Borrow	○	○	○
`SUB`	`Rm,Rn`	Rn–Rm → Rn	1	—	○	○	○
`SUBC`	`Rm,Rn`	Rn–Rm–T → Rn, Borrow → T	1	Borrow	○	○	○
`SUBV`	`Rm,Rn`	Rn–Rm → Rn, Underflow → T	1	Underflow	○	○	○

Note: * The normal minimum number of execution cycles. (The number in parentheses is the number of cycles when there is contention with following instructions.)

5.1.3 Logic Operation Instructions

Table 5.5 Logic Operation Instructions

Instruction		Operation	Cycles	T Bit	Applicable Instructions		
					SH-1	SH-2	SH-DSP
`AND`	`Rm,Rn`	Rn & Rm → Rn	1	—	○	○	○
`AND`	`#imm,R0`	R0 & imm → R0	1	—	○	○	○
`AND.B`	`#imm,@(R0,GBR)`	(R0 + GBR) & imm → (R0 + GBR)	3	—	○	○	○
`NOT`	`Rm,Rn`	~Rm → Rn	1	—	○	○	○
`OR`	`Rm,Rn`	Rn | Rm → Rn	1	—	○	○	○
`OR`	`#imm,R0`	R0 | imm → R0	1	—	○	○	○
`OR.B`	`#imm,@(R0,GBR)`	(R0 + GBR) | imm → (R0 + GBR)	3	—	○	○	○
`TAS.B`	`@Rn`	If (Rn) is 0, 1 → T; if not 0, 0 → T. Also, 1 → MSB of (Rn) regardless of value of (Rn)	4	Test result	○	○	○
`TST`	`Rm,Rn`	Rn & Rm; if the result is 0, 1 → T, If not 0, 0 → T	1	Test result	○	○	○
`TST`	`#imm,R0`	R0 & imm; if the result is 0, 1 → T, If not 0, 0 → T	1	Test result	○	○	○
`TST.B`	`#imm,@(R0,GBR)`	(R0 + GBR) & imm; if the result is 0, 1 → T, If not 0, 0 → T	3	Test result	○	○	○
`XOR`	`Rm,Rn`	Rn ^ Rm → Rn	1	—	○	○	○
`XOR`	`#imm,R0`	R0 ^ imm → R0	1	—	○	○	○
`XOR.B`	`#imm,@(R0,GBR)`	(R0 + GBR) ^ imm → (R0 + GBR)	3	—	○	○	○

5.1.4 Shift Instructions

Table 5.6 Shift Instructions

					Applicable Instructions		
Instruction		**Operation**	**Cycles**	**T Bit**	**SH-1**	**SH-2**	**SH-DSP**
`ROTL`	Rn	T ← Rn ← MSB	1	MSB	○	○	○
`ROTR`	Rn	LSB → Rn → T	1	LSB	○	○	○
`ROTCL`	Rn	T ← Rn ← T	1	MSB	○	○	○
`ROTCR`	Rn	T → Rn → T	1	LSB	○	○	○
`SHAL`	Rn	T ← Rn ← 0	1	MSB	○	○	○
`SHAR`	Rn	MSB → Rn → T	1	LSB	○	○	○
`SHLL`	Rn	T ← Rn ← 0	1	MSB	○	○	○
`SHLR`	Rn	0 → Rn → T	1	LSB	○	○	○
`SHLL2`	Rn	Rn << 2 → Rn	1	—	○	○	○
`SHLR2`	Rn	Rn >> 2 → Rn	1	—	○	○	○
`SHLL8`	Rn	Rn << 8 → Rn	1	—	○	○	○
`SHLR8`	Rn	Rn >> 8 → Rn	1	—	○	○	○
`SHLL16`	Rn	Rn << 16 → Rn	1	—	○	○	○
`SHLR16`	Rn	Rn >> 16 → Rn	1	—	○	○	○

5.1.5 Branch Instructions

Table 5.7 Branch Instructions

				Applicable Instructions		
Instruction	**Operation**	**Cycles**	**T Bit**	**SH-1**	**SH-2**	**SH-DSP**
`BF label`	If T = 0, disp × 2 + PC → PC; if T = 1, nop (where label is disp + PC)	3/1*	—	○	○	○
`BF/S label`	Delayed branch, if T = 0, disp × 2 + PC → PC; if T = 1, nop	2/1*		—	○	○
`BT label`	Delayed branch, if T = 1, disp × 2 + PC → PC; if T = 0, nop	3/1*	—	○	○	○
`BT/S label`	If T = 1, disp × 2 + PC → PC; if T = 0, nop	2/1*	—	—	○	○
`BRA label`	Delayed branch, disp × 2 + PC → PC	2	—	○	○	○
`BRAF Rm`	Delayed branch, Rm + PC → PC	2	—	—	○	○
`BSR label`	Delayed branch, PC → PR, disp × 2 + PC → PC	2	—	○	○	○
`BSRF Rm`	Delayed branch, PC → PR, Rm + PC → PC	2	—	—	○	○
`JMP @Rm`	Delayed branch, Rm → PC	2	—	○	○	○
`JSR @Rm`	Delayed branch, PC → PR, Rm → PC	2	—	○	○	○
`RTS`	Delayed branch, PR → PC	2	—	○	○	○

Note: * One state when it does not branch.

5.1.6 System Control Instructions

Table 5.8 System Control Instructions

Instruction		Operation	Cycles	T Bit	Applicable Instructions		
					SH-1	SH-2	SH-DSP
`CLRMAC`		0→MACH,MACL	1	—	○	○	○
`CLRT`		0→T	1	0	○	○	○
`LDC`	`Rm,SR`	Rm→SR	1	LSB	○	○	○
`LDC`	`Rm,GBR`	Rm→GBR	1	—	○	○	○
`LDC`	`Rm,VBR`	Rm→VBR	1	—	○	○	○
`LDC`	`Rm,MOD`	Rm→MOD	1	—	—	—	○
`LDC`	`Rm,RE`	Rm→RE	1	—	—	—	○
`LDC`	`Rm,RS`	Rm→RS	1	—	—	—	○
`LDC.L`	`@Rm+,SR`	(Rm)→SR,Rm+4→Rm	3	LSB	○	○	○
`LDC.L`	`@Rm+,GBR`	(Rm)→GBR,Rm+4→Rm	3	—	○	○	○
`LDC.L`	`@Rm+,VBR`	(Rm)→VBR,Rm+4→Rm	3	—	○	○	○
`LDC.L`	`@Rm+,MOD`	(Rm)→MOD,Rm+4→Rm	3	—	—	—	○
`LDC.L`	`@Rm+,RE`	(Rm)→RE,Rm+4→Rm	3	—	—	—	○
`LDC.L`	`@Rm+,RS`	(Rm)→RS,Rm+4→Rm	3	—	—	—	○
`LDRE`	`@(disp,PC)`	disp × 2+PC→RE	1	—	—	—	○
`LDRS`	`@(disp,PC)`	disp × 2+PC→RS	1	—	—	—	○
`LDS`	`Rm,MACH`	Rm→MACH	1	—	○	○	○
`LDS`	`Rm,MACL`	Rm→MACL	1	—	○	○	○
`LDS`	`Rm,PR`	Rm→PR	1	—	○	○	○
`LDS`	`Rm,DSR`	Rm→DSR	1	—	—	—	○
`LDS`	`Rm,A0`	Rm→A0	1	—	—	—	○
`LDS`	`Rm,X0`	Rm→X0	1	—	—	—	○
`LDS`	`Rm,X1`	Rm→X1	1	—	—	—	○
`LDS`	`Rm,Y0`	Rm→Y0	1	—	—	—	○
`LDS`	`Rm,Y1`	Rm→Y1	1	—	—	—	○
`LDS.L`	`@Rm+,MACH`	(Rm)→MACH,Rm+4→Rm	1	—	○	○	○
`LDS.L`	`@Rm+,MACL`	(Rm)→MACL,Rm+4→Rm	1	—	○	○	○

Instruction		Operation	Cycles	T Bit	Applicable Instructions		
					SH-1	SH-2	SH-DSP
LDS.L	@Rm+,PR	(Rm)→PR,Rm+4→Rm	1	—	○	○	○
LDS.L	@Rm+,DSR	(Rm)→DSR,Rm+4→Rm	1	—	—	—	○
LDS.L	@Rm+,A0	(Rm)→A0,Rm+4→Rm	1	—	—	—	○
LDS.L	@Rm+,X0	(Rm)→X0,Rm+4→Rm	1	—	—	—	○
LDS.L	@Rm+,X1	(Rm)→X1,Rm+4→Rm	1	—	—	—	○
LDS.L	@Rm+,Y0	(Rm)→Y0,Rm+4→Rm	1	—	—	—	○
LDS.L	@Rm+,Y1	(Rm)→Y1,Rm+4→Rm	1	—	—	—	○
NOP		No operation	1	—	○	○	○
RTE		Delayed branch, stack area,→PC/SR	4	LSB	○	○	○
SETRC	Rn	Rn[11:0]→RC (SR[27:16])	1	—	—	—	○
SETRC	#imm	imm→RC(SR[23:16]), zeros→SR[27:24]	1	—	—	—	○
SETT		1→T	1	—	○	○	○
SLEEP		Sleep	3*	—	○	○	○
STC	SR,Rn	SR→Rn	1	—	○	○	○
STC	GBR,Rn	GBR→Rn	1	—	○	○	○
STC	VBR,Rn	VBR→Rn	1	—	○	○	○
STC	MOD,Rn	MOD→Rn	1	—	—	—	○
STC	RE,Rn	RE→Rn	1	—	—	—	○
STC	RS,Rn	RS→Rn	1	—	—	—	○
STC.L	SR,@-Rn	Rn–4→Rn,SR→(Rn)	2	—	○	○	○
STC.L	GBR,@-Rn	Rn–4→Rn,GBR→(Rn)	2	—	○	○	○
STC.L	VBR,@-Rn	Rn–4→Rn,VBR→(Rn)	2	—	○	○	○
STC.L	MOD,@-Rn	Rn–4→Rn,MOD→(Rn)	2	—	—	—	○
STC.L	RE,@-Rn	Rn–4→Rn,RE→(Rn)	2	—	—	—	○
STC.L	RS,@-Rn	Rn–4→Rn,RS→(Rn)	2	—	—	—	○
STS	MACH,Rn	MACH→Rn	1	—	○	○	○
STS	MACL,Rn	MACL→Rn	1	—	○	○	○
STS	PR,Rn	PR→Rn	1	—	○	○	○

Instruction		Operation	Cycles	T Bit	Applicable Instructions		
					SH-1	SH-2	SH-DSP
`STS`	`DSR,Rn`	DSR→Rn	1	—	—	—	○
`STS`	`A0,Rn`	A0→Rn	1	—	—	—	○
`STS`	`X0,Rn`	X0→Rn	1	—	—	—	○
`STS`	`X1,Rn`	X1→Rn	1	—	—	—	○
`STS`	`Y0,Rn`	Y0→Rn	1	—	—	—	○
`STS`	`Y1,Rn`	Y1→Rn	1	—	—	—	○
`STS.L`	`MACH,@-Rn`	Rn–4→Rn,MACH→(Rn)	1	—	○	○	○
`STS.L`	`MACL,@-Rn`	Rn–4→Rn,MACL→(Rn)	1	—	○	○	○
`STS.L`	`PR,@-Rn`	Rn–4→Rn,PR→(Rn)	1	—	○	○	○
`STS.L`	`DSR,@-Rn`	Rn–4→Rn,DSR→(Rn)	1	—	—	—	○
`STS.L`	`A0,@-Rn`	Rn–4→Rn,A0→(Rn)	1	—	—	—	○
`STS.L`	`X0,@-Rn`	Rn–4→Rn,X0→(Rn)	1	—	—	—	○
`STS.L`	`X1,@-Rn`	Rn–4→Rn,X1→(Rn)	1	—	—	—	○
`STS.L`	`Y0,@-Rn`	Rn–4→Rn,Y0→(Rn)	1	—	—	—	○
`STS.L`	`Y1,@-Rn`	Rn–4→Rn,Y1→(Rn)	1	—	—	—	○
`TRAPA`	`#imm`	PC/SR→stack area, (imm × 4+VBR)→PC	6	—	○	○	○

Note: * 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, or when the branch destination address of a branch instruction is a 4n + 2 address.

5.1.7 CPU Instructions That Support DSP Functions

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 5.9 shows the added instructions.

Table 5.9 Added CPU Instructions

Instruction	Operation	Code	Cycles	T Bit
`LDC Rm,MOD`	Rm→MOD	`0100mmmm01011110`	1	—
`LDC Rm,RE`	Rm→RE	`0100mmmm01111110`	1	—
`LDC Rm,RS`	Rm→RS	`0100mmmm01101110`	1	—
`LDC.L @Rm+,MOD`	(Rm)→MOD,Rm+4→Rm	`0100mmmm01010111`	3	—
`LDC.L @Rm+,RE`	(Rm)→RE,Rm+4→Rm	`0100mmmm01110111`	3	—
`LDC.L @Rm+,RS`	(Rm)→RS,Rm+4→Rm	`0100mmmm01100111`	3	—
`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	—

Instruction	Operation	Code	Cycles	T Bit
`STS.L X1,@-Rn`	Rn–4→Rn,X1→(Rn)	`0100nnnn01110010`	1	—
`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]) repeat flag → RF1, RF0	`0100mmmm00010100`	1	—
`SETRC #imm`	imm→RC(SR[23:16]), zeros→SR[27:24], repeat flag → RF1, RF0	`10000010iiiiiiii`	1	—
`LDRS @(disp,pc)`	disp × 2+PC→RS	`10001100dddddddd`	1	—
`LDRE @(disp,pc)`	disp × 2+PC→RE	`10001110dddddddd`	1	—

5.2 DSP Data Transfer Instruction Set

Table 5.10 shows the DSP data transfer instructions by category.

Table 5.10 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.

5.2.1 Double Data Transfer Instructions (X Memory Data)

Table 5.11 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	—

5.2.2 Double Data Transfer Instructions (Y Memory Data)

Table 5.12 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	—

5.2.3 Single Data Transfer Instructions

Table 5.13 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 (eight-bit registers) are specified as the source operand Ds, the data is sign-extended and used.

Table 5.14 lists the correspondence between DSP data transfer operands and registers. CPU core registers are used as pointer addresses to indicate memory addresses.

Table 5.14 Correspondence between DSP Data Transfer Operands and Registers

	SuperH (CPU Core) Registers									
Operand	**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										

	DSP Registers									
Operand	**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.

5.3 DSP Operation Instruction Set

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 5.15 shows the format of DSP operation instructions. The operands are selected independently from the DSP register. Table 5.16 shows the correspondence of DSP operation instruction operands and registers.

Table 5.15 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, PMULS`

	2 operands	`ALUop. Sx, Dz`
		`ALUop. Sy, Dz`
			`PCMP, PABS, PRND`

Table 5.16 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 5.17 shows the DSP operation instructions by category.

Table 5.17 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	
			POR	Logical OR	9
			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

5.3.1 ALU Arithmetic Operation Instructions

Table 5.18 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	—

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

Table 5.19 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**********` `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	—
`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	—

Table 5.20 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	—

Table 5.21 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

5.3.2 ALU Logical Operation Instructions

Table 5.22 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	—

5.3.3 Fixed Decimal Point Multiplication Instructions

Table 5.23 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	—

5.3.4 Shift Operation Instructions

Table 5.24 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

Table 5.25 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

5.3.5 System Control Instructions

Table 5.26 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	—

5.3.6 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 5.27 shows the NOPX and NOPY instruction code.

Table 5.27 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 6 Instruction Descriptions

6.1 Instruction Descriptions

Instructions are described in alphabetical order in three sections: CPU instructions, DSP data transfer instructions, and DSP operation instructions.

This section describes instructions in alphabetical order using the format shown below in section 6.1.1. The actual descriptions begin at section 6.2.2.

6.1.1 Sample Description (Name): Classification

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

Format	Abstract	Code	Cycle	T Bit	Applicable Instructions
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	Indicates whether the instruction applies to the SH-1, SH-2, or SH-DSP.

Description: Description of operation

Notes: Notes on using the instruction

Operation: Operation written in C language. 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. When execution moves from this function to one of the following instructions and one of the listed instructions precedes it, it will be considered an illegal slot instruction (the listed instructions 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);
unsigned log IS_32bit_Inst(unsigned long Addr)
```

  If the address (Addr_4) instruction is 32-bit, 2 is returned; 0 is returned if it is 16-bit.

- 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 SRO *)(&SR)).RF10)
#define RF0 ((*struct SRO *)(&SR)).RF00)
```

- Error display function:

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

The PC should point to the location four bytes 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

Note that the SuperH Family cross assembler version 1.0 does not support the conditional assembler functions.

Notes: 1. In addressing modes that use the displacements listed below (disp), the assembler statements in this manual show the value prior to scaling (×1, ×2, and ×4) according to the operand size. This is done to clarify the LSI operation. Actual assembler statements should follow the rules of the assembler in question.
@(disp:4, Rn); Indirect register addressing with displacement
@(disp:8, GBR); Indirect GBR addressing with displacement

@(disp:8, PC); Indirect PC addressing with displacement
disp:8, disp:12:; PC relative addressing

2. 16-bit instruction code that is not assigned as instructions is handled as an ordinary illegal instruction and produces illegal instruction exception processing.

   Example: H'FFFF [ordinary illegal instruction]

3. An ordinary illegal instruction or branched instruction (i.e., an illegal slot instruction) that follows a BRA, BT/S or another delayed branch instruction will cause illegal instruction exception processing.

   Example 1:
   ```
   ....
   BRA      LABEL
   .data.w  H'FFFF   ← Illegal slot instruction
   ....              [H'FFFF is an ordinary illegal instruction from the start]
   ```

   Example 2:
   ```
   RTE
   BT/S     LABEL    ← Illegal slot instruction
   ```

4. The delayed branch actual occurs after the slot instruction is executed. Except for branches such as register updates, however, delayed branch instructions are executed before delayed slot instructions. For example, even when the contents of a register that stores a branch destination address in a delay slot are changed, the branch destination remains the register contents prior to the change.

5. When there ia an ordinary illegal instruction, branched instruction or an instruction to renew the SR, RS or RE register (SETRC, LDRS, etc.) in the last three instructions of a repeat program (loop) with three or less instructions or a program (loop) with four or more instructions, illegal instruction exception processing is started. Refer to section 4.18, DSP Repeat (Loop) Control, for more information.

6.1.2 ADD (ADD Binary): Arithmetic Instruction

Format	Abstract	Code	Cycle	T Bit	Applicable Instructions SH-1	SH-2	SH-DSP
`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
```

6.1.3 ADDC (ADD with Carry): Arithmetic Instruction

					Applicable Instructions		
Format	**Abstract**	**Code**	**Cycle**	**T Bit**	**SH-1**	**SH-2**	**SH-DSP**
`ADDC Rm,Rn`	Rn + Rm + T → Rn, carry → T	`0011nnnnmmmm1110`	1	Carry	○	○	○

Description: Adds Rm data and the T bit to general register 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
```

6.1.4 ADDV (ADD with V Flag Overflow Check): Arithmetic Instruction

Format	Abstract	Code	Cycle	T Bit	Applicable Instructions		
					SH-1	SH-2	SH-DSP
`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
```

6.1.5 AND (AND Logical): Logic Operation Instruction

Format		Abstract	Code	Cycle	T Bit	Applicable Instructions		
						SH-1	SH-2	SH-DSP
`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
```

6.1.6 BF (Branch if False): Branch Instruction

					Applicable Instructions		
Format	**Abstract**	**Code**	**Cycle**	**T Bit**	**SH-1**	**SH-2**	**SH-DSP**
`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 = 0, it branches to the branch destination address. If T = 1, BF executes the next instruction. The branch destination is an address specified by PC + displacement. However, in this case it is used for address calculation. The PC is the address 4 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. 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.

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);
    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
```

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

					Applicable Instructions		
Format	**Abstract**	**Code**	**Cycle**	**T Bit**	**SH-1**	**SH-2**	**SH-DSP**
`BF/S label`	When T = 0, disp × 2+ PC → PC; When T = 1, nop	`10001111dddddddd`	2/1	—	—	○	○

Description: Reads the T bit and conditionally branches. If T = 0, it branches after executing the next instruction. If T = 1, BF/S executes the next instruction. The branch destination is an address specified by PC + displacement. However, in this case it is used for address calculation. The PC is the address 4 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. If the displacement is too short to reach the branch destination, use BF with the BRA instruction or the like.

Note: Since this is a delay branch instruction, the instruction immediately following is executed before the branch. No interrupts and address errors are accepted between this instruction and the next instruction. When the instruction immediately following is a branch instruction, it is recognized as an illegal slot instruction. When branching, this is a two-cycle instruction; when not branching, one cycle.

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);
      Delay_Slot(temp+2);
   }
   else PC+=2;
}
```

Example:

```
        CLRT              ; T is always 0
        BT/S TRGET_T      ; Does not branch, because T = 0
        NOP               ;
        BF/S TRGET_F      ; Branches to TRGET_F, because T = 0
        ADD  R0,R1        ; Executed before branch.
        NOP               ; ← The PC location is used to calculate the branch destination
        ..........          address of the BF/S instruction
TRGET_F:                  ; ← Branch destination of the BF/S instruction
```

Note: With delayed branching, branching occurs after execution of the slot instruction. However, instructions such as register changes etc. are executed in the order of delayed branch instruction, then delay slot instruction. For example, even if the register in which the branch destination address has been loaded is changed by the delay slot instruction, the branch will still be made using the value of the register prior to the change as the branch destination address.

6.1.8 BRA (Branch): Branch Instruction

Format	Abstract	Code	Cycle	T Bit	Applicable Instructions		
					SH-1	SH-2	SH-DSP
`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 However, in this case it is used for address calculation. The PC is the address 4 bytes after this 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 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.

Operation:

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

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

Example:

```
        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: With delayed branching, branching occurs after execution of the slot instruction. However, instructions such as register changes etc. are executed in the order of delayed branch instruction, then delay slot instruction. For example, even if the register in which the branch destination address has been loaded is changed by the delay slot instruction, the branch will still be made using the value of the register prior to the change as the branch destination address.

6.1.9 BRAF (Branch Far): Branch Instruction

Format	Abstract	Code	Cycle	T Bit	Applicable Instructions SH-1	SH-2	SH-DSP
`BRAF Rm`	Rm + PC → PC	`0000mmmm00100011`	2	—	—	○	○

Description: Branches unconditionally. The branch destination is PC + the 32-bit contents of the general register Rm. However, in this case it is used for address calculation. The PC is the address 4 bytes 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.

Operation:

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

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

Example:

```
        MOV.L   #(TARGET-BSRF_PC),R0  ; Sets displacement.
        BRA     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: With delayed branching, branching occurs after execution of the slot instruction. However, instructions such as register changes etc. are executed in the order of delayed branch instruction, then delay slot instruction. For example, even if the register in which the branch destination address has been loaded is changed by the delay slot instruction, the branch will still be made using the value of the register prior to the change as the branch destination address.

6.1.10 BSR (Branch to Subroutine): 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. The PC value is stored in the PR, and the program branches to an address specified by PC + displacement However, in this case it is used for address calculation. The PC is the address 4 bytes after this 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 together for a subroutine procedure call.

Note: Since this is a delayed branch instruction, the instruction after BSR 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.

Operation:

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

   if ((d&0x800)==0) disp=(0x00000FFF & (long) d);
   else disp=(0xFFFFF000 | (long) d);
   PR=PC+Is_32bit_Inst(PR+2);
   PC=PC+(disp<<1);
   Delay_Slot(PR+2);
}
```

Example:

```
        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: With delayed branching, branching occurs after execution of the slot instruction. However, instructions such as register changes etc. are executed in the order of delayed branch instruction, then delay slot instruction. For example, even if the register in which the branch destination address has been loaded is changed by the delay slot instruction, the branch will still be made using the value of the register prior to the change as the branch destination address.

6.1.11 BSRF (Branch to Subroutine Far): Branch Instruction

					Applicable Instructions		
Format	**Abstract**	**Code**	**Cycle**	**T Bit**	**SH-1**	**SH-2**	**SH-DSP**
`BSRF Rm`	PC → PR, Rm + PC → PC	`0000mmmm00000011`	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 Rm. However, in this case it is used for address calculation. The PC is the address 4 bytes after this instruction. Used as a subroutine procedure call in combination with RTS.

Note: Since this is a delayed branch instruction, the instruction after BSR 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.

Operation:

```
BSRF(long m) /* BSRF Rm */
{
    PR=PC+Is_32bit_Inst(PR+2);
    PC+=R[m];
    Delay_Slot(PR+2);
}
```

Example:

```
        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: With delayed branching, branching occurs after execution of the slot instruction. However, instructions such as register changes etc. are executed in the order of delayed branch instruction, then delay slot instruction. For example, even if the register in which the branch destination address has been loaded is changed by the delay slot instruction, the branch will still be made using the value of the register prior to the change as the branch destination address.

6.1.12 BT (Branch if True): Branch Instruction

					Applicable Instructions		
Format	**Abstract**	**Code**	**Cycle**	**T Bit**	**SH-1**	**SH-2**	**SH-DSP**
`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. However, in this case it is used for address calculation. The PC is the address 4 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. 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.

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);
    else PC+=2;
}
```

Example:

```
          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
```

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

Format	Abstract	Code	Cycle	T Bit	Applicable Instructions SH-1	SH-2	SH-DSP
`BT/S label`	When T = 1, disp × 2 + PC → PC; When T = 0, nop	`10001101dddddddd`	2/1	—	—	○	○

Description: Reads the T bit and conditionally branches. 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. However, in this case it is used for address calculation. The PC is the address 4 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. If the displacement is too short to reach the branch destination, use BT/S with the BRA instruction or the like.

Note: Since this is a delay branch instruction, the instruction immediately following is executed before the branch. No interrupts and address errors are accepted between this instruction and the next instruction. When the immediately following instruction is a branch instruction, it is recognized as an illegal slot instruction. When branching, requires two cycles; when not branching, one cycle.

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);
        Delay_Slot(temp+2);
    }
    else PC+=2;
}
```

Example:

```
        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: With delayed branching, branching occurs after execution of the slot instruction. However, instructions such as register changes etc. are executed in the order of delayed branch instruction, then delay slot instruction. For example, even if the register in which the branch destination address has been loaded is changed by the delay slot instruction, the branch will still be made using the value of the register prior to the change as the branch destination address.

6.1.14 CLRMAC (Clear MAC Register): System Control Instruction

					Applicable Instructions		
Format	**Abstract**	**Code**	**Cycle**	**T Bit**	**SH-1**	**SH-2**	**SH-DSP**
`CLRMAC`	0 → MACH, MACL	`0000000000101000`	1	—	○	○	○

Description: Clear the MACH and MACL Register.

Operation:

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

Example:

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

6.1.15 CLRT (Clear T Bit): System Control Instruction

					Applicable Instructions		
Format	**Abstract**	**Code**	**Cycle**	**T Bit**	**SH-1**	**SH-2**	**SH-DSP**
`CLRT`	0 → T	`0000000000001000`	1	0	○	○	○

Description: Clears the T bit.

Operation:

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

Example:

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

6.1.16 CMP/cond (Compare Conditionally): Arithmetic Instruction

					Applicable Instructions		
Format	**Abstract**	**Code**	**Cycle**	**T Bit**	**SH-1**	**SH-2**	**SH-DSP**
`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. The Rn data does not change. The following eight conditions 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 6.1 shows the mnemonics for the conditions.

Table 6.1 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>>12)&0x000000FF;
    HL=(temp>>8)&0x000000FF;
    LH=(temp>>4)&0x000000FF;
    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;
}
```

Example:

```
    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
```

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

					Applicable Instructions		
Format	**Abstract**	**Code**	**Cycle**	**T Bit**	**SH-1**	**SH-2**	**SH-DSP**
`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;
}
```

Example: See DIV1.

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

					Applicable Instructions		
Format	**Abstract**	**Code**	**Cycle**	**T Bit**	**SH-1**	**SH-2**	**SH-DSP**
`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.

6.1.19 DIV1 (Divide 1 Step): Arithmetic Instruction

Format	Abstract	Code	Cycle	T Bit	Applicable Instructions SH-1	SH-2	SH-DSP
`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:

(dividend) – (divisor) × (quotient) = (remainder)

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,R1             ; 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 (one's complement)
ADDC       R2,R1          ; Increments and takes the two's 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 one's 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)
```

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

					Applicable Instructions		
Format	**Abstract**	**Code**	**Cycle**	**T Bit**	**SH-1**	**SH-2**	**SH-DSP**
`DMULS.L Rm, Rn`	With sign, Rn × Rm → MACH, MACL	`0011nnnnmmmm1101`	2 to 4	—	—	○	○

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;
}
```

Example:

```
DMULS.L  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)
```

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

					Applicable Instructions		
Format	**Abstract**	**Code**	**Cycle**	**T Bit**	**SH-1**	**SH-2**	**SH-DSP**
`DMULU.L Rm, Rn`	Without sign, Rn × Rm → MACH, MACL	`0011nnnnmmmm0101`	2 to 4	—	—	○	○

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;
}
```

Example:

```
DMULU.L 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)
```

6.1.22 DT (Decrement and Test): Arithmetic Instruction

					Applicable Instructions		
Format	**Abstract**	**Code**	**Cycle**	**T Bit**	**SH-1**	**SH-2**	**SH-DSP**
`DT Rn`	Rn – 1 → Rn; When Rn is 0, 1 → T, when Rn is nonzero, 0 → T	`0100nnnn00010000`	1	Comparison result	—	○	○

Description: The contents of general register Rn are decremented by 1 and the result compared 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.)
```

6.1.23 EXTS (Extend as Signed): Arithmetic Instruction

Format		Abstract	Code	Cycle	T Bit	Applicable Instructions		
						SH-1	SH-2	SH-DSP
`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
```

6.1.24 EXTU (Extend as Unsigned): Arithmetic Instruction

					Applicable Instructions		
Format	**Abstract**	**Code**	**Cycle**	**T Bit**	**SH-1**	**SH-2**	**SH-DSP**
`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
```

6.1.25 JMP (Jump): Branch Instruction

Class: Delayed branch instruction

					Applicable Instructions		
Format	**Abstract**	**Code**	**Cycle**	**T Bit**	**SH-1**	**SH-2**	**SH-DSP**
`JMP @Rm`	Rm → PC	`0100mmmm00101011`	2	—	○	○	○

Description: Branches unconditionally to the address specified by register indirect addressing. The branch destination is an address specified by the 32-bit data in general register Rm.

Note: Since this is a delayed branch instruction, the instruction after JMP is executed before branching. No interrupts or 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.

Operation:

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

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

Example:

```
            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
```

6.1.26 JSR (Jump to Subroutine): Branch Instruction (Class: Delayed Branch Instruction)

					Applicable Instructions		
Format	**Abstract**	**Code**	**Cycle**	**T Bit**	**SH-1**	**SH-2**	**SH-DSP**
`JSR @Rm`	PC → PR, Rm → PC	`0100mmmm00001011`	2	—	○	○	○

Description: Branches to the subroutine procedure at the address specified by register indirect addressing. The PC value is stored in the PR. The jump destination is an address specified by the 32-bit data in general register Rm. The stored/saved PC is the address four bytes after this instruction. The JSR instruction and RTS instruction are used together for subroutine procedure calls.

Note: Since this is a delayed branch instruction, the instruction after JSR 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.

Operation:

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

Example:

```
            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: When a delayed branch instruction is used, the branching operation takes place after the slot instruction is executed, but the execution of instructions (register update, etc.) takes place in the sequence delayed branch instruction → delayed slot instruction. For example, even if a delayed slot instruction is used to change the register where the branch destination address is stored, the register content previous to the change will be used as the branch destination address.

6.1.27 LDC (Load to Control Register): System Control Instruction (Class: Interrupt Disabled Instruction)

Format	Abstract	Code	Cycle	T Bit
`LDC     Rm,SR`	Rm → SR	`0100mmmm00001110`	1	LSB
`LDC     Rm,GBR`	Rm → GBR	`0100mmmm00011110`	1	—
`LDC     Rm,VBR`	Rm → VBR	`0100mmmm00101110`	1	—
`LDC     Rm,MOD`	Rm → MOD	`0100mmmm01011110`	1	—
`LDC     Rm,RE`	Rm → RE	`0100mmmm01111110`	1	—
`LDC     Rm,RS`	Rm → RS	`0100mmmm01101110`	1	—
`LDC.L @Rm+,SR`	(Rm) → SR, Rm + 4 → Rm	`0100mmmm00000111`	3	LSB
`LDC.L @Rm+,GBR`	(Rm) → GBR, Rm + 4 → Rm	`0100mmmm00010111`	3	—
`LDC.L @Rm+,VBR`	(Rm) → VBR, Rm + 4 → Rm	`0100mmmm00100111`	3	—
`LDC.L @Rm+,MOD`	(Rm) → MOD, Rm + 4 → Rm	`0100mmmm01010111`	3	—
`LDC.L @Rm+,RE`	(Rm) → RE, Rm + 4 → Rm	`0100mmmm01110111`	3	—
`LDC.L @Rm+,RS`	(Rm) → RS, Rm + 4 → Rm	`0100mmmm01100111`	3	—

Description: Store the source operand into control register SR, GBR, VBR, MOD, RE, or RS.

Note: No interrupts are accepted between this instruction and the next instruction. Address errors are accepted.

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;
}

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 */
{
   RSR=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;
}

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'0FFF0FFF
LDC.L  @R15+,GBR    ; Before execution:   R15 = H'10000000
                    ; After execution:    R15 = H'10000004, GBR = @H'10000000
```

Note: This is the execution result for the SH-DSP.

6.1.28 LDRE (Load Effective Address to RE Register): System Control Instruction

					Applicable Instructions		
Format	**Abstract**	**Code**	**Cycle**	**T Bit**	**SH-1**	**SH-2**	**SH-DSP**
`LDRE @(disp,PC)`	disp × 2 + PC → RE	`10001110dddddddd`	1	—	—	—	○

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 4.35, 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          ;
        ............
```

6.1.29 LDRS (Load Effective Address to RS Register): System Control Instruction

					Applicable Instructions		
Format	**Abstract**	**Code**	**Cycle**	**T Bit**	**SH-1**	**SH-2**	**SH-DSP**
`LDRS @(disp,PC)`	disp × 2 + PC → RS	`10001100dddddddd`	1	—	—	—	○

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 4.35. "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          ;
        ............
```

6.1.30 LDS (Load to System Register): System Control Instruction

Class: Interrupt disabled instruction

Format		Abstract	Code	Cycle	T Bit	Applicable Instructions		
						SH-1	SH-2	SH-DSP
`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	—	—	—	○

Description: Store the source operand into the system register MACH, MACL, or PR or the DSP register DSR, A0, X0, X1, Y0, or Y1. When A0 is designated as the destination, the MSB of the data is copied into A0G.

Note: No interrupts are accepted between this instruction and the next instruction. Address errors are accepted.

For the SH-1 CPU, the lower 10 bits are stored in MACH. For the SH-2 and SH-DSP CPU, 32 bits are stored in MACH.

Operation:

```
LDSMACH(long m)         /* LDS Rm,MACH */
{
   MACH=R[m];
   if ((MACH&0x00000200)==0) MACH&=0x000003FF;   For SH-1 CPU(these 2 lines
   else MACH|=0xFFFFFC00;                        not needed for SH-2 and V
        PC+=2; N                                 SH-DSP CPU)
}
LDSMACL(long m)         /* LDS Rm,MACL */
{
   MACL=R[m];
   PC+=2;
}
LDSPR(long m)           /* LDS Rm,PR */
{
   PR=R[m];
   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;
}
LDSMMACH(long m)       /* LDS.L @Rm+,MACH */
{
    MACH=Read_Long(R[m]);
    if ((MACH&0x00000200)==0) MACH&=0x000003FF;   For SH-1 CPU (these 2 lines
    else MACH|=0xFFFFFC00;                        not needed for SH-2 and
         R[m]+=4;                                 SH-DSP CPU)
    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;
}
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
```

6.1.31 MAC.L (Multiply and Accumulate Calculation Long): Arithmetic Instruction

					Applicable Instructions		
Format	**Abstract**	**Code**	**Cycle**	**T Bit**	**SH-1**	**SH-2**	**SH-DSP**
`MAC.L @Rm+, @Rn+`	Signed operation, (Rn) × (Rm) + MAC → MAC	`0000nnnnmmmm1111`	3/(2 to 4)	—	—	○	○

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, they increment Rm and Rn 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 a range of 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;
}
```

Example:

```
        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     ;
```

6.1.32 MAC.W (Multiply and Accumulate Calculation Word): Arithmetic Instruction

					Applicable Instructions		
Format	**Abstract**	**Code**	**Cycle**	**T Bit**	**SH-1**	**SH-2**	**SH-DSP**
`MAC.W @Rm+, @Rn+`	With sign, (Rn) × (Rm) + MAC → MAC	`0100nnnnmmmm1111`	3/(2)	—	—	○	○
`MAC @Rm+, @Rn+`					○	○	○

Description: Does signed multiplication of 16-bit operands obtained using the contents of general registers Rm and Rn as addresses. The 32-bit result is added to contents of the MAC register, and the final result is stored in the MAC register. Rm and Rn data are incremented by 2 after the operation.

When the S bit is cleared to 0, the operation is 16 × 16 + 64 → 64-bit multiply and accumulate and the 64-bit result is stored in the coupled MACH and MACL registers.

When the S bit is set to 1, the operation is 16 × 16 + 32 → 32-bit multiply and accumulate and addition to the MAC register is a saturation operation. For the saturation operation, only the MACL register is enabled and the result is limited to a range of H'80000000 (minimum) and H'7FFFFFFF (maximum).

If an overflow occurs, the LSB of the MACH register is set to 1. The result is stored in the MACL register. The result is limited to a value between H'80000000 (minimum) for overflows in the negative direction and H'7FFFFFFF (maximum) for overflows in the positive direction.

Note: When the S bit is 0, the SH-2 and SH-DSP CPU perform a 16 × 16 + 64 → 64 bit multiply and accumulate operation and the SH-1 CPU performs a 16 × 16 + 42 → 42 bit multiply and accumulate operation.

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)                 For SH-1 CPU (these 2 lines
                MACH|=0x00000001;                 not needed for SH-2 and
            if (src==0) MACL=0x7FFFFFFF;          SH-DSP CPU)
            if (src==2) MACL=0x80000000;
        }
    }
    else {
        MACH+=tempn;
        if (templ>MACL) MACH+=1;
        if ((MACH&0x00000200)==0)                 For SH-1 CPU (these 3 lines
            MACH&=0x000003FF;                     not needed for SH-2 and
        else MACH|=0xFFFFFC00;                    SH-DSP CPU)
    }
    PC+=2;
}
```

Example:

```
        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        ;
```

6.1.33 MOV (Move Data): Data Transfer Instruction

Format		Abstract	Code	Cycle	T Bit	Applicable Instructions		
						SH-1	SH-2	SH-DSP
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;
}
```

Example:

```
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
```

6.1.34 MOV (Move Immediate Data): Data Transfer Instruction

Format		Abstract	Code	Cycle	T Bit	Applicable Instructions		
						SH-1	SH-2	SH-DSP
`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 can be 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 can be 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 optimum table assignment is at the rear end of the module or one instruction after the unconditional branch instruction. If the optimum assignment is impossible for the reason of no unconditional branch instruction in the 510 byte/1020 byte or some other reason, means to jump past the table by the BRA instruction are required. By assigning this instruction immediately after the delayed branch instruction, the PC becomes the "first address + 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;
}
```

Example:

```
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    ;
```

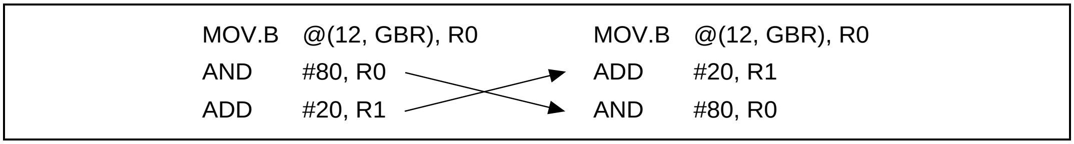
6.1.35 MOV (Move Peripheral Data): Data Transfer Instruction

					Applicable Instructions		
Format	**Abstract**	**Code**	**Cycle**	**T Bit**	**SH-1**	**SH-2**	**SH-DSP**
`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 optimum 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 6.1 will give better results.



```
MOV.B   @(12, GBR), R0          MOV.B   @(12, GBR), R0
AND     #80, R0                 ADD     #20, R1
ADD     #20, R1                 AND     #80, R0
```

Figure 6.1 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:

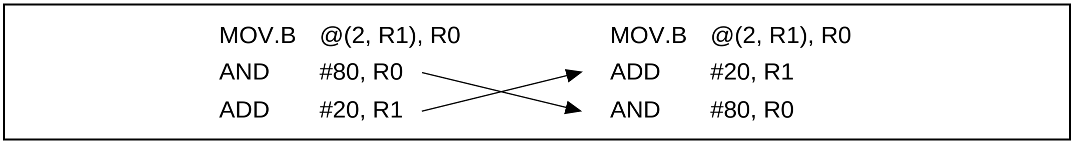
```
MOV.L  @(2,GBR),R0     ; Before execution:   @(GBR + 8) = H'12345670
                       ; After execution:    R0 = H'12345670
MOV.B  R0,@(1,GBR)     ; Before execution:   R0 = H'FFFF7F80
                       ; After execution:    @(GBR + 1) = H'80
```

6.1.36 MOV (Move Structure Data): Data Transfer Instruction

					Applicable Instructions		
Format	**Abstract**	**Code**	**Cycle**	**T Bit**	**SH-1**	**SH-2**	**SH-DSP**
`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	—	○	○	○

Description: Transfers the source operand to the destination. This instruction is optimum 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 6.2 will give better results.



```
MOV.B   @(2, R1), R0              MOV.B   @(2, R1), R0
AND     #80, R0                   ADD     #20, R1
ADD     #20, R1                   AND     #80, R0
```

Figure 6.2 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'F,R1)`	; Before execution:	R0 = H'FFFF7F80
	; After execution:	@(R1 + 60) = H'FFFF7F80

6.1.37 MOVA (Move Effective Address): Data Transfer Instruction

					Applicable Instructions		
Format	**Abstract**	**Code**	**Cycle**	**T Bit**	**SH-1**	**SH-2**	**SH-DSP**
`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 is the address four bytes after this 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;
}
```

Example:

```
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     ;
```

6.1.38 MOVT (Move T Bit): Data Transfer Instruction

					Applicable Instructions		
Format	**Abstract**	**Code**	**Cycle**	**T Bit**	**SH-1**	**SH-2**	**SH-DSP**
`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;
}
```

Example:

```
XOR     R2,R2    ; R2 = 0
CMP/PZ  R2       ; T = 1
MOVT    R0       ; R0 = 1
CLRT             ; T = 0
MOVT    R1       ; R1 = 0
```

6.1.39 MUL.L (Multiply Long): Arithmetic Instruction

Format	Abstract	Code	Cycle	T Bit	Applicable Instructions SH-1	SH-2	SH-DSP
`MUL.L  Rm,Rn`	Rn × Rm → MACL	`0000nnnnmmmm0111`	2 (to 4)	—	—	○	○

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:

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

Example:

```
MULL R0,R1        ; Before execution:  R0 = H'FFFFFFFE, R1 = H'00005555
                  ; After execution:   MACL = H'FFFF5556
STS  MACL,R0      ; Operation result
```

6.1.40 MULS.W (Multiply as Signed Word): Arithmetic Instruction

Format		Abstract	Code	Cycle	T Bit	Applicable Instructions SH-1	SH-2	SH-DSP
`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;
}
```

Example:

```
MULS R0,R1     ; Before execution:  R0 = H'FFFFFFFE, R1 = H'00005555
               ; After execution:   MACL = H'FFFF5556
STS  MACL,R0   ; Operation result
```

6.1.41 MULU.W (Multiply as Unsigned Word): Arithmetic Instruction

Format	Abstract	Code	Cycle	T Bit	Applicable Instructions SH-1	SH-2	SH-DSP

`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;
}
```

Example:

```
MULU   R0,R1     ; Before execution:   R0 = H'00000002, R1 = H'FFFFAAAA
                 ; After execution:    MACL = H'00015554
STS    MACL,R0   ; Operation result
```

6.1.42 NEG (Negate): Arithmetic Instruction

Format	Abstract	Code	Cycle	T Bit	Applicable Instructions SH-1	SH-2	SH-DSP
`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;
}
```

Example:

```
NEG   R0,R1     ; Before execution:  R0 = H'00000001
                ; After execution:   R1 = H'FFFFFFFF
```

6.1.43 NEGC (Negate with Carry): Arithmetic Instruction

Format	Abstract	Code	Cycle	T Bit	Applicable Instructions		
					SH-1	SH-2	SH-DSP
`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
```

6.1.44 NOP (No Operation): System Control Instruction

					Applicable Instructions		
Format	**Abstract**	**Code**	**Cycle**	**T Bit**	**SH-1**	**SH-2**	**SH-DSP**
`NOP`	No operation	`0000000000001001`	1	—	○	○	○

Description: Increments the PC to execute the next instruction.

Operation:

```
NOP()  /* NOP */
{
   PC+=2;
}
```

Example:

```
      NOP      ; Executes in one cycle
```

6.1.45 NOT (NOT—Logical Complement): Logic Operation Instruction

					Applicable Instructions		
Format	**Abstract**	**Code**	**Cycle**	**T Bit**	**SH-1**	**SH-2**	**SH-DSP**
`NOT Rm,Rn`	~Rm → 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;
}
```

Example:

```
NOT    R0,R1     ; Before execution:   R0 = H'AAAAAAAA
                 ; After execution:    R1 = H'55555555
```

6.1.46 OR (OR Logical) Logic Operation Instruction

Format	Abstract	Code	Cycle	T Bit	Applicable Instructions SH-1	SH-2	SH-DSP
`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
```

6.1.47 ROTCL (Rotate with Carry Left): Shift Instruction

Format	Abstract	Code	Cycle	T Bit	Applicable Instructions SH-1	SH-2	SH-DSP
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 6.3).

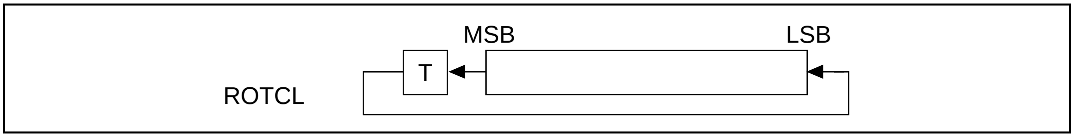


Figure 6.3 Rotate with Carry Left

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;
}
```

Example:

ROTCL R0	; Before execution:	R0 = H'80000000, T = 0
	; After execution:	R0 = H'00000000, T = 1

6.1.48 ROTCR (Rotate with Carry Right): Shift Instruction

					Applicable Instructions		
Format	**Abstract**	**Code**	**Cycle**	**T Bit**	**SH-1**	**SH-2**	**SH-DSP**
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 6.4).

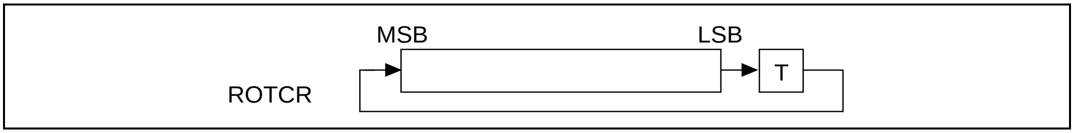


Figure 6.4 Rotate with Carry Right

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

6.1.49 ROTL (Rotate Left): Shift Instruction

					Applicable Instructions		
Format	**Abstract**	**Code**	**Cycle**	**T Bit**	**SH-1**	**SH-2**	**SH-DSP**
`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 6.5). The bit that is shifted out of the operand is transferred to the T bit.

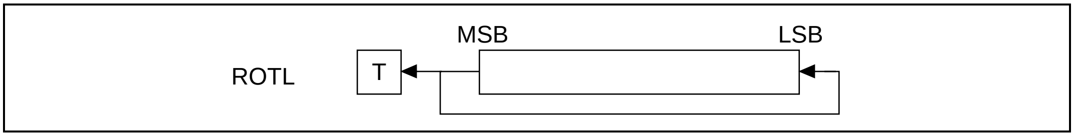


Figure 6.5 Rotate Left

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

6.1.50 ROTR (Rotate Right): Shift Instruction

					Applicable Instructions		
Format	**Abstract**	**Code**	**Cycle**	**T Bit**	**SH-1**	**SH-2**	**SH-DSP**
`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 6.6). The bit that is shifted out of the operand is transferred to the T bit.

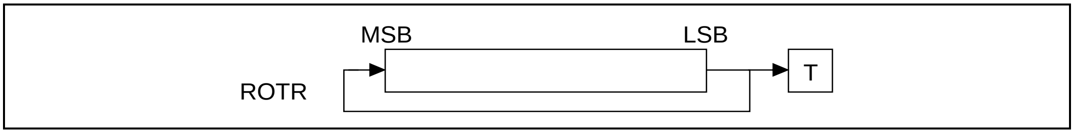


Figure 6.6 Rotate Right

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
```

6.1.51 RTE (Return from Exception): System Control Instruction

Class: Delayed branch instruction

					Applicable Instructions		
Format	**Abstract**	**Code**	**Cycle**	**T Bit**	**SH-1**	**SH-2**	**SH-DSP**
RTE	Delayed branch, Stack area → PC/SR	0000000000101011	4	LSB	○	○	○

Description: Returns from an interrupt routine. The PC and SR values are restored from the stack, and the program continues from the address specified by the restored PC value. The T bit is used as the LSB bit in the SR register restored from the stack area.

Note: Since this is a delayed branch instruction, the instruction after this RTE is executed before branching. No address errors and 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.

Operation:

```
RTE()  /* RTE */
{
    unsigned long temp;

    temp=PC;
    PC=Read_Long(R[15])+4;
    R[15]+=4;
    SR=Read_Long(R[15])&0x0FFF0FFF;
    R[15]+=4;
    Delay_Slot(temp+2);
}
```

Example:

```
RTE              ; Returns to the original routine
ADD   #8,R14     ; Executes ADD before branching
```

RENESAS

Note: With delayed branching, branching occurs after execution of the slot instruction. However, instructions such as register changes etc. are executed in the order of delayed branch instruction, then delay slot instruction. For example, even if the register in which the branch destination address has been loaded is changed by the delay slot instruction, the branch will still be made using the value of the register prior to the change as the branch destination address.

6.1.52 RTS (Return from Subroutine): Branch Instruction (Class: Delayed Branch Instruction)

					Applicable Instructions		
Format	Abstract	Code	Cycle	T Bit	SH-1	SH-2	SH-DSP
RTS	Delayed branch, 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, BSRF, or JSR instruction.

Note: Since this is a delayed branch instruction, the instruction after this RTS is executed before branching. No address errors and 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.

Operation:

```
RTS()  /* RTS */
{
    unsigned long temp;

    temp=PC;
    PC=PR+4;
    Delay_Slot(temp+2);
}
```

Example:

```
        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: With delayed branching, branching occurs after execution of the slot instruction. However, instructions such as register changes etc. are executed in the order of delayed branch instruction, then delay slot instruction. For example, even if the register in which the branch destination address has been loaded is changed by the delay slot instruction, the branch will still be made using the value of the register prior to the change as the branch destination address.

6.1.53 SETRC (Set Repeat Count to RC): System Control Instruction

					Applicable Instructions		
Format	**Abstract**	**Code**	**Cycle**	**T Bit**	**SH-1**	**SH-2**	**SH-DSP**
`SETRC Rm`	Rm[11:0] RCCSR[27:16] Repeat control flag → RF1, RF0	`0100mmmm00010100`	1	—	—	—	◯
`SETRC #imm`	imm → RC [23:26] zeros → SR[27:24], Repeat control flag → RF1, RF0	`10000010iiiiiiii`	1	—	—	—	◯

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 4.18, DSP Repeat (Loop) Control, for more information.

Operation:

```
SETRC(long m) /* SETRC Rm */
{
    long temp;

    temp=(R[m] & 0x00000FFF)<<16;
    SR&=0x00000FF3;
    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&=0x00000FFF;
    SR|=temp;
    RF1=Repeat_Control_Flag1;
    RF0=Repeat_Control_Flag0;
    PC+=2;
}
```

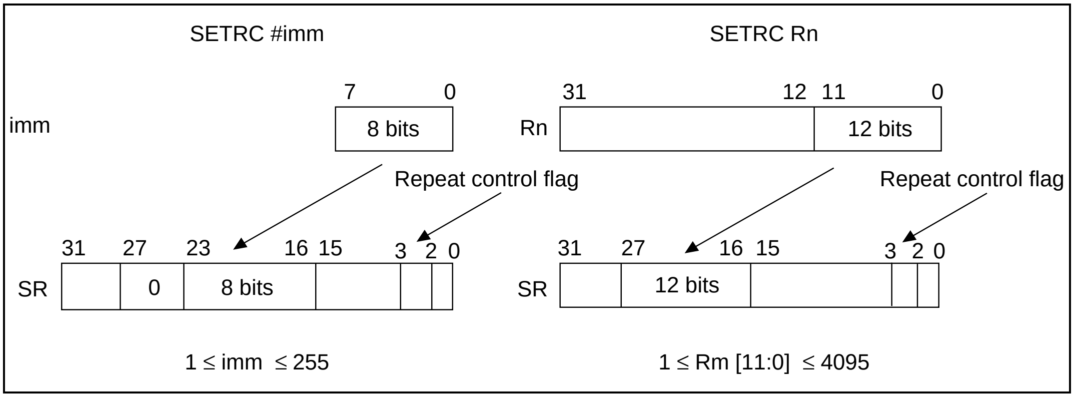


Figure 6.7   SETRC Instruction

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         ;
```

6.1.54 SETT (Set T Bit): System Control Instruction

					Applicable Instructions		
Format	**Abstract**	**Code**	**Cycle**	**T Bit**	**SH-1**	**SH-2**	**SH-DSP**
`SETT`	1 → T	`0000000000011000`	1	1	○	○	○

Description: Sets the T bit to 1.

Operation:

```
SETT() /* SETT */
{
    T=1;
    PC+=2;
}
```

Example:

```
SETT    ; Before execution:   T = 0
        ; After execution:    T = 1
```

6.1.55 SHAL (Shift Arithmetic Left): Shift Instruction

					Applicable Instructions		
Format	**Abstract**	**Code**	**Cycle**	**T Bit**	**SH-1**	**SH-2**	**SH-DSP**
`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 6.8).

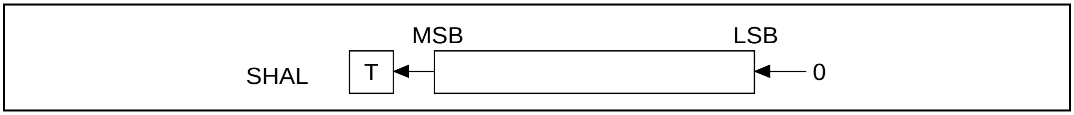


Figure 6.8 Shift Arithmetic Left

Operation:

```
SHAL(long n)  /* SHAL Rn(Same as SHLL) */
{
    if ((R[n]&0x80000000)==0) T=0;
    else T=1;
    R[n]<<=1;
    PC+=2;
}
```

Example:

```
SHAL   R0      ; Before execution:   R0 = H'80000001, T = 0
               ; After execution:    R0 = H'00000002, T = 1
```

6.1.56 SHAR (Shift Arithmetic Right): Shift Instruction

Format	Abstract	Code	Cycle	T Bit	Applicable Instructions SH-1	SH-2	SH-DSP
`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 6.9).

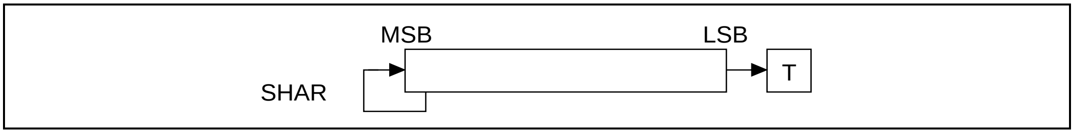


Figure 6.9 Shift Arithmetic Right

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;
}
```

Example:

```
SHAR   R0       ; Before execution:   R0 = H'80000001, T = 0
                ; After execution:    R0 = H'C0000000, T = 1
```

6.1.57 SHLL (Shift Logical Left): Shift Instruction

					Applicable Instructions		
Format	**Abstract**	**Code**	**Cycle**	**T Bit**	**SH-1**	**SH-2**	**SH-DSP**
`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 6.10).

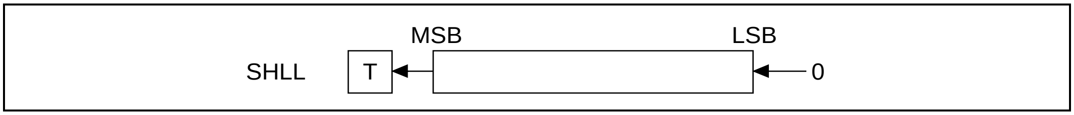


Figure 6.10 Shift Logical Left

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

6.1.58 SHLLn (Shift Logical Left n Bits): Shift Instruction

						Applicable Instructions		
Format		**Abstract**	**Code**	**Cycle**	**T Bit**	**SH-1**	**SH-2**	**SH-DSP**
`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 6.11).

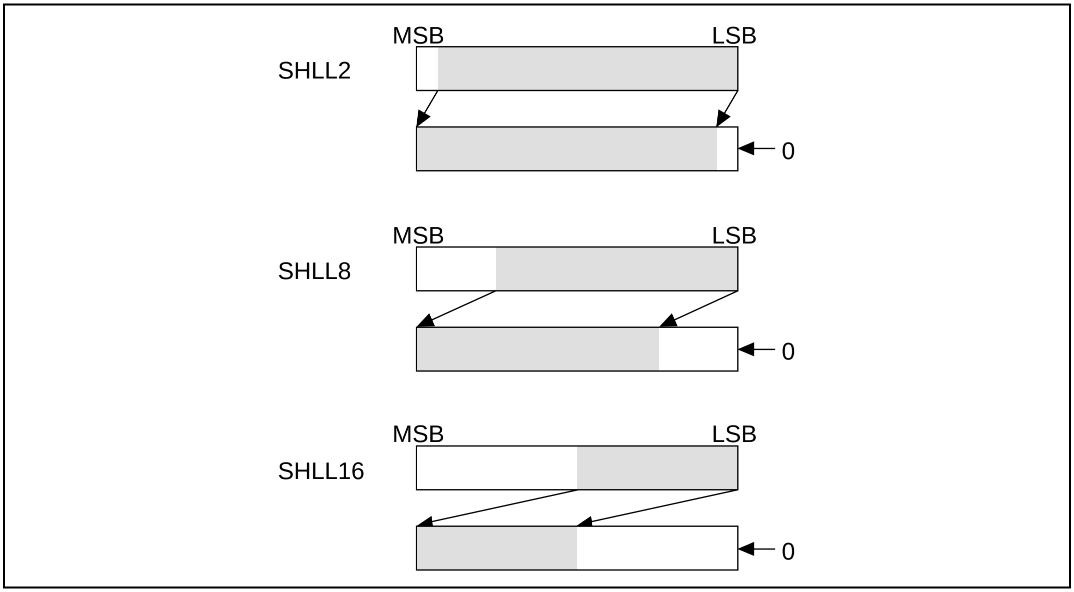


Figure 6.11 Shift Logical Left n Bits

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
```

6.1.59 SHLR (Shift Logical Right): Shift Instruction

					Applicable Instructions		
Format	**Abstract**	**Code**	**Cycle**	**T Bit**	**SH-1**	**SH-2**	**SH-DSP**
`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 6.12).

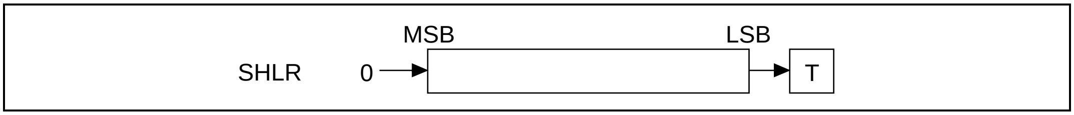


Figure 6.12 Shift Logical Right

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
```

6.1.60 SHLRn (Shift Logical Right n Bits): Shift Instruction

Format		Abstract	Code	Cycle	T Bit	Applicable Instructions SH-1	SH-2	SH-DSP
`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 6.13).

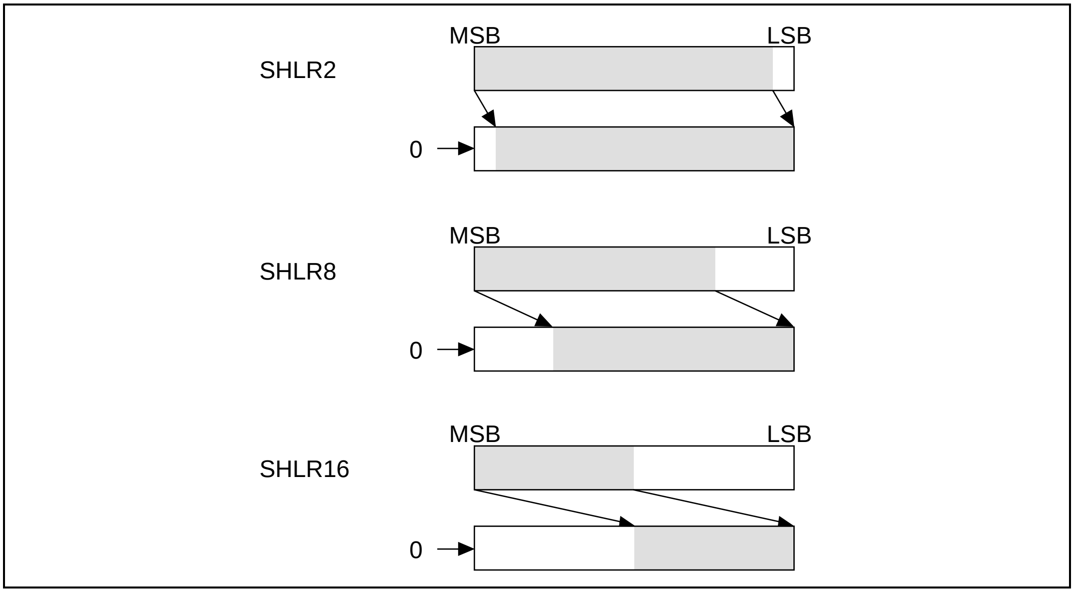


Figure 6.13 Shift Logical Right n Bits

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
```

6.1.61 SLEEP (Sleep): System Control Instruction

					Applicable Instructions		
Format	**Abstract**	**Code**	**Cycle**	**T Bit**	**SH-1**	**SH-2**	**SH-DSP**
`SLEEP`	Sleep	`0000000000011011`	3	—	○	○	○

Description: Sets the CPU into power-down mode. In power-down mode, instruction execution stops, but the CPU internal 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.

Note: The number of cycles given is for the transition to sleep mode.

Operation:

```
SLEEP()   /* SLEEP */
{
    PC-=2;
    wait_for_exception;
}
```

Example:

```
SLEEP   ; Enters power-down mode
```

6.1.62 STC (Store Control Register): System Control Instruction (Interrupt Disabled Instruction)

Format	Abstract	Code	Cycle	T Bit	Applicable Instructions SH-1	SH-2	SH-DSP
`STC SR,Rn`	SR → Rn	`0000nnnn00000010`	1	—	○	○	○
`STC GBR,Rn`	GBR → Rn	`0000nnnn00010010`	1	—	○	○	○
`STC VBR,Rn`	VBR → Rn	`0000nnnn00100010`	1	—	○	○	○
`STC MOD,Rn`	MOD → Rn	`0000nnnn01010010`	1	—	—	—	○
`STC RE,Rn`	RE → Rn	`0000nnnn01110010`	1	—	—	—	○
`STC RS,Rn`	RS → Rn	`0000nnnn01100010`	1	—	—	—	○
`STC.L SR,@-Rn`	Rn – 4 → Rn, SR → (Rn)	`0100nnnn00000011`	2	—	○	○	○
`STC.L GBR,@-Rn`	Rn – 4 → Rn, GBR → (Rn)	`0100nnnn00010011`	2	—	○	○	○
`STC.L VBR,@-Rn`	Rn – 4 → Rn, VBR → (Rn)	`0100nnnn00100011`	2	—	○	○	○
`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	—	—	—	○

Description: Stores control register SR, GBR, VBR, MOD, RE, or RS data into a specified destination.

Note: No interrupts are accepted between this instruction and the next instruction. Address errors are accepted.

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;
}

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;
}

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;
}

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
```

6.1.63 STS (Store System Register): System Control Instruction (Interrupt Disabled Instruction)

						Applicable Instructions		
Format		Abstract	Code	Cycle	T Bit	SH-1	SH-2	SH-DSP
`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	—	—	—	○

Description: Stores data from system register MACH, MACL, or PR or DSP register DSR, A0, X0, X1, Y0, or Y1 into a specified destination.

Note: No interrupts are accepted between this instruction and the next instruction. Address errors are accepted.

If the system register is MACH in the SH-1 series, the value of bit 9 is transferred to and stored in the higher 22 bits (bits 31 to 10) of the destination. With the SH-2 and SH-DSP, the 32 bits of MACH are stored directly.

Operation:

```
STSMACH(long n)  /* STS MACH,Rn */
{
   R[n]=MACH;
if ((R[n]&0x00000200)==0)
R[n]&=0x000003FF;
else R[n]|=0xFFFFFC00;
   PC+=2;
}
```

For SH-1 CPU (these 2 lines not needed for SH-2 and SH-DSP CPU)

```
STSMACL(long n)  /* STS MACL,Rn */
{
   R[n]=MACL;
   PC+=2;
}

STSPR(long n)    /* STS PR,Rn */
{
   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;
}
```

```
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)
```

For SH-1 CPU

```
Write_Long(R[n], MACH);
```

For SH-2 and SH-DSP CPU

```
   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;
}

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;
}
```

Example:

```
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
```

6.1.64 SUB (Subtract Binary): Arithmetic Instruction

					Applicable Instructions		
Format	**Abstract**	**Code**	**Cycle**	**T Bit**	**SH-1**	**SH-2**	**SH-DSP**
`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;
}
```

Example:

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

6.1.65 SUBC (Subtract with Carry): Arithmetic Instruction

					Applicable Instructions		
Format	**Abstract**	**Code**	**Cycle**	**T Bit**	**SH-1**	**SH-2**	**SH-DSP**
`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
```

6.1.66 SUBV (Subtract with V Flag Underflow Check): Arithmetic Instruction

Format	Abstract	Code	Cycle	T Bit	Applicable Instructions		
					SH-1	SH-2	SH-DSP
`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
```

6.1.67 SWAP (Swap Register Halves): Data Transfer Instruction

					Applicable Instructions		
Format	**Abstract**	**Code**	**Cycle**	**T Bit**	**SH-1**	**SH-2**	**SH-DSP**
`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]>>8)&0x000000ff;
   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
```

6.1.68 TAS (Test and Set): Logic Operation Instruction

Format	Abstract	Code	Cycle	T Bit	Applicable Instructions		
					SH-1	SH-2	SH-DSP
`TAS.B @Rn`	When (Rn) is 0, 1 → T, 1 → MSB of (Rn)	`0100nnnn00011011`	4	Test results	○	○	○

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.

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
```

6.1.69 TRAPA (Trap Always): System Control Instruction

					Applicable Instructions		
Format	**Abstract**	**Code**	**Cycle**	**T Bit**	**SH-1**	**SH-2**	**SH-DSP**
TRAPA #imm	PC/SR → Stack area, (imm × 4 + VBR) → PC	11000011iiiiiiii	8	—	○	○	○

Description: Starts the trap exception processing. The PC and SR values are stored on the stack, and the program branches to an address specified by the vector. The vector is a memory address obtained by zero-extending the 8-bit immediate data and then quadrupling it. The PC is the start address of the next instruction. TRAPA and RTE are both used together for system calls.

Operation:

```
TRAPA(long i)/* TRAPA #imm */
{
    long imm;

    imm=(0x000000FF & i);
    R[15]-=4;
    Write_Long(R[15],SR);
    R[15]-=4;
    Write_Long(R[15],PC−2);
    PC=Read_Long(VBR+(imm<<2))+4;
}
```

Example:

```
Address
VBR+H'80  .data.l 10000000 ;
   ..........
          TRAPA   #H'20    ; Branches to an address specified by data in address VBR +
                             H'80
          TST     #0,R0    ; ← Return address from the trap routine (stacked PC value)
   ...........
   ..........
100000000 XOR     R0,R0    ; ← Trap routine entrance
100000002 RTE              ; Returns to the TST instruction
100000004 NOP              ; Executes NOP before RTE
```

6.1.70 TST (Test Logical): Logic Operation Instruction

Format	Abstract	Code	Cycle	T Bit	Applicable Instructions		
					SH-1	SH-2	SH-DSP
`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
```

6.1.71 XOR (Exclusive OR Logical): Logic Operation Instruction

					Applicable Instructions		
Format	**Abstract**	**Code**	**Cycle**	**T Bit**	**SH-1**	**SH-2**	**SH-DSP**
`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
```

6.1.72 XTRCT (Extract): Data Transfer Instruction

					Applicable Instructions		
Format	**Abstract**	**Code**	**Cycle**	**T Bit**	**SH-1**	**SH-2**	**SH-DSP**
`XTRCT Rm,Rn`	Rm: Center 32 bits of Rn → Rn	`0010nnnnmmmm1101`	1	—	○	○	○

Description: Extracts the middle 32 bits from the 64 bits of coupled general registers Rm and Rn, and stores the 32 bits in Rn (figure 6.14).

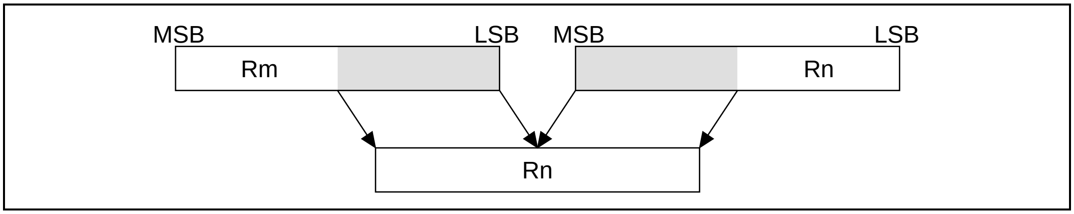


Figure 6.14 Extract

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

6.2 DSP Data Transfer Instructions

Table 6.3 lists the DSP data transfer instructions in alphabetical order.

Table 6.3 DSP Data Transfer Instructions in Alphabetical Order

					Applicable Instructions		
Instruction	**Operation**	**Code**	**Cycles**	**DC Bit**	**SH-1**	**SH-2**	**SH-DSP**
`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	—	—	—	○

					Applicable Instructions		
Instruction	**Operation**	**Code**	**Cycles**	**DC Bit**	**SH-1**	**SH-2**	**SH-DSP**
`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	—	○	○	○

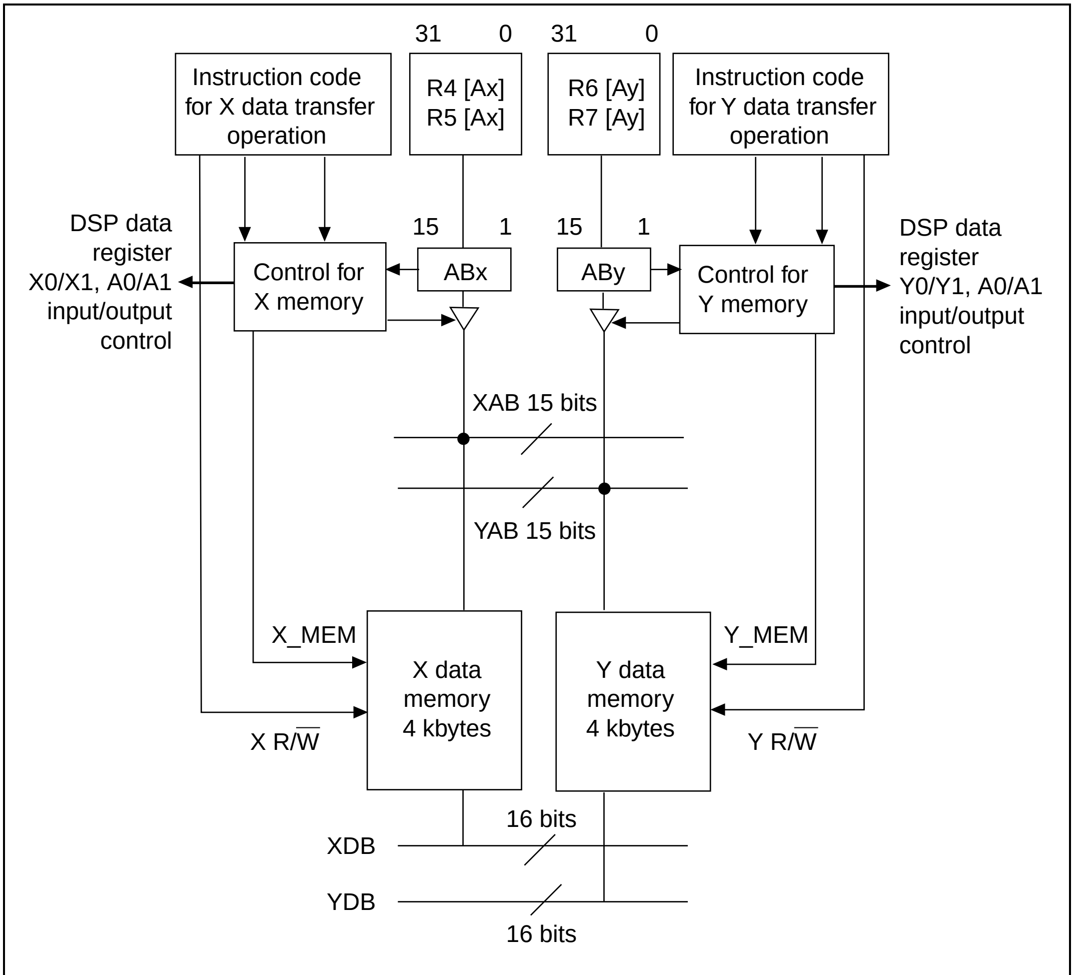
Note: MSW = High-order word of operand
LSW = Low-order word of operand

6.2.1 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 Main buses).

X and Y data transfer instructions are executed regardless of conditions even when the data operation instruction executed in parallel has conditions.

Figure 6.15 shows the load and store operations in X and Y data transfers.


31
0
31
0
Instruction code for X data transfer operation
R4 [Ax]
R5 [Ax]
R6 [Ay]
R7 [Ay]
Instruction code for Y data transfer operation
DSP data register X0/X1, A0/A1 input/output control
15
1
15
1
Control for X memory
ABx
ABy
Control for Y memory
DSP data register Y0/Y1, A0/A1 input/output control
XAB 15 bits
YAB 15 bits
X_MEM
X data memory 4 kbytes
Y data memory 4 kbytes
Y_MEM
X R/W̄
Y R/W̄
16 bits
XDB
YDB
16 bits

X_MEM, Y_MEM: Select signals for X and Y data memory

Figure 6.15 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; }
```

6.2.2 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 main 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 6.16 shows the load and store operations in single data transfers.

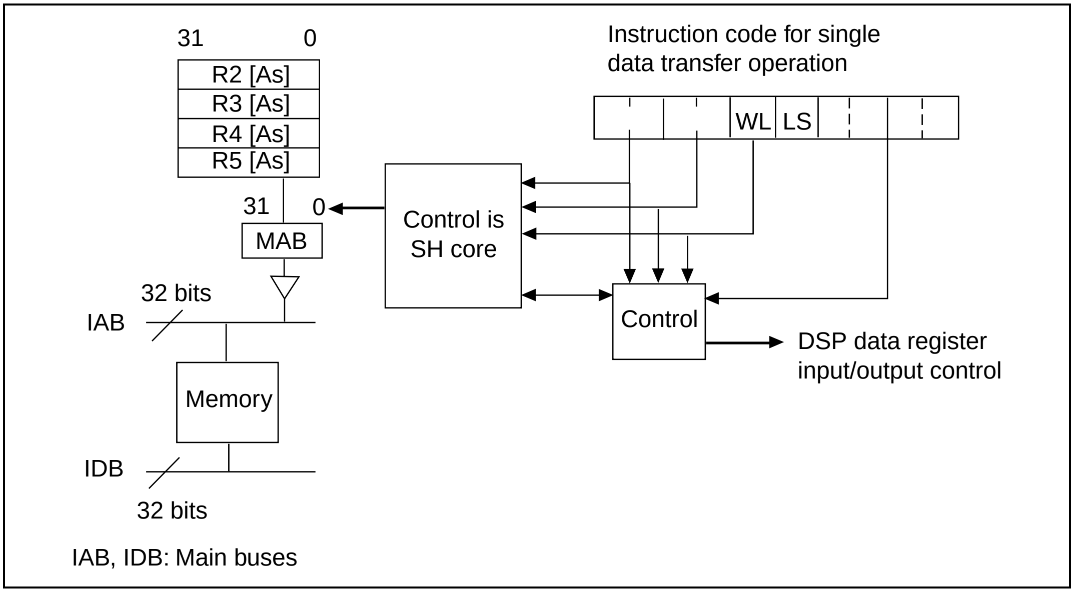


Figure 6.16 Load and Store Operations in Single Data Transfers

Load and store operations in single data transfers are shown below.

```
IAB = MAB;
if ( Ms!=NLS @@ W/L is word access {/* MOVS.W */
   if (LS==load) {
      if (DS!=A0G @@ Ds!=A1G){
            Ds[31:16] = IDB[15:0]; Ds[15:0] = 0x0000;
            if (Ds==A0) A0G[7:0] = IDB[15];
            if (Ds==A1) A1G[7:0] = IDB[15];
      }
      else Ds[7:0] = IDB[7:0]        /* Ds is A0G or A1G */
   }
   else { /* Store */
         if (DS!=A0G @@ Ds!=A1G) IDB[15:0] = Ds[31:16];
         /* Ds is A0G or A1G */
         else IDB[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] = IDB[31:0];
            if (Ds==A0) A0G[7:0] = IDB[31];
            if (Ds==A1) A1G[7:0] = IDB[31];
      }
      else Ds[7:0] = IDB[7:0]        /* Ds is A0G or A1G */
   }
   else { /* Store */
         if (DS!=A0G @@ Ds!=A1G) IDB[31:0] = Ds[31:0]
         /* Ds is A0G or A1G */
         else IDB[31:0] = Ds[7:0] with 24-bit sign extension
   }
}
```

6.2.3 Sample Description (Name): Classification

This section explains the breakdown of instructions, descriptions, etc. given in the rest of this section (section 12).

Table 6.4 Sample Description (Name): Classification

Format	Abstract	Code	Cycle	DC Bit	Applicable Instructions
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	Indicates whether the instruction applies to the SH-1, SH-2, or SH-DSP.

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 6.5 Operation Summary

Operation	Description
→, ←	Direction of transfer
(xx)	Memory operand
DC	Flag bits in the DSR
&	Logical AND of each bit
|	Logical OR of each bit
^	Exclusive OR of each bit
~	Logical NOT of each bit
<<n, >>n	n-bit shift
MSW	Most significant word (bits 16-31)
LSW	Least significant word (bits 0-15)
[n1:n2]	Bits n1 to n2

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.

6.2.4 MOVS (Move Single Data between Memory and DSP Register): DSP Data Transfer Instruction

					Applicable Instructions		
Format	Abstract	Code	Cycle	DC Bit	SH-1	SH-2	SH-DSP
`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 6.17.

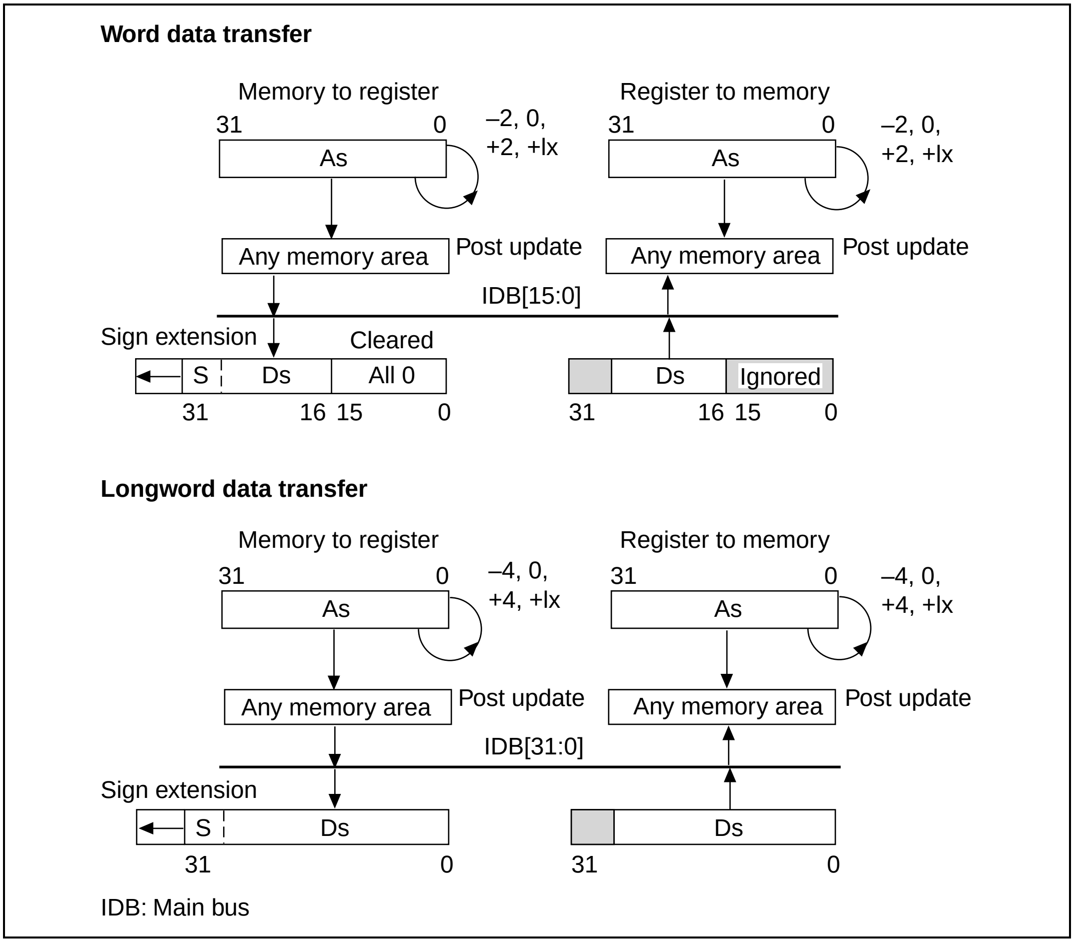


Figure 6.17 The MOVS Instruction

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

6.2.5 MOVX (Move between X Memory and DSP Register): DSP Data Transfer Instruction

					Applicable Instructions		
Format	**Abstract**	**Code**	**Cycle**	**DC Bit**	**SH-1**	**SH-2**	**SH-DSP**
`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 6.18.

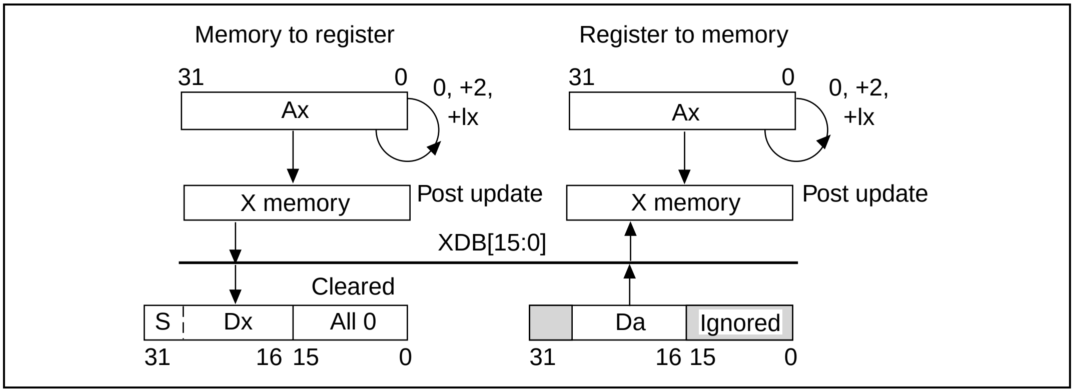


Figure 6.18 The MOVX Instruction

Examples:

```
MOVX.W @R4+,X0
```
; Before execution: R4=H'08010000, @R4=H'5555, X0=H'12345678
; After execution: R4=H'08010002, X0=H'55550000

6.2.6 MOVY (Move between Y Memory and DSP Register): DSP Data Transfer Instruction

					Applicable Instructions		
Format	**Abstract**	**Code**	**Cycle**	**DC Bit**	**SH-1**	**SH-2**	**SH-DSP**
`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 6.19.

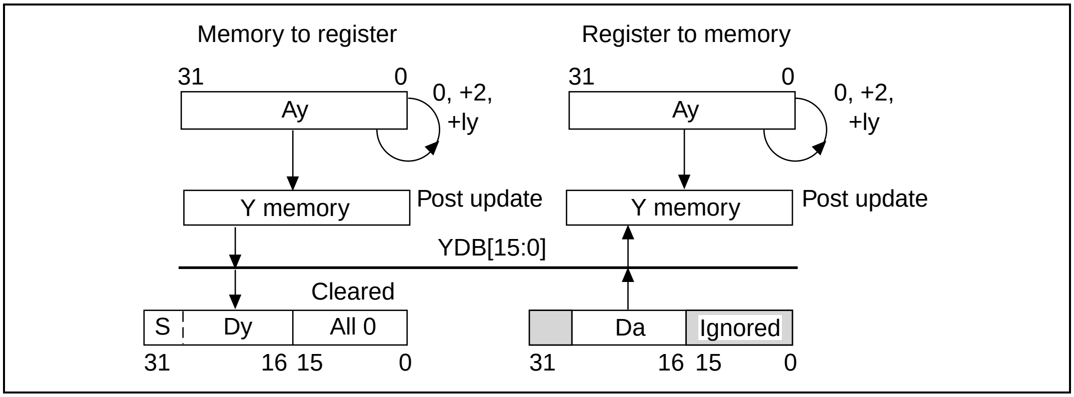


Figure 6.19 The MOVY Instruction

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

6.2.7 NOPX (No Access Operation for X Memory): DSP Data Transfer Instruction

					Applicable Instructions		
Format	**Abstract**	**Code**	**Cycle**	**DC Bit**	**SH-1**	**SH-2**	**SH-DSP**
`NOPX`	No Operation	`1111000*0*0*00**`	1	—	—	—	○

Description: No access operation for X memory.

6.2.8 NOPY (No Access Operation for Y Memory): DSP Data Transfer Instruction

					Applicable Instructions		
Format	**Abstract**	**Code**	**Cycle**	**DC Bit**	**SH-1**	**SH-2**	**SH-DSP**
`NOPY`	No Operation	`111100*0*0*0**00`	1	—	—	—	○

Description: No access operation for Y memory.

6.3 DSP Operation Instructions

The DSP operation instructions are listed below in alphabetical order. See section 6.2.3, Sample Descriptions (Name): Classification, for an explanation of the format and symbols used in this description.

Table 6.6 Alphabetical Listing of DSP Operation Instructions

					Applicable Instructions		
Instruction	**Operation**	**Code**	**Cycles**	**DC Bit**	**SH-1**	**SH-2**	**SH-DSP**

`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	—	—	—	○

					Applicable Instructions		
Instruction	**Operation**	**Code**	**Cycles**	**DC Bit**	**SH-1**	**SH-2**	**SH-DSP**
`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	—	—	○

					Applicable Instructions		
Instruction	**Operation**	**Code**	**Cycles**	**DC Bit**	**SH-1**	**SH-2**	**SH-DSP**
`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	—	—	—	○

Instruction	Operation	Code	Cycles	DC Bit	Applicable Instructions		
					SH-1	SH-2	SH-DSP
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	—	—	—	○

RENESAS

					Applicable Instructions		
Instruction	**Operation**	**Code**	**Cycles**	**DC Bit**	**SH-1**	**SH-2**	**SH-DSP**

`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	—	—	○

`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	—	—	○

`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**********`
		`110011110000zzzz`
			1	—	—	—	○

Instruction	Operation	Code	Cycles	DC Bit	Applicable Instructions		
					SH-1	SH-2	SH-DSP
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

DSP instructions are explained using the same form as for CPU instructions. However, in the description of operation using C, usage of the following DSP resources is presupposed:

1. DSP Register Definitions

The DSP register names are defined based on the union named DSP_Register_Set noted below. This union is composed of 11 longwords; each of these longwords corresponds to one of the 11 DSP registers (A0, A1, M0, M1, X0, X1, Y0, Y1, AG0, AG1, DSR).

```
/* Definition of Union DSP_Register_Set */
union {
    unsigned long int uli[11];
    unsigned short int usi[22];
    struct {
        struct {
            unsigned short int usi[2];
        } ee[11];
    } dd;
    struct {
        struct {
            union {
                unsigned long int uli;
                unsigned short int usi[2];
                struct {
                    unsigned msb:  1;
                    unsigned :     23;
                    unsigned g_msb:1;
                    unsigned :     7;
                } bb;
                struct {
                    unsigned :     24;
                    unsigned lsb8: 8;
                } cc;
            } mm;
        } a0, a1, m0, m1, x0, x1, y0, y1, a0g, a1g;
        union {
            unsigned long int uli;
            struct {
                unsigned Reserved:  24;
                unsigned gz: 1; /* Signed greater than */
                unsigned  z: 1; /* Zero value */
                unsigned  n: 1; /* Negative value */
                unsigned  v: 1; /* Overflow */
                unsigned cs: 3; /* Condition Selection */
                unsigned dc: 1; /* dsp condition bit */
            } a;
        } dsr;
    } name;
    struct {
        unsigned short int a[2][2];
        unsigned short int m[2][2];
        unsigned short int x[2][2];
        unsigned short int y[2][2];
        unsigned short int ag[2][2];
        unsigned short int dsr[2];
    } word;
} DSP_Register_Set;
```

The DSP register names are defined as follows, using the union DSP_Register_Set noted above.

```
/* Definition of DSP Register names */
#define MACL    DSP_Register_Set.name.a0.mm.uli
#define A0      DSP_Register_Set.name.a0.mm.uli
#define A0_HW   DSP_Register_Set.name.a0.mm.usi[0]
#define A0_LW   DSP_Register_Set.name.a0.mm.usi[1]
#define A0_MSB  DSP_Register_Set.name.a0.mm.bb.msb

#define MACH    DSP_Register_Set.name.a1.mm.uli
#define A1      DSP_Register_Set.name.a1.mm.uli
#define A1_HW   DSP_Register_Set.name.a1.mm.usi[0]
#define A1_LW   DSP_Register_Set.name.a1.mm.usi[1]
#define A1_MSB  DSP_Register_Set.name.a1.mm.bb.msb

#define M0      DSP_Register_Set.name.m0.mm.uli
#define M0_HW   DSP_Register_Set.name.m0.mm.usi[0]
#define M0_LW   DSP_Register_Set.name.m0.mm.usi[1]
#define M0_MSB  DSP_Register_Set.name.m0.mm.bb.msb

#define M1      DSP_Register_Set.name.m1.mm.uli
#define M1_HW   DSP_Register_Set.name.m1.mm.usi[0]
#define M1_LW   DSP_Register_Set.name.m1.mm.usi[1]
#define M1_MSB  DSP_Register_Set.name.m1.mm.bb.msb

#define X0      DSP_Register_Set.name.x0.mm.uli
#define X0_HW   DSP_Register_Set.name.x0.mm.usi[0]
#define X0_LW   DSP_Register_Set.name.x0.mm.usi[1]
```

```
#define X0_MSB  DSP_Register_Set.name.x0.mm.bb.msb

#define X1      DSP_Register_Set.name.x1.mm.uli
#define X1_HW   DSP_Register_Set.name.x1.mm.usi[0]
#define X1_LW   DSP_Register_Set.name.x1.mm.usi[1]
#define X1_MSB  DSP_Register_Set.name.x1.mm.bb.msb

#define Y0      DSP_Register_Set.name.y0.mm.uli
#define Y0_HW   DSP_Register_Set.name.y0.mm.usi[0]
#define Y0_LW   DSP_Register_Set.name.y0.mm.usi[1]
#define Y0_MSB  DSP_Register_Set.name.y0.mm.bb.msb

#define Y1      DSP_Register_Set.name.y1.mm.uli
#define Y1_HW   DSP_Register_Set.name.y1.mm.usi[0]
#define Y1_LW   DSP_Register_Set.name.y1.mm.usi[1]
#define Y1_MSB  DSP_Register_Set.name.y1.mm.bb.msb

#define A0G     DSP_Register_Set.name.a0g.mm.uli
#define A0G_HW  DSP_Register_Set.name.a0g.mm.usi[0]
#define A0G_LW  DSP_Register_Set.name.a0g.mm.usi[1]
#define A0G_LSB8  DSP_Register_Set.name.a0g.mm.cc.lsb8
#define A0G_MSB  DSP_Register_Set.name.a0g.mm.bb.g_msb

#define A1G     DSP_Register_Set.name.a1g.mm.uli
#define A1G_HW  DSP_Register_Set.name.a1g.mm.usi[0]
#define A1G_LW  DSP_Register_Set.name.a1g.mm.usi[1]
#define A1G_LSB8  DSP_Register_Set.name.a1g.mm.cc.lsb8
#define A1G_MSB  DSP_Register_Set.name.a1g.mm.bb.g_msb

#define DSR DSP_Register_Set.name.dsr.uli
```

Additionally, the individual bits of the DSR register are defined in the same manner, using the union DSP_Register_Set, as follows:

```
#define DSPGTBIT  DSP_Register_Set.name.dsr.a.gt
#define DSPZBIT   DSP_Register_Set.name.dsr.a.z
#define DSPNBIT   DSP_Register_Set.name.dsr.a.n
```

```
#define DSPVBIT   DSP_Register_Set.name.dsr.a.v
#define DSPCSBITS DSP_Register_Set.name.dsr.a.cs
#define DSPDCBIT  DSP_Register_Set.name.dsr.a.dc
```

2. ALU Input/Output and Variables Representing Operation Results

The ALU input/output is defined based on the union named DSP_ALU_Set noted below. This union is composed of six longwords. Three of these longwords correspond to two inputs and one output (src1, src2, dst). The remaining three longwords are used as guard bits for these two inputs and one output (src1g, src2g, dstg).

```
/* Definition of Union DSP_ALU_Set */
union {
   unsigned long int   uli[6];
   unsigned short int  usi[12];
   struct {
      struct {
         unsigned msb:1;
         unsigned: 31;
      } src1, src2, dst;
      struct {
         union {
            unsigned long int      uli;
            struct {
               unsigned:      24;
               unsigned bit7: 1;
               unsigned:      7;
            } a;
            struct {
               unsigned:      24;
               unsigned lsb8: 8;
            } b;
         } u;
      } src1g, src2g, dstg;
   } n;

} DSP_ALU_Set;
```

The ALU input/output names are defined as follows, using the union DSP_ALU_Set noted above.

```
/* Definition of ALU input/output in DSP operation instructions */
#define DSP_ALU_SRC1    DSP_ALU_Set.uli[0]
#define DSP_ALU_SRC2    DSP_ALU_Set.uli[1]
#define DSP_ALU_DST     DSP_ALU_Set.uli[2]

#define DSP_ALU_SRC1G   DSP_ALU_Set.uli[3]
#define DSP_ALU_SRC2G   DSP_ALU_Set.uli[4]
#define DSP_ALU_DSTG    DSP_ALU_Set.uli[5]

#define DSP_ALU_SRC1_HW DSP_ALU_Set.usi[0]
#define DSP_ALU_SRC2_HW DSP_ALU_Set.usi[2]
#define DSP_ALU_DST_HW  DSP_ALU_Set.usi[4]

#define DSP_ALU_SRC1_MSB DSP_ALU_Set.n.src1.msb
#define DSP_ALU_SRC2_MSB DSP_ALU_Set.n.src2.msb
#define DSP_ALU_DST_MSB DSP_ALU_Set.n.dst.msb

#define DSP_ALU_SRC1G_BIT7 DSP_ALU_Set.n.src1g.u.a.bit7
#define DSP_ALU_SRC2G_BIT7 DSP_ALU_Set.n.src2g.u.a.bit7
#define DSP_ALU_DSTG_BIT7  DSP_ALU_Set.n.dstg.u.a.bit7

#define DSP_ALU_SRC1G_LSB8 DSP_ALU_Set.n.src1g.u.b.lsb8
#define DSP_ALU_SRC2G_LSB8 DSP_ALU_Set.n.src2g.u.b.lsb8
#define DSP_ALU_DSTG_LSB8  DSP_ALU_Set.n.dstg.u.b.lsb8
```

Additionally, the variables representing operation results are defined as follows, using the definitions noted above. These variables are used to calculate the DSR register's DC bit within the description of operation of each instruction.

```
/* Definition of variables representing DSP operation results */
#define PLUS_OP_G_OV ((~DSP_ALU_SRC1G_BIT7 && ~DSP_ALU_SRC2G_BIT7 &&
DSP_ALU_DSTG_BIT7) || (DSP_ALU_SRC1G_BIT7 && DSP_ALU_SRC2G_BIT7 &&
~DSP_ALU_DSTG_BIT7))
```

```
#define MINUS_OP_G_OV ((~DSP_ALU_SRC1G_BIT7 && DSP_ALU_SRC2G_BIT7 &&
DSP_ALU_DSTG_BIT7) || (DSP_ALU_SRC1G_BIT7 && ~DSP_ALU_SRC2G_BIT7 &&
~DSP_ALU_DSTG_BIT7))


#define POS_NOT_OV ((DSP_ALU_DSTG_LSB8==0x00) &&
(DSP_ALU_DST_MSB==0x0))
#define NEG_NOT_OV ((DSP_ALU_DSTG_LSB8==0xff) &&
(DSP_ALU_DST_MSB==0x1))
```

3. Multiplier Input/Output

The multiplier input/output is defined based on the union named DSP_MUL_Set noted below. This union is composed of four longwords. One longword each is allocated for the two inputs, but only the upper 16 bits of both of these (usi [0], usi [2]) are used. Two longwords including guard bit usage (dst, dstg) correspond to the outputs.

```
/* Definition of Union DSP_MUL_Set */
union {
   unsigned long int   uli[4];
   struct {
      unsigned short int usi[4];
      struct {
         unsigned msb: 1;
         unsigned: 31;
      } dst;
      struct {
         unsigned: 24;
         unsigned lsb8: 8;
      } dstg;
   } aa;
} DSP_MUL_Set;
```

The multiplier input/output names are defined as follows, using the union DSP_MUL_Set noted above.

```
/* Definition of multiplier input/output in DSP operation instructions
*/
#define DSP_M_SRC1  DSP_MUL_Set.aa.usi[0]
#define DSP_M_SRC2  DSP_MUL_Set.aa.usi[2]
```

```
#define DSP_M_DST   DSP_MUL_Set.uli[2]
#define DSP_M_DST_MSB  DSP_MUL_Set.aa.dst.msb
#define DSP_M_DSTG  DSP_MUL_Set.uli[3]
#define DSP_M_DSTG_LSB8 DSP_MUL_Set.aa.dstg.lsb8
```

4. Variables Used in the Operation Descriptions of other Instructions, etc.

The following variables are used when describing the operation of DSP operation instructions for which the DCT, DCF conditions can be designated.

In the above definitions, EX_DCT and EX_DCF are variables that become true when the DCT, DCF conditions are designated in instructions. Refer to (1) DSP register definitions for DSPDCBIT.

```
#define DSP_UNCONDITIONAL_UPDATE (!EX_DCT && !EX_DCF)
#define DSP_CONDITION_MATCH ((EX_DCT && DSPDCBIT) || (EX_DCF &&
!DSPDCBIT))
#define DSP_CONDITION_NOT_MATCH ((EX_DCT && !DSPDCBIT)||(EX_DCF &&
DSPDCBIT))
```

In DSP arithmetic operations, saturation processing is performed when the SR register's saturation bit is a 1. This saturation bit is called SBIT when describing the operations.

Additionally, the following function is defined to be used in common, to simplify the notation when describing operations:

```
/* Function used in common in descriptions of DSP operation
instructions */
unsigned char carry_bit, borrow_bit, negative_bit, zero_bit,
overflow_bit;


overflow_protection()
{
    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 */
      }
  }
```

The six functions noted below are used for DSR register updating. The DC bit in the DSR register is updated in accordance with the operation results of the DSP operation instructions and the directions of the status selection bit (CS). The other bits in the DSR register are updated in accordance with the operation results of the DSP operation instructions only.

```
  /* Function to unconditionally update the DC bit (DSPDCBIT) with the
  borrow flag */
  dc_always_borrow()
  {
     /* 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;
  }

  /* Function to unconditionally update the DC bit (DSPDCBIT) with the
  carry flag */
  dc_always_carry()
  {
     /* 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;
}

/* Function to update the DC bit (DSPDCBIT) upon a subtraction */
minus_dc_bit()
{
    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;
}

/* Function to update the DC bit (DSPDCBIT) upon an addition */
```

```
plus_dc_bit()
{
    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;
}

/* Function to update the DC bit (DSPDCBIT) upon a logical operation */
logical_dc_bit()
{
    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;
}

shift_dc_bit()
{
    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;
}
```

6.3.1 PABS (Absolute): DSP Arithmetic Operation Instruction

					Applicable Instructions		
Format	**Abstract**	**Code**	**Cycle**	**DC Bit**	**SH-1**	**SH-2**	**SH-DSP**

`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 stored 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:

```
/* Case1: PABS Sx,Dz */
/* Case2: PABS Sx,Dz */
{
unsigned char carry_bit, negative_bit, zero_bit, overflow_bit,
borrow_bit;
/* ALU Sources assignment */
DSP_ALU_SRC1 = 0
DSP_ALU_SRC1G = 0
    if (Case1) {     /* PABS Sx,Dz */
        switch (xx) {/* Sx Operand selection bit (xx) */
            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  {          /* PABS Sy,Dz */
    switch (yy) {
            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;
}
/* ALU Operation */
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);
overflow_protection();
/* ALU Destination assignment */
switch (zzzz) { /* Dz Operand selection bit (zzzz) */
    case 0x5: A1 = DSP_ALU_DST;
    A1G = DSP_ALU_DSTG & 0x000000FF;
    if(DSP_ALU_DSTG_BIT7)  A1G = A1G | 0xFFFFFF00;
break
    case 0x7: A0 = DSP_ALU_DSTG;
    A0G = DSP_ALU_DSTG & 0x000000FF;
    if(DSP_ALU_DSTG_BIT7)  A0G = A0G | 0xFFFFFF00;
break;
    case 0x8: X0 = DSP_ALU_DST;
break;
    case 0x9: X1 = DSP_ALU_DST;
break;
    case 0xa: Y0 = DSP_ALU_DST;
break;
    case 0xb: Y1 = DSP_ALU_DST;
break;
    case 0xc: M0 = DSP_ALU_DST;
break;
    case 0xe: M1 = DSP_ALU_DST;
break;
    default:  printf("\nERROR: Illegal DSP Instruction");  break;
}
negative _bit = DSP_ALU_DST_BIT7;
zero_bit = (DSP_ALU_DST==0) & (DSP_ALU_DST_LSB8==0);

/* DSR register update */
if(DSP_ALU_SRC2G_BIT7==0) {
  plus_dc_bit ();
}
    else   {
        overflow_bit= MINUS_OP_G_OV || !(POS_NOT_OV || NEG_NOT_OV);
    minus_dc_bit();
     }
  }
```

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].	

6.3.2 [if cc]PADD (Addition with Condition): DSP Arithmetic Operation Instruction

					Applicable Instructions		
Format	Abstract	Code	Cycle	DC Bit	SH-1	SH-2	SH-DSP
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. If conditions are specified, the DC, N, Z, V, and GT bits are not updated even is the conditions were true and the instruction was executed.

Operation:

```
/* PADD Sx,Sy,Dz */

{
unsigned char carry_bit, negative_bit, zero_bit, overflow_bit;

/* ALU Sources assignment */

switch (xx) {  /* Sx Operand selection bit (xx) */
    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 (yy) {  /* Sy Operand selection bit (yy) */
    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;

/* ALU 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);
overflow_protection();

if(DSP_UNCONDITIONAL_UPDATE) { /* unconditional operation */

/* ALU Destination assignment */
switch (zzzz) { /* Dz Operand selection bit (zzzz) */
    case 0x5: A1 = DSP_ALU_DST;
```

```
   A1G = DSP_ALU_DSTG & 0x000000FF;
   if(DSP_ALU_DSTG_BIT7)  A1G = A1G | 0xFFFFFF00;
break
   case 0x7: A0 = DSP_ALU_DST;
   A0G = DSP_ALU_DSTG & 0x000000FF;
   if(DSP_ALU_DSTG_BIT7)  A0G = A0G | 0xFFFFFF00;
break;
   case 0x8: X0 = DSP_ALU_DST;
break;
   case 0x9: X1 = DSP_ALU_DST;
break;
   case 0xa: Y0 = DSP_ALU_DST;
break;
   case 0xb: Y1 = DSP_ALU_DST;
break;
   case 0xc: M0 = DSP_ALU_DST;
break;
   case 0xe: M1 = DSP_ALU_DST;
break;
   default:  printf(“\nERROR: Illegal DSP Instruction”);  break;
}
negative _bit = DSP_ALU_DSTG_BIT7;
zero_bit = (DSP_ALU_DST==0) & (DSP_ALU_DST_LSB8==0);

/* DSR register update */
plus_dc_bit ();
}
   else if(DSP_CONDITION_MATCH) { /* conditional operation and match */
/* ALU Destination assignment */
switch (zzzz) { /* Dz Operand selection bit (zzzz) */
   case 0x5: A1 = DSP_ALU_DST;
   A1G = DSP_ALU_DSTG & 0x000000FF;
   if(DSP_ALU_DSTG_BIT7)  A1G = A1G | 0xFFFFFF00;
break
   case 0x7: A0 = DSP_ALU_DSTG;
   A0G = DSP_ALU_DSTG & 0x000000FF;
```

```
    if(DSP_ALU_DSTG_BIT7)  A0G = A0G | 0xFFFFFF00;
break;
    case 0x8: X0 = DSP_ALU_DST;
break;
    case 0x9: X1 = DSP_ALU_DST;
break;
    case 0xa: Y0 = DSP_ALU_DST;
break;
    case 0xb: Y1 = DSP_ALU_DST;
break;
    case 0xc: M0 = DSP_ALU_DST;
break;
    case 0xe: M1 = DSP_ALU_DST;
break;
    default:  printf("\nERROR: Illegal DSP Instruction");  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.

6.3.3 PADD PMULS (Addition & Multiply Signed by Signed): DSP Arithmetic Operation Instruction

Format	Abstract	Code	Cycle	DC Bit	Applicable Instructions SH-1	SH-2	SH-DSP
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:

```
/*  PADD Sx,Sy,Du PMULS Se,Sf,Dg   */

{
unsigned char carry_bit, negative_bit, zero_bit, overflow_bit;

/* Multiplier Sources assignment */
    switch (ee) {    /* Se Operand selection bit (ee) */
        case 0x0: DSP_M_SRC1 = X0_HW;
                break;
        case 0x1: DSP_M_SRC1 = X1_HW;
                break;
        case 0x2: DSP_M_SRC1 = Y0_HW;
                break;
        case 0x3: DSP_M_SRC1 = A1_HW;
                break;
    }
```

```
     switch (ff) {    /* Sf Operand selection bit (ff) */
        case 0x0: DSP_M_SRC2 = Y0_HW;
              break;
        case 0x1: DSP_M_SRC2 = Y1_HW;
              break;
        case 0x2: DSP_M_SRC2 = X0_HW;
              break;
        case 0x3: DSP_M_SRC2 = A1_HW;
              break;
     }
  /* ALU Sources assignment */
     switch (xx) {    /* Sx Operand selection bit (xx) */
        case 0x0: DSP_ALU_SRC1  = X0;
              if (DSP_ALU_SRC1_MSB)
                 DSP_ALU_SRC1G_LSB8 = 0xff;
              else   DSP_ALU_SRC1G_LSB8 = 0x0;
              break;
        case 0x1: DSP_ALU_SRC1  = X1;
              if (DSP_ALU_SRC1_MSB)
                 DSP_ALU_SRC1G_LSB8 = 0xff;
              else   DSP_ALU_SRC1G_LSB8 = 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 (yy) {    /* Sy Operand selection bit (yy) */
        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_LSB8 = 0xff;
        else          DSP_ALU_SRC2G_LSB8 = 0x0;

/* Multiplier Operation */

        /* PMULS Se, Sf, Dg */
        if ((SBIT==1) && (DSP_M_SRC1==0x8000) && (DSP_M_SRC2==0x8000)) {
                DSP_M_DST=0x7fffffff;  /* overflow protection */
        }
        else {
                DSP_M_DST=((long)(short)DSP_M_SRC1*(long)(short)DSP_M_SRC2)
<<1;
        }
        if (DSP_M_DST_MSB) DSP_M_DSTG_LSB8 = 0xff;
        else   DSP_M_DSTG_LSB8 = 0x0;

        switch (gg) {    /* Dg Operand selection bit (gg) */
            case 0x0: M0 = DSP_M_DST;
                break;
            case 0x1:    M1 = DSP_M_DST;
                break;
            case 0x2: A0 = DSP_M_DST;
                if(DSP_M_DSTG_LSB8==0x0) A0G=0x0;
                else A0G=0xffffffff;
                break;
            case 0x3: A1 = DSP_M_DST;
                if(DSP_M_DSTG_LSB8==0x0) A1G=0x0;
                else A1G=0xffffffff;
                break;
        }

/* ALU 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);

    overflow_protection();

    switch (uu) {    /* Du Operand selection bit (uu) */
        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 & 0x000000FF;
            if(DSP_ALU_DSTG_BIT7) A0G = A0G | 0xFFFFFF00;
            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 & 0x000000FF;
            if(DSP_ALU_DSTG_BIT7) A1G = A1G | 0xFFFFFF00;
            negative_bit = DSP_ALU_DSTG_BIT7;
            zero_bit = (DSP_ALU_DST==0) & (DSP_ALU_DSTG_LSB8==0);
            break;
    }

    /* DSR register update */
    plus_dc_bit();

}
```

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].

6.3.4 PADDC (Addition with Carry): DSP Arithmetic Operation Instruction

					Applicable Instructions		
Format	**Abstract**	**Code**	**Cycle**	**DC Bit**	**SH-1**	**SH-2**	**SH-DSP**
`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:

```
/*  PADD Sx,Sy,Dz      */

{
unsigned char carry_bit, negative_bit, zero_bit, overflow_bit;

/* ALU Sources assignment */
switch (xx) {    /* Sx Operand selection bit (xx) */
    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 (yy) {    /* Sy Operand selection bit (yy) */
   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;

/* ALU Operation */
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);
overflow_protection();

   /* ALU Destination assignment */
   switch (zzzz) {  /* Dz Operand selection bit (zzzz) */
         case 0x5: A1 = DSP_ALU_DST;
            A1G = DSP_ALU_DSTG & 0x000000FF;
            if(DSP_ALU_DSTG_BIT7) A1G = A1G | 0xFFFFFF00;
      break;
         case 0x7: A0 = DSP_ALU_DST;
            A0G = DSP_ALU_DSTG & 0x000000FF;
            if(DSP_ALU_DSTG_BIT7) A0G = A0G | 0xFFFFFF00;
      break;
         case 0x8: X0 = DSP_ALU_DST;
```

```
        break;
            case 0x9: X1 = DSP_ALU_DST;
        break;
            case 0xa: Y0 = DSP_ALU_DST;
        break;
            case 0xb: Y1 = DSP_ALU_DST;
        break;
            case 0xc: M0 = DSP_ALU_DST;
        break;
            case 0xe: M1 = DSP_ALU_DST;
        break;
            default:  printf("\nERROR:Illegal DSP Instruction");
                      break;
    }

    negative_bit = DSP_ALU_DSTG_BIT7;
    zero_bit = (DSP_ALU_DST==0) & (DSP_ALU_DSTG_LSB8==0);

    /* DSR register update */
    dc_always_carry();
```

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
                             The DC bit is updated as the carry flag, regardless of
                             the state of the CS bit.
```

6.3.5 [if cc] PAND (Logical AND): DSP Logical Operation Instruction

					Applicable Instructions		
Format	Abstract	Code	Cycle	DC Bit	SH-1	SH-2	SH-DSP
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. If conditions are specified, the DC, N, Z, V, and GT bits are not updated even is the conditions were true and the instruction was executed.

Note: The bottom word of the destination register and the guard bits are ignored when the DC bit is updated.

Operation:

```
/*  PAND Sx,Sy,Dz      */

{
unsigned char carry_bit, negative_bit, zero_bit, overflow_bit;

/* ALU Sources assignment */
    switch (xx) {    /* Sx Operand selection bit (xx) */
        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 (yy) {    /* Sy Operand selection bit (yy) */
        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 */

    /* ALU Destination assignment */
    switch (zzzz) {  /* Dz Operand selection bit (zzzz) */
            case 0x5: A1_HW = DSP_ALU_DST_HW;
                A1_LW = 0x0;             /* clear LSW */
                A1G = 0x0;               /* clear Guard bits */
        break;
            case 0x7: A0_HW = DSP_ALU_DST_HW;
                A0_LW = 0x0;             /* clear LSW */
                A0G = 0x0;               /* clear Guard bits */
        break;
            case 0x8: X0_HW = DSP_ALU_DST_HW;
                X0_LW = 0x0;             /* clear LSW */
        break;
            case 0x9: X1_HW = DSP_ALU_DST;
                X1_LW = 0x0;             /* clear LSW */
        break;
            case 0xa: Y0_HW = DSP_ALU_DST;
                Y0_LW = 0x0;             /* clear LSW */
        break;
            case 0xb: Y1_HW = DSP_ALU_DST;
                Y1_LW = 0x0;             /* clear LSW */
        break;
            case 0xc: M0_HW = DSP_ALU_DST;
                M0_LW = 0x0;             /* clear LSW */
        break;
            case 0xe: M1_HW = DSP_ALU_DST;
                M1_LW = 0x0;             /* clear LSW */
        break;
            default:  printf("\nERROR:Illegal DSP Instruction");
    break;
    }

        carry_bit    = 0x0;
        negative_bit = DSP_ALU_DST_MSB;
        zero_bit     = (DSP_ALU_DST_HW==0);
        overflow_bit = 0x0;

        /* DSR register update */
        logical_dc_bit();
     }
     else if(DSP_CONDITION_MATCH) { /* conditional operation and match
*/
    /* ALU Destination assignment */
    switch (zzzz) {  /* Dz Operand selection bit (zzzz) */
            case 0x5: A1_HW = DSP_ALU_DST_HW;
                A1_LW = 0x0;             /* clear LSW */
                A1G = 0x0;               /* clear Guard bits */
        break;
            case 0x7: A0_HW = DSP_ALU_DST_HW;
                A0_LW = 0x0;             /* clear LSW */
                A0G = 0x0;               /* clear Guard bits */
        break;
            case 0x8: X0_HW = DSP_ALU_DST_HW;
                X0_LW = 0x0;            /* clear LSW */
        break;
            case 0x9: X1_HW = DSP_ALU_DST;
                X1_LW = 0x0;            /* clear LSW */
        break;
            case 0xa: Y0_HW = DSP_ALU_DST;
                Y0_LW = 0x0;            /* clear LSW */
        break;
            case 0xb: Y1_HW = DSP_ALU_DST;
                Y1_LW = 0x0;            /* clear LSW */
        break;
            case 0xc: M0_HW = DSP_ALU_DST;
                M0_LW = 0x0;            /* clear LSW */
        break;
            case 0xe: M1_HW = DSP_ALU_DST;
                M1_LW = 0x0;            /* clear LSW */
        break;
            default:  printf("\nERROR:Illegal DSP Instruction");
    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.

6.3.6 [if cc] PCLR (Clear): DSP Arithmetic Operation Instruction

Format	Abstract	Code	Cycle	DC Bit	Applicable Instructions SH-1	SH-2	SH-DSP
`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. If conditions are specified, the DC, N, Z, V, and GT bits are not updated even is the conditions were true and the instruction was executed.

Operation:

```
/*  PCLR Dz     */

{
unsigned char carry_bit, negative_bit, zero_bit, overflow_bit;

    if(DSP_UNCONDITIONAL_UPDATE) { /* unconditional operation */

   /* ALU Destination assignment */
   switch (zzzz) {  /* Dz Operand selection bit (zzzz) */
          case 0x5: A1 = 0x0;
             A1G = 0x0;
      break;
          case 0x7: A0 = 0x0;
             A0G = 0x0;
      break;
          case 0x8: X0 = 0x0;
```

```
            break;
                case 0x9: X1 = 0x0;
            break;
                case 0xa: Y0 = 0x0;
            break;
                case 0xb: Y1 = 0x0;
            break;
                case 0xc: M0 = 0x0;
            break;
                case 0xe: M1 = 0x0;
            break;
                default:  printf("\nERROR:Illegal DSP Instruction");
        break;
        }

            carry_bit    = 0;
            negative_bit = 0;
            zero_bit     = 1;
            overflow_bit = 0;


            /* DSR register update */
            plus_dc_bit();
         }
         else if(DSP_CONDITION_MATCH) { /* conditional operation and match
    */

        /* ALU Destination assignment */
        switch (zzzz) {  /* Dz Operand selection bit (zzzz) */
                case 0x5: A1 = 0x0;
                    A1G = 0x0;
            break;
                case 0x7: A0 = 0x0;
                    A0G = 0x0;
            break;
                case 0x8: X0 = 0x0;
            break;
                case 0x9: X1 = 0x0;
        break;
           case 0xa: Y0 = 0x0;
        break;
           case 0xb: Y1 = 0x0;
        break;
           case 0xc: M0 = 0x0;
        break;
           case 0xe: M1 = 0x0;
        break;
           default:  printf("\nERROR:Illegal DSP Instruction");
     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].

6.3.7 PCMP (Compare Two Data): DSP Arithmetic Operation Instruction

Format	Abstract	Code	Cycle	DC Bit	Applicable Instructions		
					SH-1	SH-2	SH-DSP
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:

```
/*  PCMP Sx,Sy     */

{
unsigned char carry_bit, borrow_bit, negative_bit, zero_bit,
overflow_bit;

/* ALU Sources assignment */
   switch (xx) {    /* Sx Operand selection bit (xx) */
      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 (yy) {    /* Sy Operand selection bit (yy) */
        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);

    overflow_protection();

    /* DSR register update */
    minus_dc_bit();

}
```

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].	

6.3.8 [if cc] PCOPY (Copy with Condition): DSP Arithmetic Operation Instruction

					Applicable Instructions		
Format	**Abstract**	**Code**	**Cycle**	**DC Bit**	**SH-1**	**SH-2**	**SH-DSP**
`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. If conditions are specified, the DC, N, Z, V, and GT bits are not updated even is the conditions were true and the instruction was executed.

Operation:

```
/*  Case1 : PCOPY Sx,Dz     */
/*  Case2 : PCOPY Sy,Dz     */

{
unsigned char carry_bit, negative_bit, zero_bit, overflow_bit;

/* ALU Sources assignment */

    if (Case1) {     /* PCOPY Sx,Dz */
```

```
        switch (xx) {    /* Sx Operand selection bit (xx) */
            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  {       /* PCOPY Sy,Dz */
        DSP_ALU_SRC1 = 0;
        DSP_ALU_SRC1G= 0;

        switch (yy) {
            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);
    overflow_protection();

     if(DSP_UNCONDITIONAL_UPDATE) { /* unconditional operation */

    /* ALU Destination assignment */
    switch (zzzz) {  /* Dz Operand selection bit (zzzz) */
           case 0x5: A1 = DSP_ALU_DST;
                A1G = DSP_ALU_DSTG & 0x000000FF;
                if(DSP_ALU_DSTG_BIT7) A1G = A1G | 0xFFFFFF00;
        break;
           case 0x7: A0 = DSP_ALU_DST;
                A0G = DSP_ALU_DSTG & 0x000000FF;
                if(DSP_ALU_DSTG_BIT7) A0G = A0G | 0xFFFFFF00;
        break;
           case 0x8: X0 = DSP_ALU_DST;
        break;
           case 0x9: X1 = DSP_ALU_DST;
        break;
           case 0xa: Y0 = DSP_ALU_DST;
        break;
           case 0xb: Y1 = DSP_ALU_DST;
        break;
           case 0xc: M0 = DSP_ALU_DST;
        break;
           case 0xe: M1 = DSP_ALU_DST;
        break;
           default:  printf("\nERROR:Illegal DSP Instruction");
    break;
    }
    negative_bit = DSP_ALU_DSTG_BIT7;
    zero_bit = (DSP_ALU_DST==0) & (DSP_ALU_DSTG_LSB8==0);

    /* DSR register update */
    plus_dc_bit();

     }
     else if(DSP_CONDITION_MATCH) { /* conditional operation and match
*/

    /* ALU Destination assignment */
    switch (zzzz) {  /* Dz Operand selection bit (zzzz) */
            case 0x5: A1 = DSP_ALU_DST;
                A1G = DSP_ALU_DSTG & 0x000000FF;
                if(DSP_ALU_DSTG_BIT7) A1G = A1G | 0xFFFFFF00;
        break;
            case 0x7: A0 = DSP_ALU_DST;
                A0G = DSP_ALU_DSTG & 0x000000FF;
                if(DSP_ALU_DSTG_BIT7) A0G = A0G | 0xFFFFFF00;
        break;
            case 0x8: X0 = DSP_ALU_DST;
        break;
            case 0x9: X1 = DSP_ALU_DST;
        break;
            case 0xa: Y0 = DSP_ALU_DST;
        break;
            case 0xb: Y1 = DSP_ALU_DST;
        break;
            case 0xc: M0 = DSP_ALU_DST;
        break;
            case 0xe: M1 = DSP_ALU_DST;
        break;
            default:  printf("\nERROR:Illegal DSP Instruction");
    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].	

6.3.9 [if cc] PDEC (Decrement by 1): DSP Arithmetic Operation Instruction

					Applicable Instructions		
Format	**Abstract**	**Code**	**Cycle**	**DC Bit**	**SH-1**	**SH-2**	**SH-DSP**
`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. If conditions are specified, the DC, N, Z, V, and GT bits are not updated even is the conditions were true and the instruction was executed.

Note: The bottom word of the destination register is ignored when the DC bit is updated.

Operation:

```
/*  Case1 : PDEC Sx,Dz     */
/*  Case2 : PDEC Sy,Dz     */

{
unsigned char carry_bit, borrow_bit, negative_bit, zero_bit,
overflow_bit;

/* ALU Sources assignment */
   DSP_ALU_SRC2 = 0x1;

   DSP_ALU_SRC2G= 0x0;

   if (Case1) {     /* MSW of Sx -1 → Dz */
       switch (xx) {   /* Sx Operand selection bit (xx) */
      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 (yy) {   /* Sy Operand selection bit (yy) */
      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);

    overflow_protection();

    if(DSP_UNCONDITIONAL_UPDATE) { /* unconditional operation */

        /* ALU Destination assignment */
    switch (zzzz) {  /* Dz Operand selection bit (zzzz) */
           case 0x5: A1_HW = DSP_ALU_DST_HW;
              A1_LW = 0x0;              /* clear LSW */
              A1G = DSP_ALU_DSTG & 0x000000FF;
              if(DSP_ALU_DSTG_BIT7) A1G = A1G | 0xFFFFFF00;
        break;
           case 0x7: A0_HW = DSP_ALU_DST_HW;
              A0_LW = 0x0;              /* clear LSW */
              A0G = DSP_ALU_DSTG & 0x000000FF;
              if(DSP_ALU_DSTG_BIT7) A0G = A0G | 0xFFFFFF00;
        break;
           case 0x8: X0_HW = DSP_ALU_DST_HW;
              X0_LW = 0x0;              /* clear LSW */
        break;
```

```
          case 0x9: X1_HW = DSP_ALU_DST_HW;
              X1_LW = 0x0;              /* clear LSW */
      break;
          case 0xa: Y0_HW = DSP_ALU_DST_HW;
              Y0_LW = 0x0;              /* clear LSW */
      break;
          case 0xb: Y1_HW = DSP_ALU_DST_HW;
              Y1_LW = 0x0;              /* clear LSW */
      break;
          case 0xc: M0_HW = DSP_ALU_DST_HW;
              M0_LW = 0x0;              /* clear LSW */
      break;
          case 0xe: M1_HW = DSP_ALU_DST_HW;
              M1_LW = 0x0;              /* clear LSW */
      break;
          default:  printf("\nERROR:Illegal DSP Instruction");
  break;
  }
  negative_bit = DSP_ALU_DSTG_BIT7;
  zero_bit = (DSP_ALU_DST_HW==0) & (DSP_ALU_DSTG_LSB8==0);

  /* DSR register update */
  minus_dc_bit.c"
   }
   else if(DSP_CONDITION_MATCH) { /* conditional operation and match
*/

      /* ALU Destination assignment */
  switch (zzzz) {  /* Dz Operand selection bit (zzzz) */
          case 0x5: A1_HW = DSP_ALU_DST_HW;
              A1_LW = 0x0;              /* clear LSW */
              A1G = DSP_ALU_DSTG & 0x000000FF;
              if(DSP_ALU_DSTG_BIT7) A1G = A1G | 0xFFFFFF00;
      break;
          case 0x7: A0_HW = DSP_ALU_DST_HW;
              A0_LW = 0x0;              /* clear LSW */
              A0G = DSP_ALU_DSTG & 0x000000FF;
```

```
                if(DSP_ALU_DSTG_BIT7) A0G = A0G | 0xFFFFFF00;
          break;
             case 0x8: X0_HW = DSP_ALU_DST_HW;
                X0_LW = 0x0;              /* clear LSW */
          break;
             case 0x9: X1_HW = DSP_ALU_DST_HW;
                X1_LW = 0x0;              /* clear LSW */
          break;
             case 0xa: Y0_HW = DSP_ALU_DST_HW;
                Y0_LW = 0x0;              /* clear LSW */
          break;
             case 0xb: Y1_HW = DSP_ALU_DST_HW;
                Y1_LW = 0x0;              /* clear LSW */
          break;
             case 0xc: M0_HW = DSP_ALU_DST_HW;
                M0_LW = 0x0;              /* clear LSW */
          break;
             case 0xe: M1_HW = DSP_ALU_DST_HW;
                M1_LW = 0x0;              /* clear LSW */
          break;
             default:  printf("\nERROR:Illegal DSP Instruction");
       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].
```

6.3.10 [if cc] PDMSB (Detect MSB with Condition): DSP Arithmetic Operation Instruction

					Applicable Instructions		
Format	**Abstract**	**Code**	**Cycle**	**DC Bit**	**SH-1**	**SH-2**	**SH-DSP**
`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. If conditions are specified, the DC, N, Z, V, and GT bits are not updated even is the conditions were true and the instruction was executed.

Operation:

```
/*  Case1 : PDMSB Sx,Dz     */
/*  Case2 : PDMSB Sy,Dz     */

{
unsigned char carry_bit, borrow_bit, negative_bit, zero_bit,
overflow_bit;

/* ALU Sources assignment */

    DSP_ALU_SRC2 = 0x0;
    DSP_ALU_SRC2G= 0x0;

    if (Case1) {     /*  msb(Sx) → Dz */
        switch (xx) {   /* Sx Operand selection bit (xx) */
        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 (yy) {   /* Sy Operand selection bit (yy) */
        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;

        /* ALU Destination assignment */
     switch (zzzz) {  /* Dz Operand selection bit (zzzz) */
           case 0x5: A1_HW = DSP_ALU_DST_HW;
              A1_LW = 0x0;              /* clear LSW */
              A1G = DSP_ALU_DSTG & 0x000000FF;
              if(DSP_ALU_DSTG_BIT7) A1G = A1G | 0xFFFFFF00;
        break;
           case 0x7: A0_HW = DSP_ALU_DST_HW;
```

```
            A0_LW = 0x0;                /* clear LSW */
            A0G = DSP_ALU_DSTG & 0x000000FF;
            if(DSP_ALU_DSTG_BIT7) A0G = A0G | 0xFFFFFF00;
        break;
          case 0x8: X0_HW = DSP_ALU_DST_HW;
            X0_LW = 0x0;                /* clear LSW */
        break;
          case 0x9: X1_HW = DSP_ALU_DST_HW;
            X1_LW = 0x0;                /* clear LSW */
        break;
          case 0xa: Y0_HW = DSP_ALU_DST_HW;
            Y0_LW = 0x0;                /* clear LSW */
        break;
          case 0xb: Y1_HW = DSP_ALU_DST_HW;
            Y1_LW = 0x0;                /* clear LSW */
        break;
          case 0xc: M0_HW = DSP_ALU_DST_HW;
            M0_LW = 0x0;                /* clear LSW */
        break;
          case 0xe: M1_HW = DSP_ALU_DST_HW;
            M1_LW = 0x0;                /* clear LSW */
        break;
          default:  printf("\nERROR:Illegal DSP Instruction");
      break;
      }
      negative_bit = DSP_ALU_DSTG_BIT7;
      zero_bit = (DSP_ALU_DST_HW==0) & (DSP_ALU_DSTG_LSB8==0);

      /* DSR register update */
      plus_dc_bit();

       }
       else if(DSP_CONDITION_MATCH) { /* conditional operation and match
  */

        /* ALU Destination assignment */
      switch (zzzz) {  /* Dz Operand selection bit (zzzz) */
```

```
            case 0x5: A1_HW = DSP_ALU_DST_HW;
                A1_LW = 0x0;                /* clear LSW */
                A1G = DSP_ALU_DSTG & 0x000000FF;
                if(DSP_ALU_DSTG_BIT7) A1G = A1G | 0xFFFFFF00;
        break;
            case 0x7: A0_HW = DSP_ALU_DST_HW;
                A0_LW = 0x0;                /* clear LSW */
                A0G = DSP_ALU_DSTG & 0x000000FF;
                if(DSP_ALU_DSTG_BIT7) A0G = A0G | 0xFFFFFF00;
        break;
            case 0x8: X0_HW = DSP_ALU_DST_HW;
                X0_LW = 0x0;                /* clear LSW */
        break;
            case 0x9: X1_HW = DSP_ALU_DST_HW;
                X1_LW = 0x0;                /* clear LSW */
        break;
            case 0xa: Y0_HW = DSP_ALU_DST_HW;
                Y0_LW = 0x0;                /* clear LSW */
        break;
            case 0xb: Y1_HW = DSP_ALU_DST_HW;
                Y1_LW = 0x0;                /* clear LSW */
        break;
            case 0xc: M0_HW = DSP_ALU_DST_HW;
                M0_LW = 0x0;                /* clear LSW */
        break;
            case 0xe: M1_HW = DSP_ALU_DST_HW;
                M1_LW = 0x0;                /* clear LSW */
        break;
            default:  printf("\nERROR:Illegal DSP Instruction");
    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].

6.3.11 [if cc] PINC (Increment by 1 with Condition): DSP Arithmetic Operation Instruction

					Applicable Instructions		
Format	**Abstract**	**Code**	**Cycle**	**DC Bit**	**SH-1**	**SH-2**	**SH-DSP**
`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. If conditions are specified, the DC, N, Z, V, and GT bits are not updated even is the conditions were true and the instruction was executed.

Note: The bottom word of the destination register is ignored when the DC bit is updated.

Operation:

```
/*  Case1 : PINC Sx,Dz      */
/*  Case2 : PINC Sy,Dz      */

{
unsigned char carry_bit, borrow_bit, negative_bit, zero_bit,
overflow_bit;

/* ALU Sources assignment */

   DSP_ALU_SRC2 = 0x1;
   DSP_ALU_SRC2G= 0x0;

   if (Case1) {    /* MSW of Sx +1 → Dz */
       switch (xx) {   /* Sx Operand selection bit (xx) */
      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 (yy) {   /* Sy Operand selection bit (yy) */
      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);
          overflow_protection();

          if(DSP_UNCONDITIONAL_UPDATE) { /* unconditional operation */

             /* ALU Destination assignment */
          switch (zzzz) {  /* Dz Operand selection bit (zzzz) */
                case 0x5: A1_HW = DSP_ALU_DST_HW;
                   A1_LW = 0x0;              /* clear LSW */
                   A1G = DSP_ALU_DSTG & 0x000000FF;
                   if(DSP_ALU_DSTG_BIT7) A1G = A1G | 0xFFFFFF00;
             break;
                case 0x7: A0_HW = DSP_ALU_DST_HW;
                   A0_LW = 0x0;              /* clear LSW */
                   A0G = DSP_ALU_DSTG & 0x000000FF;
                   if(DSP_ALU_DSTG_BIT7) A0G = A0G | 0xFFFFFF00;
             break;
                case 0x8: X0_HW = DSP_ALU_DST_HW;
                   X0_LW = 0x0;              /* clear LSW */
             break;
                case 0x9: X1_HW = DSP_ALU_DST_HW;
                   X1_LW = 0x0;              /* clear LSW */
```

```
        break;
            case 0xa: Y0_HW = DSP_ALU_DST_HW;
                Y0_LW = 0x0;              /* clear LSW */
        break;
            case 0xb: Y1_HW = DSP_ALU_DST_HW;
                Y1_LW = 0x0;              /* clear LSW */
        break;
            case 0xc: M0_HW = DSP_ALU_DST_HW;
                M0_LW = 0x0;              /* clear LSW */
        break;
            case 0xe: M1_HW = DSP_ALU_DST_HW;
                M1_LW = 0x0;              /* clear LSW */
        break;
            default:  printf("\nERROR:Illegal DSP Instruction");
    break;
    }
    negative_bit = DSP_ALU_DSTG_BIT7;
    zero_bit = (DSP_ALU_DST_HW==0) & (DSP_ALU_DSTG_LSB8==0);

    /* DSR register update */
    plus_dc_bit();

     }
     else if(DSP_CONDITION_MATCH) { /* conditional operation and match
*/

        /* ALU Destination assignment */
    switch (zzzz) {  /* Dz Operand selection bit (zzzz) */
            case 0x5: A1_HW = DSP_ALU_DST_HW;
                A1_LW = 0x0;              /* clear LSW */
                A1G = DSP_ALU_DSTG & 0x000000FF;
                if(DSP_ALU_DSTG_BIT7) A1G = A1G | 0xFFFFFF00;
        break;
            case 0x7: A0_HW = DSP_ALU_DST_HW;
                A0_LW = 0x0;              /* clear LSW */
                A0G = DSP_ALU_DSTG & 0x000000FF;
                if(DSP_ALU_DSTG_BIT7) A0G = A0G | 0xFFFFFF00;
        break;
            case 0x8: X0_HW = DSP_ALU_DST_HW;
                X0_LW = 0x0;              /* clear LSW */
        break;
            case 0x9: X1_HW = DSP_ALU_DST_HW;
                X1_LW = 0x0;              /* clear LSW */
        break;
            case 0xa: Y0_HW = DSP_ALU_DST_HW;
                Y0_LW = 0x0;              /* clear LSW */
        break;
            case 0xb: Y1_HW = DSP_ALU_DST_HW;
                Y1_LW = 0x0;              /* clear LSW */
        break;
            case 0xc: M0_HW = DSP_ALU_DST_HW;
                M0_LW = 0x0;              /* clear LSW */
        break;
            case 0xe: M1_HW = DSP_ALU_DST_HW;
                M1_LW = 0x0;              /* clear LSW */
        break;
            default:  printf("\nERROR:Illegal DSP Instruction");
    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].
```

6.3.12 [if cc] PLDS (Load System Register): DSP System Control Instruction

					Applicable Instructions		
Format	**Abstract**	**Code**	**Cycle**	**DC Bit**	**SH-1**	**SH-2**	**SH-DSP**
`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:

```
/*  Case1 : PLDS Dz,MACH     */
/*  Case2 : PLDS Dz,MACL     */

{

  if(CASE1){  /* Dz → MACH */
    if(DSP_UNCONDITIONAL_UPDATE) { /* unconditional operation */
    /* ALU Destination assignment */
    switch (zzzz) {  /* Dz Operand selection bit (zzzz) */
          case 0x5: MACH = A1;
       break;
          case 0x7: MACH = A0;
       break;
          case 0x8: MACH = X0;
       break;
          case 0x9: MACH = X1;
       break;
          case 0xa: MACH = Y0;
       break;
          case 0xb: MACH = Y1;
       break;
          case 0xc: MACH = M0;
       break;
          case 0xe: MACH = M1;
       break;
          default:  printf("\nERROR:Illegal DSPInstruction");
    break;
    }
       }
    else if(DSP_CONDITION_MATCH) { /* conditional operation and match
*/

    /* ALU Destination assignment */
    switch (zzzz) {  /* Dz Operand selection bit (zzzz) */
          case 0x5: MACH = A1;
       break;
```

```
            case 0x7: MACH = A0;
         break;
            case 0x8: MACH = X0;
         break;
            case 0x9: MACH = X1;
         break;
            case 0xa: MACH = Y0;
         break;
            case 0xb: MACH = Y1;
         break;
            case 0xc: MACH = M0;
         break;
            case 0xe: MACH = M1;
         break;
            default:  printf("\nERROR:Illegal DSP Instruction");
      break;
      }
         }
    else{    /* Dz → MACL */
      if(DSP_UNCONDITIONAL_UPDATE) { /* unconditional operation */

      /* ALU Destination assignment */
      switch (zzzz) {  /* Dz Operand selection bit (zzzz) */
            case 0x5: MACL = A1;
         break;
            case 0x7: MACL = A0;
         break;
            case 0x8: MACL = X0;
         break;
            case 0x9: MACL = X1;
         break;
            case 0xa: MACL = Y0;
         break;
            case 0xb: MACL = Y1;
         break;
            case 0xc: MACL = M0;
            break;
                case 0xe: MACL = M1;
            break;
                default:  printf("\nERROR:Illegal DSP Instruction");
        break;
        }
            }
         else if(DSP_CONDITION_MATCH) { /* conditional operation and match
  */

        /* ALU Destination assignment */
        switch (zzzz) {  /* Dz Operand selection bit (zzzz) */
                case 0x5: MACL = A1;
            break;
                case 0x7: MACL = A0;
            break;
                case 0x8: MACL = X0;
            break;
                case 0x9: MACL = X1;
            break;
                case 0xa: MACL = Y0;
            break;
                case 0xb: MACL = Y1;
            break;
                case 0xc: MACL = M0;
            break;
                case 0xe: MACL = M1;
            break;
                default:  printf("\nERROR:Illegal DSP Instruction");
        break;
      }
        }
      }
  }
```

Example:

```
PLDS A0,MACH  NOPX  NOPY    ;Before execution:   A0=H'123456789A,
                                                 MACH=H'66666666
                            ;After execution:    A0=H'123456789A,
                                                 MACH=H'3456789A
```

6.3.13 PMULS (Multiply Signed by Signed): DSP Arithmetic Operation Instruction

					Applicable Instructions		
Format	**Abstract**	**Code**	**Cycle**	**DC Bit**	**SH-1**	**SH-2**	**SH-DSP**
`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 performs fixed decimal point multiplication, the operation result will be different from that of MULS, which performs integer multiplication, even though the source data may be the same.

Operation:

```
/*  PMULS Se,Sf,Dg      */

{
/* Multiplier Sources assignment */
    switch (ee) {    /* Se Operand selection bit (ee) */
        case 0x0: DSP_M_SRC1 = X0_HW;
                break;
        case 0x1: DSP_M_SRC1 = X1_HW;
                break;
        case 0x2: DSP_M_SRC1 = Y0_HW;
                break;
        case 0x3: DSP_M_SRC1 = A1_HW;
                break;
    }
    switch (ff) {    /* Sf Operand selection bit (ff) */
        case 0x0: DSP_M_SRC2 = Y0_HW;
                break;
        case 0x1: DSP_M_SRC2 = Y1_HW;
                break;
        case 0x2: DSP_M_SRC2 = X0_HW;
                break;
        case 0x3: DSP_M_SRC2 = A1_HW;
                break;
    }

/* Multiplier Operation */
    if ((SBIT==1) && (DSP_M_SRC1==0x8000) && (DSP_M_SRC2==0x8000)) {
        DSP_M_DST=0x7fffffff;  /* overflow protection */
    }
    else {

DSP_M_DST=((long)(short)DSP_M_SRC1*(long)(short)DSP_M_SRC2)<<1;
    }
    if (DSP_M_DST_MSB) DSP_M_DSTG_LSB8 = 0xff;
    else   DSP_M_DSTG_LSB8 = 0x0;

/* Multiplier Destination assignment */
    switch (gg) {    /* Dg Operand selection bit (gg) */
        case 0x0: M0 = DSP_M_DST;
                break;
        case 0x1:    M1 = DSP_M_DST;
                break;
        case 0x2: A0 = DSP_M_DST;
                if(DSP_M_DSTG_LSB8==0x0) A0G=0x0;
                else A0G=0xffffffff;
                break;
        case 0x3: A1 = DSP_M_DST;
                if(DSP_M_DSTG_LSB8==0x0) A1G=0x0;
                else A1G=0xffffffff;
                break;
    }
}
```

Examples:

```
PMULS X0,Y0,M0 NOPX NOPY   ;Before execution:  X0=H'00010000, Y0=H'00020000,
                                                  (2^-15)            (2^-14)
                                               M0=H'33333333
                           ;After execution:   X0=H'00010000, Y0=H'00020000,
                                               M0=H'00000004
                                                  (2^-24)
                           The value is doubled when viewed as integer data.
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

                            ( ): Fixed-point value
```

6.3.14 [if cc] PNEG (Negate): DSP Arithmetic Operation Instruction

					Applicable Instructions		
Format	Abstract	Code	Cycle	DC Bit	SH-1	SH-2	SH-DSP
`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. If conditions are specified, the DC, N, Z, V, and GT bits are not updated even is the conditions were true and the instruction was executed.

Operation:

```
/*  Case1 : PNEG Sx,Dz     */
/*  Case2 : PNEG Sy,Dz     */

{
unsigned char carry_bit, borrow_bit, negative_bit, zero_bit,
overflow_bit;

   DSP_ALU_SRC1 = 0;
   DSP_ALU_SRC1G= 0;

/* ALU Sources assignment */
   if (Case1) {     /* 0 - Sx → Dz */
      switch (xx) {    /* Sx Operand selection bit (xx) */
         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 (yy) {    /* Sy Operand selection bit (yy) */
         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);

    overflow_protection();

    if(DSP_UNCONDITIONAL_UPDATE) { /* unconditional operation */

    /* ALU Destination assignment */
    switch (zzzz) {  /* Dz Operand selection bit (zzzz) */
           case 0x5: A1 = DSP_ALU_DST;
               A1G = DSP_ALU_DSTG & 0x000000FF;
               if(DSP_ALU_DSTG_BIT7) A1G = A1G | 0xFFFFFF00;
        break;
           case 0x7: A0 = DSP_ALU_DST;
               A0G = DSP_ALU_DSTG & 0x000000FF;
               if(DSP_ALU_DSTG_BIT7) A0G = A0G | 0xFFFFFF00;
        break;
           case 0x8: X0 = DSP_ALU_DST;
        break;
           case 0x9: X1 = DSP_ALU_DST;
```

```
            break;
                case 0xa: Y0 = DSP_ALU_DST;
            break;
                case 0xb: Y1 = DSP_ALU_DST;
            break;
                case 0xc: M0 = DSP_ALU_DST;
            break;
                case 0xe: M1 = DSP_ALU_DST;
            break;

                default:  printf("\nERROR:Illegal DSP Instruction");
        break;
        }

        negative_bit = DSP_ALU_DSTG_BIT7;
        zero_bit = (DSP_ALU_DST==0) & (DSP_ALU_DSTG_LSB8==0);

        /* DSR register update */
        minus_dc_bit();
         }
         else if(DSP_CONDITION_MATCH) { /* conditional operation and match
    */

        /* ALU Destination assignment */
        switch (zzzz) {  /* Dz Operand selection bit (zzzz) */
                case 0x5: A1 = DSP_ALU_DST;
                    A1G = DSP_ALU_DSTG & 0x000000FF;
                    if(DSP_ALU_DSTG_BIT7) A1G = A1G | 0xFFFFFF00;
            break;
                case 0x7: A0 = DSP_ALU_DST;
                    A0G = DSP_ALU_DSTG & 0x000000FF;
                    if(DSP_ALU_DSTG_BIT7) A0G = A0G | 0xFFFFFF00;
            break;
                case 0x8: X0 = DSP_ALU_DST;
            break;
                case 0x9: X1 = DSP_ALU_DST;
            break;
                case 0xa: Y0 = DSP_ALU_DST;
        break;
           case 0xb: Y1 = DSP_ALU_DST;
        break;
           case 0xc: M0 = DSP_ALU_DST;
        break;
           case 0xe: M1 = DSP_ALU_DST;
        break;
           default:  printf("\nERROR:Illegal DSP Instruction");
     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].

6.3.15 [if cc] POR (Logical OR): DSP Logical Operation Instruction

					Applicable Instructions		
Format	**Abstract**	**Code**	**Cycle**	**DC Bit**	**SH-1**	**SH-2**	**SH-DSP**
`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. If conditions are specified, the DC, N, Z, V, and GT bits are not updated even is the conditions were true and the instruction was executed.

Note: The bottom word of the destination register and the guard bits are ignored when the DC bit is updated.

Operation:

```
/*  POR Sx,Sy,Dz     */

{
unsigned char carry_bit, negative_bit, zero_bit, overflow_bit;

/* ALU Sources assignment */
   switch (xx) {    /* Sx Operand selection bit (xx) */
      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 (yy) {    /* Sy Operand selection bit (yy) */
      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 */

   /* ALU Destination assignment */
   switch (zzzz) {  /* Dz Operand selection bit (zzzz) */
         case 0x5: A1_HW = DSP_ALU_DST_HW;
            A1_LW = 0x0;              /* clear LSW */
            A1G = 0x0;   /* clear Guard bits */
```

```
        break;
            case 0x7: A0_HW = DSP_ALU_DST_HW;
                A0_LW = 0x0;              /* clear LSW */
                A0G = 0x0;   /* clear Guard bits */
        break;
            case 0x8: X0_HW = DSP_ALU_DST_HW;
                X0_LW = 0x0;              /* clear LSW */
        break;
            case 0x9: X1_HW = DSP_ALU_DST;
                X1_LW = 0x0;              /* clear LSW */
        break;
            case 0xa: Y0_HW = DSP_ALU_DST;
                Y0_LW = 0x0;              /* clear LSW */
        break;
            case 0xb: Y1_HW = DSP_ALU_DST;
                Y1_LW = 0x0;              /* clear LSW */
        break;
            case 0xc: M0_HW = DSP_ALU_DST;
                M0_LW = 0x0;              /* clear LSW */
        break;
            case 0xe: M1_HW = DSP_ALU_DST;
                M1_LW = 0x0;              /* clear LSW */
        break;
            default:  printf("\nERROR:Illegal DSP Instruction");
    break;
    }

        carry_bit    = 0x0;
        negative_bit = DSP_ALU_DST_MSB;
        zero_bit     = (DSP_ALU_DST_HW==0);
        overflow_bit = 0x0;

        /* DSR register update */
        logical_dc_bit();
     }
     else if(DSP_CONDITION_MATCH) { /* conditional operation and match
*/
    /* ALU Destination assignment */
    switch (zzzz) {  /* Dz Operand selection bit (zzzz) */
          case 0x5: A1_HW = DSP_ALU_DST_HW;
             A1_LW = 0x0;              /* clear LSW */
             A1G = 0x0;                /* clear Guard bits */
       break;
          case 0x7: A0_HW = DSP_ALU_DST_HW;
             A0_LW = 0x0;              /* clear LSW */
             A0G = 0x0;                /* clear Guard bits */
       break;
          case 0x8: X0_HW = DSP_ALU_DST_HW;
             X0_LW = 0x0;              /* clear LSW */
       break;
          case 0x9: X1_HW = DSP_ALU_DST;
             X1_LW = 0x0;              /* clear LSW */
       break;
          case 0xa: Y0_HW = DSP_ALU_DST;
             Y0_LW = 0x0;              /* clear LSW */
       break;
          case 0xb: Y1_HW = DSP_ALU_DST;
             Y1_LW = 0x0;              /* clear LSW */
       break;
          case 0xc: M0_HW = DSP_ALU_DST;
             M0_LW = 0x0;              /* clear LSW */
       break;
          case 0xe: M1_HW = DSP_ALU_DST;
             M1_LW = 0x0;              /* clear LSW */
       break;
          default:  printf("\nERROR:Illegal DSP Instruction");
    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].

6.3.16 PRND (Rounding): DSP Arithmetic Operation Instruction

					Applicable Instructions		
Format	**Abstract**	**Code**	**Cycle**	**DC Bit**	**SH-1**	**SH-2**	**SH-DSP**

`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:

```
/*  Case1 : PRND Sx,Dz      */
/*  Case2 : PRND Sy,Dz      */

{
unsigned char carry_bit, borrow_bit, negative_bit, zero_bit,
overflow_bit;

/* ALU Sources assignment */

    DSP_ALU_SRC2 = 0x00008000;
    DSP_ALU_SRC2G= 0x0;

    if (Case1) {  /* Sx + H'00008000 → Dz; clr Dz LW */
        switch (xx) {   /* Sx Operand selection bit (xx) */
        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 (yy) {   /* Sy Operand selection bit (yy) */
        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) & 0xFFFF0000;
    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);

    overflow_protection();

        /* ALU Destination assignment */
    switch (zzzz) {  /* Dz Operand selection bit (zzzz) */
```

```
        case 0x5: A1_HW = DSP_ALU_DST_HW;
            A1_LW = 0x0;              /* clear LSW */
            A1G = DSP_ALU_DSTG & 0x000000FF;
            if(DSP_ALU_DSTG_BIT7) A1G = A1G | 0xFFFFFF00;
    break;
        case 0x7: A0_HW = DSP_ALU_DST_HW;
            A0_LW = 0x0;              /* clear LSW */
            A0G = DSP_ALU_DSTG & 0x000000FF;
            if(DSP_ALU_DSTG_BIT7) A0G = A0G | 0xFFFFFF00;
    break;
        case 0x8: X0_HW = DSP_ALU_DST_HW;
            X0_LW = 0x0;              /* clear LSW */
    break;
        case 0x9: X1_HW = DSP_ALU_DST_HW;
            X1_LW = 0x0;              /* clear LSW */
    break;
        case 0xa: Y0_HW = DSP_ALU_DST_HW;
            Y0_LW = 0x0;              /* clear LSW */
    break;
        case 0xb: Y1_HW = DSP_ALU_DST_HW;
            Y1_LW = 0x0;              /* clear LSW */
    break;
        case 0xc: M0_HW = DSP_ALU_DST_HW;
            M0_LW = 0x0;              /* clear LSW */
    break;
        case 0xe: M1_HW = DSP_ALU_DST_HW;
            M1_LW = 0x0;              /* clear LSW */
    break;
        default:  printf("\nERROR:Illegal DSP Instruction");
break;
}
negative_bit = DSP_ALU_DSTG_BIT7;
zero_bit = (DSP_ALU_DST_HW==0) & (DSP_ALU_DSTG_LSB8==0);

/* DSR register update */
plus_dc_bit();
```

```
}
```

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].
```

6.3.17 [if cc] PSHA (Shift Arithmetically with Condition): DSP Arithmetic Shift Instruction

					Applicable Instructions		
Format	**Abstract**	**Code**	**Cycle**	**DC Bit**	**SH-1**	**SH-2**	**SH-DSP**

`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. If conditions are specified, the DC, N, Z, V, and GT bits are not updated even is the conditions were true and the instruction was executed.

Operation:

```
/*  PSHA Sx,Sy,Dz     */
<When register operand is used>
{
unsigned char carry_bit, negative_bit, zero_bit, overflow_bit;

/* ALU Sources assignment */
   switch (xx) {    /* Sx Operand selection bit (xx) */
      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 (yy) {    /* Sy Operand selection bit (yy) */
      case 0x0: DSP_ALU_SRC2  = Y0 & 0x007F0000;
            break;
      case 0x1: DSP_ALU_SRC2  = Y1 & 0x007F0000;
            break;
      case 0x2: DSP_ALU_SRC2  = M0 & 0x007F0000;
            break;
      case 0x3: DSP_ALU_SRC2  = M1 & 0x007F0000;
            break;
   }
   if (DSP_ALU_SRC2_MSB) DSP_ALU_SRC2G = 0xff;
   else         DSP_ALU_SRC2G = 0x0;
```

```
    if((DSP_ALU_SRC2_HW & 0x0040)==0) {       /* Left Shift 0<=cnt<=32
*/
        char cnt = (DSP_ALU_SRC2_HW & 0x003F);
        if(cnt > 32) {
            printf("\nPSHA Sz,Sy,Dz \nError! 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))) & 0x000000FF;
        carry_bit = ((DSP_ALU_DSTG & 0x00000001)==0x1);
    }
    else   {            /* Right Shift 0< cnt <=32 */
        char cnt = ((~DSP_ALU_SRC2_HW & 0x003F)+1);
        if(cnt > 32) {
           printf("\nPSHA Sz,Sy,Dz \nError! 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) & 0x00000001)==0x1);
    }

    overflow_bit = !(POS_NOT_OV || NEG_NOT_OV);
    overflow_protection();

    if(DSP_UNCONDITIONAL_UPDATE) { /* unconditional operation */
```

```
/* ALU Destination assignment */
    switch (zzzz) { /* Dz Operand selection bit (zzzz) */
       case 0x5: A1 = DSP_ALU_DST;
           A1G = DSP_ALU_DSTG & 0x000000FF;
           if(DSP_ALU_DSTG_BIT7) A1G = A1G | 0xFFFFFF00;
   break;
       case 0x7: A0 = DSP_ALU_DST;
           A0G = DSP_ALU_DSTG & 0x000000FF;
           if(DSP_ALU_DSTG_BIT7) A0G = A0G | 0xFFFFFF00;
   break;
       case 0x8: X0 = DSP_ALU_DST;
   break;
       case 0x9: X1 = DSP_ALU_DST;
   break;
       case 0xa: Y0 = DSP_ALU_DST;
   break;
       case 0xb: Y1 = DSP_ALU_DST;
   break;
       case 0xc: M0 = DSP_ALU_DST;
   break;
       case 0xe: M1 = DSP_ALU_DST;
   break;
       default:  printf("\nERROR:Illegal DSP Instruction");
break;
    }

negative_bit = DSP_ALU_DSTG_BIT7;
zero_bit = (DSP_ALU_DST==0) & (DSP_ALU_DSTG_LSB8==0);

/* DSR register update */
shift_dc_bit();

}
else if(DSP_CONDITION_MATCH) { /* conditional operation and match */

/* ALU Destination assignment */
    switch (zzzz) { /* Dz Operand selection bit (zzzz) */
```

```
          case 0x5: A1 = DSP_ALU_DST;
             A1G = DSP_ALU_DSTG & 0x000000FF;
             if(DSP_ALU_DSTG_BIT7) A1G = A1G | 0xFFFFFF00;
       break;
          case 0x7: A0 = DSP_ALU_DST;
             A0G = DSP_ALU_DSTG & 0x000000FF;
             if(DSP_ALU_DSTG_BIT7) A0G = A0G | 0xFFFFFF00;
       break;
          case 0x8: X0 = DSP_ALU_DST;
       break;
          case 0x9: X1 = DSP_ALU_DST;
       break;
          case 0xa: Y0 = DSP_ALU_DST;
       break;
          case 0xb: Y1 = DSP_ALU_DST;
       break;
          case 0xc: M0 = DSP_ALU_DST;
       break;
          case 0xe: M1 = DSP_ALU_DST;
       break;
          default:  printf("\nERROR:Illegal DSPInstruction");
    break;
        }
    }
}

/*  PSHA #Imm,Dz      */
<When register operand is used>
{
unsigned char carry_bit, negative_bit, zero_bit, overflow_bit;
unsigned short tmp_imm;

/* ALU Sources assignment */
        switch (zzzz) { /* Dz Operand selection bit (zzzz) */
          case 0x5: DSP_ALU_SRC1 = A1;
             DSP_ALU_SRC1G = A1G;
        break;
           case 0x7: DSP_ALU_SRC1 = A0;
              DSP_ALU_SRC1G = A1G;
        break;
           case 0x8: DSP_ALU_SRC1 = X0;
        break;
           case 0x9: DSP_ALU_SRC1 = X1;
        break;
           case 0xa: DSP_ALU_SRC1 = Y0;
        break;
           case 0xb: DSP_ALU_SRC1 = Y1;
        break;
           case 0xc: DSP_ALU_SRC1 = M0;
        break;
           case 0xe: DSP_ALU_SRC1 = M1;
        break;
           default:  printf("\nERROR:Illegal DSP Instruction");
     break;
         }
         if (DSP_ALU_SRC1_MSB) DSP_ALU_SRC1G = 0xff;
             else                   DSP_ALU_SRC1G = 0x0;

     tmp_imm = (#Imm) & 0x0000007F); /* Extract 7bit Immidiate Data */

     if((tmp_imm & 0x0040)==0) { /* Left Shift 0<= cnt <=32 */
         char cnt = (tmp_imm & 0x003F);
         if(cnt > 32) {
         printf("\nPSHA Dz,#Imm,Dz \nError! #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))) & 0x000000FF;
         carry_bit = ((DSP_ALU_DSTG & 0x00000001)==0x1);
     }
```

```
    else {          /* Right Shift 0< cnt <=32 */
        char cnt = ((~tmp_imm & 0x003F)+1);
        if(cnt > 32) {
        printf("\nPSHL Dz,#Imm,Dz \nError! #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) & 0x00000001)==0x1);
    }

    overflow_bit = !(POS_NOT_OV || NEG_NOT_OV);
    overflow_protection();

    { /* unconditional operation */
    /* ALU Destination assignment */
        switch (zzzz) { /* Dz Operand selection bit (zzzz) */
          case 0x5: A1 = DSP_ALU_DST;
              A1G = DSP_ALU_DSTG & 0x000000FF;
              if(DSP_ALU_DSTG_BIT7) A1G = A1G | 0xFFFFFF00;
       break;
          case 0x7: A0 = DSP_ALU_DST;
              A0G = DSP_ALU_DSTG & 0x000000FF;
              if(DSP_ALU_DSTG_BIT7) A0G = A0G | 0xFFFFFF00;
       break;
          case 0x8: X0 = DSP_ALU_DST;
       break;
          case 0x9: X1 = DSP_ALU_DST;
       break;
```

```
            case 0xa: Y0 = DSP_ALU_DST;
        break;
            case 0xb: Y1 = DSP_ALU_DST;
        break;
            case 0xc: M0 = DSP_ALU_DST;
        break;
            case 0xe: M1 = DSP_ALU_DST;
        break;
            default:  printf("\nERROR:Illegal DSP Instruction");
    break;
         }

    negative_bit = DSP_ALU_DSTG_BIT7;
    zero_bit = (DSP_ALU_DST==0) & (DSP_ALU_DSTG_LSB8==0);

    /* DSR register update */
    shift_dc_bit();
    }
}
```

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].

6.3.18 [if cc] PSHL (Shift Logically with Condition): DSP Logical Shift Instruction

					Applicable Instructions		
Format	**Abstract**	**Code**	**Cycle**	**DC Bit**	**SH-1**	**SH-2**	**SH-DSP**
`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. If conditions are specified, the DC, N, Z, V, and GT bits are not updated even is the conditions were true and the instruction was executed.

Operation:

```
<When register operand is used>
/*  PSHL Sx,Sy,Dz     */

unsigned char carry_bit, negative_bit, zero_bit, overflow_bit;

/* ALU Sources assignment */
   switch (xx) {    /* Sx Operand selection bit (xx) */
      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 (yy) {    /* Sy Operand selection bit (yy) */
      case 0x0: DSP_ALU_SRC2  = Y0 & 0x003F0000;
            break;
      case 0x1: DSP_ALU_SRC2  = Y1 & 0x003F0000;
            break;
      case 0x2: DSP_ALU_SRC2  = M0 & 0x003F0000;
            break;
      case 0x3: DSP_ALU_SRC2  = M1 & 0x003F0000;
            break;
   }
   if((DSP_ALU_SRC2_HW & 0x0020)==0) {       /* Left Shift 0<=cnt<=16
*/
       char cnt = (DSP_ALU_SRC2_HW & 0x001F);
       if(cnt > 16) {
       printf("PSHL Sx,Sy,Dz \nError! Shift %2X exceed range\n",cnt);
       exit();
           }
       DSP_ALU_DST_HW = DSP_ALU_SRC1_HW << cnt--;
       carry_bit = (((DSP_ALU_SRC1_HW << cnt) & 0x8000)==0x8000);
```

```
}
else   {            /* Right Shift 0<cnt<=16 */
    char cnt = ((~DSP_ALU_SRC2_HW & 0x000F)+1);
    if(cnt > 16) {
    printf("PSHL Sx,Sy,Dz \nError! Shift -%2X exceed range\n",cnt);
    exit();
         }
    DSP_ALU_DST_HW = DSP_ALU_SRC1_HW >> cnt--;
    carry_bit = (((DSP_ALU_SRC1_HW >> cnt) & 0x0001)==0x1);
}

if(DSP_UNCONDITIONAL_UPDATE) { /* unconditional operation */
/* ALU Destination assignment */
switch (zzzz) {  /* Dz Operand selection bit (zzzz) */
      case 0x5: A1_HW = DSP_ALU_DST_HW;
          A1_LW = 0x0;              /* clear LSW */
          A1G = 0x0;                /* clear Guard bits */
   break;
      case 0x7: A0_HW = DSP_ALU_DST_HW;
          A0_LW = 0x0;              /* clear LSW */
          A0G = 0x0;                /* clear Guard bits */
   break;
      case 0x8: X0_HW = DSP_ALU_DST_HW;
          X0_LW = 0x0;              /* clear LSW */
   break;
      case 0x9: X1_HW = DSP_ALU_DST;
          X1_LW = 0x0;              /* clear LSW */
   break;
      case 0xa: Y0_HW = DSP_ALU_DST;
          Y0_LW = 0x0;              /* clear LSW */
   break;
      case 0xb: Y1_HW = DSP_ALU_DST;
          Y1_LW = 0x0;              /* clear LSW */
   break;
      case 0xc: M0_HW = DSP_ALU_DST;
          M0_LW = 0x0;              /* clear LSW */
```

```
        break;
            case 0xe: M1_HW = DSP_ALU_DST;
                M1_LW = 0x0;              /* clear LSW */
        break;
            default:  printf("\nERROR:Illegal DSP Instruction");
    break;
    }

        carry_bit    = 0x0;
        negative_bit = DSP_ALU_DST_MSB;
        zero_bit     = (DSP_ALU_DST_HW==0);
        overflow_bit = 0x0;


        /* DSR register update */
        shift_dc_bit();
    }

    else if(DSP_CONDITION_MATCH) { /* conditional operation and match */

    /* ALU Destination assignment */
         switch (zzzz) { /* Dz Operand selection bit (zzzz) */
            case 0x5: A1_HW = DSP_ALU_DST_HW;
                A1_LW = 0x0;              /* clear LSW */
                A1G = 0x0;                /* clear Guard bits */
        break;
            case 0x7: A0_HW = DSP_ALU_DST_HW;
                A0_LW = 0x0;              /* clear LSW */
                A0G = 0x0;                /* clear Guard bits */
        break;
            case 0x8: X0_HW = DSP_ALU_DST_HW;
                X0_LW = 0x0;              /* clear LSW */
        break;
            case 0x9: X1_HW = DSP_ALU_DST;
                X1_LW = 0x0;              /* clear LSW */
        break;
            case 0xa: Y0_HW = DSP_ALU_DST;
                Y0_LW = 0x0;              /* clear LSW */
        break;
           case 0xb: Y1_HW = DSP_ALU_DST;
              Y1_LW = 0x0;              /* clear LSW */
        break;
           case 0xc: M0_HW = DSP_ALU_DST;
              M0_LW = 0x0;              /* clear LSW */
        break;
           case 0xe: M1_HW = DSP_ALU_DST;
              M1_LW = 0x0;              /* clear LSW */
        break;
           default:  printf("\nERROR:Illegal DSP Instruction");
     break;
         }
     }
  }

/*  PSHL #Imm,Dz      */
<When immediate operand is used>
{
unsigned char carry_bit, negative_bit, zero_bit, overflow_bit;
unsigned short tmp_imm;

/* ALU Sources assignment */
   switch (xx) {    /* Sx Operand selection bit (xx) */
      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 (yy) {    /* Sy Operand selection bit (yy) */
      case 0x0: DSP_ALU_SRC2  = Y0 & 0x003F0000;
             break;
        case 0x1: DSP_ALU_SRC2  = Y1 & 0x003F0000;
              break;
        case 0x2: DSP_ALU_SRC2  = M0 & 0x003F0000;
              break;
        case 0x3: DSP_ALU_SRC2  = M1 & 0x003F0000;
              break;
    }

    tmp_imm = (#Imm) & 0x0000007F); /* Extract 7bit Immediate Data */

    if((tmp_imm & 0x0020)==0) { /* Left Shift 0<= cnt <16 */
        char cnt = (tmp_imm & 0x001F);
        if(cnt > 16) {
        printf("PSHL Dz,#Imm,Dz \nError! #Imm=%6X exceed
 range\n",tmp_imm);
        exit();
        }
        DSP_ALU_DST_HW = DSP_ALU_SRC1_HW << cnt--;
        carry_bit = (((DSP_ALU_SRC1_HW << cnt) & 0x8000)==0x8000);
    }
    else   {        /* Right Shift 0< cnt <=16 */
        char cnt = ((~tmp_imm & 0x001F)+1);
        if(cnt > 16) {
        printf("PSHL Dz,#Imm,Dz \nError! #Imm=%6X exceed
 range\n",tmp_imm);
        exit();
        }
        DSP_ALU_DST_HW = DSP_ALU_SRC1_HW >> cnt--;
        carry_bit = (((DSP_ALU_SRC1_HW >> cnt) & 0x0001)==0x1);
    }

    { /* unconditional operation */

    /* ALU Destination assignment */
        switch (zzzz) { /* Dz Operand selection bit (zzzz) */
          case 0x5: A1_HW = DSP_ALU_DST_HW;
              A1_LW = 0x0;              /* clear LSW */
              A1G = 0x0;   /* clear Guard bits */
        break;
           case 0x7: A0_HW = DSP_ALU_DST_HW;
              A0_LW = 0x0;              /* clear LSW */
              A0G = 0x0;                /* clear Guard bits */
        break;
           case 0x8: X0_HW = DSP_ALU_DST_HW;
              X0_LW = 0x0;              /* clear LSW */
        break;
           case 0x9: X1_HW = DSP_ALU_DST;
              X1_LW = 0x0;              /* clear LSW */
        break;
           case 0xa: Y0_HW = DSP_ALU_DST;
              Y0_LW = 0x0;              /* clear LSW */
        break;
           case 0xb: Y1_HW = DSP_ALU_DST;
              Y1_LW = 0x0;              /* clear LSW */
        break;
           case 0xc: M0_HW = DSP_ALU_DST;
              M0_LW = 0x0;              /* clear LSW */
        break;
           case 0xe: M1_HW = DSP_ALU_DST;
              M1_LW = 0x0;              /* clear LSW */
        break;
           default:  printf("\nERROR:Illegal DSPInstruction");
     break;
         }

        carry_bit    = 0x0;
        negative_bit = DSP_ALU_DST_MSB;
        zero_bit     = (DSP_ALU_DST_HW==0);
        overflow_bit = 0x0;

        /* DSR register update */
        shift_dc_bit();
     }
  }
```

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].

6.3.19 [if cc] PSTS (Store System Register): DSP System Control Instruction

					Applicable Instructions		
Format	Abstract	Code	Cycle	DC Bit	SH-1	SH-2	SH-DSP
`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:

```
/*  Case1 : PSTS MACH,Dz     */
/*  Case2 : PSTS MACL,Dz     */

{

  if(CASE1){  /* MACH → Dz */
    if(DSP_UNCONDITIONAL_UPDATE) { /* unconditional operation */

    /* ALU Destination assignment */
    switch (zzzz) {  /* Dz Operand selection bit (zzzz) */
           case 0x5: A1 = MACH;
               A1G = DSP_ALU_DSTG & 0x000000FF;
               if(DSP_ALU_DSTG_BIT7) A1G = A1G | 0xFFFFFF00;
       break;
           case 0x7: A0 = MACH;
               A0G = DSP_ALU_DSTG & 0x000000FF;
               if(DSP_ALU_DSTG_BIT7) A0G = A0G | 0xFFFFFF00;
       break;
           case 0x8: X0 = MACH;
       break;
           case 0x9: X1 = MACH;
       break;
           case 0xa: Y0 = MACH;
       break;
           case 0xb: Y1 = MACH;
       break;
           case 0xc: M0 = MACH;
       break;
           case 0xe: M1 = MACH;
       break;
           default:  printf("\nERROR:Illegal DSP Instruction");
    break;
    }
     }
    else if(DSP_CONDITION_MATCH) { /* conditional operation and match
*/

   /* ALU Destination assignment */
   switch (zzzz) {  /* Dz Operand selection bit (zzzz) */
          case 0x5: A1 = MACH;
             A1G = DSP_ALU_DSTG & 0x000000FF;
             if(DSP_ALU_DSTG_BIT7) A1G = A1G | 0xFFFFFF00;
       break;
          case 0x7: A0 = MACH;
             A0G = DSP_ALU_DSTG & 0x000000FF;
             if(DSP_ALU_DSTG_BIT7) A0G = A0G | 0xFFFFFF00;
       break;
          case 0x8: X0 = MACH;
       break;
          case 0x9: X1 = MACH;
       break;
          case 0xa: Y0 = MACH;
       break;
          case 0xb: Y1 = MACH;
       break;
          case 0xc: M0 = MACH;
       break;
          case 0xe: M1 = MACH;
       break;
          default:  printf("\nERROR:Illegal DSP Instruction");
   break;
   }
    }
  else{    /* MACL → Dz */
    if(DSP_UNCONDITIONAL_UPDATE) { /* unconditional operation */

   /* ALU Destination assignment */
   switch (zzzz) {  /* Dz Operand selection bit (zzzz) */
          case 0x5: A1 = MACL;
             A1G = DSP_ALU_DSTG & 0x000000FF;
             if(DSP_ALU_DSTG_BIT7) A1G = A1G | 0xFFFFFF00;
```

```
        break;
           case 0x7: A0 = MACL;
              A0G = DSP_ALU_DSTG & 0x000000FF;
              if(DSP_ALU_DSTG_BIT7) A0G = A0G | 0xFFFFFF00;
        break;
           case 0x8: X0 = MACL;
        break;
           case 0x9: X1 = MACL;
        break;
           case 0xa: Y0 = MACL;
        break;
           case 0xb: Y1 = MACL;
        break;
           case 0xc: M0 = MACL;
        break;
           case 0xe: M1 = MACL;
        break;
           default:  printf("\nERROR:Illegal DSP Instruction");
     break;
     }
        }
      else if(DSP_CONDITION_MATCH) { /* conditional operation and match
  */

     /* ALU Destination assignment */
     switch (zzzz) {  /* Dz Operand selection bit (zzzz) */
           case 0x5: A1 = MACL;
              A1G = DSP_ALU_DSTG & 0x000000FF;
              if(DSP_ALU_DSTG_BIT7) A1G = A1G | 0xFFFFFF00;
        break;
           case 0x7: A0 = MACL;
              A0G = DSP_ALU_DSTG & 0x000000FF;
              if(DSP_ALU_DSTG_BIT7) A0G = A0G | 0xFFFFFF00;
        break;
           case 0x8: X0 = MACL;
        break;
```

```
            case 0x9: X1 = MACL;
         break;
            case 0xa: Y0 = MACL;
         break;
            case 0xb: Y1 = MACL;
         break;
            case 0xc: M0 = MACL;
         break;
            case 0xe: M1 = MACL;
         break;
            default:  printf("\nERROR:Illegal DSP Instruction");
      break;
     }
      }
     }
   }
```

Examples:

`PSTS MACH,A0 NOPX NOPY`	; Before execution:	A0=H'123456789A, MACH=H'88888888
	; After execution:	A0=H'FF88888888, MACH=H'88888888

6.3.20 [if cc]PSUB (Subtract with Condition): DSP Arithmetic Operation Instruction

					Applicable Instructions		
Format	**Abstract**	**Code**	**Cycle**	**DC Bit**	**SH-1**	**SH-2**	**SH-DSP**
`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. If conditions are specified, the DC, N, Z, V, and GT bits are not updated even is the conditions were true and the instruction was executed.

Operation:

```
/*  PSUB Sx,Sy,Dz      */

{
unsigned char carry_bit, borrow_bit, negative_bit, zero_bit,
overflow_bit;

/* ALU Sources assignment */
switch (xx) {    /* Sx Operand selection bit (xx) */
   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 (yy) {    /* Sy Operand selection bit (yy) */
   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);
overflow_protection();

    if(DSP_UNCONDITIONAL_UPDATE) { /* unconditional operation */

   /* ALU Destination assignment */
   switch (zzzz) {  /* Dz Operand selection bit (zzzz) */
         case 0x5: A1 = DSP_ALU_DST;
            A1G = DSP_ALU_DSTG & 0x000000FF;
            if(DSP_ALU_DSTG_BIT7) A1G = A1G | 0xFFFFFF00;
      break;
         case 0x7: A0 = DSP_ALU_DST;
            A0G = DSP_ALU_DSTG & 0x000000FF;
            if(DSP_ALU_DSTG_BIT7) A0G = A0G | 0xFFFFFF00;
      break;
         case 0x8: X0 = DSP_ALU_DST;
      break;
         case 0x9: X1 = DSP_ALU_DST;
      break;
         case 0xa: Y0 = DSP_ALU_DST;
      break;
         case 0xb: Y1 = DSP_ALU_DST;
      break;
         case 0xc: M0 = DSP_ALU_DST;
      break;
         case 0xe: M1 = DSP_ALU_DST;
      break;
         default:  printf("\nERROR:Illegal DSP Instruction");
   break;
   }
```

```
    negative_bit = DSP_ALU_DSTG_BIT7;
    zero_bit = (DSP_ALU_DST==0) & (DSP_ALU_DSTG_LSB8==0);

    /* DSR register update */
    minus_dc_bit();
     }
     else if(DSP_CONDITION_MATCH) { /* conditional operation and match
*/


    /* ALU Destination assignment */
    switch (zzzz) {  /* Dz Operand selection bit (zzzz) */
            case 0x5: A1 = DSP_ALU_DST;
               A1G = DSP_ALU_DSTG & 0x000000FF;
               if(DSP_ALU_DSTG_BIT7) A1G = A1G | 0xFFFFFF00;
       break;
            case 0x7: A0 = DSP_ALU_DST;
               A0G = DSP_ALU_DSTG & 0x000000FF;
               if(DSP_ALU_DSTG_BIT7) A0G = A0G | 0xFFFFFF00;
       break;
            case 0x8: X0 = DSP_ALU_DST;
       break;
            case 0x9: X1 = DSP_ALU_DST;
       break;
            case 0xa: Y0 = DSP_ALU_DST;
       break;
            case 0xb: Y1 = DSP_ALU_DST;
       break;
            case 0xc: M0 = DSP_ALU_DST;
       break;
            case 0xe: M1 = DSP_ALU_DST;
       break;
            default:  printf("\nERROR:Illegal DSPInstruction");
    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].

6.3.21 PSUB PMULS (Subtraction & Multiply Signed by Signed): DSP Arithmetic Operation Instruction

					Applicable Instructions		
Format	**Abstract**	**Code**	**Cycle**	**DC Bit**	**SH-1**	**SH-2**	**SH-DSP**
`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:

```
/*  PSUB Sx,Sy,Du PMULS Se,Sf,Dg   */

{
unsigned char carry_bit, borrow_bit, negative_bit, zero_bit,
overflow_bit;

/* Multiplier Sources assignment */
    switch (ee) {    /* Se Operand selection bit (ee) */
        case 0x0: DSP_M_SRC1 = X0_HW;
                break;
        case 0x1: DSP_M_SRC1 = X1_HW;
                break;
        case 0x2: DSP_M_SRC1 = Y0_HW;
                break;
        case 0x3: DSP_M_SRC1 = A1_HW;
                break;
    }
    switch (ff) {    /* Sf Operand selection bit (ff) */
       case 0x0: DSP_M_SRC2 = Y0_HW;
             break;
       case 0x1: DSP_M_SRC2 = Y1_HW;
             break;
       case 0x2: DSP_M_SRC2 = X0_HW;
             break;
       case 0x3: DSP_M_SRC2 = A1_HW;
             break;
    }

 /* ALU Sources assignment */
    switch (xx) {    /* Sx Operand selection bit (xx) */
       case 0x0: DSP_ALU_SRC1  = X0;
             if (DSP_ALU_SRC1_MSB)
                DSP_ALU_SRC1G_LSB8 = 0xff;
             else   DSP_ALU_SRC1G_LSB8 = 0x0;
             break;
       case 0x1: DSP_ALU_SRC1  = X1;
             if (DSP_ALU_SRC1_MSB)
                DSP_ALU_SRC1G_LSB8 = 0xff;
             else   DSP_ALU_SRC1G_LSB8 = 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 (yy) {    /* Sy Operand selection bit (yy) */
       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_LSB8 = 0xff;
    else          DSP_ALU_SRC2G_LSB8 = 0x0;

/* Multiplier Operation */

    /* PMULS Se, Sf, Dg */
    if ((SBIT==1) && (DSP_M_SRC1==0x8000) && (DSP_M_SRC2==0x8000)) {
            DSP_M_DST=0x7fffffff;  /* overflow protection */
    }
    else {
            DSP_M_DST=((long)(short)DSP_M_SRC1*(long)(short)DSP_M_SRC2)
<<1;
    }
    if (DSP_M_DST_MSB) DSP_M_DSTG_LSB8 = 0xff;
    else   DSP_M_DSTG_LSB8 = 0x0;

    switch (gg) {    /* Dg Operand selection bit (gg) */
        case 0x0: M0 = DSP_M_DST;
                break;
        case 0x1:    M1 = DSP_M_DST;
                break;
        case 0x2: A0 = DSP_M_DST;
                if(DSP_M_DSTG_LSB8==0x0) A0G=0x0;
                else A0G=0xffffffff;
                break;
        case 0x3: A1 = DSP_M_DST;
                if(DSP_M_DSTG_LSB8==0x0) A1G=0x0;
                else A1G=0xffffffff;
                break;
    }

/* ALU 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);
    overflow_protection();

    switch (uu) {    /* Du Operand selection bit (uu) */
       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 & 0x000000FF;
          if(DSP_ALU_DSTG_BIT7) A0G = A0G | 0xFFFFFF00;
          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 & 0x000000FF;
          if(DSP_ALU_DSTG_BIT7) A1G = A1G | 0xFFFFFF00;
          negative_bit = DSP_ALU_DSTG_BIT7;
          zero_bit = (DSP_ALU_DST==0) & (DSP_ALU_DSTG_LSB8==0);
          break;
    }
    /* DSR register update */
    minus_dc_bit();

}
```

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
```

6.3.22 PSUBC (Subtraction with Carry): DSP Arithmetic Operation Instruction

					Applicable Instructions		
Format	Abstract	Code	Cycle	DC Bit	SH-1	SH-2	SH-DSP
`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:

```
/*  PSUBC Sx,Sy,Dz     */

{
unsigned char carry_bit, borrow_bit, negative_bit, zero_bit,
overflow_bit;

/* ALU Sources assignment */
switch (xx) {    /* Sx Operand selection bit (xx) */
    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 (yy) {    /* Sy Operand selection bit (yy) */
   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);
overflow_protection();

   /* ALU Destination assignment */
   switch (zzzz) {  /* Dz Operand selection bit (zzzz) */
          case 0x5: A1 = DSP_ALU_DST;
             A1G = DSP_ALU_DSTG & 0x000000FF;
             if(DSP_ALU_DSTG_BIT7) A1G = A1G | 0xFFFFFF00;
       break;
          case 0x7: A0 = DSP_ALU_DST;
             A0G = DSP_ALU_DSTG & 0x000000FF;
             if(DSP_ALU_DSTG_BIT7) A0G = A0G | 0xFFFFFF00;
       break;
          case 0x8: X0 = DSP_ALU_DST;
```

```
        break;
            case 0x9: X1 = DSP_ALU_DST;
        break;
            case 0xa: Y0 = DSP_ALU_DST;
        break;
            case 0xb: Y1 = DSP_ALU_DST;
        break;
            case 0xc: M0 = DSP_ALU_DST;
        break;
            case 0xe: M1 = DSP_ALU_DST;
        break;
            default:  printf("\nERROR:Illegal DSPInstruction");
    break;
    }

    negative_bit = DSP_ALU_DSTG_BIT7;
    zero_bit = (DSP_ALU_DST==0) & (DSP_ALU_DSTG_LSB8==0);

    /* DSR register update */
    dc_always_borrow();
}
```

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'00 12345678, 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'00 12345678, DC=1
                            ; After execution:  X0=H'33333333, Y0=H'55555555
                                                M0=H'FFDDDDDDDD, DC=1
```

6.3.23 [if cc] PXOR (Logical Exclusive OR): DSP Logical Operation Instruction

Format	Abstract	Code	Cycle	DC Bit	Applicable Instructions		
					SH-1	SH-2	SH-DSP
`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. If conditions are specified, the DC, N, Z, V, and GT bits are not updated even is the conditions were true and the instruction was executed.

Note: The bottom word of the destination register and the guard bits are ignored when the DC bit is updated.

Operation:

```
/*  PXOR Sx,Sy,Dz     */

{
unsigned char carry_bit, negative_bit, zero_bit, overflow_bit;

/* ALU Sources assignment */
    switch (xx) {    /* Sx Operand selection bit (xx) */
        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 (yy) {    /* Sy Operand selection bit (yy) */
        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 */

    /* ALU Destination assignment */
    switch (zzzz) {  /* Dz Operand selection bit (zzzz) */
            case 0x5: A1_HW = DSP_ALU_DST_HW;
                A1_LW = 0x0;              /* clear LSW */
                A1G = 0x0;   /* clear Guard bits */
    break;
        case 0x7: A0_HW = DSP_ALU_DST_HW;
            A0_LW = 0x0;              /* clear LSW */
            A0G = 0x0;                /* clear Guard bits */
    break;
        case 0x8: X0_HW = DSP_ALU_DST_HW;
            X0_LW = 0x0;              /* clear LSW */
    break;
        case 0x9: X1_HW = DSP_ALU_DST;
            X1_LW = 0x0;              /* clear LSW */
    break;
        case 0xa: Y0_HW = DSP_ALU_DST;
            Y0_LW = 0x0;              /* clear LSW */
    break;
        case 0xb: Y1_HW = DSP_ALU_DST;
            Y1_LW = 0x0;              /* clear LSW */
    break;
        case 0xc: M0_HW = DSP_ALU_DST;
            M0_LW = 0x0;              /* clear LSW */
    break;
        case 0xe: M1_HW = DSP_ALU_DST;
            M1_LW = 0x0;              /* clear LSW */
    break;
        default:  printf("\nERROR:Illegal DSP Instruction");
break;
}

    carry_bit    = 0x0;
    negative_bit = DSP_ALU_DST_MSB;
    zero_bit     = (DSP_ALU_DST_HW==0);
    overflow_bit = 0x0;

    /* DSR register update */
    logical_dc_bit();

 }
```

```
     else if(DSP_CONDITION_MATCH) { /* conditional operation and match
 */

    /* ALU Destination assignment */
    switch (zzzz) {  /* Dz Operand selection bit (zzzz) */
            case 0x5: A1_HW = DSP_ALU_DST_HW;
               A1_LW = 0x0;                /* clear LSW */
               A1G = 0x0;                  /* clear Guard bits */
         break;
            case 0x7: A0_HW = DSP_ALU_DST_HW;
               A0_LW = 0x0;                /* clear LSW */
               A0G = 0x0;                  /* clear Guard bits */
         break;
            case 0x8: X0_HW = DSP_ALU_DST_HW;
               X0_LW = 0x0;                /* clear LSW */
         break;
            case 0x9: X1_HW = DSP_ALU_DST;
               X1_LW = 0x0;                /* clear LSW */
         break;
            case 0xa: Y0_HW = DSP_ALU_DST;
               Y0_LW = 0x0;                /* clear LSW */
         break;
            case 0xb: Y1_HW = DSP_ALU_DST;
               Y1_LW = 0x0;                /* clear LSW */
         break;
            case 0xc: M0_HW = DSP_ALU_DST;
               M0_LW = 0x0;                /* clear LSW */
         break;
            case 0xe: M1_HW = DSP_ALU_DST;
               M1_LW = 0x0;                /* clear LSW */
         break;
            default:  printf("\nERROR:Illegal DSP Instruction");
    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 7 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).

7.1 Basic Configuration of Pipelines

7.1.1 The Five-Stage Pipeline

Pipelines are composed of the following five stages:

1. IF (Instruction fetch)

   Fetches instruction from the memory where the program is stored.
2. ID (Instruction decode)

   Decodes the instruction fetched.
3. EX (Instruction execution)

   Does data operations and address calculations according to the results of decoding.
4. MA (Memory access)

   Accesses data in memory. Generated by instructions that involve memory access, with some exceptions.
5. WB/DSP (W/D) (Write back (CPU core) or DSP (DSP unit))

   Write Back: Returns the results of the memory access (data) to a register. Generated by instructions that involve memory loads, with some exceptions.

   DSP: Does operations using the DSP unit’s ALU and MAC. Also, the results of memory accesses (data) are returned to registers; not generated during writes to memory or no operation (NOP).

These stages flow with the execution of the instructions and thereby constitute a pipeline. At a given instant, five instructions are being executed simultaneously. The basic pipeline flow is as shown in figure 7.1. The period in which a single stage is operating is called a slot and is indicated by two-way arrows (←→).

All instructions have at least the 3 stages IF, ID and EX, but not all have stages MA and WB/DSP. The way the pipeline flows also varies with the type of instruction. Some pipelines differ, however, because of contention between IF and MA.

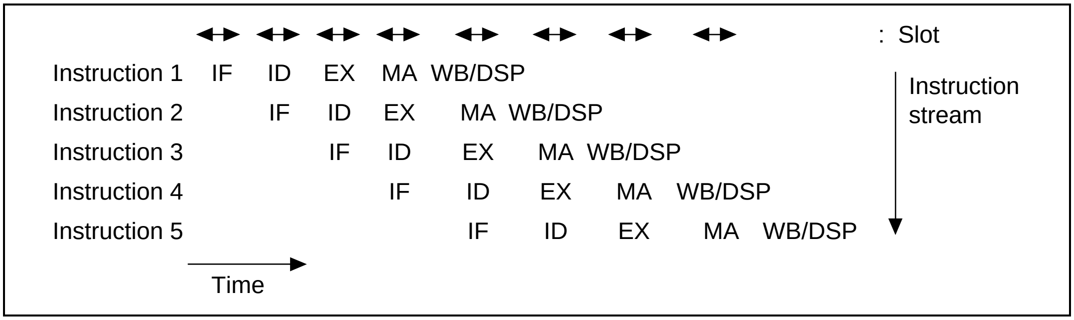

: Slot
Instruction 1 IF ID EX MA WB/DSP
Instruction 2 IF ID EX MA WB/DSP
Instruction 3 IF ID EX MA WB/DSP
Instruction 4 IF ID EX MA WB/DSP
Instruction 5 IF ID EX MA WB/DSP
Instruction stream
Time

Figure 7.1 Basic Structure of Pipeline Flow

7.1.2 Slot and Pipeline Flow

The time period in which a single stage operates called a slot. Slots must follow the rules described below.

All stages (IF, ID, EX, MA, WB/DSP) of an instruction must be executed in 1 slot. Two or more stages cannot be executed within 1 slot. Since WB/DSP is executed immediately after MA, however, some instructions may execute MA and WB/DSP within the same slot. Figures 7.2 and 7.3 show impossible pipeline flows.

Instruction Execution: Each stage (IF, ID, EX, MA, WB/DSP) of an instruction must be executed in one slot. Two or more stages cannot be executed within one slot (figure 7.2), with exception of WB and MA. Since WB is executed immediately after MA, however, some instructions may execute MA and WB within the same slot.

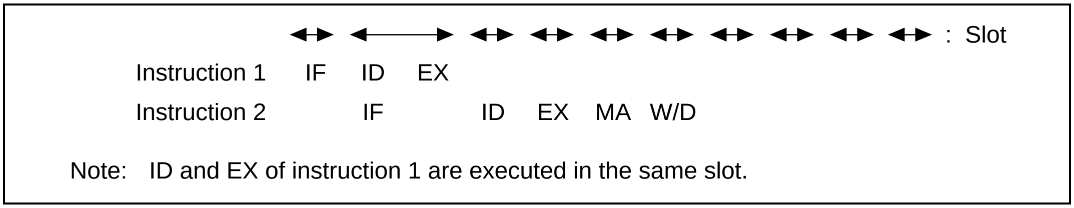

: Slot
Instruction 1 IF ID EX
Instruction 2 IF ID EX MA W/D
Note: ID and EX of instruction 1 are executed in the same slot.

Figure 7.2 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 7.3).

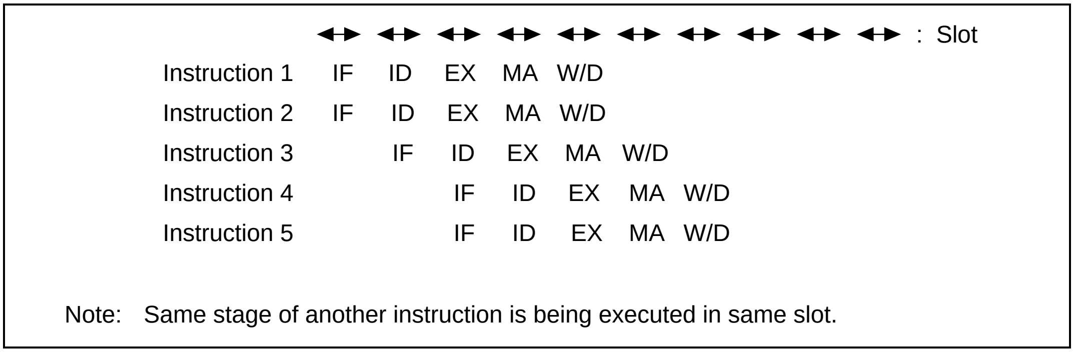

: Slot
Instruction 1 IF ID EX MA W/D
Instruction 2 IF ID EX MA W/D
Instruction 3 IF ID EX MA W/D
Instruction 4 IF ID EX MA W/D
Instruction 5 IF ID EX MA W/D
Note: Same stage of another instruction is being executed in same slot.

Figure 7.3 Impossible Pipeline Flow 2

7.1.3 Slot Length

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/DSP	Always one cycle

As an example, figure 7.4 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.

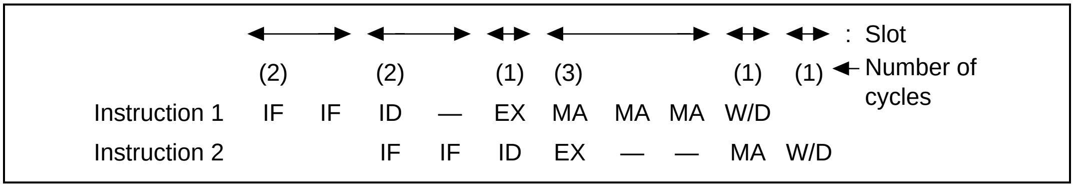

: Slot
(2) (2) (1) (3) (1) (1)
Number of cycles
Instruction 1 IF IF ID — EX MA MA MA W/D
Instruction 2 IF IF ID EX — — MA W/D

Figure 7.4 Slots Requiring Multiple Cycles

7.1.4 Number of Instruction Execution Cycles

The number of instruction execution cycles is counted as 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.

For example, in a pipeline flow like that shown in figure 7.5, the EX stage interval between instructions 1 and 2 is five cycles, so the execution time for instruction 1 is five cycles. Since the interval between EX stages for instructions 2 and 3 is one cycle, the execution time of instruction 2 is one cycle.

If a program ends with instruction 3, the execution time for instruction 3 should be calculated as the interval between the EX stage of instruction 3 and the EX stage of a hypothetical instruction 4, using a MOV Rm, Rn that follows instruction 3. (In figure 7.5, the execution time of instruction 3 would thus be one cycle.) In this example, the MA of instruction 1 and the IF of instruction 4 are in contention. For operation during the contention between the MA and IF, see section 7.2.1, Contention between Instruction Fetch (IF) and Memory Access (MA).

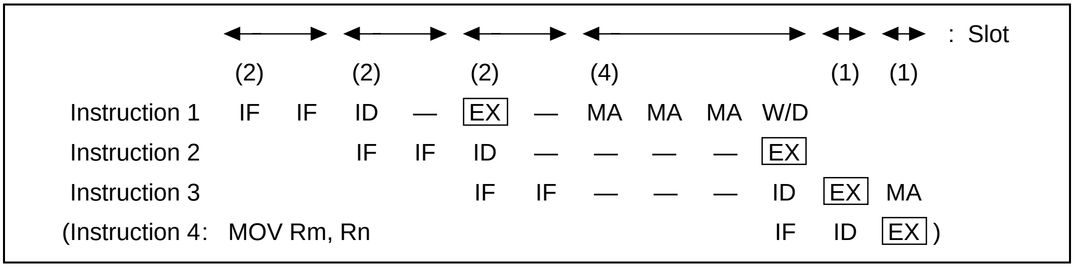

: Slot
(2) (2) (2) (4) (1) (1)
Instruction 1 IF IF ID — EX — MA MA MA W/D
Instruction 2 IF IF ID — — — — EX
Instruction 3 IF IF — — — ID EX MA
(Instruction 4: MOV Rm, Rn IF ID EX)

Figure 7.5 Method for Counting Instruction Execution Cycles

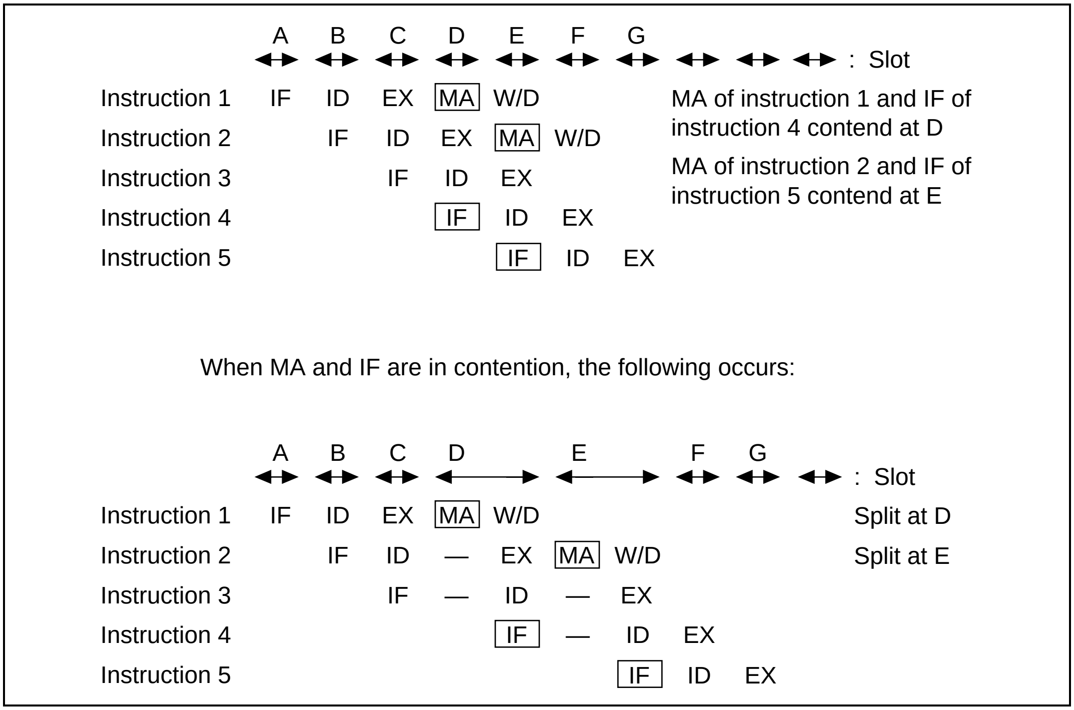
7.2 Contention

Contention occurs in four cases. When it occurs, the slot splits and requires at least two cycles.

1. Contention between instruction fetch (IF) and memory access (MA)
2. Contention when the previous instruction's destination register is used
3. Multiplier access contention
4. Contention between memory stores (MA) and either DSP operations or memory loads (WB/DSP)

7.2.1 Contention between Instruction Fetch (IF) and Memory Access (MA)

Basic Operation when IF and MA Are in Contention (Common): The IF and MA stages both access memory, so they cannot operate simultaneously. When the IF and MA stages both try to access memory within the same slot, the slot splits as shown in figure 7.6. When there is a WB, it is executed immediately after the MA ends.



	A	B	C	D	E	F	G				: Slot
Instruction 1	IF	ID	EX	$\boxed{\text{MA}}$	W/D						
Instruction 2		IF	ID	EX	$\boxed{\text{MA}}$	W/D					
Instruction 3			IF	ID	EX						
Instruction 4				$\boxed{\text{IF}}$	ID	EX					
Instruction 5					$\boxed{\text{IF}}$	ID	EX				

MA of instruction 1 and IF of instruction 4 contend at D

MA of instruction 2 and IF of instruction 5 contend at E

When MA and IF are in contention, the following occurs:

	A	B	C	D		E		F	G		: Slot
Instruction 1	IF	ID	EX	$\boxed{\text{MA}}$	W/D						Split at D
Instruction 2		IF	ID	—	EX	$\boxed{\text{MA}}$	W/D				Split at E
Instruction 3			IF	—	ID	—	EX				
Instruction 4					$\boxed{\text{IF}}$	—	ID	EX			
Instruction 5							$\boxed{\text{IF}}$	ID	EX		

Figure 7.6 Operation when IF and MA Are in Contention

The slots in which MA and IF contend are split into two cycles. MA is given priority to execute in the first half (when there is a WB, it immediately follows the MA), and the EX, ID, and IF are executed simultaneously in the latter half. For example, in figure 7.6 the MA of instruction 1 is executed in slot D while the EX of instruction 2, the ID of instruction 3 and IF of instruction 4 are executed simultaneously thereafter. In slot E, the MA of instruction 2 is given priority and the EX of instruction 3, the ID of instruction 4 and the IF of instruction 5 executed thereafter.

The number of cycles for a slot in which MA and IF are in contention is the sum of the number of memory access cycles for the MA and the number of memory access cycles for the IF.

The Relationship Between IF and the Location of Instructions in On-Chip ROM/RAM or On-Chip Memory (SH1 and SH2): When the instruction is located in the on-chip memory (ROM or RAM) or on-chip cache of the SuperH microcomputer, the SuperH microcomputer accesses the on-chip 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.

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 on-chip 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 ‘if’s always take one state.

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 7.7 illustrates these operations.

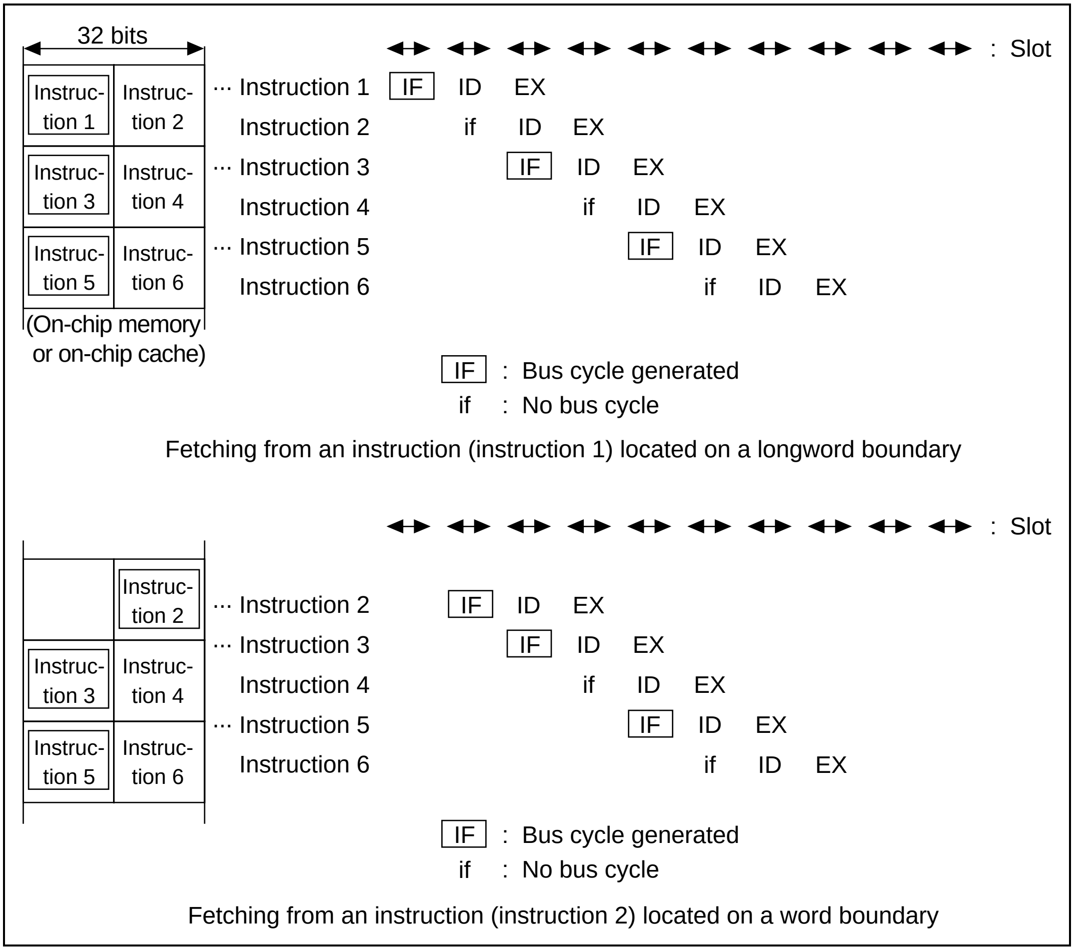

32 bits
Slot
Instruction 1
Instruction 2
Instruction 3
Instruction 4
Instruction 5
Instruction 6
IF ID EX
if ID EX
(On-chip memory or on-chip cache)
IF : Bus cycle generated
if : No bus cycle
Fetching from an instruction (instruction 1) located on a longword boundary
Fetching from an instruction (instruction 2) located on a word boundary

Figure 7.7 Relationship Between IF and Location of Instructions in On-Chip Memory

Relationship Between Position of Instructions Located in On-Chip ROM/RAM or On-Chip Memory and Contention Between IF and MA (SH-1 and SH-2): When an instruction is located in on-chip memory (ROM/RAM) or on-chip cache, there are instruction fetch stages ('if' written in lower case) that do not generate bus cycles as explained in section 7.4.2 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 states the MA requires for memory access, as illustrated in figure 7.8.

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 start from the longword boundaries in on-chip 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.

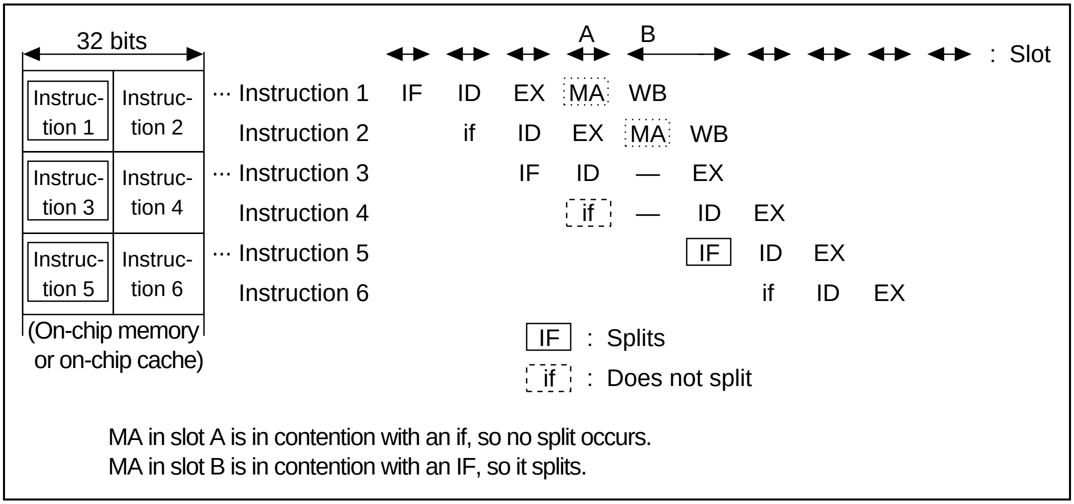


Figure 7.8 Relationship Between the Location of Instructions in On-Chip Memory and Contention Between IF and MA

Relationship between Position of Instructions Located in On-Chip Memory and Contention between IF and MA: When an instruction is located in on-chip memory, there are instruction fetch stages ("if", written in lower case) that do not generate bus cycles. 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.

When programming, avoid contention of MA and IF whenever possible and pair MAs with ifs to increase the instruction execution speed.

7.2.2 Contention when the Previous Instruction's Destination Register Is Used

Relationship between Load Instructions and the Instructions that Follow: Instructions that involve loading from memory return data to the destination register during the WB/DSP stage, which comes at the end of the pipeline. The WB/DSP stage of such a load instruction (load instruction 1) will thus not have ended before after the EX stage of the instruction that immediately follows it (instruction 2) begins.

When instruction 2 uses the same destination register as load instruction 1, the contents of that register will not be ready, so any slot containing the MA of instruction 1 and EX of instruction 2 will split. When the destination register of load instruction 1 is the same as the destination, not the source, of instruction 2 it will still split.

When the destination of load instruction 1 is the status register (SR) and the flag in it is fetched by instruction 2 (as ADDC does), a split occurs. No split occurs, however, in the following cases:

- When instruction 2 is a load instruction and its destination is the same as that of load instruction 1
- 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 shown in figure 7.9. 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.

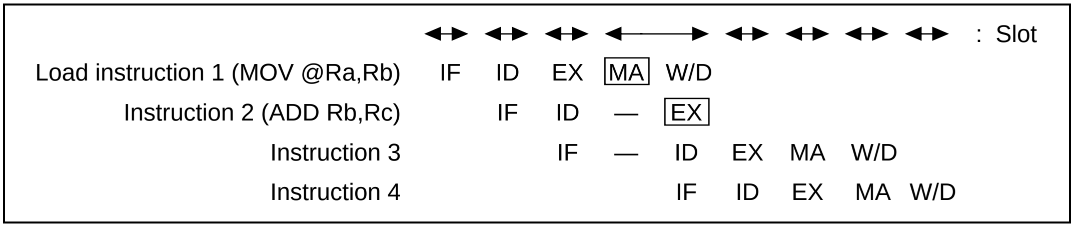

: Slot
Load instruction 1 (MOV @Ra,Rb) IF ID EX MA W/D
Instruction 2 (ADD Rb,Rc) IF ID — EX
Instruction 3 IF — ID EX MA W/D
Instruction 4 IF ID EX MA W/D

Figure 7.9 Effects of Memory Load Instructions on the Pipeline (1)

When data is loaded to a register in the previous instruction and the following memory access instruction uses that register as an address pointer, the memory access is extended until the data load of the MA stage of the previous instruction ends.

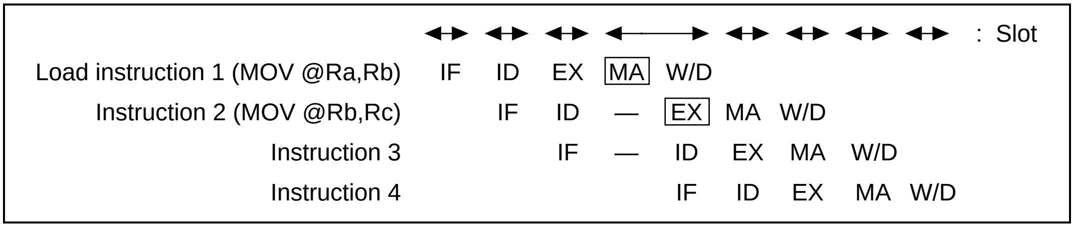

: Slot
Load instruction 1 (MOV @Ra,Rb) IF ID EX MA W/D
Instruction 2 (MOV @Rb,Rc) IF ID — EX MA W/D
Instruction 3 IF — ID EX MA W/D
Instruction 4 IF ID EX MA W/D

Figure 7.10 Effects of Memory Load Instructions on the Pipeline (2)

In the DSP unit, all operation instructions are executed in the WB/DSP stage, so transfers and operations do not contend. When the destination of the previous MOV instruction is used as the address pointer for the following instruction, however, contention can occur.

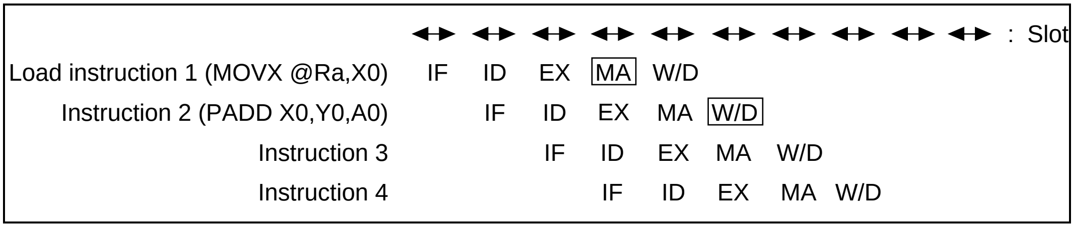

: Slot
Load instruction 1 (MOVX @Ra,X0) IF ID EX MA W/D
Instruction 2 (PADD X0,Y0,A0) IF ID EX MA W/D
Instruction 3 IF ID EX MA W/D
Instruction 4 IF ID EX MA W/D

Figure 7.11 Effects of Memory Load Instructions in the DSP Unit on the Pipeline

Relationship between Data Operation Instructions and Store Instructions: 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 7.12 shows the relationship between DSP unit data operation instructions and store instructions; figure 7.13 shows the relationship to the CPU core.

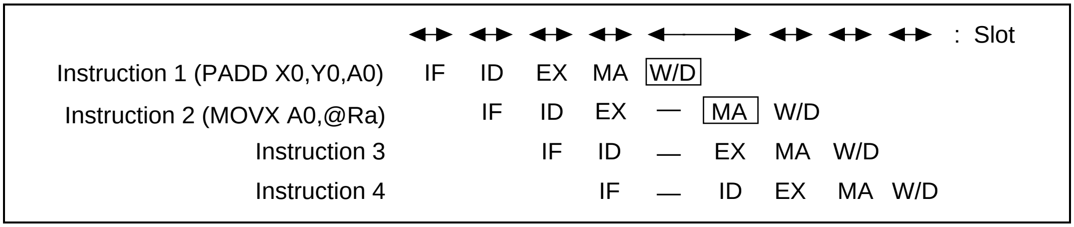

: Slot
Instruction 1 (PADD X0,Y0,A0) IF ID EX MA W/D
Instruction 2 (MOVX A0,@Ra) IF ID EX — MA W/D
Instruction 3 IF ID — EX MA W/D
Instruction 4 IF — ID EX MA W/D

Figure 7.12 Relationship between DSP Engine Operation Instructions and Store Instructions

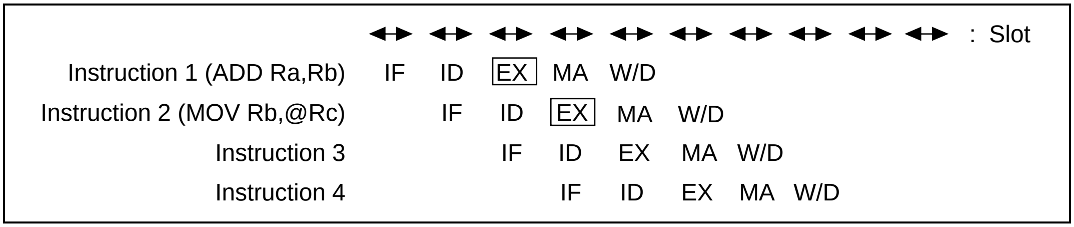

: Slot
Instruction 1 (ADD Ra,Rb) IF ID EX MA W/D
Instruction 2 (MOV Rb,@Rc) IF ID EX MA W/D
Instruction 3 IF ID EX MA W/D
Instruction 4 IF ID EX MA W/D

Figure 7.13 Relationship between CPU Core Operation Instructions and Store Instructions

Relationship between Load and Store Instructions: 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.

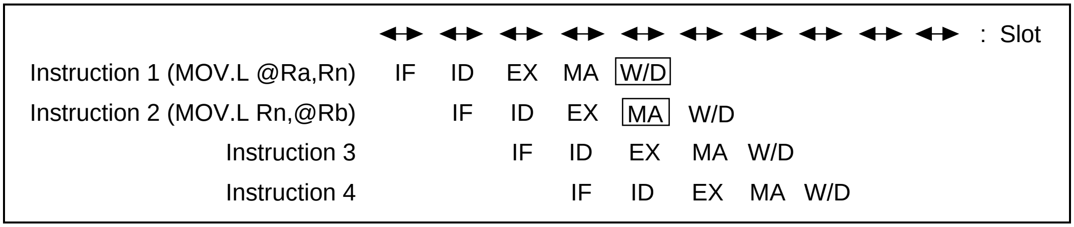

: Slot
Instruction 1 (MOV.L @Ra,Rn) IF ID EX MA W/D
Instruction 2 (MOV.L Rn,@Rb) IF ID EX MA W/D
Instruction 3 IF ID EX MA W/D
Instruction 4 IF ID EX MA W/D

Figure 7.14 Relationship between Load and Store Instructions in the CPU Core

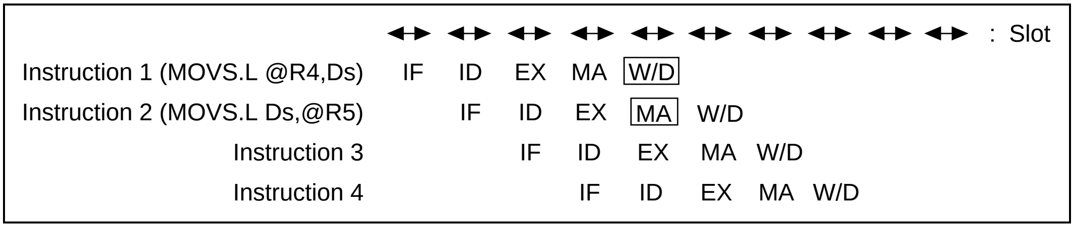

: Slot
Instruction 1 (MOVS.L @R4,Ds) IF ID EX MA W/D
Instruction 2 (MOVS.L Ds,@R5) IF ID EX MA W/D
Instruction 3 IF ID EX MA W/D
Instruction 4 IF ID EX MA W/D

Figure 7.15 Relationship between Load and Store Instructions in the DSP Unit

Relationship between MAC and STS Instructions: The MAC.W instruction has two MA stages and two mm (multiplier access) stages. When an STS instruction that stores a MACL or MACH register in the Rn register comes after a MAC.W instruction, the MA stage of the STS instruction is executed after the mm stage of the MAC.W instruction ends. Likewise, when an STS instruction that stores a MACL or MACH register in memory comes after a MAC.W instruction, the MA stage of the STS instruction is executed after the mm stage of the MAC.W instruction ends.

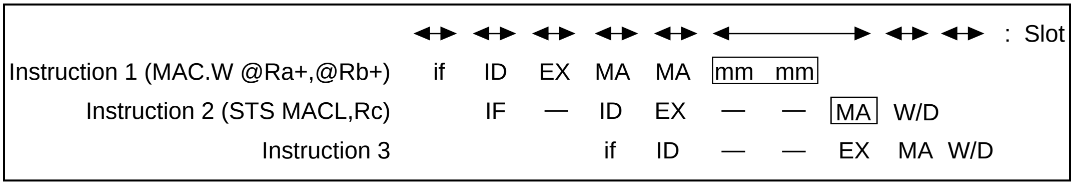

: Slot
Instruction 1 (MAC.W @Ra+,@Rb+) if ID EX MA MA mm mm
Instruction 2 (STS MACL,Rc) IF — ID EX — — MA W/D
Instruction 3 if ID — — EX MA W/D

Figure 7.16 Relationship between MAC.W and STS Instructions

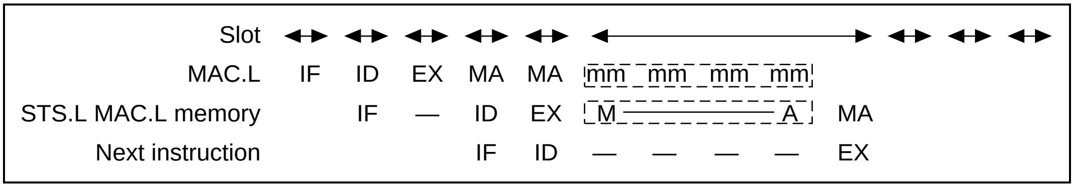

Slot
MAC.L IF ID EX MA MA mm mm mm mm
STS.L MAC.L memory IF — ID EX M A MA
Next instruction IF ID — — — — EX

Figure 7.17 Example of Multiplier Access Contention—MAC.L and STS.L Instructions

7.2.3 Multiplier Access Contention

Instructions that access multiplier type instructions (Multiply/Accumulate instructions and multiplication instructions) or the multiply and accumulate calculation registers (MACH and MACL) contend with multiplier accesses.

In multiplier type instructions, the multiplier operates for either four cycles (for double-length 64 bits instructions) or two cycles (single-length 32 bit instructions) after the MA ends, regardless of the slot. When the MA (or the second MA, if there are two) of a multiplier type instruction (Multiply/Accumulate instructions and multiplication instructions) contends with the multiplier access (mm) of the previous multiplier type instruction, the bus cycle of the MA is extended until the mm ends. The extended MA becomes a single slot.

The ID of the instruction following a double-length instruction also stalls until one slot later.

Multiplier type instructions and instructions that access the multiply and accumulate calculation registers have MA stages, so they also contend with IFs. Figure 7.18 shows an example of multiplier access contention, but it does not address MA and IF contention.

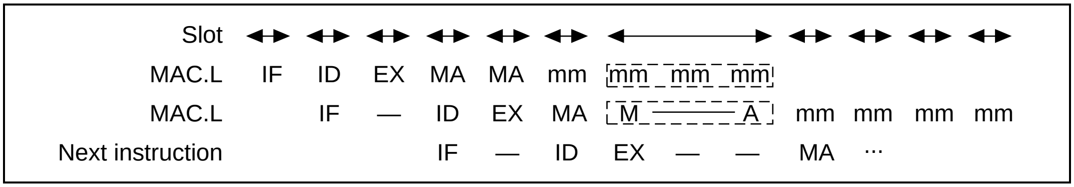

Slot
MAC.L IF ID EX MA MA mm mm mm mm
MAC.L IF — ID EX MA M A mm mm mm mm
Next instruction IF — ID EX — — MA ···

Figure 7.18 Example of Multiplier Access Contention—MAC.L and MAC.L Instructions

7.2.4 Contention between Memory Stores and DSP Operations

When an instruction that will store the result of a DSP operation instruction is written immediately after the DSP operation instruction is executed, the execution will be too late. To prevent this, a stall cycle is inserted. For more information, see section 4.17.2, Single Data Transfers.

7.3 Programming Guide

7.3.1 Types of Contention and Affected Instructions

Types of contention and the instructions they affect are summarized below.

- Instructions without contention
- Instructions with memory accesses (MA) that contend with instruction fetches (IF)
- Instructions that store the result of the immediately preceding DSP operation in memory using the X bus or Y bus
- Instructions with memory accesses (MA) that contend with instruction fetches (IF), also have write backs (WB/DSP), and may cause contention with memory loads
- Instructions with memory accesses (MA) that contend with instruction fetches (IF), also access the multiplier (mm), and may cause contention with the multiplier
- Instructions that store DSP operation results in memory, because the memory access (MA) contends with an instruction fetch (IF)
- Instructions with memory accesses (MA) that contend with instruction fetches (IF), access the multiplier (mm), and may cause contention with the multiplier, and also have write backs (WB/DSP) and may cause contention with memory loads
- Instructions that cause contention with MOV.X, MOV.Y, or MOVS.L instructions

Table 7.1 shows the correspondence between types of contention and instructions.

Table 7.1 Types of Contention and Instructions

Contention	Cycles	Stages	Instructions
None	1	3	Inter-register transfer instructions Inter-register operations (except multiplier type instructions) Inter-register logic operation instructions Shift instructions System control ALU instructions
	2	3	Unconditional branch instructions
	3/1	3	Conditional branch instructions
	2/1	3	Delayed conditional branch instruction
	3	3	SLEEP instruction
	4	5	RTE instruction
	8	9	TRAP instruction
	1	5	DSP operation instructions MOVX.W (load) and MOVY.W (load) instructions
MA contends with IF	1	4	Memory store instructions STS.L instruction (PR)
	2	4	STC.L instruction
	3	6	Memory logic operations
	4	6	TAS instruction
	1	5	MOVS.W (load) and MOVS.L (load) instructions
Causes DSP operation contention	1	4	MOVX.W (store) and MOVY.W (store) instructions
MA contends with IF Causes memory load contention	1	5	Memory load instructions LDS.L instruction (PR)
	3	5	LDC.L instruction
MA contends with IF Causes multiplier contention	1	4	Register to MAC transfer instructions (MACH/MACL) Memory to MAC transfer instructions (MACH/MACL) MAC to memory transfer instructions (MACH/MACL)
	1 (to 3)*	6	Multiplication instructions

Contention	Cycles	Stages	Instructions
MA contends with IF Causes multiplier contention (cont)	2 (to 3)*	7	Multiply and accumulate calculation instructions
	2 (to 4)*	9	Double-length multiplication instructions
	2 (to 4)*	9	Double-length multiply and accumulate calculation 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 DSP operation contention Causes memory load contention	1	5	STS instruction (except PR)
Causes MOVX.W, MOVY.W, MOVS.W or MOVS.L instruction contention	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.

7.3.2 Increasing Instruction Execution Speed

Instruction execution speed can be increased by trying, at the programming stage, to keep contention from occurring. Follow these rules when writing programs to minimize contention:

1. A 32-bit DSP instruction can require up to three memory accesses per cycle: one instruction (I-bus), one X-data (X-bus), and one Y-data (Y-bus). The SH-DSP has four independently accessible on-chip memory areas: X-ROM, X-RAM, Y-ROM, and Y-RAM. If more than one access is performed in the same memory area in a cycle, a stall occurs. Locate the program (instructions) and the data arrays that the program accesses in different on-chip memory areas. This prevents memory bank contention in DSP instructions.
2. Follow instructions that compute a value in the DSP unit and write it to a DSP register with instructions that do not store the same register to memory. This prevents DSP register contention because storing a DSP register that was the destination of a DSP calculation in the previous cycle will cause a stall.
3. Instruction fetch (IF) can conflict with an SH data memory access (MA) because both use the same bus. Whether the instruction fetch occurs in a specific cycle depends on the locations and size (16 bit or 32 bit) of the preceding instructions. Try to locate the SH instructions that perform memory access at longword boundries in on-chip memory and use a 16-bit instruction as the next instruction. This prevents contention between memory accesses and instruction fetches.

4. Follow instructions that load an SH register (R0 to R15) from memory with instructions that do not use the same register as the load instruction's destination register. This prevents memory load contention caused by write backs (WB/DSP).

   Note: The DSP registers (A0 to Y1) loaded in the previous cycle can be used in this cycle without causing any stalls.
5. Do not place two instructions that use the multiplier consecutively (the PMULS instruction is excepted from this rule). Also try to keep accesses of MACH and MACL registers for getting the results from the multiplier away from instructions that use the multiplier. This prevents multiplier contention caused by multiplier accesses (mm).
6. Avoid data transfers to memory or CPU core registers immediately after DSP unit data operations from those registers storing the operation results. Avoid contention by placing another instruction before the transfer.

7.3.3 Cycles

Basic instructions are designed to execute in one cycle. One-cycle instructions include both instructions that cause contention and instructions that do not. Operations and transfers that occur between registers do not create contention.

There are instructions that require two or more cycles even when there is no contention. Instructions that change the branch destination addresses, such as branch instructions or the like, memory logic operation instructions, instructions that execute memory accesses twice or more, such as some system control instructions, and instructions that have memory accesses and multiplier accesses such as multiplication instructions and multiply and accumulate instructions, (excluding PMULS) all take two or more cycles.

Instructions that require two or more cycles also include both instructions that cause contention and instructions that do not.

To write efficient programs, it is essential to avoid contention, keep instruction execution speed up, and use instructions with fewer stages.

7.4 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 7.2.1, Contention between Instruction Fetch (IF) and Memory Access (MA). Base your response on the rules for pipeline operation given there.

Table 7.2 shows the number of instruction stages and number of execution cycles as follows:

- Type: Given by function
- Category: Categorized by differences in instruction operation
- Stages: The number of stages in the instruction
- Cycles: The number of execution cycles when there is no contention
- Contention: Indicates the contention that occurs
- Instructions: Gives a mnemonic for the instruction concerned

Table 7.2 Number of Instruction Stages and Execution Cycles

Type	Category	Instruction	Stages	Cycles	Contention
Data transfer instructions	Register-register transfer instructions	`MOV #imm,Rn`	3	1	—
		`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`	5	1	• Contention occurs if the instruction placed immediately after this CPU instruction 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`			

Type	Category	Instruction	Stages	Cycles	Contention
Data transfer instructions (cont)	Memory store instructions	`MOV.B @Rm,Rn`	4	1	MA contends with IF
		`MOV.W @Rm,Rn`			
		`MOV.L @Rm,Rn`			
		`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)`			

Type	Category	Instruction		Stages	Cycles	Contention
Arithmetic instructions	Arithmetic instructions between registers (except multiplication instructions)	`ADD`	`Rm,Rn`	3	1	—
		`ADD`	`#imm,Rn`			
		`ADDC`	`Rm,Rn`			
		`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/PZ`	`Rn`			
		`CMP/PL`	`Rn`			
		`CMP/STR`	`Rm,Rn`			
		`DIV1`	`Rm,Rn`			
		`DIV0S`	`Rm,Rn`			
		`DIV0U`				
		`DT`	`Rn`			
		`EXTS.B`	`Rm,Rn`			
		`EXTS.W`	`Rm,Rn`			
		`EXTU.B`	`Rm,Rn`			
		`EXTU.W`	`Rm,Rn`			
		`NEG`	`Rm,Rn`			
		`NEGC`	`Rm,Rn`			
		`SUB`	`Rm,Rn`			
		`SUBC`	`Rm,Rn`			
		`SUBV`	`Rm,Rn`			
	Multiply/add instructions	`MAC.W`	`@Rm+,@Rn+`	7/8*3	2 (to 3)*1	• Multiplier contention occurs when an instruction that uses the multiplier follows a MAC instruction • MA contends with IF

Type	Category	Instruction	Stages	Cycles	Contention
Arithmetic instructions (cont)	Double-length multiply/ accumulate instruction	`MAC.L @Rm+,@Rn+`	9	2 (to 4)*1	• Multiplier contention occurs when an instruction that uses the multiplier follows a MAC instruction • MA contends with IF

	Multiplication instructions	`MULS.W Rm,Rn`
		`MULU.W Rm,Rn`
			6/7*3	1 (to 3)*1	• Multiplier contention occurs when an instruc-tion that uses the multiplier follows a MUL instruction
					• MA contends with IF

	Double-length multiply/ accumulate instruction	`DMULS.L Rm,Rn`
		`DMULU.L Rm,Rn`
		`MUL.L Rm,Rn`
			9	2 (to 4)*1	• Multiplier contention occurs when an instruction that uses the multiplier follows a MAC instruction
					• MA contends with IF

Logic operation instructions	Register-register logic operation instructions	`AND Rm,Rn`
		`AND #imm,R0`
		`NOT Rm,Rn`
		`OR Rm,Rn`
		`OR #imm,R0`
		`TST Rm,Rn`
		`TST #imm,R0`
		`XOR Rm,Rn`
		`XOR #imm,R0`
			3	1	—

Type	Category	Instruction		Stages	Cycles	Contention
Logic operation instructions (cont)	Memory logic operations instructions	`AND.B`	`#imm,@(R0,GBR)`	6	3	MA contends with IF
		`OR.B`	`#imm,@(R0,GBR)`			
		`TST.B`	`#imm,@(R0,GBR)`			
		`XOR.B`	`#imm,@(R0,GBR)`			
	TAS instruction	`TAS.B`	`@Rn`	6	4	MA contends with IF
Shift instructions	Shift instructions	`ROTL`	`Rn`	3	1	—
		`ROTR`	`Rn`			
		`ROTCL`	`Rn`			
		`ROTCR`	`Rn`			
		`SHAL`	`Rn`			
		`SHAR`	`Rn`			
		`SHLL`	`Rn`			
		`SHLR`	`Rn`			
		`SHLL2`	`Rn`			
		`SHLR2`	`Rn`			
		`SHLL8`	`Rn`			
		`SHLR8`	`Rn`			
		`SHLL16`	`Rn`			
		`SHLR16`	`Rn`			
Branch instructions	Conditional branch instructions	`BF`	`label`	3	3/1*2	—
		`BT`	`label`			
	Delayed conditional branch instructions	`BF/S`	`label`	3	2/1*2	—
		`BT/S`	`label`			
	Unconditional branch instructions	`BRA`	`label`	3	2	—
		`BRAF`	`Rm`			
		`BSR`	`label`			
		`BSRF`	`Rm`			
		`JMP`	`@Rm`			
		`JSR`	`@Rm`			
		`RTS`				

Type	Category	Instruction	Stages	Cycles	Contention
System control instructions	System control ALU instructions	`CLRT`	3	1	—
		`LDC Rm,SR`			
		`LDC Rm,GBR`			
		`LDC Rm,VBR`			
		`LDC Rm,MOD`			
		`LDC Rm,RE`			
		`LDC Rm,RS`			
		`LDRE @(disp,PC)`			
		`LDRS @(disp,PC)`			
		`LDS Rm,PR`			
		`NOP`			
		`SETRC Rm`			
		`SETRC #imm`			
		`SETT`			
		`STC SR,Rn`			
		`STC GBR,Rn`			
		`STC VBR,Rn`			
		`STC MOD,Rn`			
		`STC RE,Rn`			
		`STC RS,Rn`			
		`STS PR,Rn`			

Type	Category	Instruction	Stages	Cycles	Contention
System control instructions (cont)	LDS.L instructions (PR)	`LDS.L @Rm+,PR`	5	1	• 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`	4	1	MA contends with IF

	LDC.L instructions	`LDC.L @Rm+,SR`
		`LDC.L @Rm+,GBR`
		`LDC.L @Rm+,VBR`
		`LDC.L @Rm+,MOD`
		`LDC.L @Rm+,RE`
		`LDC.L @Rm+,RS`
			5	3	• Contention occurs when an instruction that uses the same destination register is placed immediately after this instruction
					• MA contends with IF

	STC.L instructions	`STC.L SR,@–Rn`
		`STC.L GBR,@–Rn`
		`STC.L VBR,@–Rn`
		`STC.L MOD,@–Rn`
		`STC.L RE,@–Rn`
		`STC.L RS,@–Rn`
			4	2	MA contends with IF

Type	Category	Instruction		Stages	Cycles	Contention
System control instructions (cont)	Register → MAC transfer instruction	`CLRMAC`		4	1	• Contention occurs with multiplier • MA contends with IF
		`LDS`	`Rm,MACH`			
		`LDS`	`Rm,MACL`			
	Register → DSP transfer instruction	`LDS`	`Rm,DSR`	4	1	—
		`LDS`	`Rm,A0`			
		`LDS`	`Rm,X0`			
		`LDS`	`Rm,X1`			
		`LDS`	`Rm,Y0`			
		`LDS`	`Rm,Y1`			
	Memory → MAC transfer instructions	`LDS.L`	`@Rm+,MACH`	4	1	• Contention occurs with multiplier • MA contends with IF
		`LDS.L`	`@Rm+,MACL`			
	Memory → DSP transfer instructions	`LDS.L`	`@Rm+,DSR`	4	1	—
		`LDS.L`	`@Rm+,A0`			
		`LDS.L`	`@Rm+,X0`			
		`LDS.L`	`@Rm+,X1`			
		`LDS.L`	`@Rm+,Y0`			
		`LDS.L`	`@Rm+,Y1`			

Type	Category	Instruction	Stages	Cycles	Contention

System control instructions (cont)	MAC → register transfer instruction	`STS MACH,Rn`
		`STS MACL,Rn`
			5	1	• Contention occurs with multiplier

	DSP → register transfer instruction	`STS DSR,Rn`
		`STS A0,Rn`
		`STS X0,Rn`
		`STS X1,Rn`
		`STS Y0,Rn`
		`STS Y1,Rn`
					• Contention occurs when an instruction that uses the same destination register is placed immediately after this instruction
					• MA contends with IF

	MAC → memory transfer instruction	`STS.L MACH,@–Rn`
		`STS.L MACL,@–Rn`
			4	1	• Contention occurs with multiplier
					• MA contends with IF

	DSP → memory transfer instruction	`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	—

	RTE instruction	`RTE`	5	4	—
	TRAP instruction	`TRAPA #imm`	9	8	—
	SLEEP instruction	`SLEEP`	3	3	—

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 there is no branch.

3. Number of stages of the SH-1 CPU.

7.4.1 Data Transfer Instructions

Register-Register Transfer Instructions (Common): Includes the following 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

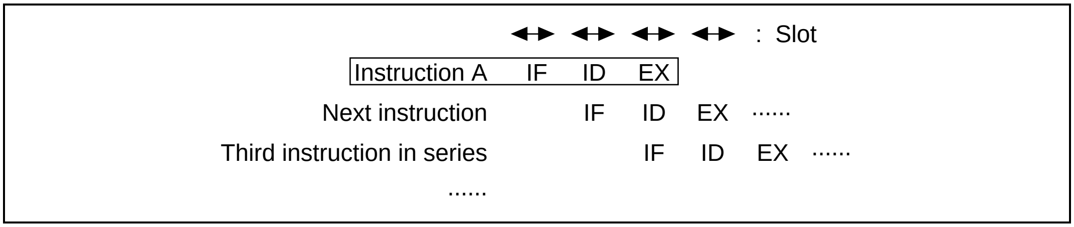

: Slot
Instruction A IF ID EX
Next instruction IF ID EX ......
Third instruction in series IF ID EX ......
......

Figure 7.19 Register-Register Transfer Instruction Pipeline

Operation: The pipeline ends after three stages: IF, ID, and EX. Data is transferred in the EX stage via the ALU.

Memory Load Instructions (Common): Include the following 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

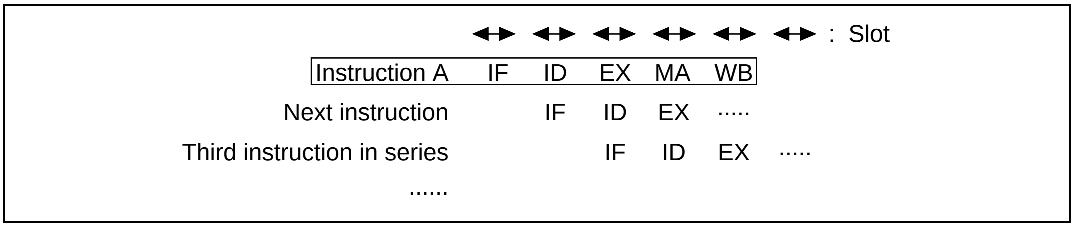
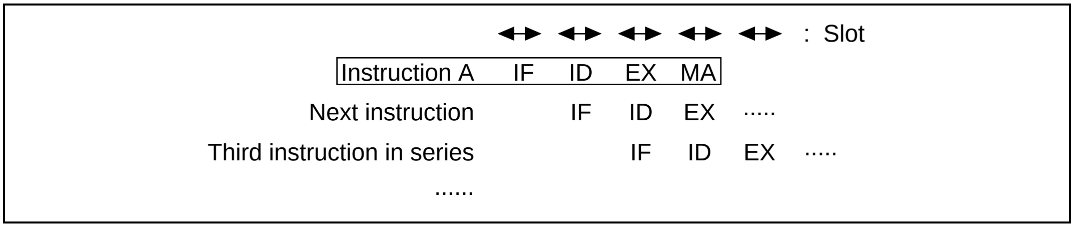

: Slot
Instruction A IF ID EX MA WB
Next instruction IF ID EX .....
Third instruction in series IF ID EX .....
......

Figure 7.20 Memory Load Instruction Pipeline

The pipeline has five stages: IF, ID, EX, MA, and WB (figure 7.20). If an instruction that uses the same destination register as this instruction is placed immediately after it, contention will occur. (See section 7.2.2, Contention when the Previous Instruction's Destination Register Is Used.)

Memory Store Instructions (Common): Include the following instruction types:

- 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 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)



: Slot
Instruction A IF ID EX MA
Next instruction IF ID EX .....
Third instruction in series IF ID EX .....
......

Figure 7.21 Memory Store Instructions Pipeline

The pipeline has four stages: IF, ID, EX, and MA (figure 7.21). Data is not returned to the register so there is no WB stage.

7.4.2 Arithmetic Instructions

Arithmetic Instructions between Registers (Except Multiplication Instructions) (Common, or SH-2 CPU, SH-DSP): Include the following instruction types:

• ADD	Rm, Rn
• ADD	#imm, Rn
• ADDC	Rm, Rn
• 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/PZ	Rn
• CMP/PL	Rn
• CMP/STR	Rm, Rn
• DIV1	Rm, Rn
• DIV0S	Rm, Rn
• DIV0U	
• DT	Rn (SH-2 CPU, SH-DSP)
• EXTS.B	Rm, Rn
• EXTS.W	Rm, Rn
• EXTU.B	Rm, Rn
• EXTU.W	Rm, Rn
• NEG	Rm, Rn
• NEGC	Rm, Rn
• SUB	Rm, Rn
• SUBC	Rm, Rn
• SUBV	Rm, Rn

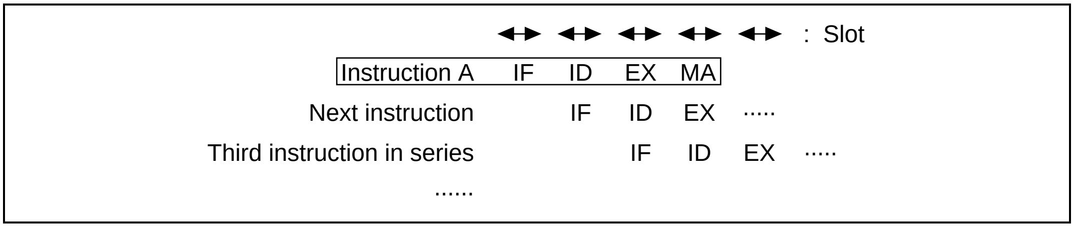

: Slot
Instruction A IF ID EX MA
Next instruction IF ID EX .....
Third instruction in series IF ID EX .....
......

Figure 7.22 Pipeline for Arithmetic Instructions between Registers Except Multiplication Instructions

The pipeline has three stages: IF, ID, and EX (figure 7.22). The data operation is completed in the EX stage via the ALU.

Multiply/Accumulate Instruction (SH-1 CPU): Includes the following instruction type:

- MAC.W @Rm+, @Rn+

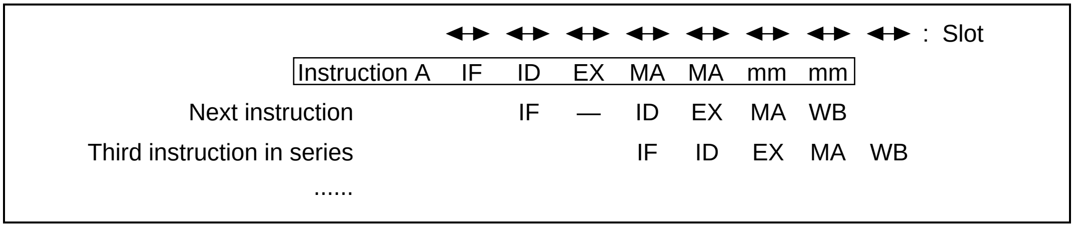

: Slot
Instruction A IF ID EX MA MA mm mm
Next instruction IF — ID EX MA WB
Third instruction in series IF ID EX MA WB
......

Figure 7.23 Multiply/Accumulate Instruction Pipeline

The pipeline has seven stages: IF, ID, EX, MA, MA, mm, and mm. The second MA reads the memory and accesses the multiplier. mm indicates that the multiplier is operating. mm operates for two cycles after the final MA ends, regardless of slot. The ID of the instruction after the MAC.W instruction is stalled for 1 slot. The two MAs of the MAC.W instruction, when they contend with IF, split the slots as described in Section 7.2.1, Contention between Instruction Fetch (IF) and Memory Access (MA).

When an instruction that does not use the multiplier comes after 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 operates like a normal pipeline. When an instruction that uses the multiplier comes after the MAC.W instruction, however, contention occurs with the multiplier, so operation is different from normal.

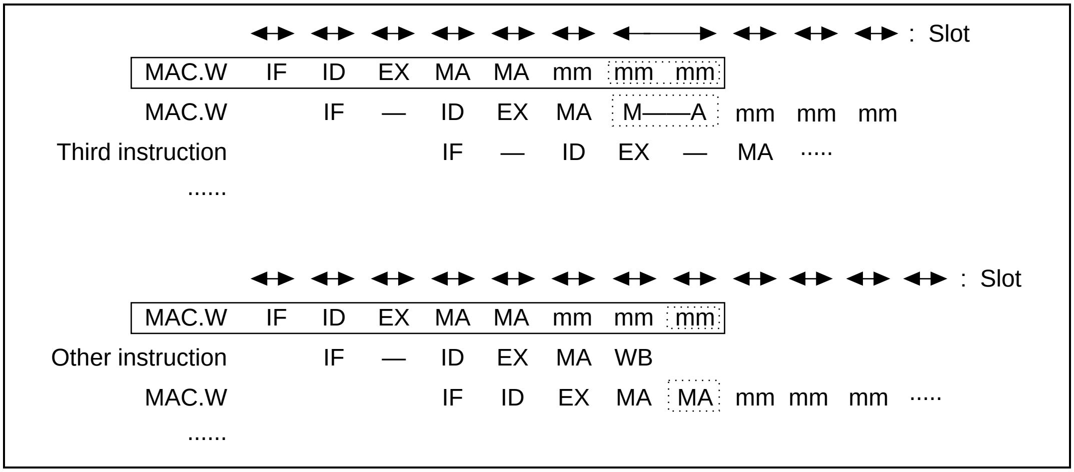
This occurs in the following cases:

1. When a MAC.W instruction is located immediately after another MAC.W instruction
2. When a MULS.W instruction is located immediately after a MAC.W instruction
3. When an STS (register) instruction is located immediately after a MAC.W instruction
4. When an STS.L (memory) instruction is located immediately after a MAC.W instruction
5. When an LDS (register) instruction is located immediately after a MAC.W instruction
6. When an LDS.L (memory) instruction is located immediately after a MAC.W instruction

1. When a MAC.W instruction is located immediately after another MAC.W instruction

   When the second MA of a MAC.W instruction contends with an mm generated by a preceding multiplier-type instruction, the bus cycle of that MA is extended until the mm ends (the M—A shown in the dotted line box below) and that extended MA occupies one slot.

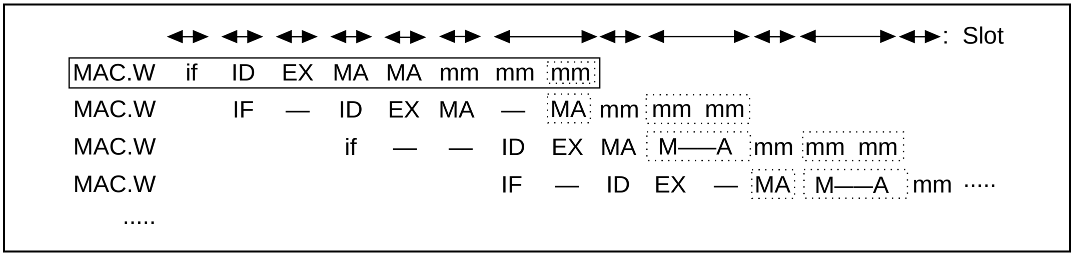
   If one or more instruction not related to the multiplier is located between the MAC.W instructions, multiplier contention between MAC instructions does not cause stalls (figure 7.24).



: Slot
MAC.W IF ID EX MA MA mm mm mm
MAC.W IF — ID EX MA M——A mm mm mm
Third instruction IF — ID EX — MA .....
......
: Slot
MAC.W IF ID EX MA MA mm mm mm
Other instruction IF — ID EX MA WB
MAC.W IF ID EX MA MA mm mm mm .....
......

Figure 7.24 Unrelated Instructions between MAC.W Instructions

Sometimes consecutive MAC.Ws may not have multiplier contention even when MA and IF contention causes misalignment of instruction execution. Figure 7.25 illustrates a case of this type. This figure assumes MA and IF contention.



: Slot
MAC.W if ID EX MA MA mm mm mm
MAC.W IF — ID EX MA — MA mm mm mm
MAC.W if — — ID EX MA M——A mm mm mm
MAC.W IF — ID EX — MA M——A mm .....
.....

Figure 7.25 Consecutive MAC.Ws without Misalignment

When the second MA of the MAC.W instruction is extended until the mm ends, contention between MA and IF will split the slot, as usual. Figure 7.26 illustrates a case of this type. This figure assumes MA and IF contention.

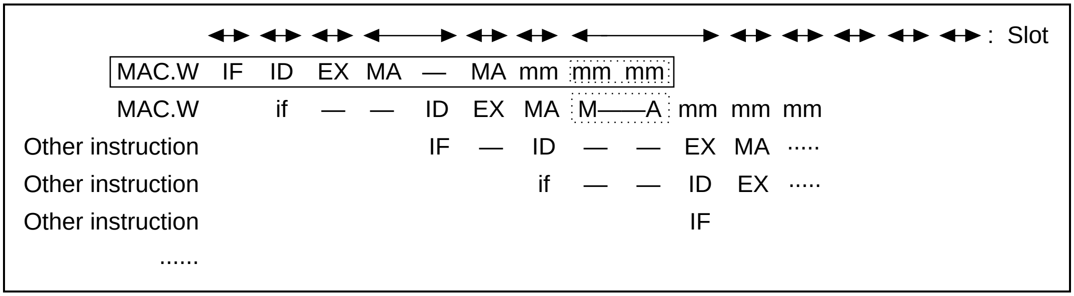
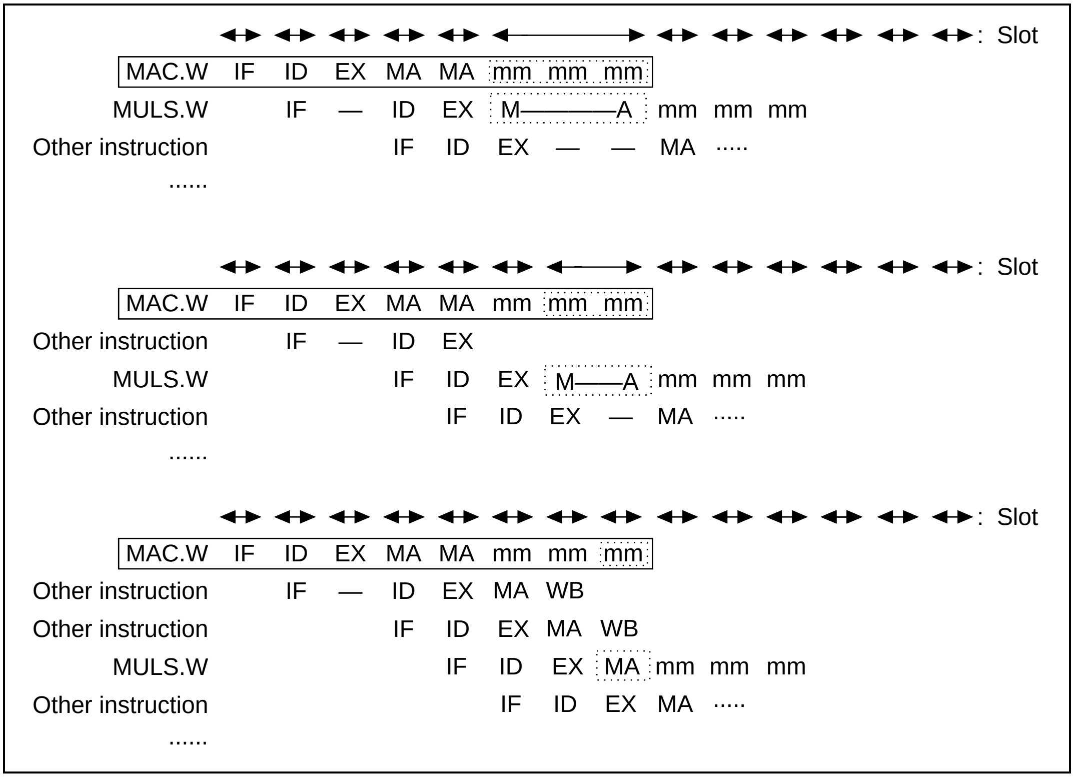

: Slot
MAC.W IF ID EX MA — MA mm mm mm
MAC.W if — — ID EX MA M——A mm mm mm
Other instruction IF — ID — — EX MA .....
Other instruction if — — ID EX .....
Other instruction IF
......

Figure 7.26 MA and IF Contention

2. When a MULS.W instructions is located immediately after a MAC.W instruction

   A MULS.W instruction has an MA stage for accessing the multiplier. When the MA of the MULS.W instruction contends with an operating MAC instruction multiplier (mm), the MA is extended until the mm ends (the M—A shown in the dotted line box in figure 7.27) to create a single slot. When two or more instructions not related to the multiplier come between the MAC.W and MULS.W instructions, MAC.W and MULS.W contention does not cause stalling. When the MULS.W MA and IF contend, the slot is split.



: Slot
MAC.W IF ID EX MA MA mm mm mm
MULS.W IF — ID EX M——A mm mm mm
Other instruction IF ID EX — — MA ·····
......
: Slot
MAC.W IF ID EX MA MA mm mm mm
Other instruction IF — ID EX
MULS.W IF ID EX M——A mm mm mm
Other instruction IF ID EX — MA ·····
......
: Slot
MAC.W IF ID EX MA MA mm mm mm
Other instruction IF — ID EX MA WB
Other instruction IF ID EX MA WB
MULS.W IF ID EX MA mm mm mm
Other instruction IF ID EX MA ·····
......

Figure 7.27 MULS.W Instruction Immediately After a MAC.W Instruction

3. When an STS (register) instruction is located immediately after a MAC.W instruction

When the contents of a MAC register are stored in a general-purpose register using an STS instruction, an MA stage for accessing the multiplier is added to the STS instruction, as described later. When the MA of the STS instruction contends with the operating multiplier (mm), the MA is extended until the mm ends (the M—A shown in the dotted line box in figure 7.28) to create a single slot. The MA of the STS contends with the IF. Figure 7.28 illustrates how this occurs, assuming MA and IF contention.

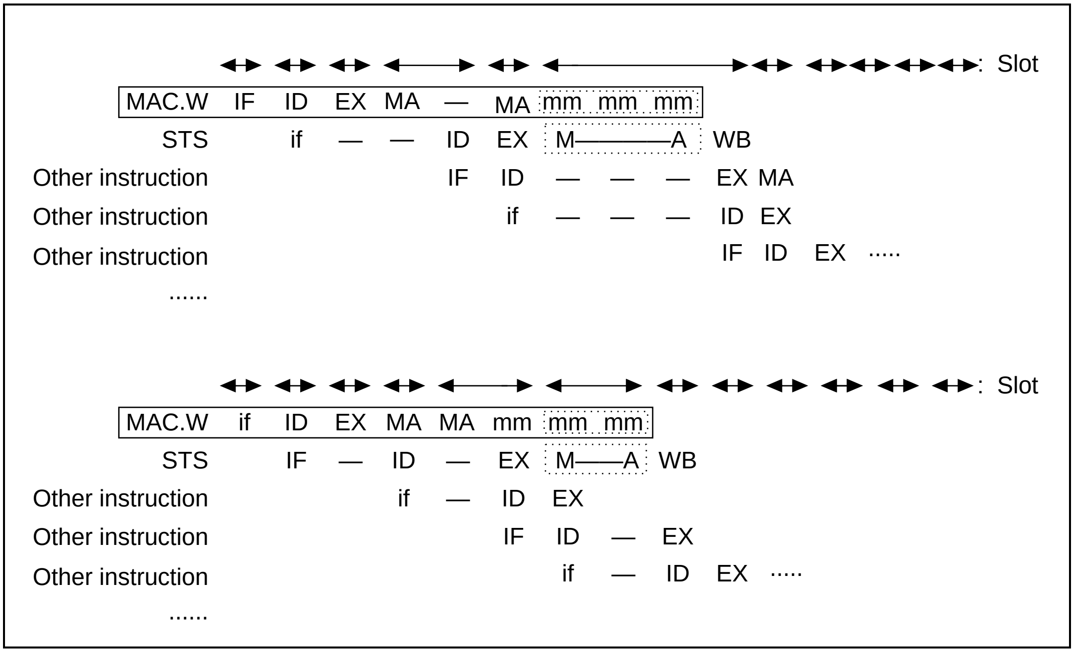

Slot
MAC.W IF ID EX MA — MA mm mm mm
STS if — — ID EX M———A WB
Other instruction IF ID — — — EX MA
Other instruction if — — — ID EX
Other instruction IF ID EX .....
......
Slot
MAC.W if ID EX MA MA mm mm mm
STS IF — ID — EX M——A WB
Other instruction if — ID EX
Other instruction IF ID — EX
Other instruction if — ID EX .....
......

Figure 7.28 STS (Register) Instruction Immediately After a MAC.W Instruction

4. When an STS.L (memory) instruction is located immediately after a MAC.W instruction

When the contents of a MAC register are stored in memory using an STS instruction, an MA stage for accessing the multiplier and writing to memory is added to the STS instruction, as described later. When the MA of the STS instruction contends with the operating multiplier (mm), the MA is extended until one state after the mm ends (the M—A shown in the dotted line box in figure 7.29) to create a single slot. The MA of the STS contends with the IF. Figure 7.29 illustrates how this occurs, assuming MA and IF contention.

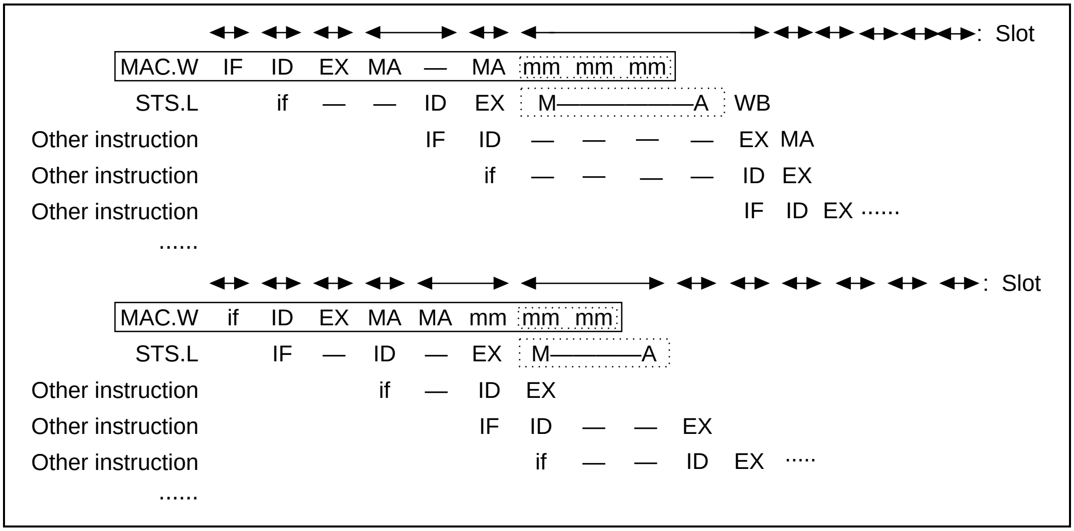
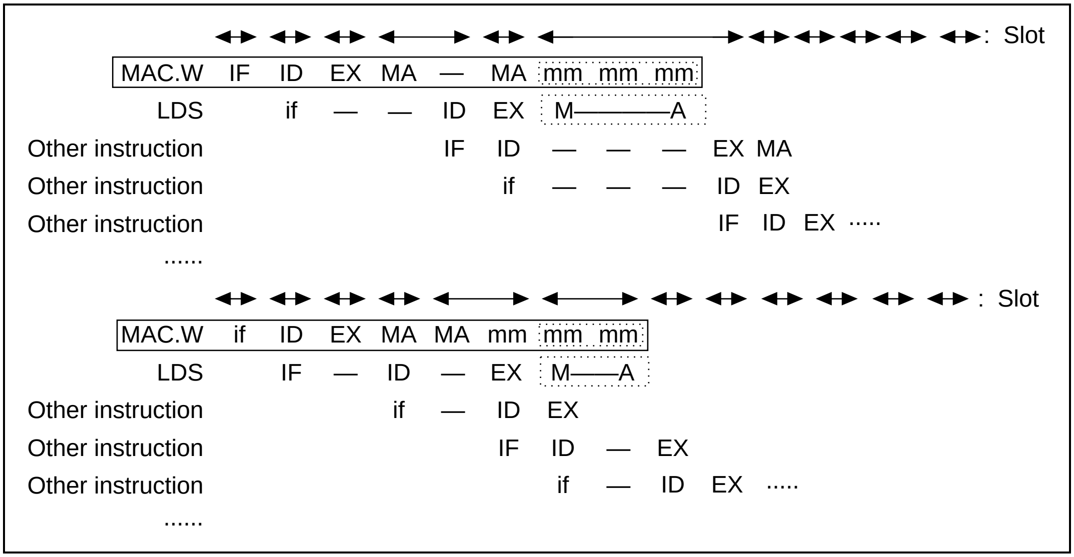

Slot
MAC.W IF ID EX MA — MA mm mm mm
STS.L if — — ID EX M———A WB
Other instruction IF ID — — — — EX MA
Other instruction if — — — — ID EX
Other instruction IF ID EX ......
......
Slot
MAC.W if ID EX MA MA mm mm mm
STS.L IF — ID — EX M———A
Other instruction if — ID EX
Other instruction IF ID — — EX
Other instruction if — — ID EX .....
......

Figure 7.29 STS.L (Memory) Instruction Immediately After a MAC.W Instruction

5. When an LDS (register) instruction is located immediately after a MAC.W instruction

When the contents of a MAC register are loaded from a general-purpose register using an LDS instruction, an MA stage for accessing the multiplier is added to the LDS instruction, as described later. When the MA of the LDS instruction contends with the operating multiplier (mm), the MA is extended until the mm ends (the M—A shown in the dotted line box in figure 7.30) to create a single slot. The MA of this LDS contends with IF. Figure 7.30 illustrates how this occurs, assuming MA and IF contention.

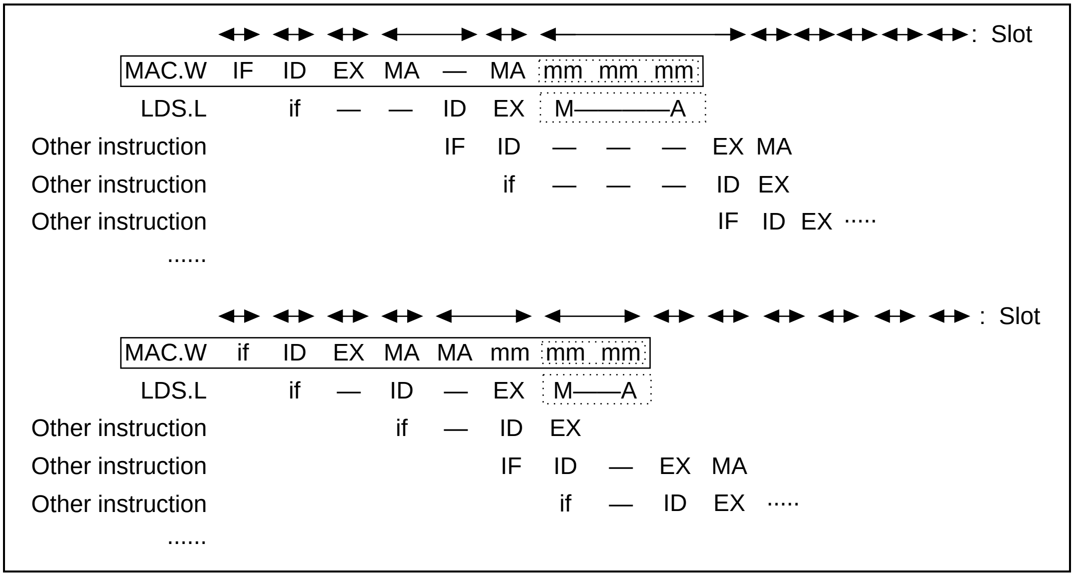


: Slot
MAC.W IF ID EX MA — MA mm mm mm
LDS if — — ID EX M————A
Other instruction IF ID — — — EX MA
Other instruction if — — — ID EX
Other instruction IF ID EX .....
......
: Slot
MAC.W if ID EX MA MA mm mm mm
LDS IF — ID — EX M——A
Other instruction if — ID EX
Other instruction IF ID — EX
Other instruction if — ID EX .....
......

Figure 7.30 LDS (Register) Instruction Immediately After a MAC.W Instruction

6. When an LDS.L (memory) instruction is located immediately after a MAC.W instruction

   When the contents of a MAC register are loaded from memory using an LDS instruction, an MA stage for accessing the memory and the multiplier is added to the LDS instruction, as described later. When the MA of the LDS instruction contends with the operating multiplier (mm), the MA is extended until the mm ends (the M—A shown in the dotted line box in figure 7.31) to create a single slot. The MA of the LDS contends with IF. Figure 7.31 illustrates how this occurs, assuming MA and IF contention.



: Slot
MAC.W IF ID EX MA — MA mm mm mm
LDS.L if — — ID EX M———A
Other instruction IF ID — — — EX MA
Other instruction if — — — ID EX
Other instruction IF ID EX .....
......
: Slot
MAC.W if ID EX MA MA mm mm mm
LDS.L if — ID — EX M——A
Other instruction if — ID EX
Other instruction IF ID — EX MA
Other instruction if — ID EX .....
......

Figure 7.31 LDS.L (Memory) Instruction Immediately After a MAC.W Instruction

Double-Length Multiply/Accumulate Instruction (SH-2 CPU, SH-DSP): Includes the following instruction type:

- MAC.L @Rm+, @Rn+

	◀▶	◀▶	◀▶	◀▶	◀▶	◀▶	◀▶	◀▶	◀▶ : Slot
Instruction A	IF	ID	EX	MA	MA	mm	mm	mm	mm
Next instruction		IF	—	ID	EX	MA	WB		
Third instruction				IF	ID	EX	MA	WB	
......									

Figure 7.32 Multiply/Accumulate Instruction Pipeline

The pipeline has nine stages: IF, ID, EX, MA, MA, mm, mm, mm, and mm (figure 7.32). The second MA reads the memory and accesses the multiplier. The mm indicates that the multiplier is operating. The mm operates for four 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 7.2.1, 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 different from normal.

This occurs in the following cases:

1. When a MAC.W instruction is located immediately after another MAC.W instruction
2. When a MAC.L instruction is located immediately after a MAC.W instruction
3. When a MULS.W instruction is located immediately after a MAC.W instruction
4. When a DMULS.L instruction is located immediately after a MAC.W instruction
5. When an STS (register) instruction is located immediately after a MAC.W instruction
6. When an STS.L (memory) instruction is located immediately after a MAC.W instruction
7. When an LDS (register) instruction is located immediately after a MAC.W instruction
8. When an LDS.L (memory) instruction is located immediately after a MAC.W instruction

1. When a MAC.W instruction is located immediately after another MAC.W instruction

   The second MA of a MAC.W instruction does not contend with an mm generated by a preceding multiplication instruction.

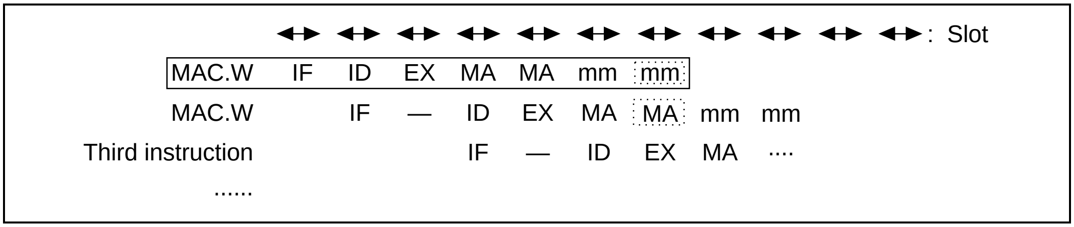

: Slot
MAC.W IF ID EX MA MA mm mm
MAC.W IF — ID EX MA MA mm mm
Third instruction IF — ID EX MA ····
......

Figure 7.33   MAC.W Instruction That Immediately Follows Another MAC.W instruction

Sometimes consecutive MAC.Ws may have misalignment of instruction execution caused by MA and IF contention. Figure 7.34 illustrates a case of this type. This figure assumes MA and IF contention.

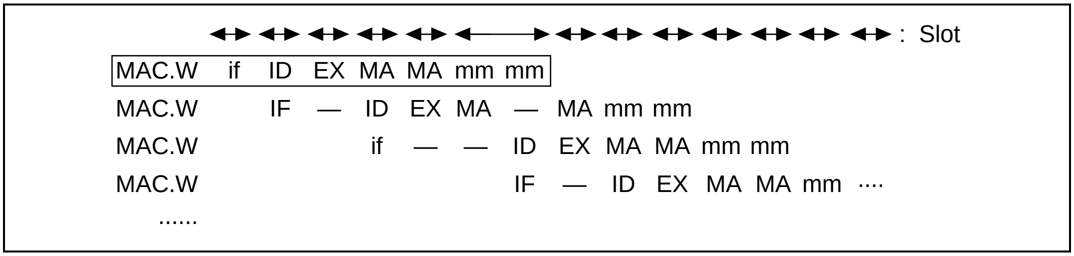

: Slot
MAC.W if ID EX MA MA mm mm
MAC.W IF — ID EX MA — MA mm mm
MAC.W if — — ID EX MA MA mm mm
MAC.W IF — ID EX MA MA mm ····
......

Figure 7.34   Consecutive MAC.Ws with Misalignment

When the second MA of the MAC.W instruction contends with IF, the slot will split as usual. Figure 7.35 illustrates a case of this type. This figure assumes MA and IF contention.

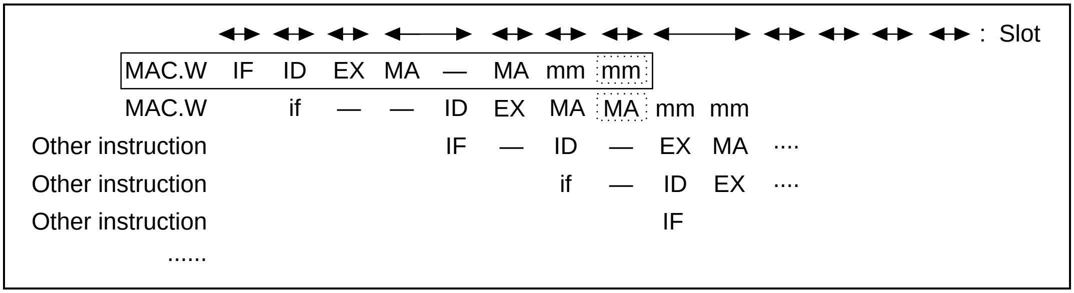

: Slot
MAC.W IF ID EX MA — MA mm mm
MAC.W if — — ID EX MA MA mm mm
Other instruction IF — ID — EX MA ····
Other instruction if — ID EX ····
Other instruction IF
......

Figure 7.35 MA and IF Contention

2. When a MAC.L instruction is located immediately after a MAC.W instruction

The second MA of a MAC.W instruction does not contend with an mm generated by a preceding multiplication instruction (figure 7.36).

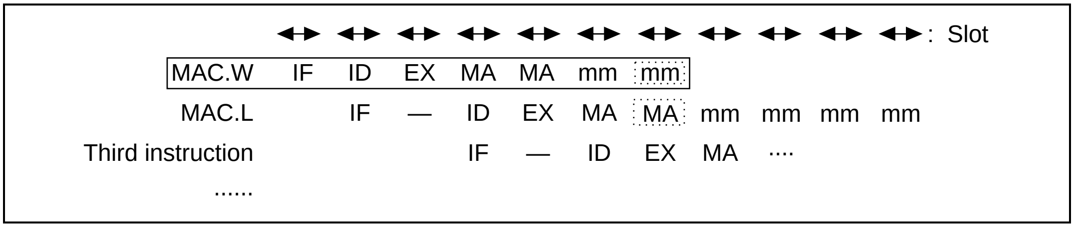

: Slot
MAC.W IF ID EX MA MA mm mm
MAC.L IF — ID EX MA MA mm mm mm mm
Third instruction IF — ID EX MA ····
......

Figure 7.36 MAC.L Instructions Immediately After a MAC.W Instruction

3. When a MULS.W instruction is located immediately after a MAC.W instruction

MULS.W instructions have an MA stage for accessing the multiplier. When the MA of the MULS.W instruction contends with an operating MAC.W instruction multiplier (mm), the MA is extended until the mm ends (the M—A shown in the dotted line box in figure 7.37) to create a single slot. When one or more instructions not related to the multiplier come between the MAC.W and MULS.W instructions, MAC.W and MULS.W contention does not cause stalling. There is no MULS.W MA contention while the MAC.W instruction multiplier is operating (mm). When the MULS.W MA and IF contend, the slot is split.

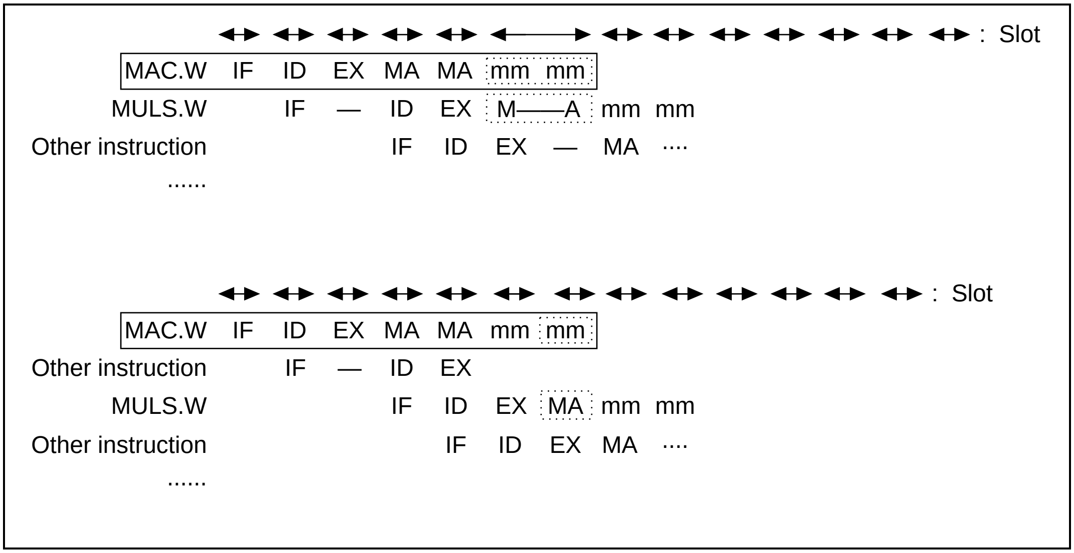

: Slot
MAC.W IF ID EX MA MA mm mm
MULS.W IF — ID EX M——A mm mm
Other instruction IF ID EX — MA ····
......
: Slot
MAC.W IF ID EX MA MA mm mm
Other instruction IF — ID EX
MULS.W IF ID EX MA mm mm
Other instruction IF ID EX MA ····
......

Figure 7.37 MULS.W Instruction Immediately After a MAC.W Instruction

4. When a DMULS.L instruction is located immediately after a MAC.W instruction

DMULS.L instructions have an MA stage for accessing the multiplier, but there is no DMULS.L MA contention while the MAC.W instruction multiplier is operating (mm). When the DMULS.L MA and IF contend, the slot is split (figure 7.38).

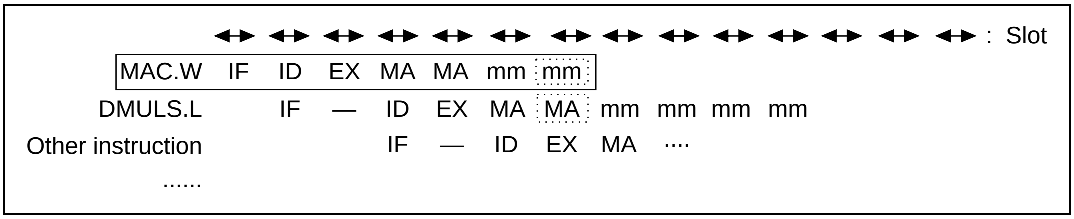

: Slot
MAC.W IF ID EX MA MA mm mm
DMULS.L IF — ID EX MA MA mm mm mm mm
Other instruction IF — ID EX MA ····
......

Figure 7.38 DMULS.L Instructions Immediately After a MAC.W Instruction

5. When an STS (register) instruction is located immediately after a MAC.W instruction

   When the contents of a MAC register are stored in a general-purpose register using an STS instruction, an MA stage for accessing the multiplier is added to the STS instruction, as described later. When the MA of the STS instruction contends with the operating multiplier (mm), the MA is extended until the mm ends (the M—A shown in the dotted line box in figure 7.39) to create a single slot. The MA of the STS contends with the IF. Figure 7.39 illustrates how this occurs, assuming MA and IF contention.

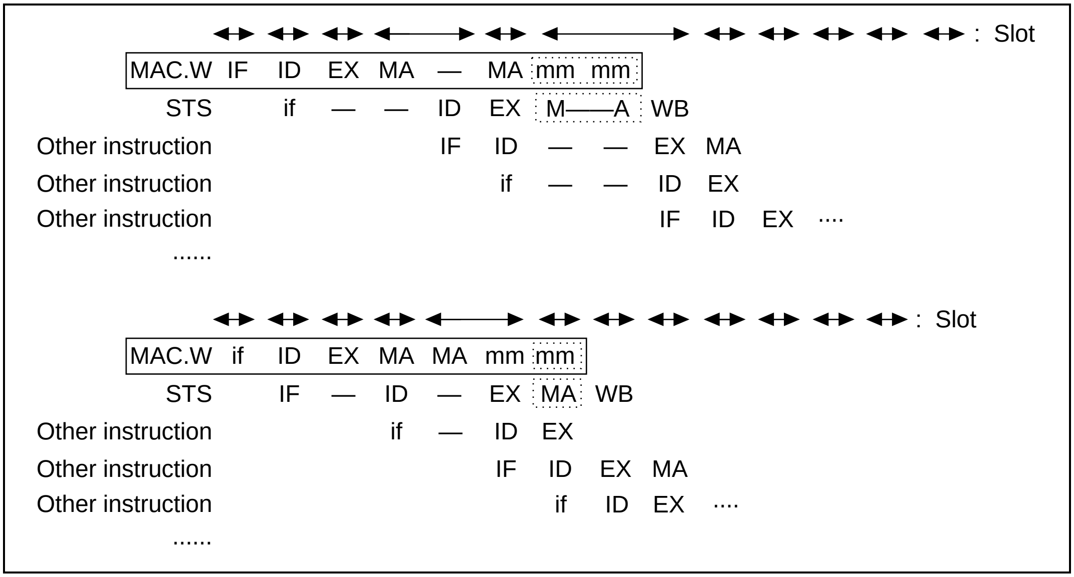

: Slot
MAC.W IF ID EX MA — MA mm mm
STS if — — ID EX M——A WB
Other instruction IF ID — — EX MA
Other instruction if — — ID EX
Other instruction IF ID EX ····
......
: Slot
MAC.W if ID EX MA MA mm mm
STS IF — ID — EX MA WB
Other instruction if — ID EX
Other instruction IF ID EX MA
Other instruction if ID EX ····
......

Figure 7.39 STS (Register) Instruction Immediately After a MAC.W Instruction

6. When an STS.L (memory) instruction is located immediately after a MAC.W instruction

When the contents of a MAC register are stored in memory using an STS instruction, an MA stage for accessing the memory and the multiplier and writing to memory is added to the STS instruction, as described later. Figure 7.40 illustrates how this occurs, assuming MA and IF contention.

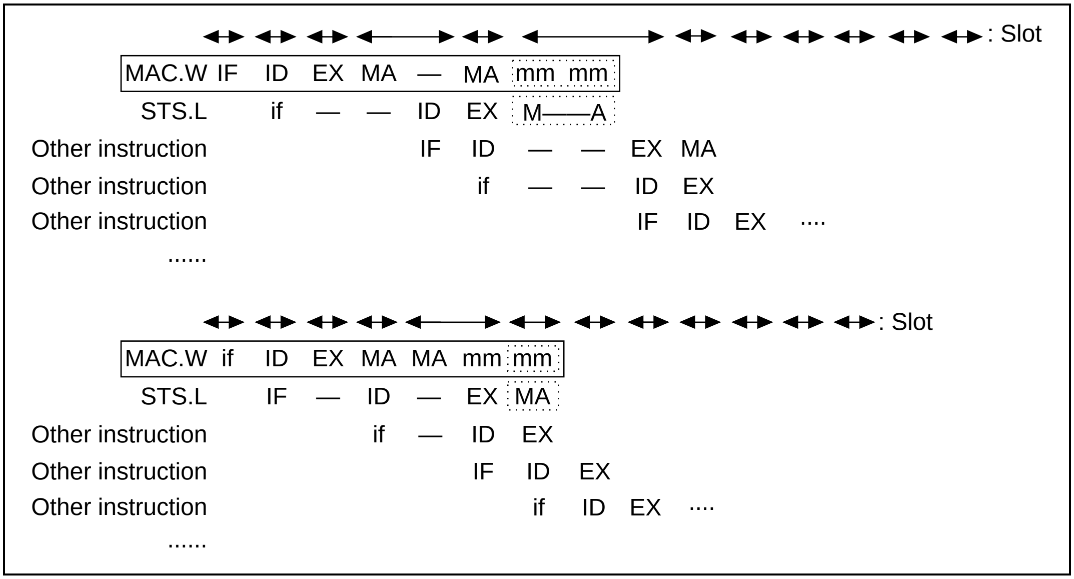

: Slot
MAC.W IF ID EX MA — MA mm mm
STS.L if — — ID EX M——A
Other instruction IF ID — — EX MA
Other instruction if — — ID EX
Other instruction IF ID EX ····
......
: Slot
MAC.W if ID EX MA MA mm mm
STS.L IF — ID — EX MA
Other instruction if — ID EX
Other instruction IF ID EX
Other instruction if ID EX ····
......

Figure 7.40 STS.L (Memory) Instruction Immediately After a MAC.W Instruction

7. When an LDS (register) instruction is located immediately after a MAC.W instruction

When the contents of a MAC register are loaded from a general-purpose register using an LDS instruction, an MA stage for accessing the multiplier is added to the LDS instruction, as described later. When the MA of the LDS instruction contends with the operating multiplier (mm), the MA is extended until the mm ends (the M—A shown in the dotted line box in figure 7.41) to create a single slot. The MA of this LDS contends with IF. Figure 7.41 illustrates how this occurs, assuming MA and IF contention.

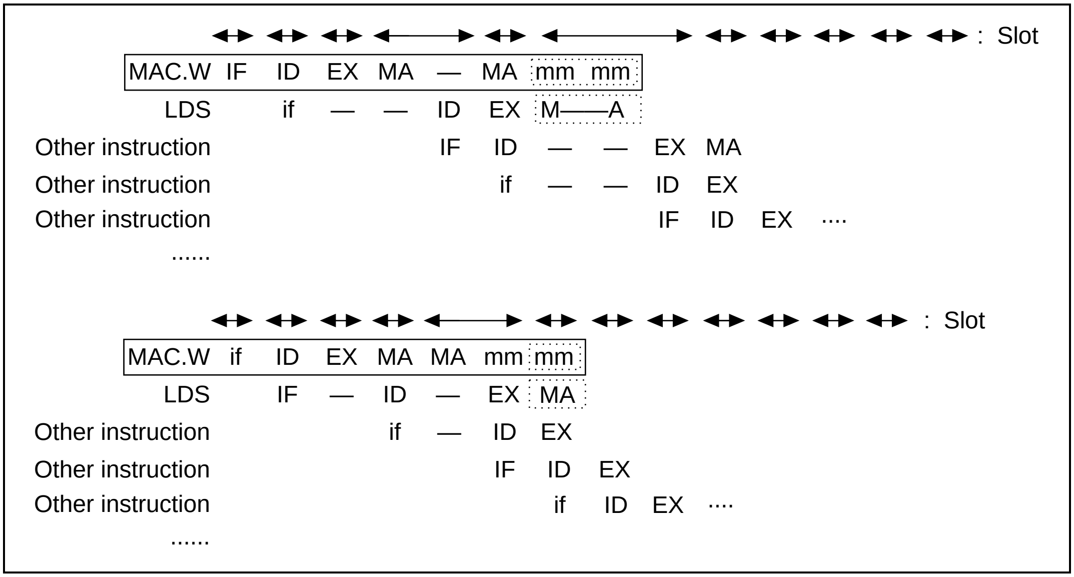

: Slot
MAC.W IF ID EX MA — MA mm mm
LDS if — — ID EX M——A
Other instruction IF ID — — EX MA
Other instruction if — — ID EX
Other instruction IF ID EX ····
......
: Slot
MAC.W if ID EX MA MA mm mm
LDS IF — ID — EX MA
Other instruction if — ID EX
Other instruction IF ID EX
Other instruction if ID EX ····
......

Figure 7.41 LDS (Register) Instruction Immediately After a MAC.W Instruction

8. When an LDS.L (memory) instruction is located immediately after a MAC.W instruction

When the contents of a MAC register are loaded from memory using an LDS instruction, an MA stage for accessing the multiplier is added to the LDS instruction, as described later. When the MA of the LDS instruction contends with the operating multiplier (mm), the MA is extended until the mm ends (the M—A shown in the dotted line box in figure 7.42) to create a single slot. The MA of the LDS contends with IF. Figure 7.42 illustrates how this occurs, assuming MA and IF contention.

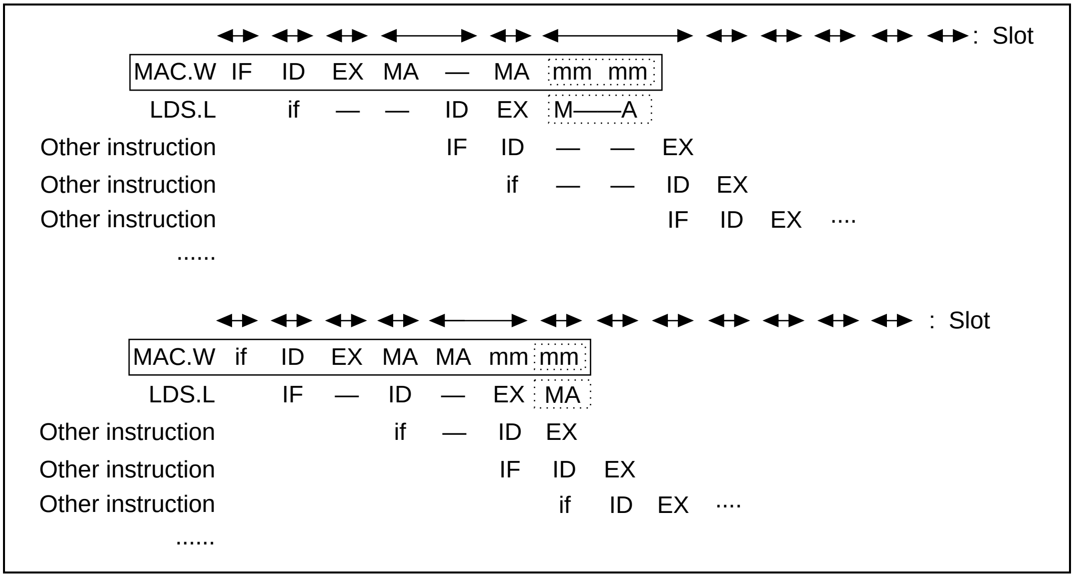

: Slot
MAC.W IF ID EX MA — MA mm mm
LDS.L if — — ID EX M——A
Other instruction IF ID — — EX
Other instruction if — — ID EX
Other instruction IF ID EX ····
......
: Slot
MAC.W if ID EX MA MA mm mm
LDS.L IF — ID — EX MA
Other instruction if — ID EX
Other instruction IF ID EX
Other instruction if ID EX ····
......

Figure 7.42 LDS.L (Memory) Instruction Immediately After a MAC.W Instruction

Double-Length Multiply/Accumulate Instruction (SH-2 CPU, SH-DSP): Includes the following instruction type:

- MAC.L @Rm+, @Rn+ (SH-2 CPU only)

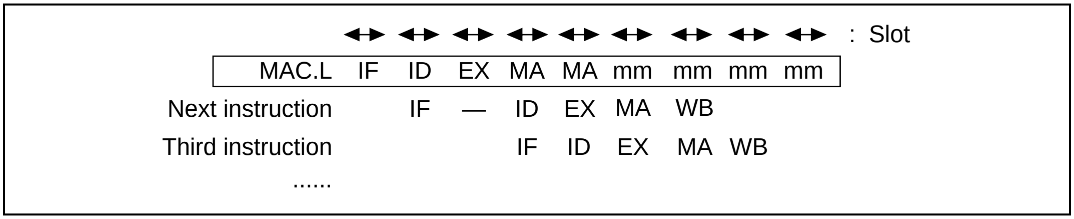

: Slot
MAC.L IF ID EX MA MA mm mm mm mm
Next instruction IF — ID EX MA WB
Third instruction IF ID EX MA WB
......

Figure 7.43 Multiply/Accumulate Instruction Pipeline

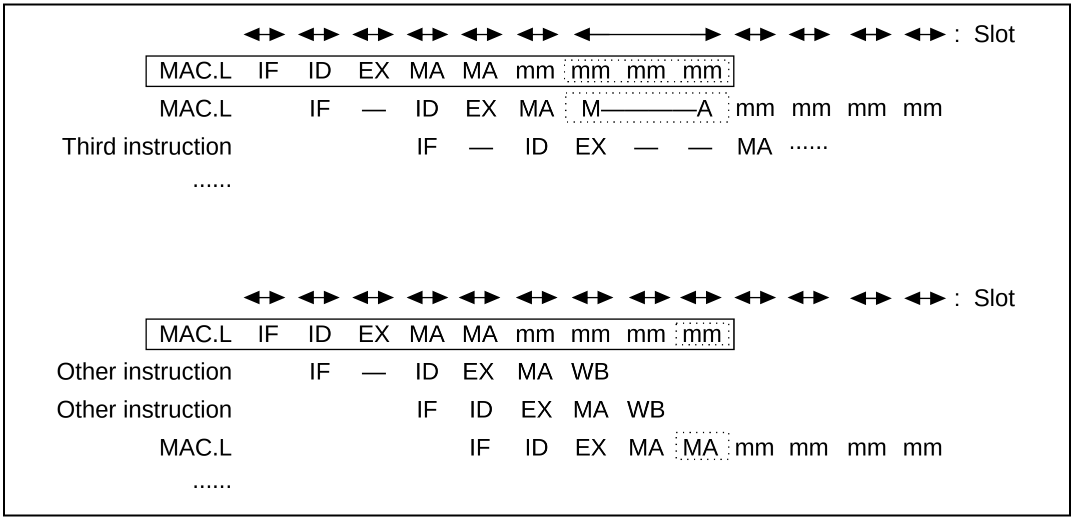
Operation: The pipeline has nine stages: IF, ID, EX, MA, MA, mm, mm, mm, and mm (figure 7.43). The second MA reads the memory and accesses the multiplier. The mm indicates that the multiplier is operating. The mm operates for four cycles after the final MA ends, regardless of a slot. The ID of the instruction 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 7.4, 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 five-stage pipeline instructions of IF, ID, EX, MA, and 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. This occurs in the following cases:

1. When a MAC.L instruction is located immediately after another MAC.L instruction
2. When a MAC.W instruction is located immediately after a MAC.L instruction
3. When a DMULS.L instruction is located immediately after a MAC.L instruction
4. When a MULS.W instruction is located immediately after a MAC.L instruction
5. When an STS (register) instruction is located immediately after a MAC.L instruction
6. When an STS.L (memory) instruction is located immediately after a MAC.L instruction
7. When an LDS (register) instruction is located immediately after a MAC.L instruction
8. When an LDS.L (memory) instruction is located immediately after a MAC.L instruction

1. When a MAC.L instruction is located immediately after another MAC.L instruction

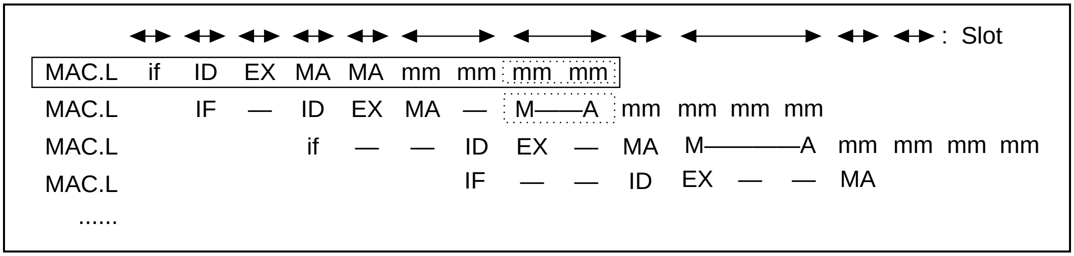
   When the second MA of the MAC.L instruction contends with the mm produced by the previous multiplication instruction, the MA bus cycle is extended until the mm ends (the M—A shown in the dotted line box in figure 7.44) to create a single slot. When two or more instructions that do not use the multiplier occur between two MAC.L instructions, the stall caused by multiplier contention between MAC.L instructions is eliminated.



: Slot
MAC.L IF ID EX MA MA mm mm mm mm
MAC.L IF — ID EX MA M——A mm mm mm mm
Third instruction IF — ID EX — — MA ......
......
: Slot
MAC.L IF ID EX MA MA mm mm mm mm
Other instruction IF — ID EX MA WB
Other instruction IF ID EX MA WB
MAC.L IF ID EX MA MA mm mm mm mm
......

Figure 7.44 MAC.L Instruction Immediately After Another MAC.L Instruction

Sometimes consecutive MAC.Ls may have less multiplier contention even when there is misalignment of instruction execution caused by MA and IF contention. Figure 7.45 illustrates a case of this type, assuming MA and IF contention.



: Slot
MAC.L if ID EX MA MA mm mm mm mm
MAC.L IF — ID EX MA — M——A mm mm mm mm
MAC.L if — — ID EX — MA M————A mm mm mm mm
MAC.L IF — — ID EX — — MA
......

Figure 7.45 Consecutive MAC.Ls with Misalignment

When the second MA of the MAC.L instruction is extended to the end of the mm, contention between the MA and IF will split the slot in the usual way. Figure 7.46 illustrates a case of this type, assuming MA and IF contention.

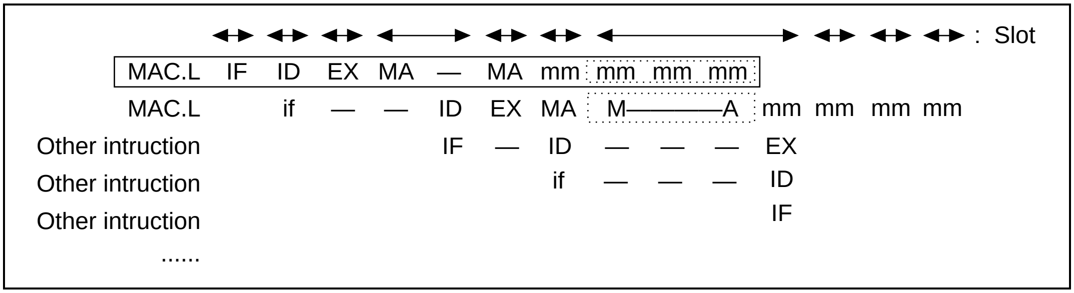

: Slot
MAC.L IF ID EX MA — MA mm mm mm mm
MAC.L if — — ID EX MA M———A mm mm mm mm
Other intruction IF — ID — — — EX
Other intruction if — — — ID
Other intruction IF
......

Figure 7.46 MA and IF Contention

2. When a MAC.W instruction is located immediately after a MAC.L instruction

When the second MA of the MAC.W instruction contends with the mm produced by the previous multiplication instruction, the MA bus cycle is extended until the mm ends (the M—A shown in the dotted line box in figure 7.47) to create a single slot. When two or more instructions that do not use the multiplier occur between the MAC.L and MAC.W instructions, the stall caused by multiplier contention between MAC.L instructions is eliminated.

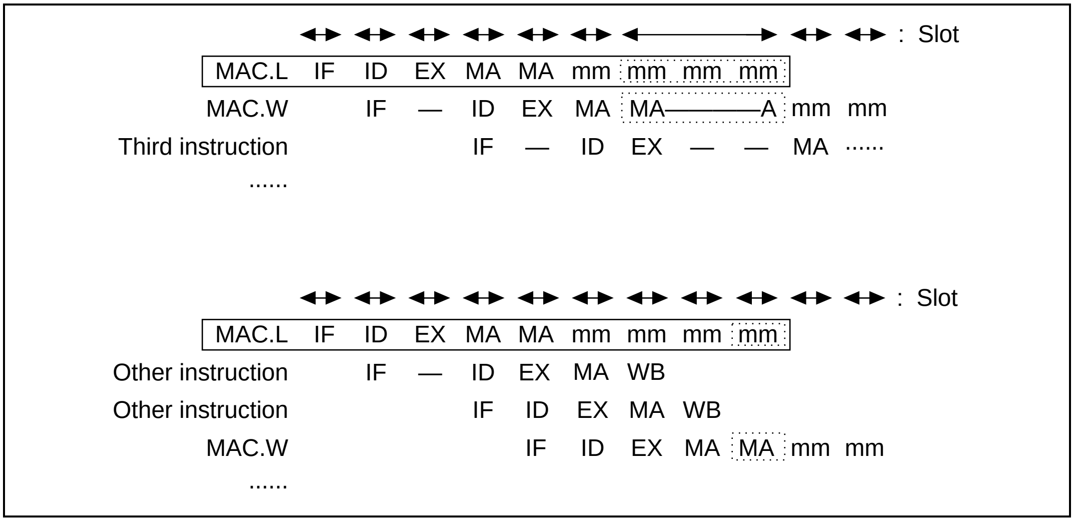
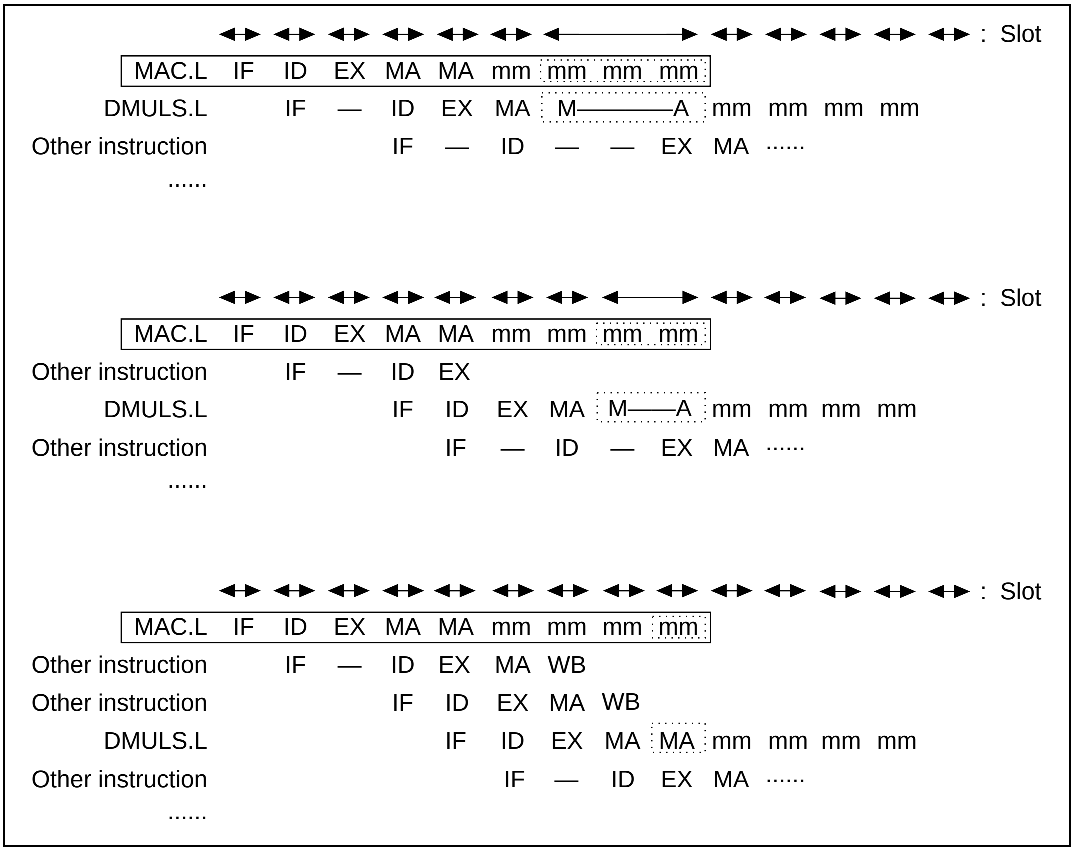

: Slot
MAC.L IF ID EX MA MA mm mm mm mm
MAC.W IF — ID EX MA MA———A mm mm
Third instruction IF — ID EX — — MA ......
......
: Slot
MAC.L IF ID EX MA MA mm mm mm mm
Other instruction IF — ID EX MA WB
Other instruction IF ID EX MA WB
MAC.W IF ID EX MA MA mm mm
......

Figure 7.47 MAC.W Instruction Immediately After a MAC.L Instruction

3. When a DMULS.L instruction is located immediately after a MAC.L instruction

   DMULS.L instructions have an MA stage for accessing the multiplier. When the second MA of the DMULS.L instruction contends with an operating MAC.L instruction multiplier (mm), the MA is extended until the mm ends (the M—A shown in the dotted line box in figure 7.48) to create a single slot. When two or more instructions not related to the multiplier come between the MAC.L and DMULS.L instructions, MAC.L and DMULS.L contention does not cause stalling. When the DMULS.L MA and IF contend, the slot is split.



: Slot
MAC.L IF ID EX MA MA mm mm mm mm
DMULS.L IF — ID EX MA M———A mm mm mm mm
Other instruction IF — ID — — EX MA ······
······
: Slot
MAC.L IF ID EX MA MA mm mm mm mm
Other instruction IF — ID EX
DMULS.L IF ID EX MA M——A mm mm mm mm
Other instruction IF — ID — EX MA ······
······
: Slot
MAC.L IF ID EX MA MA mm mm mm mm
Other instruction IF — ID EX MA WB
Other instruction IF ID EX MA WB
DMULS.L IF ID EX MA MA mm mm mm mm
Other instruction IF — ID EX MA ······
······

Figure 7.48 DMULS.L Instruction Immediately After a MAC.L Instruction

4. When a MULS.W instruction is located immediately after a MAC.L instruction

   MULS.W instructions have an MA stage for accessing the multiplier. When the MA of the MULS.W instruction contends with an operating MAC.L instruction multiplier (mm), the MA is extended until the mm ends (the M—A shown in the dotted line box in figure 7.49) to create a single slot. When three or more instructions not related to the multiplier come between the MAC.L and MULS.W instructions, MAC.L and MULS.W contention does not cause stalling. When the MULS.W MA and IF contend, the slot is split.

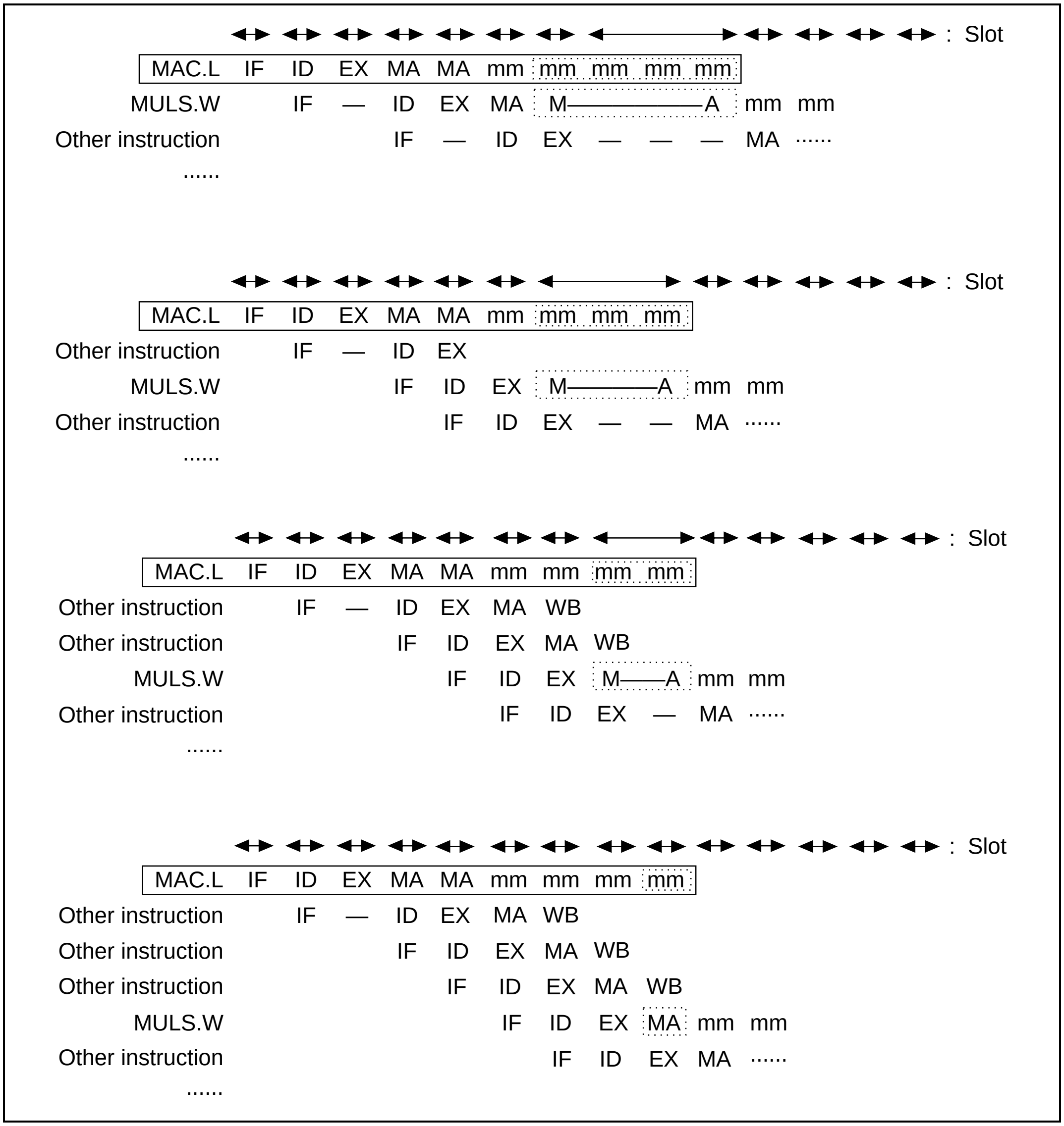

Slot
MAC.L IF ID EX MA MA mm mm mm mm mm
MULS.W IF — ID EX MA M——A mm mm
Other instruction IF — ID EX — — — MA ......
......
Slot
MAC.L IF ID EX MA MA mm mm mm mm
Other instruction IF — ID EX
MULS.W IF ID EX M——A mm mm
Other instruction IF ID EX — — MA ......
......
Slot
MAC.L IF ID EX MA MA mm mm mm mm
Other instruction IF — ID EX MA WB
Other instruction IF ID EX MA WB
MULS.W IF ID EX M——A mm mm
Other instruction IF ID EX — MA ......
......
Slot
MAC.L IF ID EX MA MA mm mm mm mm
Other instruction IF — ID EX MA WB
Other instruction IF ID EX MA WB
Other instruction IF ID EX MA WB
MULS.W IF ID EX MA mm mm
Other instruction IF ID EX MA ......
......

Figure 7.49 MULS.W Instruction Immediately After a MAC.L Instruction

5. When an STS (register) instruction is located immediately after a MAC.L instruction

When the contents of a MAC register are stored in a general-purpose register using an STS instruction, an MA stage for accessing the multiplier is added to the STS instruction, as described later. When the MA of the STS instruction contends with the operating multiplier (mm), the MA is extended until the mm ends (the M—A shown in the dotted line box in figure 7.50) to create a single slot. The MA of the STS contends with the IF. Figure 7.50 illustrates how this occurs, assuming MA and IF contention.

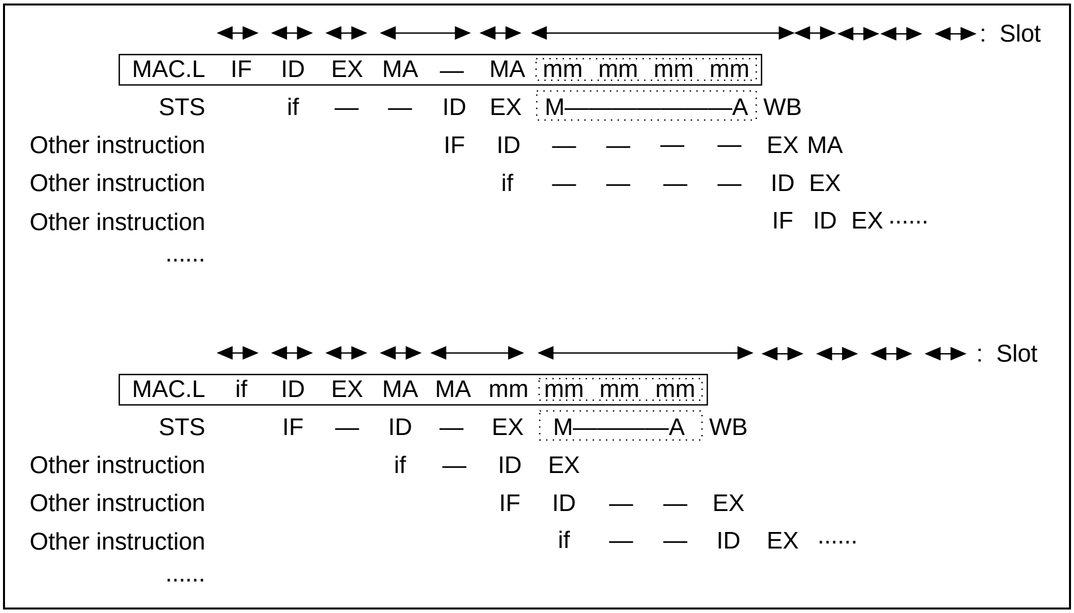
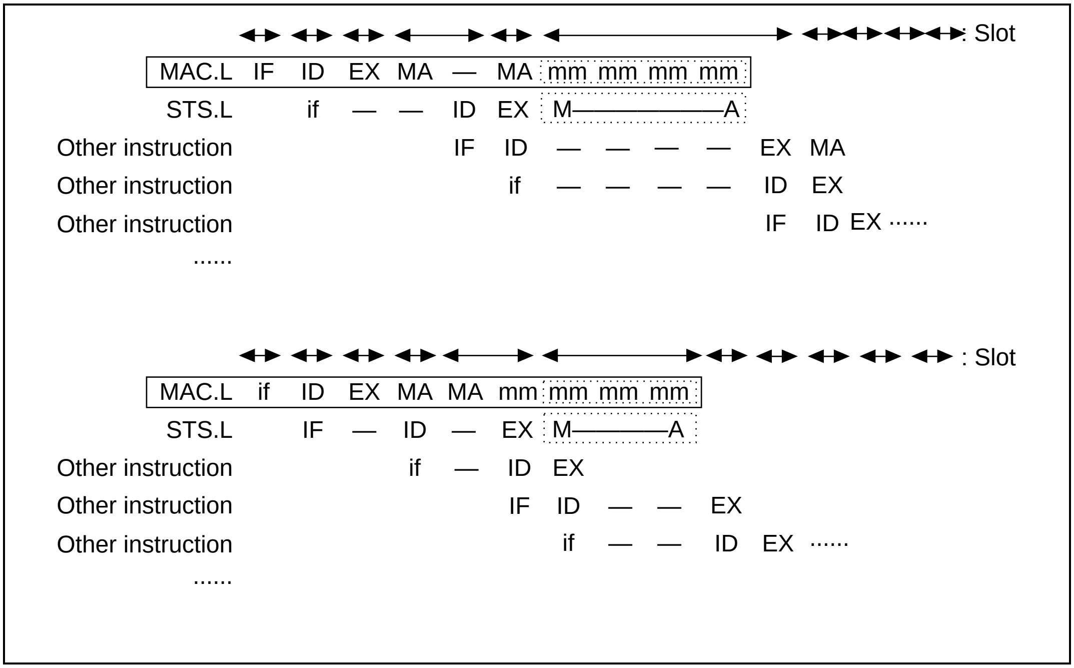

: Slot
MAC.L IF ID EX MA — MA mm mm mm mm
STS if — — ID EX M————A WB
Other instruction IF ID — — — — EX MA
Other instruction if — — — — ID EX
Other instruction IF ID EX ......
......
: Slot
MAC.L if ID EX MA MA mm mm mm mm
STS IF — ID — EX M————A WB
Other instruction if — ID EX
Other instruction IF ID — — EX
Other instruction if — — ID EX ......
......

Figure 7.50 STS (Register) Instruction Immediately After a MAC.L Instruction

6. When an STS.L (memory) instruction is located immediately after a MAC.L instruction

When the contents of a MAC register are stored in memory using an STS instruction, an MA stage for accessing the multiplier and writing to memory is added to the STS instruction, as described later. The MA of the STS contends with the IF. Figure 7.51 illustrates how this occurs, assuming MA and IF contention.



: Slot
MAC.L IF ID EX MA — MA mm mm mm mm
STS.L if — — ID EX M——A
Other instruction IF ID — — — — EX MA
Other instruction if — — — — ID EX
Other instruction IF ID EX ......
......
: Slot
MAC.L if ID EX MA MA mm mm mm mm
STS.L IF — ID — EX M——A
Other instruction if — ID EX
Other instruction IF ID — — EX
Other instruction if — — ID EX ......
......

Figure 7.51 STS.L (Memory) Instruction Immediately After a MAC.L Instruction

7. When an LDS (register) instruction is located immediately after a MAC.L instruction

When the contents of a MAC register are loaded from a general-purpose register using an LDS instruction, an MA stage for accessing the multiplier is added to the LDS instruction, as described later. When the MA of the LDS instruction contends with the operating multiplier (mm), the MA is extended until the mm ends (the M—A shown in the dotted line box in figure 7.52) to create a single slot. The MA of this LDS contends with IF. Figure 7.52 illustrates how this occurs, assuming MA and IF contention.

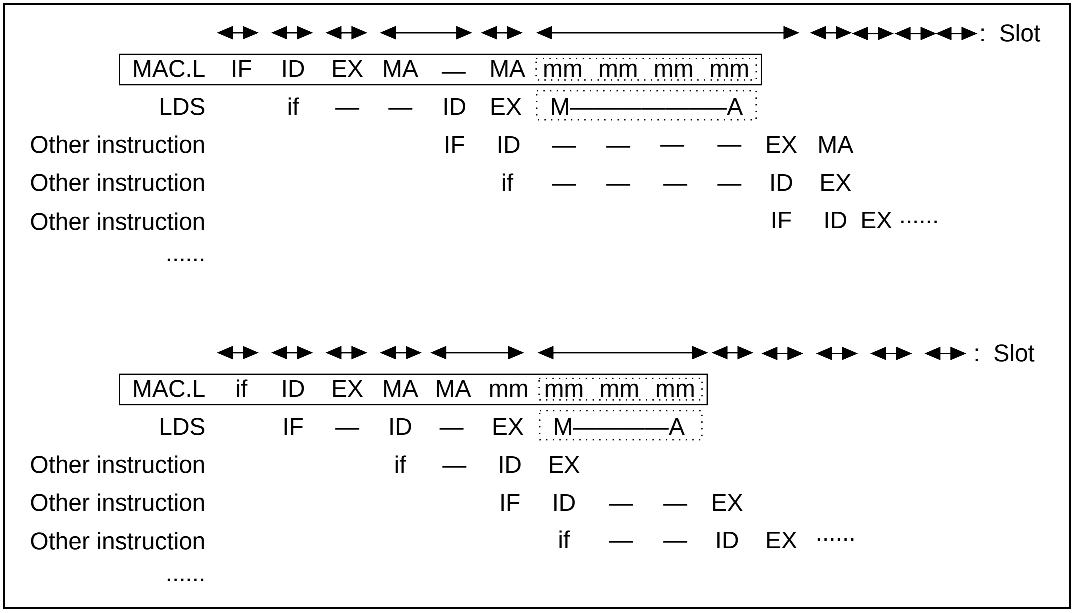

Slot
MAC.L IF ID EX MA — MA mm mm mm mm
LDS if — — ID EX M—A
Other instruction IF ID — — — — EX MA
Other instruction if — — — — ID EX
Other instruction IF ID EX ......
......
Slot
MAC.L if ID EX MA MA mm mm mm mm
LDS IF — ID — EX M—A
Other instruction if — ID EX
Other instruction IF ID — — EX
Other instruction if — — ID EX ......
......

Figure 7.52 LDS (Register) Instruction Immediately After a MAC.L Instruction

8. When an LDS.L (memory) instruction is located immediately after a MAC.L instruction

When the contents of a MAC register are loaded from memory using an LDS instruction, an MA stage for accessing the memory and the memory and the multiplier is added to the LDS instruction, as described later. When the MA of the LDS instruction contends with the operating multiplier (mm), the MA is extended until the mm ends (the M—A shown in the dotted line box in figure 7.53) to create a single slot. The MA of the LDS contends with IF. Figure 7.53 illustrates how this occurs, assuming MA and IF contention.

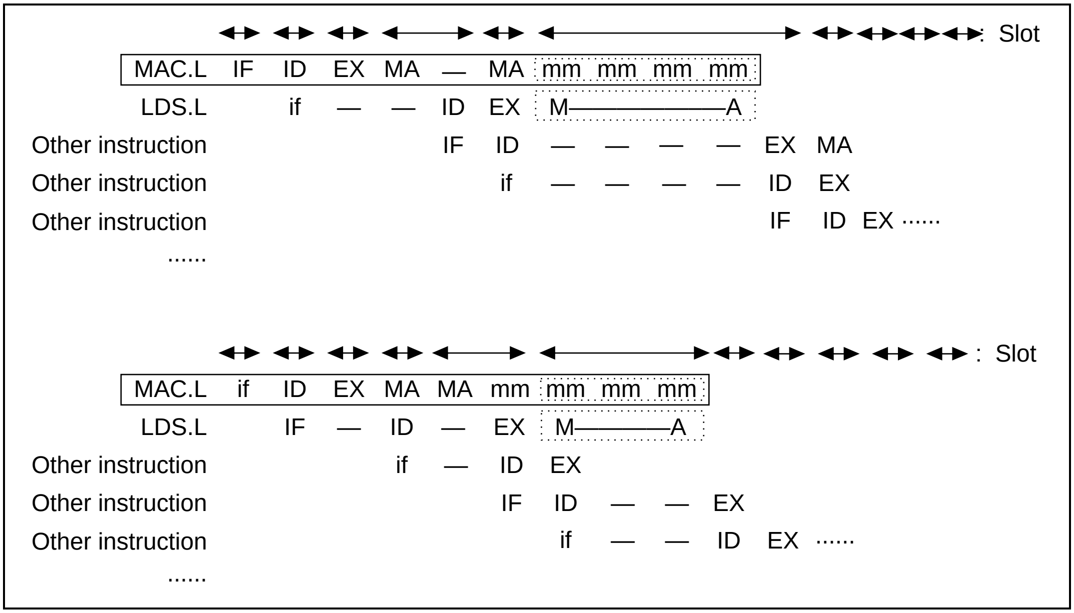

: Slot
MAC.L IF ID EX MA — MA mm mm mm mm
LDS.L if — — ID EX M——————A
Other instruction IF ID — — — — EX MA
Other instruction if — — — — ID EX
Other instruction IF ID EX ......
......
: Slot
MAC.L if ID EX MA MA mm mm mm mm
LDS.L IF — ID — EX M————A
Other instruction if — ID EX
Other instruction IF ID — — EX
Other instruction if — — ID EX ......
......

Figure 7.53 LDS.L (Memory) Instruction Immediately After a MAC.L Instruction

Multiplication Instructions (SH-1 CPU): Include the following instruction types:

- MULS.W Rm, Rn
- MULU.W Rm, Rn

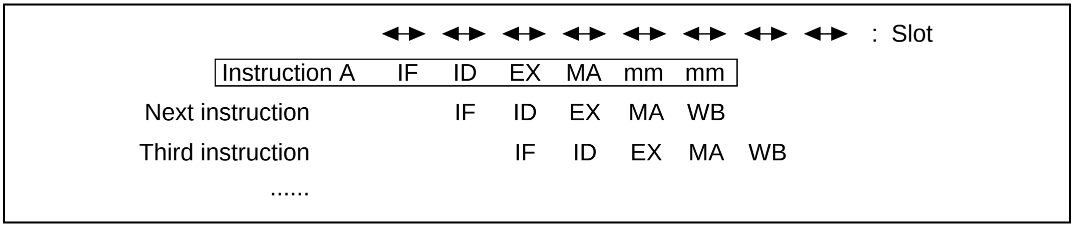

: Slot
Instruction A IF ID EX MA mm mm
Next instruction IF ID EX MA WB
Third instruction IF ID EX MA WB
......

Figure 7.54 Multiplication Instruction Pipeline

The pipeline has six stages: IF, ID, EX, MA, mm, and mm. The MA accesses the multiplier. mm indicates that the multiplier is operating. mm operates for three cycles after the MA ends, regardless of slot. The MA of the MULS.W instruction, when it contends with IF, splits the slot as described in Section 7.2.1, 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 comes after the MULS.W instruction, however, contention occurs with the multiplier, so operation is different from normal.

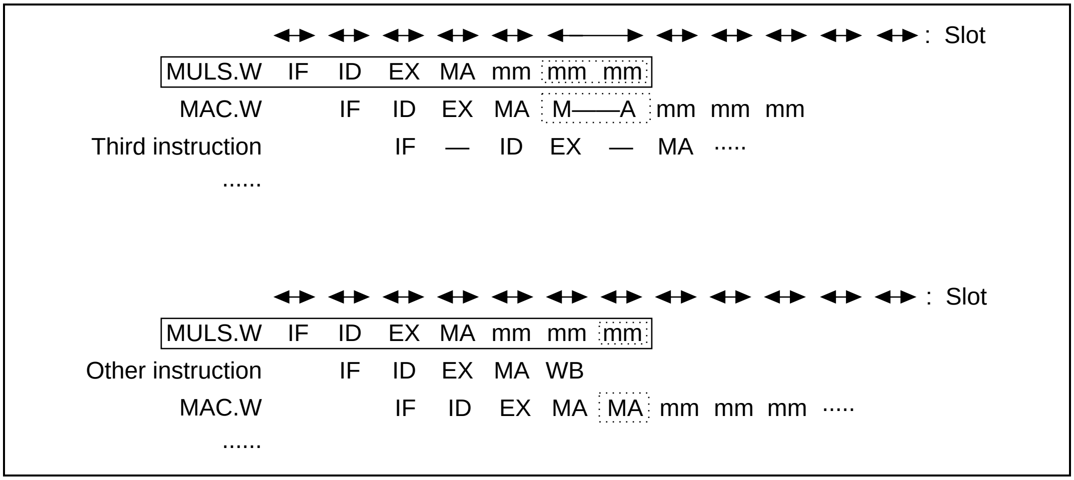
This occurs in the following cases:

1. When a MAC.W instruction is located immediately after a MULS.W instruction
2. When a MULS.W instruction is located immediately after another MULS.W instruction
3. When an STS (register) instruction is located immediately after a MULS.W instruction
4. When an STS.L (memory) instruction is located immediately after a MULS.W instruction
5. When an LDS (register) instruction is located immediately after a MULS.W instruction
6. When an LDS.L (memory) instruction is located immediately after a MULS.W instruction

1. When a MAC.W instruction is located immediately after a MULS.W instruction

   When the second MA of a MAC.W instruction contends with the mm generated by a preceding multiplication instruction, the bus cycle of that MA is extended until the mm ends (the M—A shown in the dotted line box below) and that extended MA occupies one slot.

   If one or more instructions not related to the multiplier comes between the MULS.W and MAC.W instructions, multiplier contention between the MULS.W and MAC.W instructions does not cause stalls (figure 7.55).



: Slot
MULS.W IF ID EX MA mm mm mm
MAC.W IF ID EX MA M——A mm mm mm
Third instruction IF — ID EX — MA .....
......
: Slot
MULS.W IF ID EX MA mm mm mm
Other instruction IF ID EX MA WB
MAC.W IF ID EX MA MA mm mm mm .....
......

Figure 7.55 MAC.W Instruction Immediately After a MULS.W Instruction

2. When a MULS.W instruction is located immediately after another MULS.W instruction

MULS.W instructions have an MA stage for accessing the multiplier. When the MA of the MULS.W instruction contends with the operating multiplier (mm) of another MULS.W instruction, the MA is extended until the mm ends (the M—A shown in the dotted line box in figure 7.56) to create a single slot. When two or more instructions not related to the multiplier are located between the two MULS.W instructions, contention between the MULS.Ws does not cause stalling. When the MULS.W MA and IF contend, the slot is split.

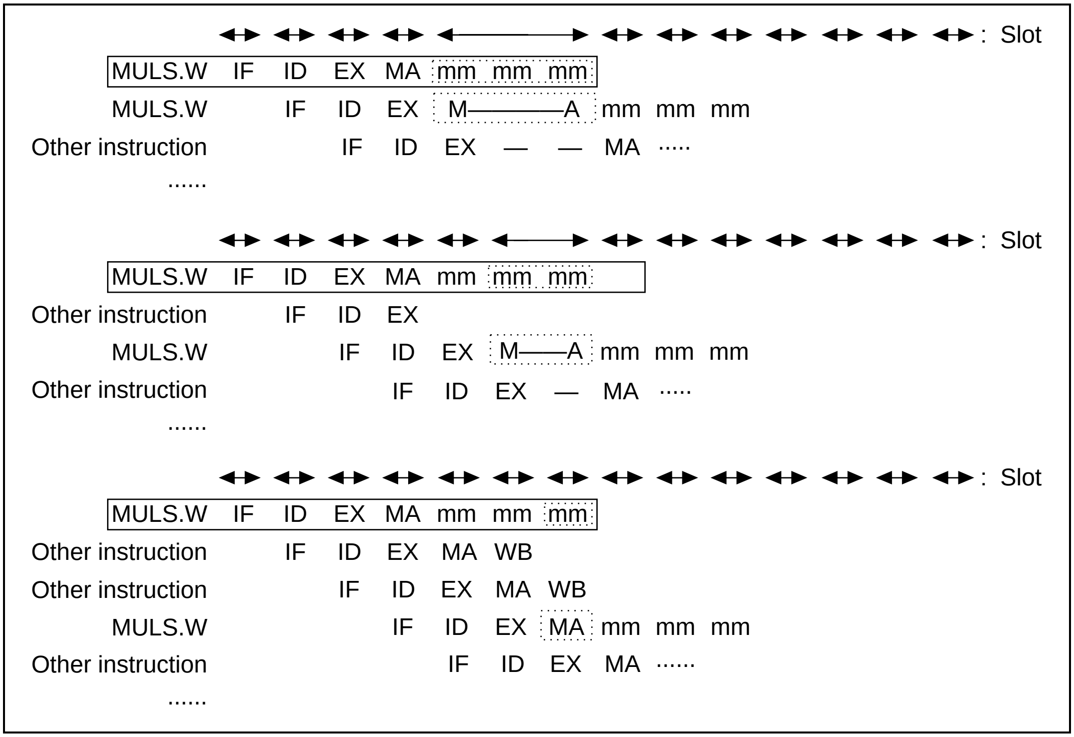

: Slot
MULS.W IF ID EX MA mm mm mm
MULS.W IF ID EX M———A mm mm mm
Other instruction IF ID EX — — MA ·····
......
: Slot
MULS.W IF ID EX MA mm mm mm
Other instruction IF ID EX
MULS.W IF ID EX M——A mm mm mm
Other instruction IF ID EX — MA ·····
......
: Slot
MULS.W IF ID EX MA mm mm mm
Other instruction IF ID EX MA WB
Other instruction IF ID EX MA WB
MULS.W IF ID EX MA mm mm mm
Other instruction IF ID EX MA ······
......

Figure 7.56 MULS.W Instruction Immediately After Another MULS.W Instruction

When the MA of the MULS.W instruction is extended until the mm ends, contention between MA and IF will split the slot, as is normal. Figure 7.57 illustrates a case of this type, assuming MA and IF contention.

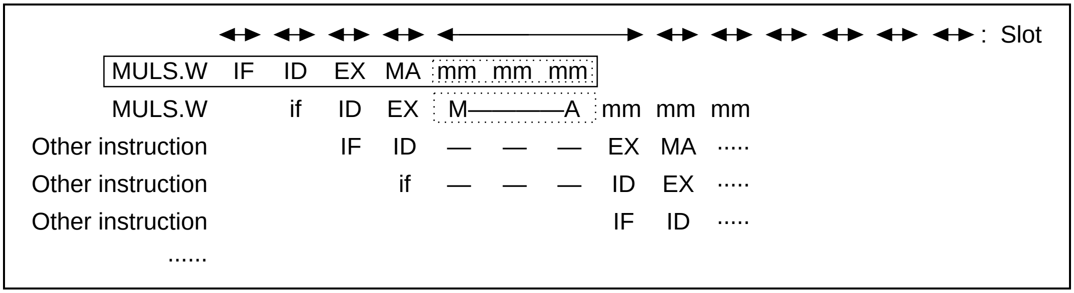
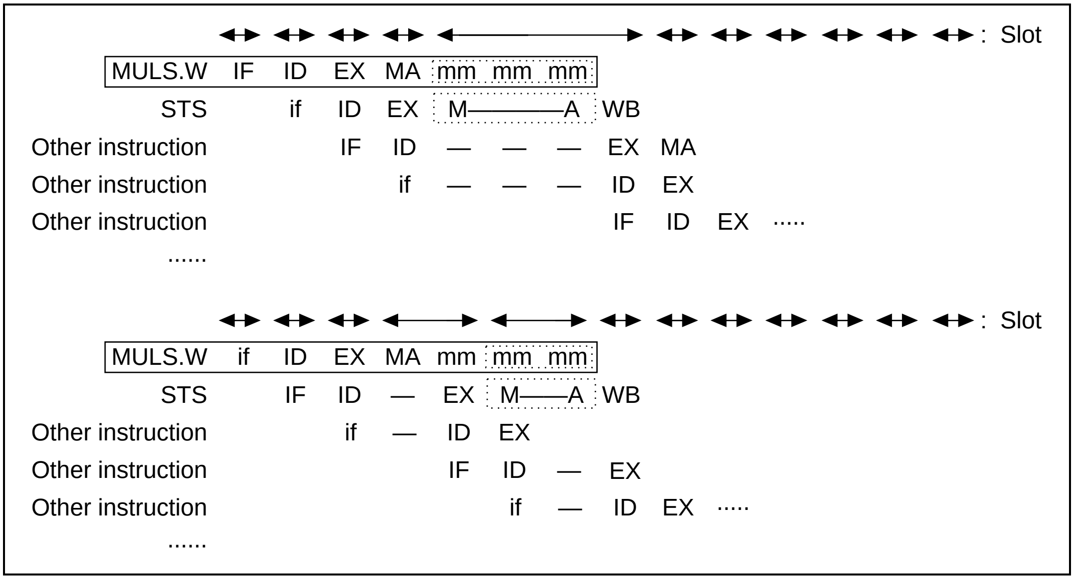

: Slot
MULS.W IF ID EX MA mm mm mm
MULS.W if ID EX M——A mm mm mm
Other instruction IF ID — — — EX MA .....
Other instruction if — — — ID EX .....
Other instruction IF ID .....
......

Figure 7.57 MULS.W Instruction Immediately After Another MULS.W Instruction (IF and MA Contention)

3. When an STS (register) instruction is located immediately after a MULS.W instruction

   When the contents of a MAC register are stored in a general-purpose register using an STS instruction, an MA stage for accessing the multiplier is added to the STS instruction, as described later. When the MA of the STS instruction contends with the operating multiplier (mm), the MA is extended until the mm ends (the M—A shown in the dotted line box in figure 7.58) to create a single slot. The MA of the STS contends with the IF. Figure 7.58 illustrates how this occurs, assuming MA and IF contention.



Slot
MULS.W IF ID EX MA mm mm mm
STS if ID EX M——A WB
Other instruction IF ID — — — EX MA
Other instruction if — — — ID EX
Other instruction IF ID EX .....
......
Slot
MULS.W if ID EX MA mm mm mm
STS IF ID — EX M——A WB
Other instruction if — ID EX
Other instruction IF ID — EX
Other instruction if — ID EX .....
......

Figure 7.58 STS (Register) Instruction Immediately After a MULS.W Instruction

4. When an STS.L (memory) instruction is located immediately after a MULS.W instruction

When the contents of a MAC register are loaded from memory using an STS instruction, an MA stage for accessing the multiplier and writing to memory is added to the STS instruction, as described later. When the MA of the STS instruction contends with the operating multiplier (mm), the MA is extended until one cycle after the mm ends (the M—A shown in the dotted line box in figure 7.59) to create a single slot. The MA of the STS contends with the IF. Figure 7.59 illustrates how this occurs, assuming MA and IF contention.

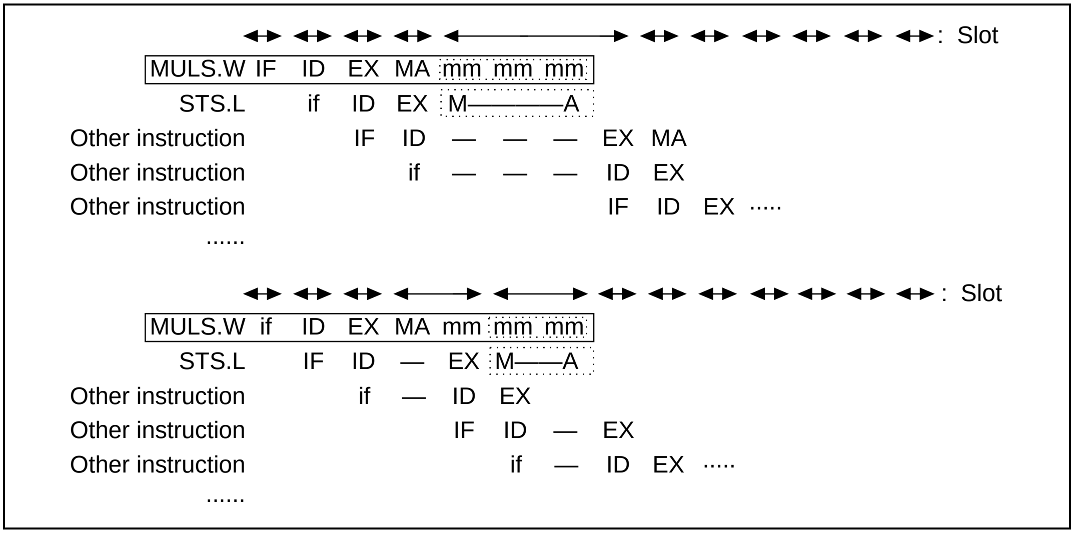
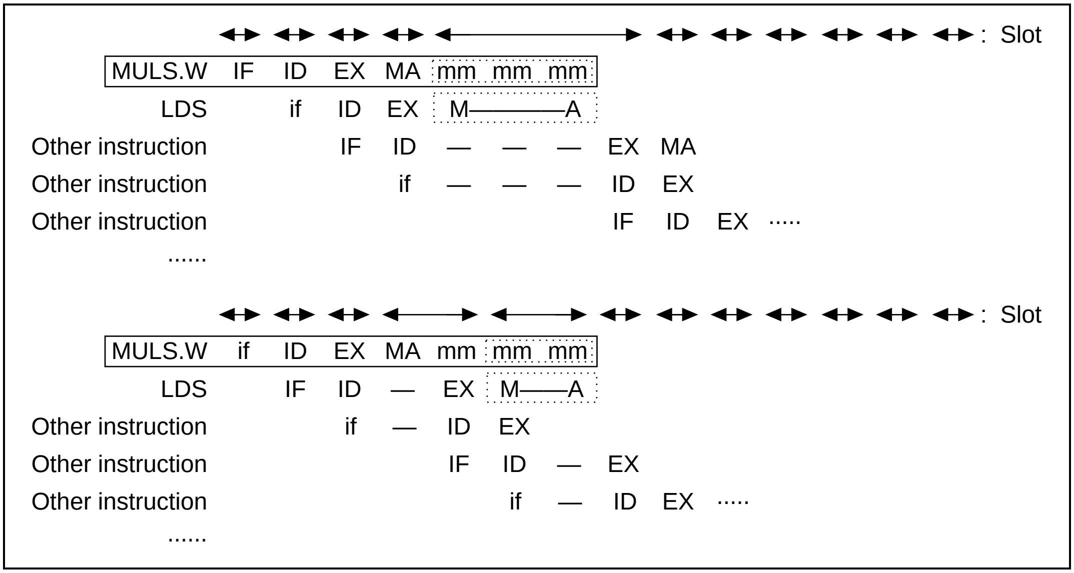

Slot
MULS.W IF ID EX MA mm mm mm
STS.L if ID EX M——A
Other instruction IF ID — — — EX MA
Other instruction if — — — ID EX
Other instruction IF ID EX .....
......
Slot
MULS.W if ID EX MA mm mm mm
STS.L IF ID — EX M——A
Other instruction if — ID EX
Other instruction IF ID — EX
Other instruction if — ID EX .....
......

Figure 7.59 STS.L (Memory) Instruction Immediately After a MULS.W Instruction

5. When an LDS (register) instruction is located immediately after a MULS.W instruction

When the contents of a MAC register are loaded from a general-purpose register using an LDS instruction, an MA stage for accessing the multiplier is added to the LDS instruction, as described later. When the MA of the LDS instruction contends with the operating multiplier (mm), the MA is extended until the mm ends (the M—A shown in the dotted line box below) to create a single slot. The MA of this LDS contends with IF. Figure 7.60 illustrates how this occurs, assuming MA and IF contention.

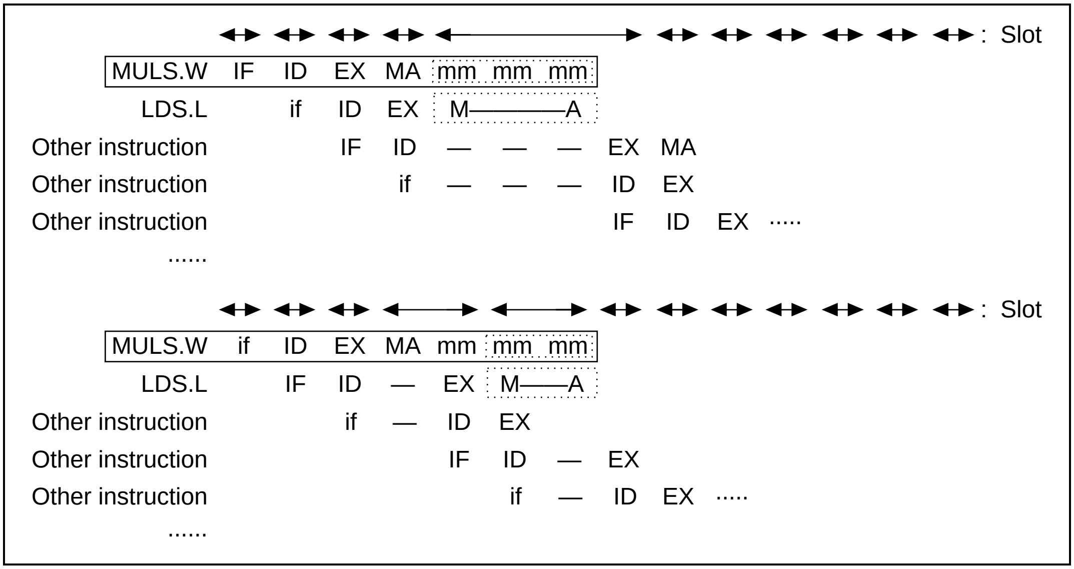


: Slot
MULS.W IF ID EX MA mm mm mm
LDS if ID EX M————A
Other instruction IF ID — — — EX MA
Other instruction if — — — ID EX
Other instruction IF ID EX ·····
......
: Slot
MULS.W if ID EX MA mm mm mm
LDS IF ID — EX M——A
Other instruction if — ID EX
Other instruction IF ID — EX
Other instruction if — ID EX ·····
......

Figure 7.60 LDS (Register) Instruction Immediately After a MULS.W Instruction

6. When an LDS.L (memory) instruction is located immediately after a MULS.W instruction

   When the contents of a MAC register are loaded from memory using an LDS instruction, an MA stage for accessing the memory and the multiplier is added to the LDS instruction, as described later. When the MA of the LDS instruction contends with the operating multiplier (mm), the MA is extended until the mm ends (the M—A shown in the dotted line box in figure 7.61) to create a single slot. The MA of the LDS contends with IF. Figure 7.61 illustrates how this occurs, assuming MA and IF contention.

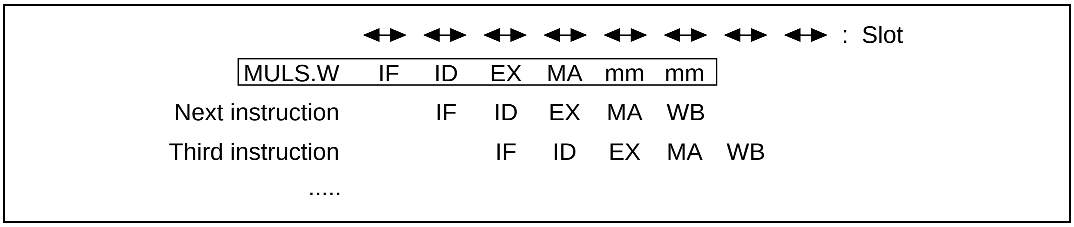


: Slot
MULS.W IF ID EX MA mm mm mm
LDS.L if ID EX M————A
Other instruction IF ID — — — EX MA
Other instruction if — — — ID EX
Other instruction IF ID EX ·····
......
: Slot
MULS.W if ID EX MA mm mm mm
LDS.L IF ID — EX M——A
Other instruction if — ID EX
Other instruction IF ID — EX
Other instruction if — ID EX ·····
......

Figure 7.61 LDS.L (Memory) Instruction Immediately After a MULS.W Instruction

Multiplication Instructions (SH-2 CPU, SH-DSP): Include the following instruction types:

- MULS.W Rm, Rn
- MULU.W Rm, Rn



: Slot
MULS.W IF ID EX MA mm mm
Next instruction IF ID EX MA WB
Third instruction IF ID EX MA WB
.....

Figure 7.62 Multiplication Instruction Pipeline

Operation: The pipeline has six stages: IF, ID, EX, MA, mm, and mm (figure 8.62). 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, when it contends with IF, splits the slot as described in Section 7.4, 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 four-stage pipeline instructions of IF, ID, EX, and MA. In such cases, it operates like a normal pipeline. When an instruction that uses the multiplier is located after the MULS.W instruction, however, contention occurs with the multiplier, so operation is not as normal. This occurs in the following cases:

1. When a MAC.W instruction is located immediately after a MULS.W instruction
2. When a MAC.L instruction is located immediately after a MULS.W instruction
3. When a MULS.W instruction is located immediately after another MULS.W instruction
4. When a DMULS.L instruction is located immediately after a MULS.W instruction
5. When an STS (register) instruction is located immediately after a MULS.W instruction
6. When an STS.L (memory) instruction is located immediately after a MULS.W instruction
7. When an LDS (register) instruction is located immediately after a MULS.W instruction
8. When an LDS.L (memory) instruction is located immediately after a MULS.W instruction

1. When a MAC.W instruction is located immediately after a MULS.W instruction

   The second MA of a MAC.W instruction does not contend with the mm generated by a preceding multiplication instruction.

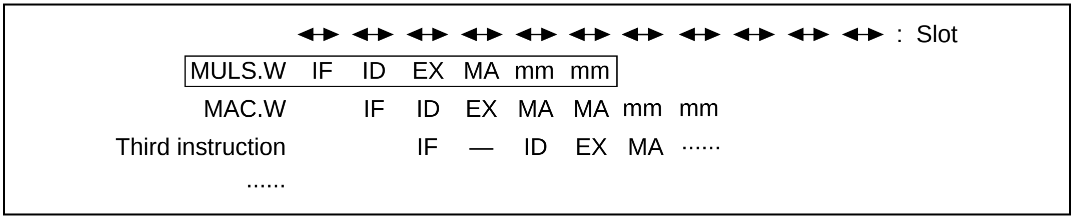

: Slot
MULS.W IF ID EX MA mm mm
MAC.W IF ID EX MA MA mm mm
Third instruction IF — ID EX MA ······
......

Figure 7.63 MAC.W Instruction Immediately After a MULS.W Instruction

2. When a MAC.L instruction is located immediately after a MULS.W instruction

   The second MA of a MAC.W instruction does not contend with the mm generated by a preceding multiplication instruction.

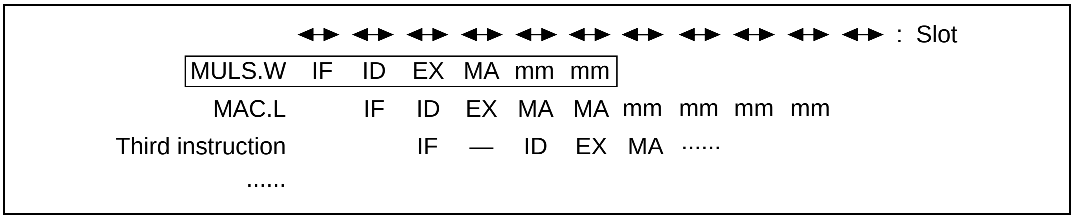
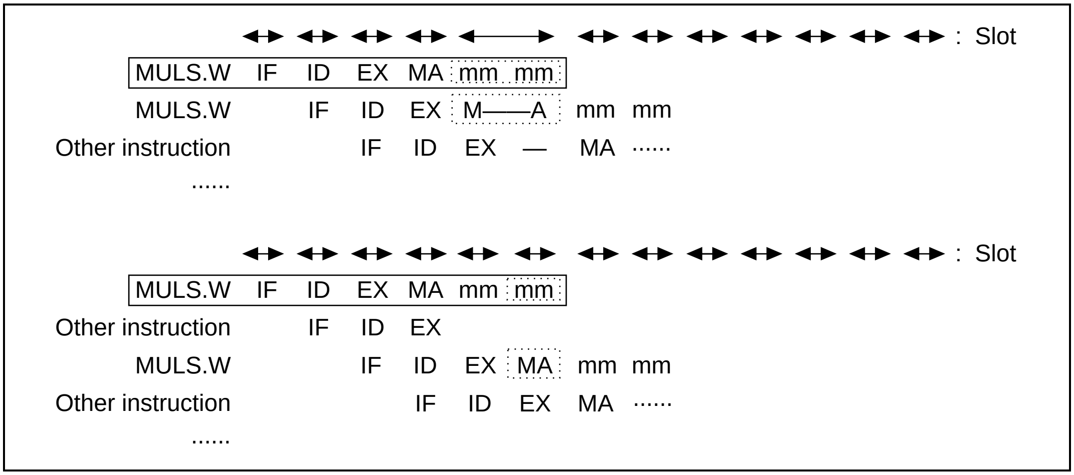

: Slot
MULS.W IF ID EX MA mm mm
MAC.L IF ID EX MA MA mm mm mm mm
Third instruction IF — ID EX MA ······
......

Figure 7.64 MAC.L Instruction Immediately After a MULS.W Instruction

3. When a MULS.W instruction is located immediately after another MULS.W instruction

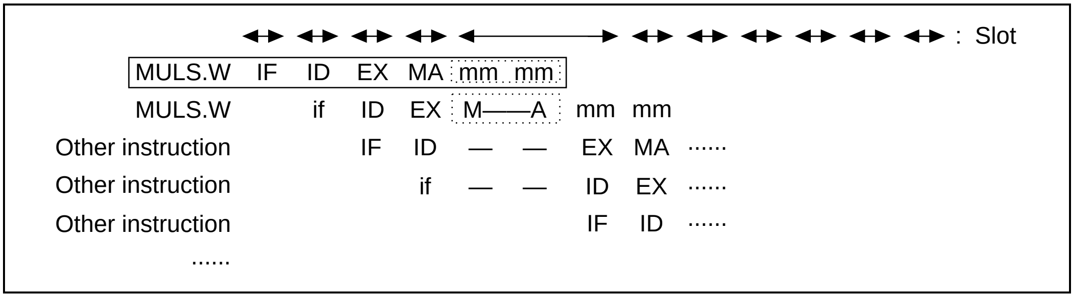
MULS.W instructions have an MA stage for accessing the multiplier. When the MA of the MULS.W instruction contends with the operating multiplier (mm) of another MULS.W instruction, the MA is extended until the mm ends (the M—A shown in the dotted line box in figure 7.65) to create a single slot. When one or more instructions not related to the multiplier is located between the two MULS.W instructions, contention between the MULS.Ws does not cause stalling. When the MULS.W MA and IF contend, the slot is split.



: Slot
MULS.W IF ID EX MA mm mm
MULS.W IF ID EX M——A mm mm
Other instruction IF ID EX — MA ......
......
: Slot
MULS.W IF ID EX MA mm mm
Other instruction IF ID EX
MULS.W IF ID EX MA mm mm
Other instruction IF ID EX MA ......
......

Figure 7.65 MULS.W Instruction Immediately After Another MULS.W Instruction

When the MA of the MULS.W instruction is extended until the mm ends, contention between the MA and IF will split the slot in the usual way. Figure 7.66 illustrates a case of this type, assuming MA and IF contention.

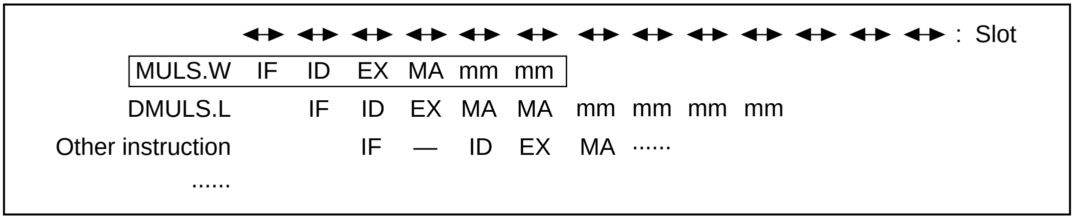


: Slot
MULS.W IF ID EX MA mm mm
MULS.W if ID EX M——A mm mm
Other instruction IF ID — — EX MA ......
Other instruction if — — ID EX ......
Other instruction IF ID ......
......

Figure 7.66 MULS.W Instruction Immediately After Another MULS.W Instruction (IF and MA contention)

4. When a DMULS.L instruction is located immediately after a MULS.W instruction

   Though the second MA in the DMULS.L instruction makes an access to the multiplier, it does not contend with the operating multiplier (mm) generated by the MULS.W instruction.

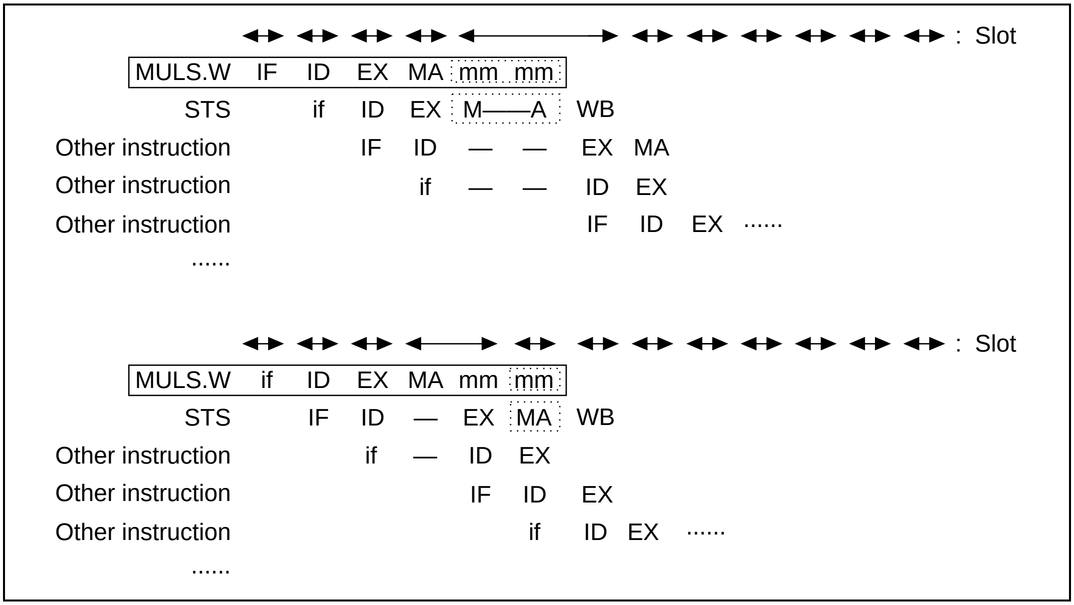


: Slot
MULS.W IF ID EX MA mm mm
DMULS.L IF ID EX MA MA mm mm mm mm
Other instruction IF — ID EX MA ······
······

Figure 7.67 DMULS.L Instruction Immediately After a MULS.W Instruction

5. When an STS (register) instruction is located immediately after a MULS.W instruction

When the contents of a MAC register are stored in a general-purpose register using an STS instruction, an MA stage for accessing the multiplier is added to the STS instruction, as described later. When the MA of the STS instruction contends with the operating multiplier (mm), the MA is extended until the mm ends (the M—A shown in the dotted line box in figure 7.68) to create a single slot. The MA of the STS contends with the IF. Figure 7.68 illustrates how this occurs, assuming MA and IF contention.



: Slot
MULS.W IF ID EX MA mm mm
STS if ID EX M——A WB
Other instruction IF ID — — EX MA
Other instruction if — — ID EX
Other instruction IF ID EX ......
......
: Slot
MULS.W if ID EX MA mm mm
STS IF ID — EX MA WB
Other instruction if — ID EX
Other instruction IF ID EX
Other instruction if ID EX ......
......

Figure 7.68 STS (Register) Instruction Immediately After a MULS.W Instruction

6. When an STS.L (memory) instruction is located immediately after a MULS.W instruction

When the contents of a MAC register are stored in memory using an STS instruction, an MA stage for accessing the multiplier and writing to memory is added to the STS instruction, as described later. The MA of the STS contends with the IF. Figure 7.69 illustrates how this occurs, assuming MA and IF contention.

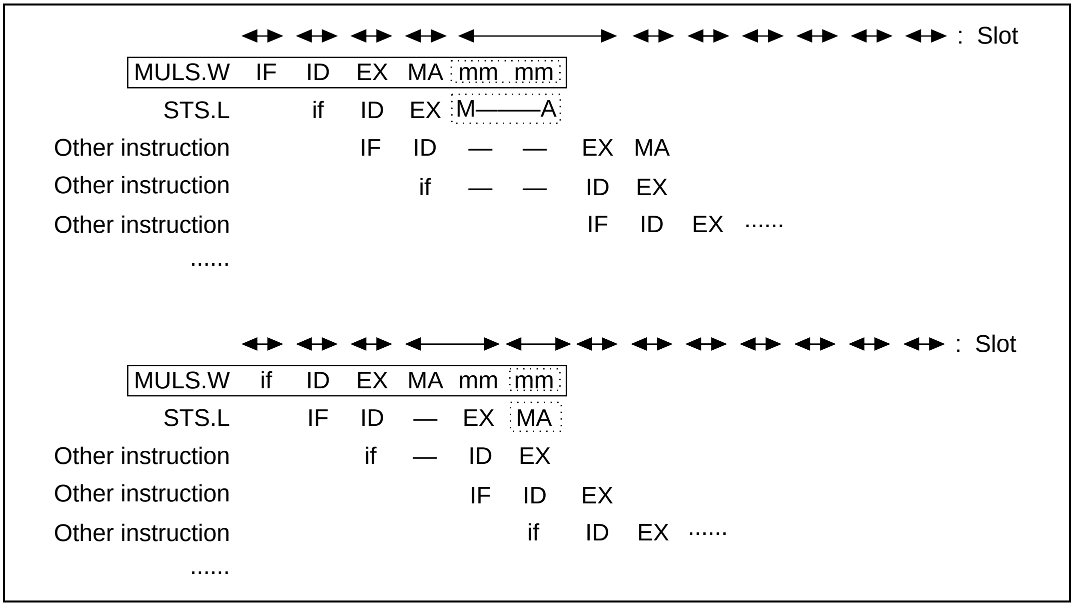
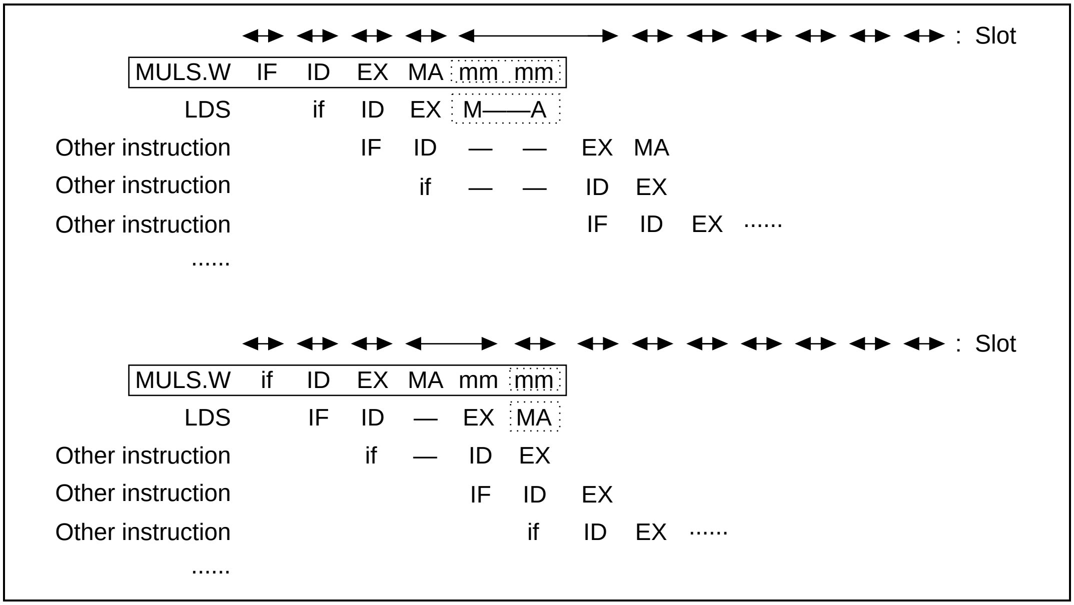

Slot
MULS.W IF ID EX MA mm mm
STS.L if ID EX M——A
Other instruction IF ID — — EX MA
Other instruction if — — ID EX
Other instruction IF ID EX ......
......
Slot
MULS.W if ID EX MA mm mm
STS.L IF ID — EX MA
Other instruction if — ID EX
Other instruction IF ID EX
Other instruction if ID EX ......
......

Figure 7.69 STS.L (Memory) Instruction Immediately After a MULS.W Instruction

7. When an LDS (register) instruction is located immediately after a MULS.W instruction

When the contents of a MAC register are loaded from a general-purpose register using an LDS instruction, an MA stage for accessing the multiplier is added to the LDS instruction, as described later. When the MA of the LDS instruction contends with the operating multiplier (mm), the MA is extended until the mm ends (the M—A shown in the dotted line box below) to create a single slot. The MA of this LDS contends with IF. The following figures illustrates how this occurs, assuming MA and IF contention.

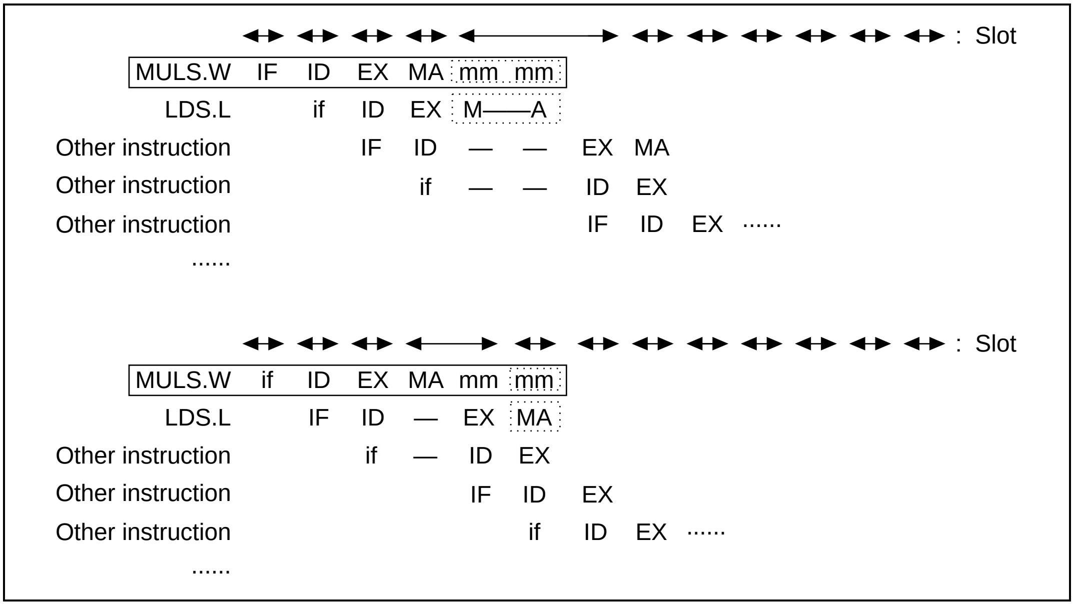


Slot
MULS.W IF ID EX MA mm mm
LDS if ID EX M——A
Other instruction IF ID — — EX MA
Other instruction if — — ID EX
Other instruction IF ID EX ......
......
Slot
MULS.W if ID EX MA mm mm
LDS IF ID — EX MA
Other instruction if — ID EX
Other instruction IF ID EX
Other instruction if ID EX ......
......

Figure 7.70 LDS (Register) Instruction Immediately After a MULS.W Instruction

8. When an LDS.L (memory) instruction is located immediately after a MULS.W instruction

When the contents of a MAC register are loaded from memory using an LDS instruction, an MA stage for accessing the multiplier is added to the LDS instruction, as described later. When the MA of the LDS instruction contends with the operating multiplier (mm), the MA is extended until the mm ends (the M—A shown in the dotted line box in figure 7.71) to create a single slot. The MA of the LDS contends with IF. Figure 7.71 illustrates how this occurs, assuming MA and IF contention.



Slot
MULS.W IF ID EX MA mm mm
LDS.L if ID EX M——A
Other instruction IF ID — — EX MA
Other instruction if — — ID EX
Other instruction IF ID EX ......
......
Slot
MULS.W if ID EX MA mm mm
LDS.L IF ID — EX MA
Other instruction if — ID EX
Other instruction IF ID EX
Other instruction if ID EX ......
......

Figure 7.71 LDS.L (Memory) Instruction Immediately After a MULS.W Instruction

Double-Length Multiplication Instructions (SH-2 CPU, SH-DSP): Include the following instruction types:

- DMULS.L Rm, Rn
- DMULU.L Rm, Rn
- MUL.L Rm, Rn

	↔	↔	↔	↔	↔	↔	↔	↔	↔ : Slot
Instruction A	IF	ID	EX	MA	MA	mm	mm	mm	mm
Next instruction		IF	—	ID	EX	MA	WB		
Third instruction				IF	ID	EX	MA	WB	
......									

Figure 7.72 Multiplication Instruction Pipeline

The pipeline has nine stages: IF, ID, EX, MA, MA, mm, mm, mm, and mm (figure 7.72). The second MA accesses the multiplier. The mm indicates that the multiplier is operating. The mm operates for four 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 7.2.1, 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 different from normal.

This occurs in the following cases:

1. When a MAC.L instruction is located immediately after a DMULS.L instruction
2. When a MAC.W instruction is located immediately after a DMULS.L instruction
3. When a DMULS.L instruction is located immediately after another DMULS.L instruction
4. When a MULS.W instruction is located immediately after a DMULS.L instruction
5. When an STS (register) instruction is located immediately after a DMULS.L instruction
6. When an STS.L (memory) instruction is located immediately after a DMULS.L instruction
7. When an LDS (register) instruction is located immediately after a DMULS.L instruction
8. When an LDS.L (memory) instruction is located immediately after a DMULS.L instruction

1. When a MAC.L instruction is located immediately after a DMULS.L instruction

   When the second MA of a MAC.L instruction contends with the mm generated by a preceding multiplication instruction, the bus cycle of that MA is extended until the mm ends (the M—A shown in the dotted line box below) and that extended MA occupies one slot.

   If two or more instructions not related to the multiplier are located between the DMULS.L and MAC.L instructions, multiplier contention between the DMULS.L and MAC.L instructions does not cause stalls (figure 7.73).

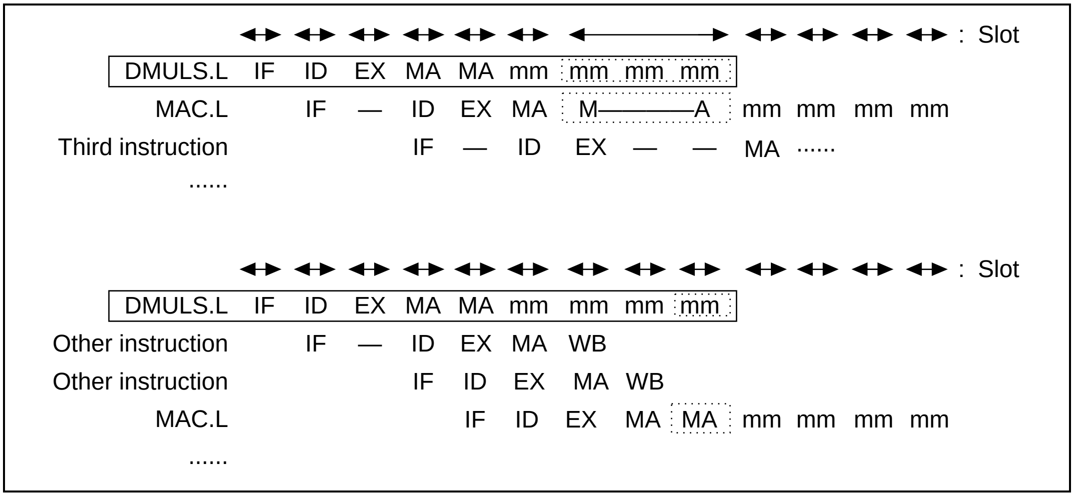

: Slot
DMULS.L IF ID EX MA MA mm mm mm mm
MAC.L IF — ID EX MA M———A mm mm mm mm
Third instruction IF — ID EX — — MA ......
......
: Slot
DMULS.L IF ID EX MA MA mm mm mm mm
Other instruction IF — ID EX MA WB
Other instruction IF ID EX MA WB
MAC.L IF ID EX MA MA mm mm mm mm
......

Figure 7.73 MAC.L Instruction Immediately After a DMULS.L Instruction

7.4.3 Logic Operation Instructions

Register-Register Logic Operation Instructions (Common): Include the following instruction types:

- AND Rm, Rn
- AND #imm, R0
- NOT Rm, Rn
- OR Rm, Rn
- OR #imm, R0
- TST Rm, Rn
- TST #imm, R0
- XOR Rm, Rn
- XOR #imm, R0

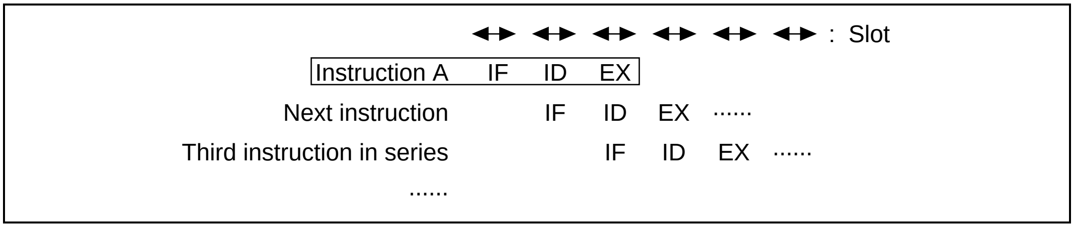

: Slot
Instruction A IF ID EX
Next instruction IF ID EX ......
Third instruction in series IF ID EX ......
......

Figure 7.74 Register-Register Logic Operation Instruction Pipeline

The pipeline has three stages: IF, ID, and EX (figure 7.74). The data operation is completed in the EX stage via the ALU.

Memory Logic Operations Instructions (Common): Include the following instruction types:

- AND.B #imm, @(R0, GBR)
- OR.B #imm, @(R0, GBR)
- TST.B #imm, @(R0, GBR)
- XOR.B #imm, @(R0, GBR)

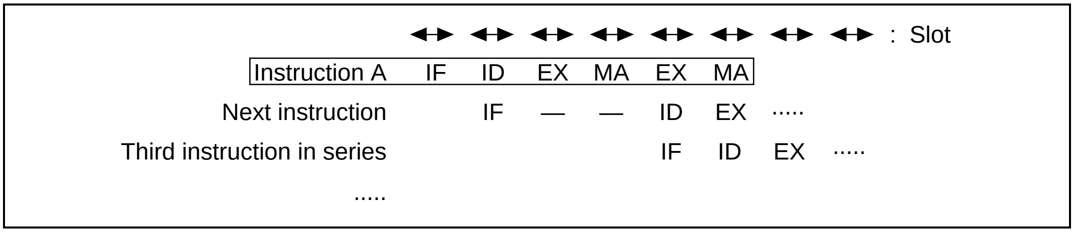

: Slot
Instruction A IF ID EX MA EX MA
Next instruction IF — — ID EX .....
Third instruction in series IF ID EX .....
.....

Figure 7.75 Memory Logic Operation Instruction Pipeline

The pipeline has six stages: IF, ID, EX, MA, EX, and MA (figure 7.75). The ID of the next instruction stalls for 2 slots. The MAs of these instructions contend with IF.

TAS Instruction (Common): Includes the following instruction type:

- TAS.B @Rn

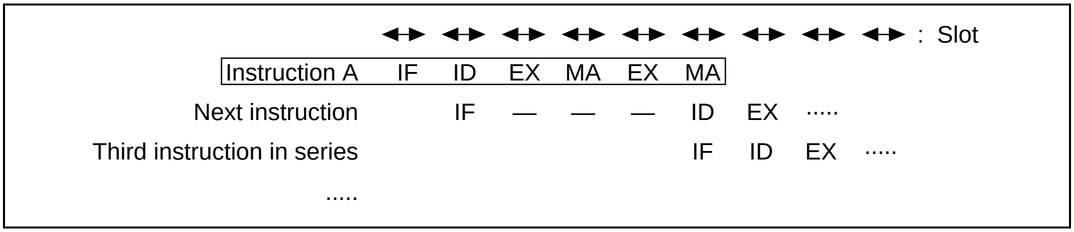

: Slot
Instruction A IF ID EX MA EX MA
Next instruction IF — — — ID EX .....
Third instruction in series IF ID EX .....
.....

Figure 7.76 TAS Instruction Pipeline

The pipeline has six stages: IF, ID, EX, MA, EX, and MA (figure 7.76). The ID of the next instruction stalls for 3 slots. The MA of the TAS instruction contends with IF.

7.4.4 Shift Instructions (Common)

• ROTL Rn	• SHLR Rn
• ROTR Rn	• SHLL2 Rn
• ROTCL Rn	• SHLR2 Rn
• ROTCR Rn	• SHLL8 Rn
• SHAL Rn	• SHLR8 Rn
• SHAR Rn	• SHLL16 Rn
• SHLL Rn	• SHLR16 Rn

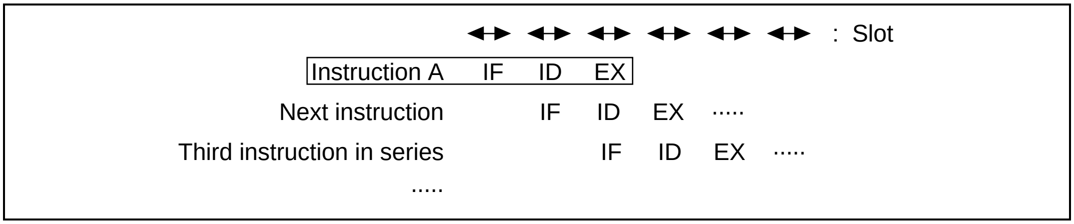

: Slot
Instruction A IF ID EX
Next instruction IF ID EX .....
Third instruction in series IF ID EX .....
.....

Figure 7.77 General Shift Instruction Pipeline

The pipeline has three stages: IF, ID, and EX (figure 7.77). The data operation is completed in the EX stage via the ALU.

7.4.5 Branch Instructions

Conditional Branch Instructions (Common): Include the following instruction types:

- BF label
- BT label

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 7.78).

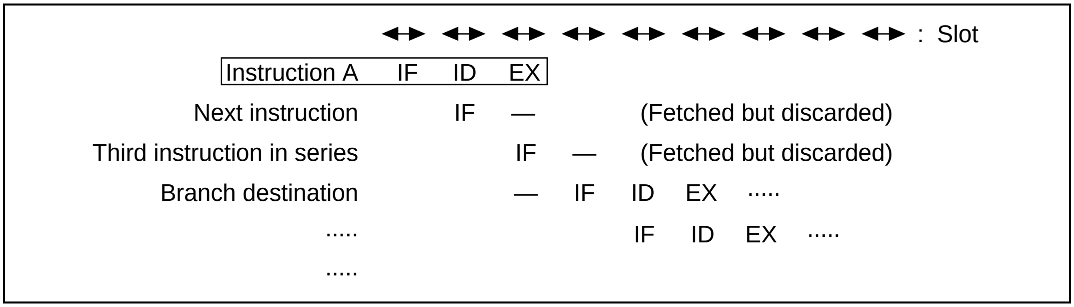
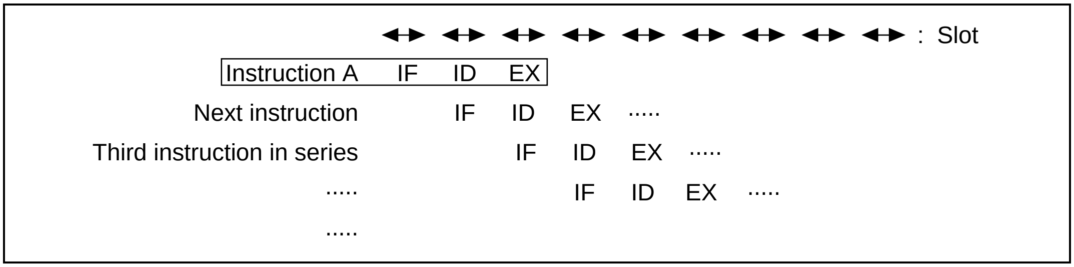


Figure 7.78 Branch Instruction when Condition Is Satisfied

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 7.79).



: Slot
Instruction A IF ID EX
Next instruction IF ID EX .....
Third instruction in series IF ID EX .....
..... IF ID EX .....
.....

Figure 7.79 Branch Instruction when Condition Is Not Satisfied

Delayed Conditional Branch Instructions (SH-2 CPU, SH-DSP): Include the following instruction types:

- BF/S label
- BT/S label

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 7.80).

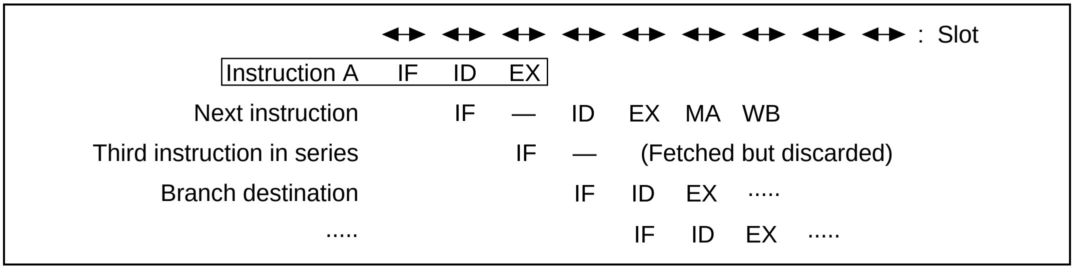

: Slot
Instruction A IF ID EX
Next instruction IF — ID EX MA WB
Third instruction in series IF — (Fetched but discarded)
Branch destination IF ID EX .....
..... IF ID EX .....

Figure 7.80 Branch Instruction when Condition Is Satisfied

2. When condition is not satisfied

   If it is determined that a condition is not satisfied at the ID stage, the EX stage proceeds without doing anything. The next instruction also executes a fetch (figure 7.81).

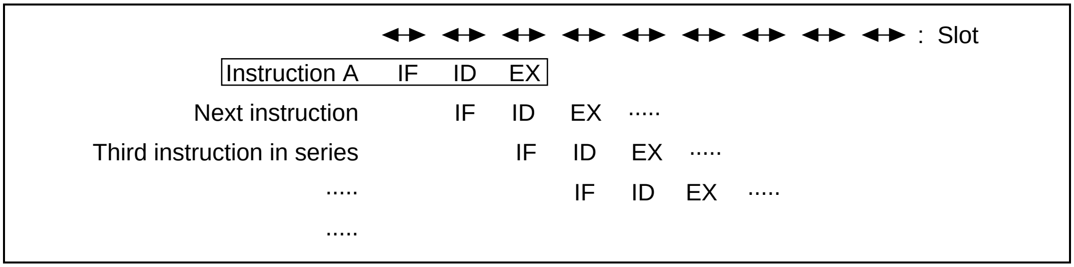

: Slot
Instruction A IF ID EX
Next instruction IF ID EX .....
Third instruction in series IF ID EX .....
..... IF ID EX .....
.....

Figure 7.81 Branch Instruction when Condition Is Not Satisfied

Unconditional Branch Instructions (Common, or SH-2 CPU, SH-DSP): Include the following instruction types:

- BRA label
- BRAF Rm (SH-2, SH-DSP CPU)
- BSR label
- BSRF Rm (SH-2, SH-DSP CPU)
- JMP @Rm
- JSR @Rm
- RTS

```
                         ◄► ◄► ◄► ◄► ◄► ◄► ◄► ◄► ◄►  :  Slot
       Instruction A     IF  ID  EX
          Delay slot         IF  —   ID  EX  MA  WB
  Branch destination                 IF  ID  EX  .....
               .....                     IF  ID  EX  .....
               .....
```

Figure 7.82 Unconditional Branch Instruction Pipeline

The pipeline has three stages: IF, ID, and EX (figure 7.82). 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 conditional branch instructions are, but is instead 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.

7.4.6 System Control Instructions

System Control ALU Instructions (Common, or SH-DSP): Include the following instruction types:

- CLRT
- LDC Rm,SR
- LDC Rm,GBR
- LDC Rm,VBR
- LDC Rm,MOD (SH-DSP)
- LDC Rm,RE (SH-DSP)
- LDC Rm,RS (SH-DSP)
- LDRE @(disp,PC)
- LDRS @(disp,PC)
- LDS Rm,PR
- NOP
- SETRC Rm (SH-DSP)
- SETRC #imm (SH-DSP)
- SETT
- STC SR,Rn
- STC GBR,Rn
- STC VBR,Rn
- STC MOD,Rn (SH-DSP)
- STC RE,Rn (SH-DSP)
- STC RS,Rn (SH-DSP)
- STS PR,Rn

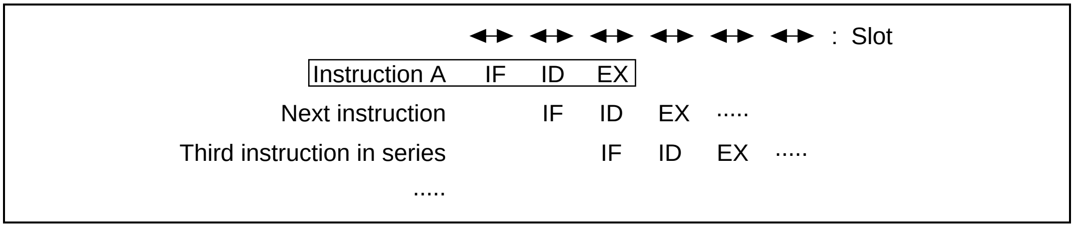

: Slot
Instruction A IF ID EX
Next instruction IF ID EX .....
Third instruction in series IF ID EX .....
.....

Figure 7.83 System Control ALU Instruction Pipeline

The pipeline has three stages: IF, ID, and EX (figure 7.83). The data operation is completed in the EX stage via the ALU.

LDC.L Instructions (Common, or SH-DSP): Include the following instruction types:

- LDC.L @Rm+, SR
- LDC.L @Rm+, GBR
- LDC.L @Rm+, VBR
- LDC.L @Rm+, MOD (SH-DSP)
- LDC.L @Rm+, RE (SH-DSP)
- LDC.L @Rm+, RS (SH-DSP)

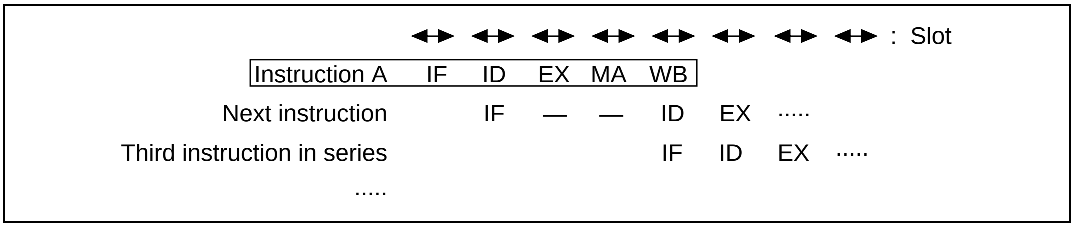

: Slot
Instruction A IF ID EX MA WB
Next instruction IF — — ID EX .....
Third instruction in series IF ID EX .....
.....

Figure 7.84 LDC.L Instruction Pipeline

The pipeline has five stages: IF, ID, EX, MA, and EX (figure 7.84). The ID of the following instruction is stalled two slots.

STC.L Instructions (Common, or SH-DSP): Include the following instruction types:

- STC.L SR, @–Rn
- STC.L GBR, @–Rn
- STC.L VBR, @–Rn
- STC.L MOD, @–Rn (SH-DSP)
- STC.L RE, @–Rn (SH-DSP)
- STC.L RS, @–Rn (SH-DSP)

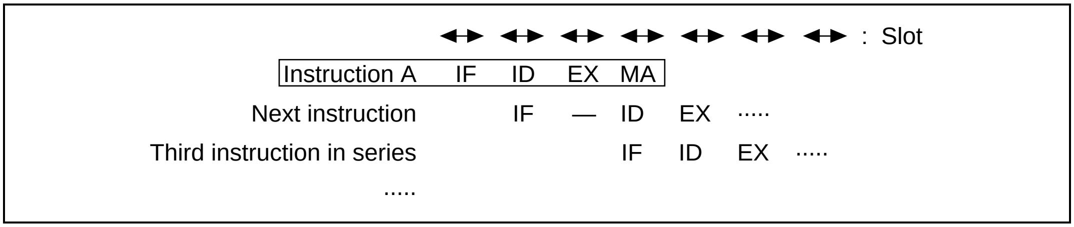

: Slot
Instruction A IF ID EX MA
Next instruction IF — ID EX .....
Third instruction in series IF ID EX .....
.....

Figure 7.85 STC.L Instruction Pipeline

The pipeline has four stages: IF, ID, EX, and MA (figure 7.85). The ID of the next instruction is stalled one slot.

LDS.L Instruction (Common): Includes the following instruction type:

- LDS.L @Rm+, PR

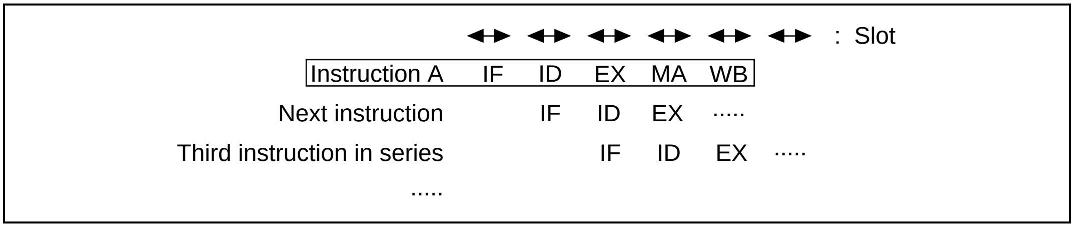

: Slot
Instruction A IF ID EX MA WB
Next instruction IF ID EX .....
Third instruction in series IF ID EX .....
.....

Figure 7.86 LDS.L Instructions (PR) Pipeline

The pipeline has five stages: IF, ID, EX, MA, and WB (figure 7.86). It is the same as an ordinary load instruction.

STS.L Instruction (Common): Includes the following instruction type:

- STS.L PR, @–Rn

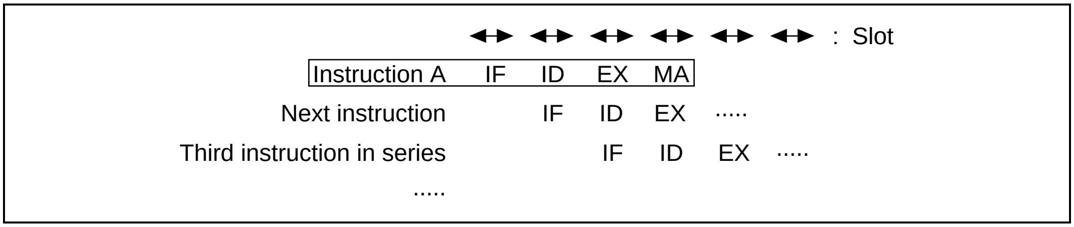

: Slot
Instruction A IF ID EX MA
Next instruction IF ID EX .....
Third instruction in series IF ID EX .....
.....

Figure 7.87 STS.L Instruction (PR) Pipeline

The pipeline has four stages: IF, ID, EX, and MA (figure 7.87). It is the same as an ordinary load instruction.

Register → MAC Transfer Instructions (Common, or SH-DSP): Include the following instruction types:

- CLRMAC
- LDS Rm, MACH
- LDS Rm, MACL
- LDS Rm,DSR (SH-DSP)
- LDS Rm,A0 (SH-DSP)
- LDS Rm,X0 (SH-DSP)
- LDS Rm,X1 (SH-DSP)
- LDS Rm,Y0 (SH-DSP)
- LDS Rm,Y1 (SH-DSP)

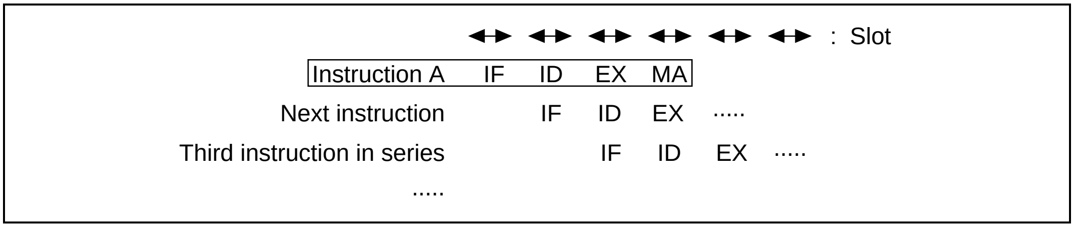

: Slot
Instruction A IF ID EX MA
Next instruction IF ID EX .....
Third instruction in series IF ID EX .....
.....

Figure 7.88 Register → MAC Transfer Instruction Pipeline

The pipeline has four stages: IF, ID, EX, and MA (figure 7.88). MA is a stage for accessing the multiplier. MA contends with IF. This makes it the same as ordinary store instructions. Since the multiplier does contend with the MA, however, the items noted for the multiplication, Multiply/Accumulate, double-length multiplication, and double-length multiply/accumulate instructions apply.

Memory → MAC Transfer Instructions (Common, or SH-DSP): Include the following instruction types:

- LDS.L @Rm+, MACH
- LDS.L @Rm+, MACL
- LDS.L @Rm+,DSR (SH-DSP)
- LDS.L @Rm+,A0 (SH-DSP)
- LDS.L @Rm+,X0 (SH-DSP)
- LDS.L @Rm+,X1 (SH-DSP)
- LDS.L @Rm+,Y0 (SH-DSP)
- LDS.L @Rm+,Y1 (SH-DSP)

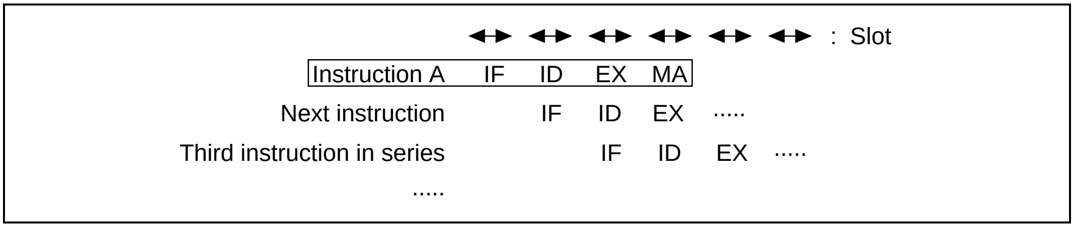

: Slot
Instruction A IF ID EX MA
Next instruction IF ID EX .....
Third instruction in series IF ID EX .....
.....

Figure 7.89 Memory → MAC Transfer Instruction Pipeline

The pipeline has four stages: IF, ID, EX, and MA (figure 7.89). MA contends with IF. MA is a stage for memory access and multiplier access. This makes it the same as ordinary load instructions. Since the multiplier does contend with the MA, however, the items noted for the multiplication, Multiply/Accumulate, double-length multiplication, and double-length multiply/accumulate instructions apply.

MAC → Register Transfer Instructions (Common, or SH-DSP): Include the following instruction types:

- STS MACH, Rn
- STS MACL, Rn
- STS DSR,Rn
- STS A0,Rn
- STS X0,Rn
- STS X1,Rn
- STS Y0,Rn
- STS Y1,Rn

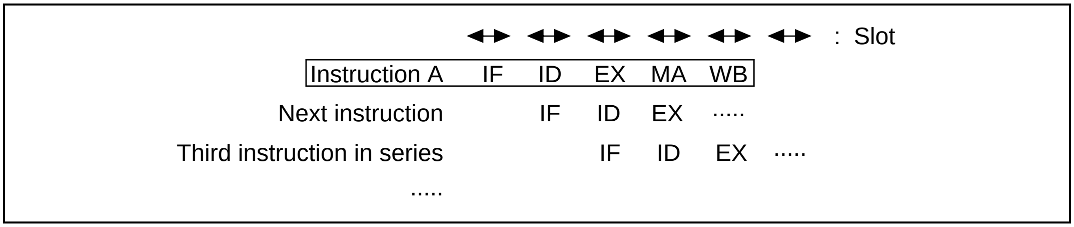


Figure 7.90 MAC → Register Transfer Instruction Pipeline

The pipeline has five stages: IF, ID, EX, MA, and WB (figure 7.90). MA is a stage for accessing the multiplier. MA contends with IF. This makes it the same as ordinary load instructions. Since the multiplier does contend with the MA, however, the items noted for the multiplication, Multiply/Accumulate, double-length multiplication, and double-length multiply/accumulate instructions apply.

MAC → Memory Transfer Instructions (Common, or SH-DSP): Include the following instruction types:

- STS.L MACH, @–Rn
- STS.L MACL, @–Rn
- STS.L DSR,@–Rn (SH-DSP)
- STS.L A0,@–Rn (SH-DSP)
- STS.L X0,@–Rn (SH-DSP)
- STS.L X1,@–Rn (SH-DSP)
- STS.L Y0,@–Rn (SH-DSP)
- STS.L Y1,@–Rn (SH-DSP)

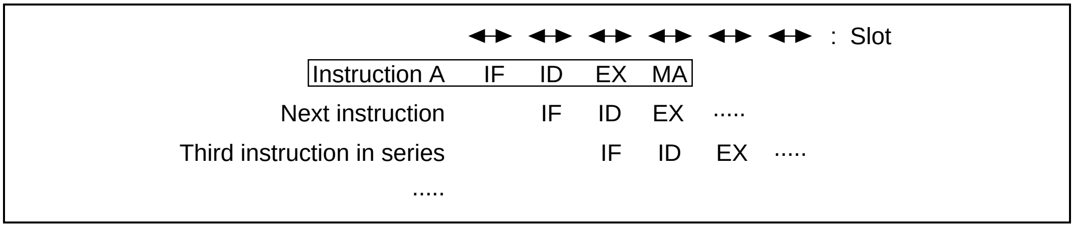

: Slot
Instruction A IF ID EX MA
Next instruction IF ID EX .....
Third instruction in series IF ID EX .....
.....

Figure 7.91 MAC → Memory Transfer Instruction Pipeline

The pipeline has four stages: IF, ID, EX, and MA (figure 7.91). MA is a stage for accessing the memory and multiplier. MA contends with IF. This makes it the same as ordinary store instructions. Since the multiplier does contend with the MA, however, the items noted for the multiplication, Multiply/Accumulate, double-length multiplication, and double-length multiply/accumulate instructions apply.

RTE Instruction (Common): RTE

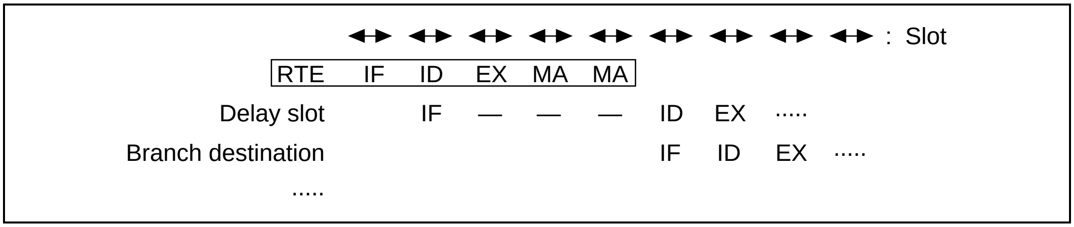

: Slot
RTE IF ID EX MA MA
Delay slot IF — — — ID EX .....
Branch destination IF ID EX .....
.....

Figure 7.92 RTE Instruction Pipeline

The pipeline has five stages: IF, ID, EX, MA, and MA (figure 7.92). The MAs do not contend with IF. RTE is a delayed branch instruction. The ID of the delay slot instruction is stalled 3 slots. The IF of the branch destination instruction starts from the slot following the MA of the RTE.

TRAP Instruction (Common): TRAPA #imm

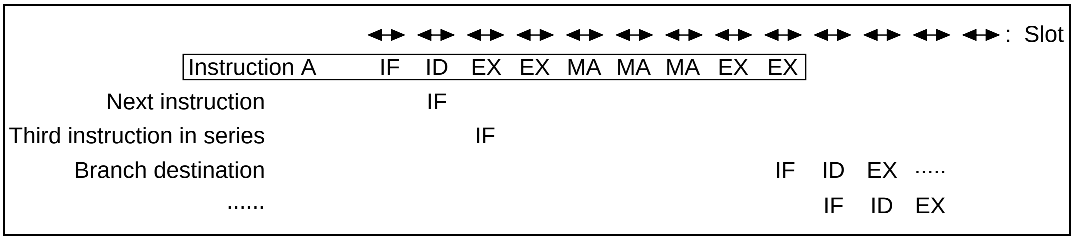

: Slot
Instruction A IF ID EX EX MA MA MA EX EX
Next instruction IF
Third instruction in series IF
Branch destination IF ID EX .....
...... IF ID EX

Figure 7.93 TRAP Instruction Pipeline

The pipeline has nine stages: IF, ID, EX, EX, MA, MA, MA, EX, and EX (figure 7.93). The MAs do not contend with IF. 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 slot of the EX in the ninth stage of the TRAP instruction.

SLEEP Instruction (Common): SLEEP

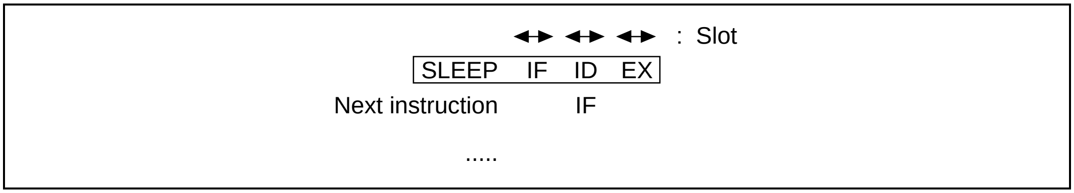


Figure 7.94 SLEEP Instruction Pipeline

The pipeline has three stages: IF, ID and EX (figure 7.94). It is issued until the IF of the next instruction. After the SLEEP instruction is executed, the CPU enters sleep mode or standby mode.

7.4.7 Exception Processing

Interrupt Exception Processing (Common): 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 ten stages: IF, ID, EX, EX, MA, MA, EX, MA, EX, and EX (figure 7.95). 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 that has the final EX in the interrupt exception processing.

Interrupt sources are external interrupt request pins such as NMI, user breaks, IRQ, and on-chip peripheral module interrupts.

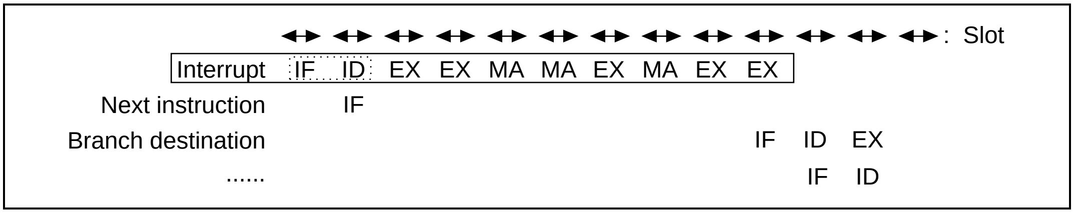


Figure 7.95 Interrupt Exception Processing Pipeline

Address Error Exception Processing: 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 ten stages: IF, ID, EX, EX, MA, MA, EX, MA, EX, and EX (figure 7.96). 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 that has the final EX in the address error exception processing.

Address errors are caused by instruction fetches and by data reads or writes. See the Hardware Manual for information on the causes of address errors.

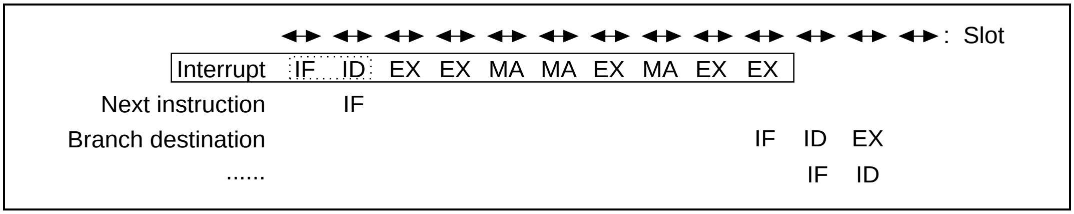
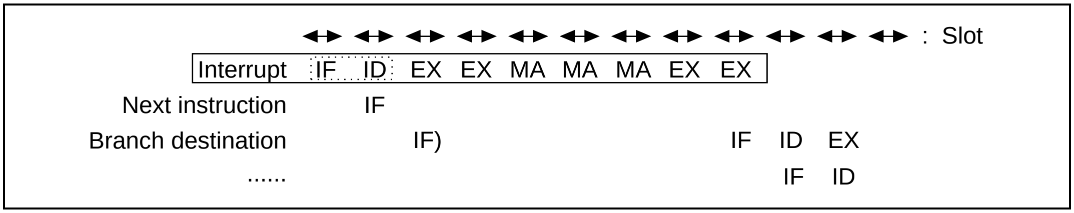


Figure 7.96 Address Error Exception Processing Pipeline

Illegal Instruction Exception Processing (Common): 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 nine stages: IF, ID, EX, EX, MA, MA, MA, EX, and EX (figure 7.97). Illegal instruction exception processing is not a delayed branch. In illegal instruction exception processing, overrun fetches (IF) occur. 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 that has 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.



: Slot
Interrupt IF ID EX EX MA MA MA EX EX
Next instruction IF
Branch destination IF) IF ID EX
...... IF ID

Figure 7.97 Illegal Instruction Exception Processing Pipeline

Appendix A CPU Instructions

A.1 CPU Instructions

Instructions executed by the CPU core are described in alphabetical order.

Table A.1 CPU Instructions in Alphabetical Order

Instruction		Operation	Code	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	Over-flow
`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 × 2 + PC → PC; if T = 1, nop	`10001011dddddddd`	3/1*[1]	—
`BF/S`	`label`	If T = 0, disp × 2 + PC → PC; if T = 1, nop	`10001111dddddddd`	2/1*[1]	—
`BRA`	`label`	Delayed branch, disp × 2 + PC → PC	`1010dddddddddddd`	2	—
`BRAF`	`Rm`	Delayed branch, Rm + PC → PC	`0000mmmm00100011`	2	—
`BSR`	`label`	Delayed branch, PC → PR, disp × 2 + PC → PC	`1011dddddddddddd`	2	—
`BSRF`	`Rm`	Delayed branch, PC → PR, Rm + PC → PC	`0000mmmm00000011`	2	—
`BT`	`label`	If T = 1, disp × 2 + PC → PC; if T = 0, nop	`10001001dddddddd`	3/1*[1]	—
`BT/S`	`label`	If T = 1, disp × 2 + PC → PC; if T = 0, nop	`10001101dddddddd`	2/1*[1]	—

Instruction		Operation	Code	Cycles	T Bit
CLRMAC		0 → MACH, MACL	0000000000101000	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, MACHL	0011nnnnmmmm1101	2 to 4*2	—
DMULU.L	Rm,Rn	Unsigned operation of Rn × Rm → MACH, MACL	0011nnnnmmmm0101	2 to 4*2	—

Instruction		Operation	Code	Cycles	T Bit
`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	—
`JMP`	`@Rm`	Delayed branch, Rm → PC	`0100mmmm00101011`	2	—
`JSR`	`@Rm`	Delayed branch, PC → PR, Rm → PC	`0100mmmm00001011`	2	—
`LDC`	`Rm,GBR`	Rm → GBR	`0100mmmm00011110`	1	—
`LDC`	`Rm,MOD`	Rm→MOD	`0100mmmm01011110`	1	—
`LDC`	`Rm,RE`	Rm→RE	`0100mmmm01111110`	1	—
`LDC`	`Rm,RS`	Rm→RS	`0100mmmm01101110`	1	—
`LDC`	`Rm,SR`	Rm→SR	`0100mmmm00001110`	1	LSB
`LDC`	`Rm,VBR`	Rm→VBR	`0100mmmm00101110`	1	—
`LDC.L`	`@Rm+,GBR`	(Rm)→GBR,Rm+4→Rm	`0100mmmm00010111`	3	—
`LDC.L`	`@Rm+,MOD`	(Rm)→MOD,Rn+4→Rn	`0100mmmm01010111`	3	—
`LDC.L`	`@Rm+,RE`	(Rm)→RE,Rn+4→Rn	`0100mmmm01110111`	3	—
`LDC.L`	`@Rm+,RS`	(Rm)→RS,Rn+4→Rn	`0100mmmm01100111`	3	—
`LDC.L`	`@Rm+,SR`	(Rm)→SR,Rm+4→Rm	`0100mmmm00000111`	3	LSB
`LDC.L`	`@Rm+,VBR`	(Rm)→VBR,Rm+4→Rm	`0100mmmm00100111`	3	—
`LDRE`	`@(disp,PC)`	disp × 2 +PC→RE	`10001110dddddddd`	1	—
`LDRS`	`@(disp,PC)`	disp × 2 +PC→RS	`10001100dddddddd`	1	—
`LDS`	`Rm,A0`	Rm → A0	`0100mmmm01111010`	1	—
`LDS`	`Rm,DSR`	Rm → DSR	`0100mmmm01101010`	1	—
`LDS`	`Rm,MACH`	Rm → MACH	`0100mmmm00001010`	1	—
`LDS`	`Rm,MACL`	Rm → MACL	`0100mmmm00011010`	1	—
`LDS`	`Rm,PR`	Rm → PR	`0100mmmm00101010`	1	—

Instruction		Operation	Code	Cycles	T Bit
`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+,A0`	(Rm) → A0, Rm + 4 → Rm	`0100mmmm01110110`	1	—
`LDS.L`	`@Rm+,DSR`	(Rm) → DSR, Rm + 4 → Rm	`0100mmmm01100110`	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+,X0`	(Rm)→X0,Rm+4→Rm	`0100mmmm10000110`	1	—
`LDS.L`	`@Rm+,X1`	(Rm)→X1,Rm+4→Rm	`0100mmmm10010110`	1	—
`LDS.L`	`@Rm+,Y0`	(Rm)→Y0,Rm+4→Rm	`0100mmmm10100110`	1	—
`LDS.L`	`@Rm+,Y1`	(Rm)→Y1,Rm+4→Rm	`0100mmmm10110110`	1	—
`MAC.L`	`@Rm+,@Rn+`	Signed operation of (Rn) × (Rm) + MAC → MAC	`0000nnnnmmmm1111`	3 (2 to 4)*2	—
`MAC.W`	`@Rm+,@Rn+`	Signed operation of (Rn) × (Rm) + MAC → MAC	`0100nnnnmmmm1111`	3/(2)*2	—
`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	—

Instruction		Operation	Code	Cycles	T Bit
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	—
MOV.L	@(disp,GBR), R0	(disp × 4 + GBR) → R0	11000110dddddddd	1	—
MOV.L	@(disp,PC), Rn	(disp × 4 + PC) → Rn	1101nnnndddddddd	1	—
MOV.L	@(disp,Rm), Rn	(disp × 4 + 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 × 4 + GBR)	11000010dddddddd	1	—
MOV.L	Rm,@(disp, Rn)	Rm → (disp × 4 + 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 × 2 + GBR) → Sign extension → R0	11000101dddddddd	1	—
MOV.W	@(disp,PC), Rn	(disp × 2 + PC) → Sign extension → Rn	1001nnnndddddddd	1	—
MOV.W	@(disp,Rm), R0	(disp × 2 + 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	—

Instruction		Operation	Code	Cycles	T Bit
MOV.W	@Rm,Rn	(Rm) → Sign extension → Rn	0110nnnnmmmm0001	1	—
MOV.W	R0,@(disp, GBR)	R0 → (disp × 2 + GBR)	11000001dddddddd	1	—
MOV.W	R0,@(disp, Rn)	R0 → (disp × 2 + 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 × 4 + PC → R0	11000111dddddddd	1	—
MOVT	Rn	T → Rn	0000nnnn00101001	1	—
MUL.L	Rm,Rn	Rn × Rm → MACL	0000nnnnmmmm0111	2 to 4*2	—
MULS.W	Rm,Rn	Signed operation of Rn × Rm → MAC	0010nnnnmmmm1111	1 to 3*2	—
MULU.W	Rm,Rn	Unsigned operation of 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	Bor-row
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	—
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, stack area→PC/SR	0000000000101011	4	LSB

Instruction		Operation	Code	Cycles	T Bit
RTS		Delayed branch, PR → PC	0000000000001011	2	—
SETRC	#imm	imm → RC (SR[23:16]), 0 → SR[27:24]	10000010iiiiiiii	1	—
SETRC	Rm	Rm [11:0]), 0 → RC(SR[27:16])	0100mmmm00010100	1	—
SETT		1 → T	0000000000011000	1	1
SHAL	Rn	T ← Rn ← 0	0100nnnn00100000	1	MSB
SHAR	Rn	MSB → Rn → T	0100nnnn00100001	1	LSB
SHLL	Rn	T ← Rn ← 0	0100nnnn00000000	1	MSB
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	3	—
STC	GBR,Rn	GBR → Rn	0000nnnn00010010	1	—
STC	MOD,Rn	MOD → Rn	0000nnnn01010010	1	—
STC	RE,Rn	RE → Rn	0000nnnn01110010	1	—
STC	RS,Rn	RS → Rn	0000nnnn01100010	1	—
STC	SR,Rn	SR → Rn	0000nnnn00000010	1	—
STC	VBR,Rn	VBR → Rn	0000nnnn00100010	1	—
STC.L	GBR,@–Rn	Rn–4 → Rn, GBR → (Rn)	0100nnnn00010011	2	—
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	—

Instruction		Operation	Code	Cycles	T Bit
STC.L	SR,@–Rn	Rn–4 → Rn, SR → (Rn)	0100nnnn00000011	2	—
STC.L	VBR,@–Rn	Rn–4 → Rn, VBR → (Rn)	0100nnnn00100011	2	—
STS	A0,Rn	A0 → Rn	0000nnnn01111010	1	—
STS	DSR,Rn	DSR → Rn	0000nnnn01101010	1	—
STS	MACH,Rn	MACH → Rn	0000nnnn00001010	1	—
STS	MACL,Rn	MACL → Rn	0000nnnn00011010	1	—
STS	PR,Rn	PR → Rn	0000nnnn00101010	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	A0,@–Rn	Rn–4 → Rn, A0 → (Rn)	0100nnnn01110010	1	—
STS.L	DSR,@–Rn	Rn–4 → Rn, DSR → (Rn)	0100nnnn01100010	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, R → (Rn)	0100nnnn00100010	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	—
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

Instruction		Operation	Code	Cycles	T Bit
`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`	4	Test result
`TRAPA`	`#imm`	PC/SR → Stack area, (imm × 4 + VBR) → PC	`11000011iiiiiiii`	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.

Added CPU Instructions: Table A.2 shows the CPU instructions in the SH-DSP added since the SH-2 (3 types, 24 instructions). Table A.3 shows the CPU instructions in the SH-2 added since the SH-1 (6 types, 9 instructions).

Table A.2 CPU Instructions in the SH-DSP Added since the SH-2

Instruction		Operation	Code	Cycles	T Bit
`LDC`	`Rm,MOD`	Rm → MOD	`0100mmmm01011110`	1	—
`LDC`	`Rm,RE`	Rm → RE	`0100mmmm01111110`	1	—
`LDC`	`Rm,RS`	Rm → RS	`0100mmmm01101110`	1	—
`LDC.L`	`@Rm+,MOD`	(Rm) → MOD, Rm + 4 → Rm	`0100mmmm01010111`	3	—
`LDC.L`	`@Rm+,RE`	(Rm) → RE, Rm + 4 → Rm	`0100mmmm01110111`	3	—
`LDC.L`	`@Rm+,RS`	(Rm) → RS, Rm + 4 → Rm	`0100mmmm01100111`	3	—
`LDRE`	`@(disp,PC)`	disp × 2 + PC → RE	`10001110dddddddd`	1	—
`LDRS`	`@(disp,PC)`	disp × 2 + PC → RS	`10001100dddddddd`	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+,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	—
`SETRC`	`Rm`	Rm[11:0] → RC (SR[27:16])	`0100nnnn00010100`	1	—
`SETRC`	`#imm`	imm → RC (SR [23:16]), zeros → SR[27:24]	`10000010iiiiiiii`	1	—
`STC`	`MOD,Rn`	MOD → Rn	`0000nnnn01010010`	1	—
`STC`	`RE,Rn`	RE → Rn	`0000nnnn01110010`	1	—
`STC`	`RS,Rn`	RS → Rn	`0000nnnn01100010`	1	—

Instruction	Operation	Code	Cycles	T Bit
`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	—
`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  DSR,@–Rn`	Rn–4 → Rn, DSR → (Rn)	`0100nnnn01100010`	1	—
`STS.L  A0,@–Rn`	Rn–4 → Rn, A0 → (Rn)	`0100nnnn01110010`	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	—

Table A.3 CPU Instructions in the SH-2 Added since the SH-1

Instruction		Operation	Code	Cycles	T Bit
`BF/S`	`label`	When T = 0, disp × 2 + PC → PC; When T = 1, nop	`10001111dddddddd`	2/1	—
`BRAF`	`Rm`	Delayed branch, Rm + PC → PC	`0000mmmm00100011`	2	—
`BSRF`	`Rm`	Delayed branch, PC → PR, Rm + PC → PC	`0000mmmm00000011`	2	—
`BT/S`	`label`	When T = 1, disp × 2 + PC → PC; When T = 0, nop	`10001101dddddddd`	2/1	—
`DMULS.L`	`Rm,Rn`	Signed Rn x Rm → MACH, MACL 32 × 32 → 64 bits	`0011nnnnmmmm1101`	2 (to 4)	—
`DMULU.L`	`Rm,Rn`	Unsigned Rn x Rm → MACH, MACL 32 × 32 → 64 bits	`0011nnnnmmmm0101`	2 (to 4)	—
`DT`	`Rn`	Rn - 1 → Rn, When Rn is 0, 1 → T, when Rn is nonzero, 0 → T	`0100nnnn00010000`	1	Compa-rison result
`MAC.L`	`@Rm+,@Rn+`	Signed (Rn) × (Rm) + MAC → MAC	`0000nnnnmmmm1111`	2 (to 4)	—
`MUL.L`	`Rm,Rn`	Rn × Rm → MACL	`0000nnnnmmmm0111`	2 (to 4)	—

Renesas 32-Bit RISC Microcomputer
Software Manual
SH-1/SH-2/SH-DSP

Publication Date:	1st Edition, September 1994
	Rev.5.00, June 30, 2004
Published by:	Sales Strategic Planning Div.
	Renesas Technology Corp.
Edited by:	Technical Documentation & Information Department
	Renesas Kodaira Semiconductor Co., Ltd.

Renesas Technology Corp. Sales Strategic Planning Div. Nippon Bldg., 2-6-2, Ohte-machi, Chiyoda-ku, Tokyo 100-0004, Japan

RENESAS SALES OFFICES

http://www.renesas.com

Renesas Technology America, Inc.
450 Holger Way, San Jose, CA 95134-1368, U.S.A
Tel: <1> (408) 382-7500 Fax: <1> (408) 382-7501

Renesas Technology Europe Limited.
Dukes Meadow, Millboard Road, Bourne End, Buckinghamshire, SL8 5FH, United Kingdom
Tel: <44> (1628) 585 100, Fax: <44> (1628) 585 900

Renesas Technology Europe GmbH
Dornacher Str. 3, D-85622 Feldkirchen, Germany
Tel: <49> (89) 380 70 0, Fax: <49> (89) 929 30 11

Renesas Technology Hong Kong Ltd.
7/F., North Tower, World Finance Centre, Harbour City, Canton Road, Hong Kong
Tel: <852> 2265-6688, Fax: <852> 2375-6836

Renesas Technology Taiwan Co., Ltd.
FL 10, #99, Fu-Hsing N. Rd., Taipei, Taiwan
Tel: <886> (2) 2715-2888, Fax: <886> (2) 2713-2999

Renesas Technology (Shanghai) Co., Ltd.
26/F., Ruijin Building, No.205 Maoming Road (S), Shanghai 200020, China
Tel: <86> (21) 6472-1001, Fax: <86> (21) 6415-2952

Renesas Technology Singapore Pte. Ltd.
1, Harbour Front Avenue, #06-10, Keppel Bay Tower, Singapore 098632
Tel: <65> 6213-0200, Fax: <65> 6278-8001

SH-1/SH-2/SH-DSP
Software Manual

Renesas Technology Corp.
2-6-2, Ote-machi, Chiyoda-ku, Tokyo, 100-0004, Japan