SH7000/7600 Series

Super H RISC Engine

Programming Manual

HITACHI®

Hitachi
Single-Chip RISC Microcomputer
SH7000 and SH7600 Series

Programming Manual

Introduction

The SH7000 and SH7600 series are new-generation RISC (Reduced instruction set computer) microcomputers that integrate a RISC-type CPU and the peripheral functions required for system configuration onto a single chip to achieve high-performance operation. It can operate in a power-down state, which is an essential feature for portable equipment.

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

This programming manual describes in detail the instructions for the SH7000 and SH7600 series and is intended as a reference on instruction operation and architecture. It also covers the pipeline operation, which is a feature of the SH7000 and SH7600 series. For information on the hardware, refer to the hardware manual for the product in question.

Related Manuals

- *SH7032, SH7034 Hardware Manual* (Document No. ADE-602-062).
- *SH7020, SH7021 Hardware Manual* (Document No. ADE-602-074)
- *SH7604 Hardware Manual*

For development support tools, contact your Hitachi sales office.

Organization of This Manual

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

Table 1 Manual Organization

Category	Section Title	Contents
Introduction	1. Features	CPU features
Architecture (1)	2. Register Configuration	Types and configuration of general registers, control registers and system registers
	3. Data Formats	Data formats for registers and memory
Introduction to instructions	4. Instruction Features	Instruction features, addressing modes, and instruction formats
	5. Instruction Sets	Summary of instructions by category and list in alphabetic order
Detailed information on instructions	6. Instruction Descriptions	Operation of each instruction in alphabetical order
Architecture (2)	7. Processing States	Power-down and other processing states
	8. Pipeline Operation	Pipeline flow, and pipeline flows with operation for each instruction
Instruction code	Appendixes: Instruction Code	Operation code map

Table 2 Subjects and Corresponding Sections

Category	Topic	Section Title
Introduction and features	CPU features	1. Features
	Instruction features	4.1 RISC-Type Instruction Set
	Pipelines	8.1 Basic Configuration of Pipelines 8.2 Slot and Pipeline Flow
Architecture	Register configuration	2. Register Configuration
	Data formats	3. Data Formats
	Processing states, reset state, exception processing state, bus release state, program execution state, power-down state, sleep mode and standby mode	7. Processing States
	Pipeline operation	8. Pipeline Operation
Introduction to instructions	Instruction features	4. Instruction Features
	Addressing modes	4.2 Addressing Modes
	Instruction formats	4.3 Instruction Formats
List of instructions	Instruction sets	5.1 Instruction Set by Classification 5.2 Instruction Set in Alphabetical Order Appendix A.1 Instruction Set by Addressing Mode Appendix A.2 Instruction Set by Instruction Format
	Instruction code	Appendix A.3 Instruction Set in Order by Instruction Code Appendix A.4 Operation Code Map
Detailed information on instructions	Detailed information on instruction operation	6. Instruction Description 8.7 Instruction Pipeline Operations
	Number of instruction execution states	8.3 Number of Instruction Execution States

Functions Listed by CPU Type

This manual is common for both the SH7000 and SH7600 series. However, not all CPUs can use all the instructions and functions. Table 3 lists the usable functions by CPU type.

Table 3 Functions by CPU Type

Item		SH7000 Series	SH7600 Series
Instructions	BF/S	No	Yes
	BRAF	No	Yes
	BSRF	No	Yes
	BT/S	No	Yes
	DMULS.L	No	Yes
	DMULU.L	No	Yes
	DT	No	Yes
	MAC.L	No	Yes
	MAC.W*1 (MAC)*2	16 x 16 + 42 → 42	16 x 16 + 64 → 64
	MUL.L	No	Yes
	All others	Yes	Yes
States for multiplication operation	16 x 16 → 32 (MULS.W, MULU.W)*2	Executed in 1–3*3 states	Executed in 1–3*3 states
	32 x 32 → 32 (MUL.L)	No	Executed in 2–4 *3 states
	32 x 32 → 64 (DMULS.L, DMULU.L)	No	Executed in 2–4 *3 states
States for multiply and accumulate operation	16 x 16 + 42 → 42 (SH7000, MAC.W)	Executed in 3/(2)*3 states	No
	16 x 16 + 64 → 64 (SH7600, MAC.W)	No	Executed in states 3/(2)*3
	32 x 32 + 64 → 64 (MAC.L)	No	Executed in 2–4 states 3/(2~4)*3
Processing status	Module stop mode	No	Yes (Supply of clock to specified module can be halted)

Notes: 1. MAC.W works differently on different LSIs.

2. MAC and MAC.W are the same. MULS is also the same as MULS.W and MULU the same as MULU.W.

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

Contents

Section 1 Features

The SH7000 and SH7600 series 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 SH7000 and SH7600-series CPU features.

Table 1.1 SH7000 and SH7600-Series CPU Features

Item	Feature
Architecture	• Original Hitachi architecture • 32-bit internal data paths
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 (SH7000)	• 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 (SH7600)	• 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
Processing states	• Reset state • Exception processing state • Program execution state • Power-down state • Bus release state
Power-down states	• Sleep mode • Standby mode • Module stop mode (SH7600 only)

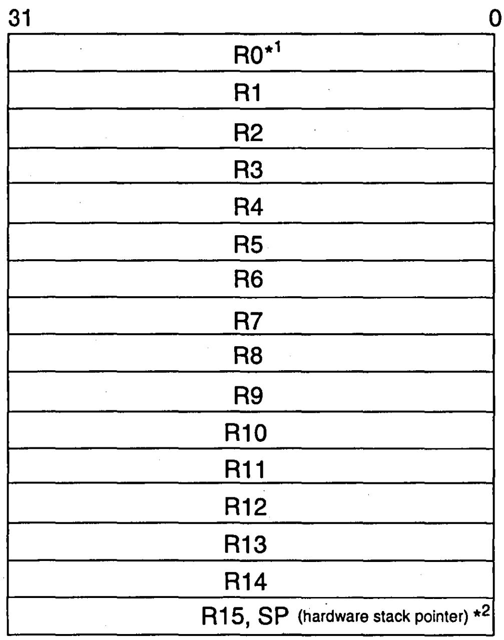
Note: The normal minimum number of execution cycles (The number in parentheses in the mumber in contention with preceding/following instructions).

Section 2 Register Configuration

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

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.



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

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 2.2). 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).

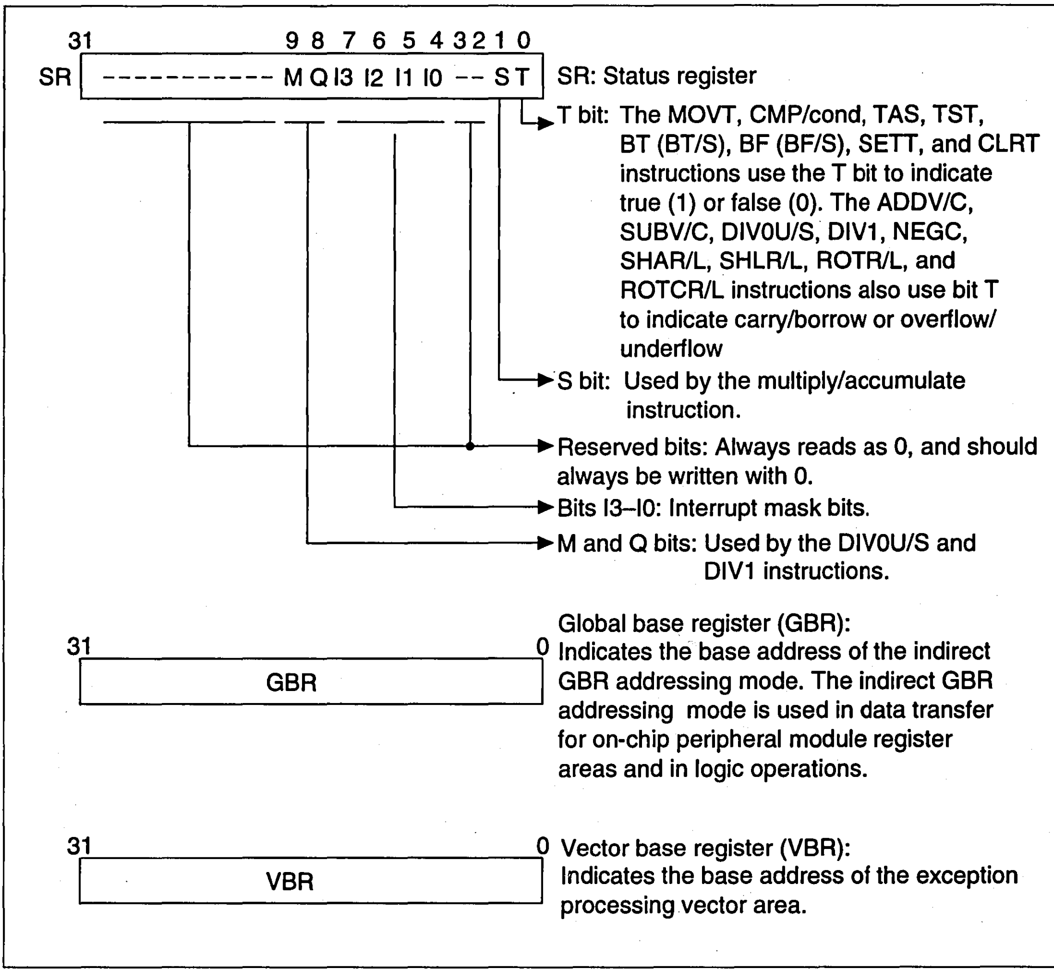
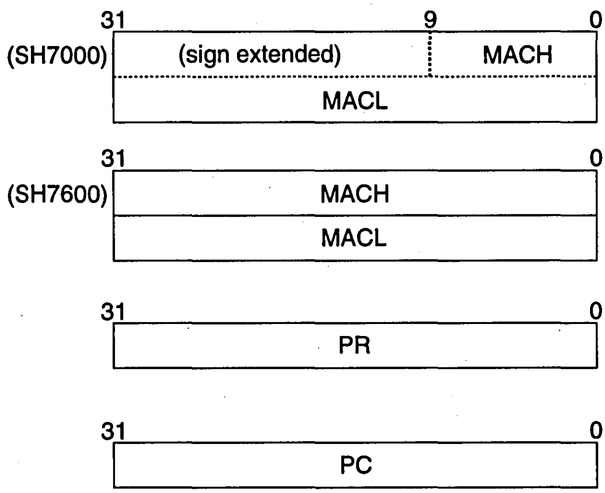


Figure 2.2 Control Registers

2.3 System Registers

The 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) (figure 2.3). The multiply and accumulate registers store the results of multiply and accumulate operations. The procedure register stores the return address from the subroutine procedure. The program counter stores program addresses to control the flow of the processing.



Multiply and accumulate (MAC) registers high and low (MACH/L): Store the results of multiply and accumulate operations. In the SH7000, MACH is sign-extended to 32 bits when read because only the lowest 10 bits are valid. In the SH7600, all 32 bits of MACH are valid.

Procedure register (PR): Stores a return address from a subroutine procedure.

Program counter (PC): Indicates the fourth byte (second instruction) after the current instruction.

Figure 2.3 System Registers

2.4 Initial Values of Registers

Table 2.1 lists the values of the registers after reset.

Table 2.1 Initial Values of Registers

Classification	Register	Initial Value
General register	R0–R14	Undefined
	R15 (SP)	Value of the stack pointer in the vector address table
Control register	SR	Bits I3–I0 are 1111 (H'F), reserved bits are 0, and other bits are undefined
	GBR	Undefined
	VBR	H'00000000
System register	MACH, MACL, PR	Undefined
	PC	Value of the program counter in the vector address table

Section 3 Data Formats

3.1 Data Format in Registers

Register operands are always longwords (32 bits) (figure 3.1). When the memory operand is only a byte (8 bits) or a word (16 bits), it is sign-extended into a longword when loaded into a register.

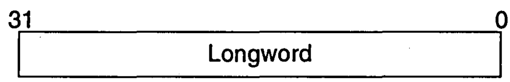


Figure 3.1 Longword Operand

3.2 Data Format in Memory

Memory data formats are classified into bytes, words, and longwords. 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 (figure 3.2). 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 holds the program counter and status register. See the *SH Hardware Manual* for more information on address errors.

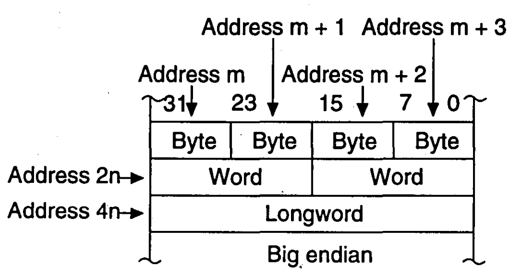


Figure 3.2 Byte, Word, and Longword Alignment

SH7604 has a function that allows access of CS2 space (area 2) in little endian format, which enables memory to be shared with processors that access memory in little endian format (figure 3.3). Byte data is arranged differently for little endian and the usual big endian.

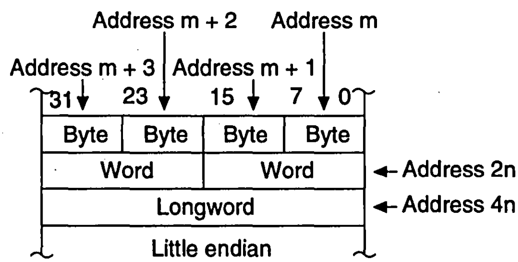


Figure 3.3 Byte, Word, and Longword Alignment in little endian format (SH7604 only)

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 calculated with registers and longword data. Immediate data accessed by the TST, AND, OR, and XOR instructions is zero-extended and calculated with 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. Rather, it 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. Specific examples are given in section 4.1.8, Immediate Data.

Section 4 Instruction Features

4.1 RISC-Type Instruction Set

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

4.1.1 16-Bit Fixed Length

All instructions are 16 bits long, increasing program coding efficiency.

4.1.2 One Instruction/Cycle

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

4.1.3 Data Length

Longword is the standard data length for all operations. Memory can be accessed in bytes, words, or longwords. Byte or word data accessed from memory is sign-extended and 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

SH7000/SH7600-Series CPU		Description	Example for Other CPU	
`MOV.W`	`@(disp,PC),R1`	Data is sign-extended to 32 bits, and R1 becomes H'00001234. It is next operated upon by an ADD instruction.	`ADD.W`	`#H'1234,R0`
`ADD`	`R1,R0`			
	`.........`			
`.DATA.W`	`H'1234`			

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

4.1.4 Load-Store Architecture

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

4.1.5 Delayed Branch Instructions

Unconditional branch instructions are delayed. Pipeline disruption during branching is reduced by first executing the instruction that follows the branch instruction, and then branching (table 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

SH7000/7600-Series CPU		Description	Example for Other CPU	
BRA	TRGET	Executes an ADD before branching to TRGET.	ADD.W	R1,R0
ADD	R1,R0		BRA	TRGET

4.1.6 Multiplication/Accumulation Operation

SH7000: 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.

SH7600: 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.

4.1.7 T Bit

The T bit in the status register changes according to the result of the comparison, and in turn is the condition (true/false) that determines if the program will branch (table 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

SH7000/7600-Series CPU		Description	Example for Other CPU	
CMP/GE	R1,R0	T bit is set when R0 ≥ R1. The program branches to TRGET0 when R0 ≥ R1 and to TRGET1 when R0 < R1.	CMP.W	R1,R0
BT	TRGET0		BGE	TRGET0
BF	TRGET1		BLT	TRGET1
ADD	#-1,R0	T bit is not changed by ADD. T bit is set when R0 = 0. The program branches if R0 = 0.	SUB.W	#1,R0
CMP/EQ	#0,R0		BEQ	TRGET
BT	TRGET			

4.1.8 Immediate Data

Byte immediate data is located in instruction code. Word or longword immediate data is not input via instruction codes but is stored in a memory table. The memory table is accessed by an immediate data transfer instruction (MOV) using the PC relative addressing mode with displacement (table 4.4).

Table 4.4 Immediate Data Accessing

Classification	SH7000/7600-Series 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).

4.1.9 Absolute Address

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

Table 4.5 Absolute Address

Classification	SH7000/7600 Series 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`

4.1.10 16-Bit/32-Bit Displacement

When data is accessed by 16-bit or 32-bit displacement, the pre-existing displacement value is placed in the memory table. Loading the immediate data when the instruction is executed transfers that value to the register and the data is accessed in the indirect indexed register addressing mode.

Table 4.6 Displacement Accessing

Classification	SH7000/7600 Series 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 and effective address calculation are described in table 4.7.

Table 4.7 Addressing Modes and Effective Addresses

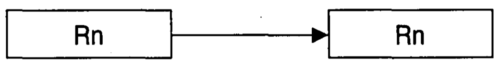
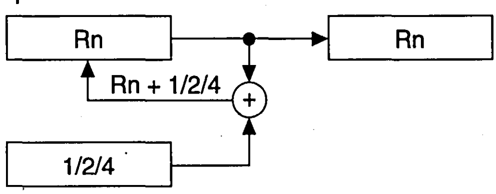
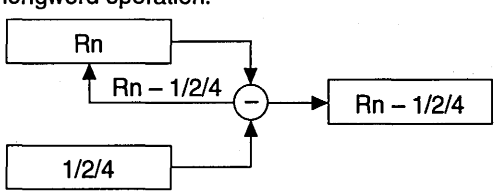
Addressing Mode	Instruction Format	Effective Addresses Calculation	Formula
Direct register addressing	Rn	The effective address is register Rn. (The operand is the contents of register Rn.)	—
Indirect register addressing	@Rn	The effective address is the content of register Rn. 	Rn
Post-increment indirect 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)

Table 4.7 Addressing Modes and Effective Addresses (cont)

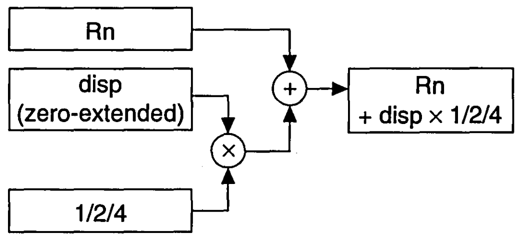
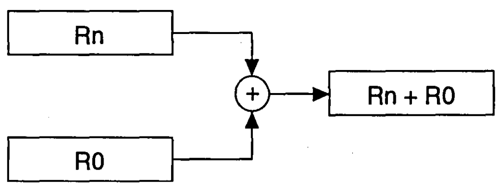
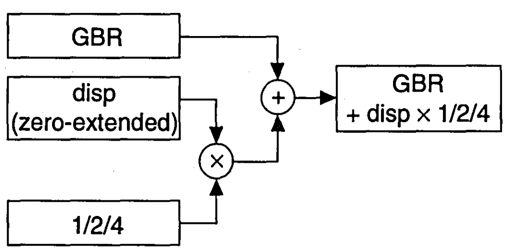
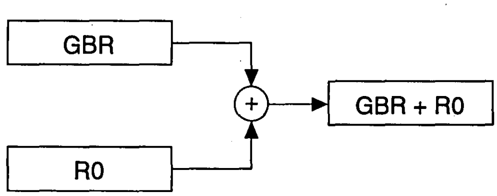
<table>
<tr><th>Addressing Mode</th><th>Instruction Format</th><th>Effective Addresses Calculation</th><th>Formula</th></tr>
<tr><td>Indirect register addressing with displacement</td><td>@(disp:4, Rn)</td><td>The effective address is Rn plus a 4-bit displacement (disp). The value of disp is zero-extended, and remains the same for a byte operation, is doubled for a word operation, or is quadrupled for a longword operation.

</td><td>Byte: Rn + disp
Word: Rn + disp × 2
Longword: Rn + disp × 4</td></tr>
<tr><td>Indirect indexed register addressing</td><td>@(R0, Rn)</td><td>The effective address is the Rn value plus R0.

</td><td>Rn + R0</td></tr>
<tr><td>Indirect GBR addressing with displacement</td><td>@(disp:8, GBR)</td><td>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.

</td><td>Byte: GBR + disp
Word: GBR + disp × 2
Longword: GBR + disp × 4</td></tr>
<tr><td>Indirect indexed GBR addressing</td><td>@(R0, GBR)</td><td>The effective address is the GBR value plus R0.

</td><td>GBR + R0</td></tr>
</table>

Table 4.7 Addressing Modes and Effective Addresses (cont)

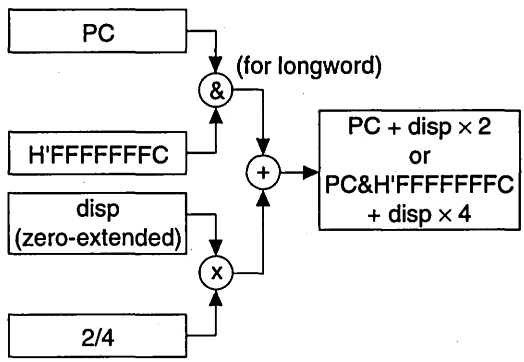
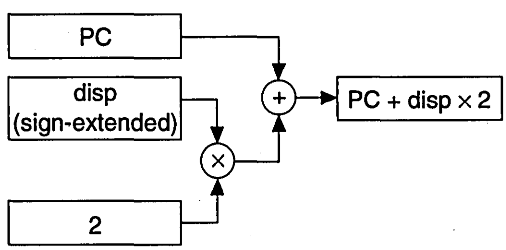
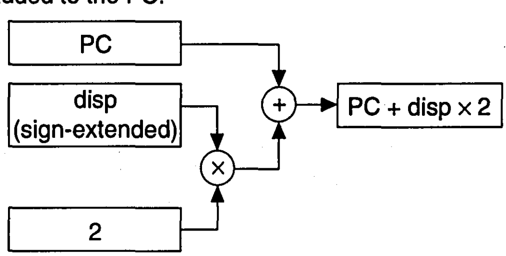
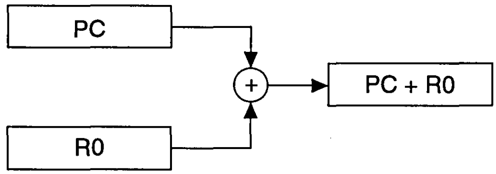
<table>
<tr><th>Addressing Mode</th><th>Instruction Format</th><th>Effective Addresses Calculation</th><th>Formula</th></tr>
<tr><td>PC relative addressing with displace-ment</td><td>@(disp:8, PC)</td><td>The effective address is the PC value plus an 8-bit displacement (disp). The value of disp is zero-extended, 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.

</td><td>Word: PC + disp × 2
Longword: PC & H'FFFFFFFC + disp × 4</td></tr>
<tr><td>PC relative addressing</td><td>disp:8</td><td>The effective address is the PC value sign-extended with an 8-bit displacement (disp), doubled, and added to the PC.

</td><td>PC + disp × 2</td></tr>
<tr><td></td><td>disp:12</td><td>The effective address is the PC value sign-extended with a 12-bit displacement (disp), doubled, and added to the PC.

</td><td>PC + disp × 2</td></tr>
</table>

Table 4.7 Addressing Modes and Effective Addresses (cont)

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.	—

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 15 0 xxxx xxxx xxxx xxxx	—	—	NOP
n format 15 0 xxxx | nnnn | xxxx xxxx	—	nnnn: Direct register	MOVT Rn
	Control register or system register	nnnn: Direct register	STS MACH,Rn

Table 4.8 Instruction Formats (cont)

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

Table 4.8 Instruction Formats (cont)

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

Section 5 Instruction Set

5.1 Instruction Set by Classification

Table 5.1 lists instructions by classification.

Table 5.1 Classification of Instructions

Classification	Types	Operation Code	Function	Applicable Instructions SH 7600	SH 7000	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	✔	✔	
		EXTU	Zero extension	✔	✔	
		MAC	Multiply/accumulate, double-length multiply/accumulate operation*1	✔	✔	
		MUL	Double-length multiplication	✔	✔	
		MULS	Signed multiplication	✔	✔	
		MULU	Unsigned multiplication	✔	✔	
		NEG	Negation	✔	✔	
		NEGC	Negation with borrow	✔	✔	
		SUB	Binary subtraction	✔	✔	
		SUBC	Binary subtraction with borrow	✔	✔	
		SUBV	Binary subtraction with underflow check	✔	✔	

Notes 1. Double-length multiply/accumulate is an SH7600 function.

Table 5.1 Classification of Instructions (cont)

Classification	Types	Operation Code	Function	Applicable Instructions SH 7600	SH 7000	No. of Instructions
Logic operations	6	AND	Logical AND	✔	✔	14
		NOT	Bit inversion	✔	✔	
		OR	Logical OR	✔	✔	
		TAS	Memory test and bit set	✔	✔	
		TST	Logical AND and T-bit set	✔	✔	
		XOR	Exclusive OR	✔	✔	
Shift	10	ROTL	One-bit left rotation	✔	✔	14
		ROTR	One-bit right rotation	✔	✔	
		ROTCL	One-bit left rotation with T bit	✔	✔	
		ROTCR	One-bit right rotation with T bit	✔	✔	
		SHAL	One-bit arithmetic left shift	✔	✔	
		SHAR	One-bit arithmetic right shift	✔	✔	
		SHLL	One-bit logical left shift	✔	✔	
		SHLLn	n-bit logical left shift	✔	✔	
		SHLR	One-bit logical right shift	✔	✔	
		SHLRn	n-bit logical right shift	✔	✔	
Branch	9	BF	Conditional branch, conditional branch with delay*2 (T = 0)	✔	✔	11
		BT	Conditional branch, conditional branch with delay*2 (T = 1)	✔	✔	
		BRA	Unconditional branch	✔	✔	
		BRAF	Unconditional branch	✔		
		BSR	Branch to subroutine procedure	✔	✔	
		BSRF	Branch to subroutine procedure	✔		
		JMP	Unconditional branch	✔	✔	
		JSR	Branch to subroutine procedure	✔	✔	
		RTS	Return from subroutine procedure	✔	✔	

Notes 2. Conditional branch with delay is an SH7600 function.

Table 5.1 Classification of Instructions (cont)

Classification	Types	Operation Code	Function	Applicable Instructions SH 7600	SH 7000	No. of Instructions
System control	11	CLRT	T-bit clear	✔	✔	31
		CLRMAC	MAC register clear	✔	✔	
		LDC	Load to control register	✔	✔	
		LDS	Load to system register	✔	✔	
		NOP	No operation	✔	✔	
		RTE	Return from exception processing	✔	✔	
		SETT	T-bit set	✔	✔	
		SLEEP	Shift into power-down mode	✔	✔	
		STC	Storing control register data	✔	✔	
		STS	Storing system register data	✔	✔	
		TRAPA	Trap exception processing	✔	✔	
Total:	62					142

Instruction codes, operation, and execution states are listed in table 5.2 in order 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*
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 left/right shift
Execution cycle		Value when no wait states are inserted
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		Value of T bit after instruction is executed
—		No change

Note: Scaling (x1, x2, x4) is performed according to the instruction operand size. See "6. Instruction Descriptions" for details.

5.1.1 Data Transfer Instructions

Tables 5.3 to 5.8 list the minimum number of clock states required for execution.

Table 5.3 Data Transfer Instructions

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

Table 5.3 Data Transfer Instructions (cont)

Instruction		Instruction Code	Operation	Execution State	T Bit
MOV.L	Rm,@(R0,Rn)	0000nnnnmmmm0110	Rm → (R0 + Rn)	1	—
MOV.B	@(R0,Rm),Rn	0000nnnnmmmm1100	(R0 + Rm) → Sign extension → Rn	1	—
MOV.W	@(R0,Rm),Rn	0000nnnnmmmm1101	(R0 + Rm) → Sign extension → Rn	1	—
MOV.L	@(R0,Rm),Rn	0000nnnnmmmm1110	(R0 + Rm) → Rn	1	—
MOV.B	R0,@(disp,GBR)	11000000dddddddd	R0 → (disp + GBR)	1	—
MOV.W	R0,@(disp,GBR)	11000001dddddddd	R0 → (disp × 2 + GBR)	1	—
MOV.L	R0,@(disp,GBR)	11000010dddddddd	R0 → (disp × 4+ GBR)	1	—
MOV.B	@(disp,GBR),R0	11000100dddddddd	(disp + GBR) → Sign extension → R0	1	—
MOV.W	@(disp,GBR),R0	11000101dddddddd	(disp × 2 + GBR) → Sign extension → R0	1	—
MOV.L	@(disp,GBR),R0	11000110dddddddd	(disp × 4 + GBR) → R0	1	—
MOVA	@(disp,PC),R0	11000111dddddddd	disp × 4 + PC → R0	1	—
MOVT	Rn	0000nnnn00101001	T → Rn	1	—
SWAP.B	Rm,Rn	0110nnnnmmmm1000	Rm → Swap upper and lower 2 bytes→ Rn	1	—
SWAP.W	Rm,Rn	0110nnnnmmmm1001	Rm → Swap upper and lower word → Rn	1	—
XTRCT	Rm,Rn	0010nnnnmmmm1101	Center 32 bits of Rm and Rn→ Rn	1	—

5.1.2 Arithmetic Instructions

Table 5.4 Arithmetic Instructions

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

Table 5.4 Arithmetic Instructions (cont)

Instruction		Instruction Code	Operation	Execution State	T Bit
DMULS.L	Rm,Rn*2	0011nnnnmmmm1101	Signed operation of Rn x Rm → MACH, MACL 32 x 32 → 64 bits	2 to 4*1	—
DMULU.L	Rm,Rn*2	0011nnnnmmmm0101	Unsigned operation of Rn x Rm → MACH, MACL 32 x 32 → 64 bits	2 to 4*1	—
DT	Rn*2	0100nnnn00010000	Rn - 1 → Rn, when Rn is 0, 1 → T. When Rn is nonzero, 0 → T	1	Comparison result
EXTS.B	Rm,Rn	0110nnnnmmmm1110	A byte in Rm is sign-extended → Rn	1	—
EXTS.W	Rm,Rn	0110nnnnmmmm1111	A word in Rm is sign-extended → Rn	1	—
EXTU.B	Rm,Rn	0110nnnnmmmm1100	A byte in Rm is zero-extended → Rn	1	—
EXTU.W	Rm,Rn	0110nnnnmmmm1101	A word in Rm is zero-extended → Rn	1	—
MAC.L	@Rm+,@Rn+ *2	0000nnnnmmmm1111	Signed operation of (Rn) x (Rm) + MAC → MAC 32 x 32 + 64 → 64 bits	3/(2 to 4)*1	—
MAC.W	@Rm+,@Rn+	0100nnnnmmmm1111	Signed operation of (Rn) × (Rm) + MAC → MAC (SH7600) 16 x 16 + 64 → 64 bits (SH7000) 16 x 16 + 42 → 42 bits	3/(2)*1	—
MUL.L	Rm,Rn*2	0000nnnnmmmm0111	Rn x Rm → MACL, 32 x 32 → 32 bits	2 to 4*1	—
MULS.W	Rm,Rn	0010nnnnmmmm1111	Signed operation of Rn × Rm → MAC 16 x 16 → 32 bits	1 to 3*1	—

Notes: 1. The normal minimum number of execution states (The number in parentheses is the number of states when there is contention with preceding/following instructions)

2. SH7600 instructions

Table 5.4 Arithmetic Instructions (cont)

Instruction		Instruction Code	Operation	Execution State	T Bit
MULU.W	Rm,Rn	0010nnnnmmmm1110	Unsigned operation of Rn × Rm → MAC 16 x 16 → 32 bits	1 to 3*1	—
NEG	Rm,Rn	0110nnnnmmmm1011	0–Rm → Rn	1	—
NEGC	Rm,Rn	0110nnnnmmmm1010	0–Rm–T → Rn, Borrow → T	1	Borrow
SUB	Rm,Rn	0011nnnnmmmm1000	Rn–Rm → Rn	1	—
SUBC	Rm,Rn	0011nnnnmmmm1010	Rn–Rm–T → Rn, Borrow → T	1	Borrow
SUBV	Rm,Rn	0011nnnnmmmm1011	Rn–Rm → Rn, Underflow → T	1	Underflow

Notes: 1. The normal minimum number of execution states (The number in parentheses is the number of states when there is contention with preceding/following instructions)

5.1.3 Logic Operation Instructions

Table 5.5 Logic Operation Instructions

Instruction		Instruction Code	Operation	Execution State	T Bit
AND	Rm,Rn	0010nnnnmmmm1001	Rn & Rm → Rn	1	—
AND	#imm,R0	11001001iiiiiiii	R0 & imm → R0	1	—
AND.B	#imm,@(R0,GBR)	11001101iiiiiiii	(R0 + GBR) & imm → (R0 + GBR)	3	—
NOT	Rm,Rn	0110nnnnmmmm0111	~Rm → Rn	1	—
OR	Rm,Rn	0010nnnnmmmm1011	Rn I Rm → Rn	1	—
OR	#imm,R0	11001011iiiiiiii	R0 I imm → R0	1	—
OR.B	#imm,@(R0,GBR)	11001111iiiiiiii	(R0 + GBR) I imm → (R0 + GBR)	3	—
TAS.B	@Rn	0100nnnn00011011	If (Rn) is 0, 1 → T; 1 → MSB of (Rn)	4	Test result
TST	Rm,Rn	0010nnnnmmmm1000	Rn & Rm; if the result is 0, 1 →T	1	Test result
TST	#imm,R0	11001000iiiiiiii	R0 & imm; if the result is 0, 1 →T	1	Test result

Table 5.5 Logic Operation Instructions (cont)

Instruction		Instruction Code	Operation	Execution State	T Bit
TST.B	#imm,@(R0,GBR)	11001100iiiiiiii	(R0 + GBR) & imm; if the result is 0, 1 → T	3	Test result
XOR	Rm,Rn	0010nnnnmmmm1010	Rn ^ Rm → Rn	1	—
XOR	#imm,R0	11001010iiiiiiii	R0 ^ imm → R0	1	—
XOR.B	#imm,@(R0,GBR)	11001110iiiiiiii	(R0 + GBR) ^ imm → (R0 + GBR)	3	—

5.1.4 Shift Instructions

Table 5.6 Shift Instructions

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

5.1.5 Branch Instructions

Table 5.7 Branch Instructions

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

Notes: 2. SH7600 instruction

3. One state when it does not branch

5.1.6 System Control Instructions

Table 5.8 System Control Instructions

Instruction		Instruction Code	Operation	Execution State	T Bit
CLRT		0000000000001000	0 →T	1	0
CLRMAC		0000000000101000	0 → MACH, MACL	1	—
LDC	Rm,SR	0100mmmm00001110	Rm → SR	1	LSB
LDC	Rm,GBR	0100mmmm00011110	Rm → GBR	1	—
LDC	Rm,VBR	0100mmmm00101110	Rm → VBR	1	—
LDC.L	@Rm+,SR	0100mmmm00000111	(Rm) → SR, Rm + 4 → Rm	3	LSB
LDC.L	@Rm+,GBR	0100mmmm00010111	(Rm) → GBR, Rm + 4 → Rm	3	—
LDC.L	@Rm+,VBR	0100mmmm00100111	(Rm) → VBR, Rm + 4 → Rm	3	—
LDS	Rm,MACH	0100mmmm00001010	Rm → MACH	1	—
LDS	Rm,MACL	0100mmmm00011010	Rm → MACL	1	—
LDS	Rm,PR	0100mmmm00101010	Rm → PR	1	—
LDS.L	@Rm+,MACH	0100mmmm00000110	(Rm) → MACH, Rm + 4 → Rm	1	—
LDS.L	@Rm+,MACL	0100mmmm00010110	(Rm) → MACL, Rm + 4 → Rm	1	—
LDS.L	@Rm+,PR	0100mmmm00100110	(Rm) → PR, Rm + 4 → Rm	1	—
NOP		0000000000001001	No operation	1	—
RTE		0000000000101011	Delayed branch, stack area → PC/SR	4	LSB
SETT		0000000000011000	1 →T	1	1
SLEEP		0000000000011011	Sleep	3*4	—
STC	SR,Rn	0000nnnn00000010	SR → Rn	1	—
STC	GBR,Rn	0000nnnn00010010	GBR → Rn	1	—
STC	VBR,Rn	0000nnnn00100010	VBR → Rn	1	—
STC.L	SR,@-Rn	0100nnnn00000011	Rn–4 → Rn, SR → (Rn)	2	—
STC.L	GBR,@-Rn	0100nnnn00010011	Rn–4 → Rn, GBR → (Rn)	2	—
STC.L	VBR,@-Rn	0100nnnn00100011	Rn–4 → Rn, VBR → (Rn)	2	—
STS	MACH,Rn	0000nnnn00001010	MACH → Rn	1	—
STS	MACL,Rn	0000nnnn00011010	MACL → Rn	1	—
STS	PR,Rn	0000nnnn00101010	PR → Rn	1	—

Table 5.8 System Control Instructions (cont)

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

Notes: 4. The number of execution states before the chip enters the sleep state

The above 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.

5.2 Instruction Set in Alphabetical Order

Table 5.9 alphabetically lists instruction codes and number of execution cycles for each instruction.

Table 5.9 Instruction Set

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

Table 5.9 Instruction Set (cont)

Instruction		Instruction Code	Operation	Execution State	T Bit
BRA	label	1010dddddddddddd	Delayed branch, disp × 2 + PC → PC	2	—
BRAF	Rn*2	0000nnnn00100011	Delayed branch, Rn + PC → PC	2	—
BSR	label	1011dddddddddddd	Delayed branch, PC → PR, disp × 2 + PC → PC	2	—
BSRF	Rn*2	0000nnnn00000011	Delayed branch, PC → PR, Rn + PC → PC	2	—
BT	label	10001001dddddddd	If T = 1, disp × 2+ PC → PC; if T = 0, nop	3/1*3	—
BT/S	label*2	10001101dddddddd	If T = 1, disp × 2 + PC → PC; if T = 0, nop	2/1*3	—
CLRMAC		0000000000101000	0 → MACH, MACL	1	—
CLRT		0000000000001000	0 →T	1	0
CMP/EQ	#imm,R0	10001000iiiiiiii	If R0 = imm, 1 →T	1	Comparison result
CMP/EQ	Rm,Rn	0011nnnnmmmm0000	If Rn = Rm, 1 →T	1	Comparison result
CMP/GE	Rm,Rn	0011nnnnmmmm0011	If Rn ≥ Rm with signed data, 1 → T	1	Comparison result
CMP/GT	Rm,Rn	0011nnnnmmmm0111	If Rn > Rm with signed data, 1 → T	1	Comparison result
CMP/HI	Rm,Rn	0011nnnnmmmm0110	If Rn > Rm with unsigned data, 1 → T	1	Comparison result
CMP/HS	Rm,Rn	0011nnnnmmmm0010	If Rn ≥ Rm with unsigned data, 1 → T	1	Comparison result
CMP/PL	Rn	0100nnnn00010101	If Rn>0, 1 →T	1	Comparison result
CMP/PZ	Rn	0100nnnn00010001	If Rn ≥ 0, 1 → T	1	Comparison result

Notes: 2. SH7600 instructions
3. One state when it does not branch

Table 5.9 Instruction Set (cont)

Instruction		Instruction Code	Operation	Execution State	T Bit
CMP/STR	Rm,Rn	0010nnnnmmmm1100	If Rn and Rm have an equivalent byte, 1 → T	1	Comparison result
DIV0S	Rm,Rn	0010nnnnmmmm0111	MSB of Rn → Q, MSB of Rm → M, M ^ Q → T	1	Calculation result
DIV0U		0000000000011001	0 → M/Q/T	1	0
DIV1	Rm,Rn	0011nnnnmmmm0100	Single-step division (Rn/Rm)	1	Calculation result
DMULS.L	Rm,Rn*2	0011nnnnmmmm1101	Signed operation of Rn x Rm → MACH, MACL	2 to 4*1	—
DMULU.L	Rm,Rn*2	0011nnnnmmmm0101	Unsigned operation of Rn x Rm → MACH, MACL	2 to 4*1	—
DT	Rn*2	0100nnnn00010000	Rn - 1 → Rn, when Rn is 0, 1 → T. When Rn is nonzero, 0 → T	1	Comparison result
EXTS.B	Rm,Rn	0110nnnnmmmm1110	A byte in Rm is sign-extended → Rn	1	—
EXTS.W	Rm,Rn	0110nnnnmmmm1111	A word in Rm is sign-extended → Rn	1	—
EXTU.B	Rm,Rn	0110nnnnmmmm1100	A byte in Rm is zero-extended → Rn	1	—
EXTU.W	Rm,Rn	0110nnnnmmmm1101	A word in Rm is zero-extended → Rn	1	—
JMP	@Rn	0100nnnn00101011	Delayed branch, Rn → PC	2	—

Notes: 1. The normal minimum number of execution states
2. SH7600 instructions

Table 5.9 Instruction Set (cont)

Instruction		Instruction Code	Operation	Execu-tion State	T Bit
JSR	@Rn	0100nnnn00001011	Delayed branch, PC → PR, Rn → PC	2	—
LDC	Rm,GBR	0100mmmm00011110	Rm → GBR	1	—
LDC	Rm,SR	0100mmmm00001110	Rm → SR	1	LSB
LDC	Rm,VBR	0100mmmm00101110	Rm → VBR	1	—
LDC.L	@Rm+,GBR	0100mmmm00010111	(Rm) → GBR, Rm + 4 → Rm	3	—
LDC.L	@Rm+,SR	0100mmmm00000111	(Rm) → SR, Rm + 4 → Rm	3	LSB
LDC.L	@Rm+,VBR	0100mmmm00100111	(Rm) → VBR, Rm + 4 → Rm	3	—
LDS	Rm,MACH	0100mmmm00001010	Rm → MACH	1	—
LDS	Rm,MACL	0100mmmm00011010	Rm → MACL	1	—
LDS	Rm,PR	0100mmmm00101010	Rm → PR	1	—
LDS.L	@Rm+,MACH	0100mmmm00000110	(Rm) → MACH, Rm + 4 → Rm	1	—
LDS.L	@Rm+,MACL	0100mmmm00010110	(Rm) → MACL, Rm + 4 → Rm	1	—
LDS.L	@Rm+,PR	0100mmmm00100110	(Rm) → PR, Rm + 4 → Rm	1	—
MAC.L	@Rm+,@Rn+*2	0000nnnnmmmm1111	Signed operation of (Rn) × (Rm) + MAC → MAC	3/(2 to 4)*1	—
MAC.W	@Rm+,@Rn+	0100nnnnmmmm1111	Signed operation of (Rn) × (Rm) + MAC → MAC	3/(2)*1	—
MOV	#imm,Rn	1110nnnniiiiiiii	imm → Sign extension → Rn	1	—
MOV	Rm,Rn	0110nnnnmmmm0011	Rm → Rn	1	—

Notes: 1. The normal minimum number of execution states (the number in parentheses is the number of states when there is contention with preceding/following instructions)

2. SH7600 instructions

Table 5.9 Instruction Set (cont)

Instruction	Instruction Code	Operation	Execution State	T Bit
MOV.B @(disp,GBR),R0	11000100dddddddd	(disp + GBR) → Sign extension → R0	1	—
MOV.B @(disp,Rm),R0	10000100mmmmdddd	(disp + Rm) → Sign extension → R0	1	—
MOV.B @(R0,Rm),Rn	0000nnnnmmmm1100	(R0 + Rm) → Sign extension → Rn	1	—
MOV.B @Rm+,Rn	0110nnnnmmmm0100	(Rm) → Sign extension → Rn, Rm + 1 → Rm	1	—
MOV.B @Rm,Rn	0110nnnnmmmm0000	(Rm) → Sign extension → Rn	1	—
MOV.B R0,@(disp,GBR)	11000000dddddddd	R0 → (disp + GBR)	1	—
MOV.B R0,@(disp,Rn)	10000000nnnndddd	R0 → (disp + Rn)	1	—
MOV.B Rm,@(R0,Rn)	0000nnnnmmmm0100	Rm → (R0 + Rn)	1	—
MOV.B Rm,@-Rn	0010nnnnmmmm0100	Rn–1 → Rn, Rm → (Rn)	1	—
MOV.B Rm,@Rn	0010nnnnmmmm0000	Rm → (Rn)	1	—
MOV.L @(disp,GBR),R0	11000110dddddddd	(disp × 4 + GBR) → R0	1	—
MOV.L @(disp,PC),Rn	1101nnnndddddddd	(disp × 4 + PC) → Rn	1	—
MOV.L @(disp,Rm),Rn	0101nnnnmmmmdddd	(disp × 4 + Rm) → Rn	1	—
MOV.L @(R0,Rm),Rn	0000nnnnmmmm1110	(R0 + Rm) → Rn	1	—
MOV.L @Rm+,Rn	0110nnnnmmmm0110	(Rm) → Rn, Rm + 4 → Rm	1	—
MOV.L @Rm,Rn	0110nnnnmmmm0010	(Rm) → Rn	1	—
MOV.L R0,@(disp,GBR)	11000010dddddddd	R0 → (disp × 4 + GBR)	1	—
MOV.L Rm,@(disp,Rn)	0001nnnnmmmmdddd	Rm → (disp × 4 + Rn)	1	—
MOV.L Rm,@(R0,Rn)	0000nnnnmmmm0110	Rm → (R0 + Rn)	1	—
MOV.L Rm,@-Rn	0010nnnnmmmm0110	Rn–4 → Rn, Rm → (Rn)	1	—
MOV.L Rm,@Rn	0010nnnnmmmm0010	Rm → (Rn)	1	—
MOV.W @(disp,GBR),R0	11000101dddddddd	(disp × 2 + GBR) → Sign extension → R0	1	—

Table 5.9 Instruction Set (cont)

Instruction		Instruction Code	Operation	Execution State	T Bit
MOV.W	@(disp,PC),Rn	1001nnnndddddddd	(disp × 2 + PC) → Sign extension → Rn	1	—
MOV.W	@(disp,Rm),R0	10000101mmmmdddd	(disp × 2 + Rm) → Sign extension → R0	1	—
MOV.W	@(R0,Rm),Rn	0000nnnnmmmm1101	(R0 + Rm) → Sign extension → Rn	1	—
MOV.W	@Rm+,Rn	0110nnnnmmmm0101	(Rm) → Sign extension → Rn, Rm + 2 → Rm	1	—
MOV.W	@Rm,Rn	0110nnnnmmmm0001	(Rm) → Sign extension → Rn	1	—
MOV.W	R0,@(disp,GBR)	11000001dddddddd	R0 → (disp × 2+ GBR)	1	—
MOV.W	R0,@(disp,Rn)	10000001nnnndddd	R0 → (disp × 2 + Rn)	1	—
MOV.W	Rm,@(R0,Rn)	0000nnnnmmmm0101	Rm → (R0 + Rn)	1	—
MOV.W	Rm,@-Rn	0010nnnnmmmm0101	Rn–2 → Rn, Rm → (Rn)	1	—
MOV.W	Rm,@Rn	0010nnnnmmmm0001	Rm → (Rn)	1	—
MOVA	@(disp,PC),R0	11000111dddddddd	disp × 4 + PC → R0	1	—
MOVT	Rn	0000nnnn00101001	T → Rn	1	—
MUL.L	Rm,Rn*2	0000nnnnmmmm0111	Rn × Rm → MACL	2 to 4*1	—
MULS.W	Rm,Rn	0010nnnnmmmm1111	Signed operation of Rn × Rm → MAC	1 to 3*1	—
MULU.W	Rm,Rn	0010nnnnmmmm1110	Unsigned operation of Rn × Rm → MAC	1 to 3*1	—
NEG	Rm,Rn	0110nnnnmmmm1011	0–Rm → Rn	1	—
NEGC	Rm,Rn	0110nnnnmmmm1010	0–Rm–T → Rn, Borrow → T	1	Borrow
NOP		0000000000001001	No operation	1	—
NOT	Rm,Rn	0110nnnnmmmm0111	~Rm → Rn	1	—
OR	#imm,R0	11001011iiiiiiii	R0 | imm → R0	1	—
OR	Rm,Rn	0010nnnnmmmm1011	Rn | Rm → Rn	1	—

Notes: 1. The normal minimum number of execution states
2. SH7600 instructions

Table 5.9 Instruction Set (cont)

Instruction		Instruction Code	Operation	Execution State	T Bit
OR.B	#imm,@(R0,GBR)	11001111iiiiiiii	(R0 + GBR) | imm → (R0 + GBR)	3	—
ROTCL	Rn	0100nnnn00100100	T ← Rn ← T	1	MSB
ROTCR	Rn	0100nnnn00100101	T → Rn → T	1	LSB
ROTL	Rn	0100nnnn00000100	T ← Rn ← MSB	1	MSB
ROTR	Rn	0100nnnn00000101	LSB → Rn → T	1	LSB
RTE		0000000000101011	Delayed branch, stack area → PC/SR	4	LSB
RTS		0000000000001011	Delayed branch, PR → PC	2	—
SETT		0000000000011000	1 → T	1	1
SHAL	Rn	0100nnnn00100000	T ← Rn ← 0	1	MSB
SHAR	Rn	0100nnnn00100001	MSB → Rn → T	1	LSB
SHLL	Rn	0100nnnn00000000	T ← Rn ← 0	1	MSB
SHLL2	Rn	0100nnnn00001000	Rn<<2 → Rn	1	—
SHLL8	Rn	0100nnnn00011000	Rn<<8 → Rn	1	—
SHLL16	Rn	0100nnnn00101000	Rn<<16 → Rn	1	—
SHLR	Rn	0100nnnn00000001	0 → Rn → T	1	LSB
SHLR2	Rn	0100nnnn00001001	Rn>>2 → Rn	1	—
SHLR8	Rn	0100nnnn00011001	Rn>>8 → Rn	1	—
SHLR16	Rn	0100nnnn00101001	Rn>>16 → Rn	1	—
SLEEP		0000000000011011	Sleep	3	—
STC	GBR,Rn	0000nnnn00010010	GBR → Rn	1	—
STC	SR,Rn	0000nnnn00000010	SR → Rn	1	—
STC	VBR,Rn	0000nnnn00100010	VBR → Rn	1	—
STC.L	GBR,@-Rn	0100nnnn00010011	Rn–4 → Rn, GBR → (Rn)	2	—
STC.L	SR,@-Rn	0100nnnn00000011	Rn–4 → Rn, SR → (Rn)	2	—
STC.L	VBR,@-Rn	0100nnnn00100011	Rn–4 → Rn, VBR → (Rn)	2	—
STS	MACH,Rn	0000nnnn00001010	MACH → Rn	1	—

Table 5.9 Instruction Set (cont)

Instruction		Instruction Code	Operation	Execution State	T Bit
STS	MACL,Rn	0000nnnn00011010	MACL → Rn	1	—
STS	PR,Rn	0000nnnn00101010	PR → Rn	1	—
STS.L	MACH,@-Rn	0100nnnn00000010	Rn–4 → Rn, MACH → (Rn)	1	—
STS.L	MACL,@-Rn	0100nnnn00010010	Rn–4 → Rn, MACL → (Rn)	1	—
STS.L	PR,@-Rn	0100nnnn00100010	Rn–4 → Rn, PR → (Rn)	1	—
SUB	Rm,Rn	0011nnnnmmmm1000	Rn–Rm → Rn	1	—
SUBC	Rm,Rn	0011nnnnmmmm1010	Rn–Rm–T → Rn, Borrow → T	1	Borrow
SUBV	Rm,Rn	0011nnnnmmmm1011	Rn–Rm → Rn, Underflow → T	1	Underflow
SWAP.B	Rm,Rn	0110nnnnmmmm1000	Rm → Swap upper and lower 2 bytes → Rn	1	—
SWAP.W	Rm,Rn	0110nnnnmmmm1001	Rm → Swap upper and lower word → Rn	1	—
TAS.B	@Rn	0100nnnn00011011	If (Rn) is 0, 1 → T; 1 → MSB of (Rn)	4	Test result
TRAPA	#imm	11000011iiiiiiii	PC/SR → stack area, (imm × 4 + VBR) → PC	8	—
TST	#imm,R0	11001000iiiiiiii	R0 & imm; if the result is 0, 1 → T	1	Test result
TST	Rm,Rn	0010nnnnmmmm1000	Rn & Rm; if the result is 0, 1 → T	1	Test result
TST.B	#imm,@(R0,GBR)	11001100iiiiiiii	(R0 + GBR) & imm; if the result is 0, 1 → T	3	Test result
XOR	#imm,R0	11001010iiiiiiii	R0 ^ imm → R0	1	—
XOR	Rm,Rn	0010nnnnmmmm1010	Rn ^ Rm → Rn	1	—
XOR.B	#imm,@(R0,GBR)	11001110iiiiiiii	(R0 + GBR) ^ imm → (R0 + GBR)	3	—
XTRCT	Rm,Rn	0010nnnnmmmm1101	Center 32 bits of Rm and Rn → Rn	1	—

Section 6 Instruction Descriptions

This section describes instructions in alphabetical order using the format shown below in section 6.1. The actual descriptions begin at section 6.2.

6.1 Sample Description (Name): Classification

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

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

Description: Description of operation

Notes: Notes on using the instruction

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

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

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

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

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

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

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

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

- List registers:

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

- Definition of SR structures:

```
struct SR0 {
    unsigned long dummy0:22;
    unsigned long M0:1;
    unsigned long Q0:1;
    unsigned long I0:4;
    unsigned long dummy1:2;
    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)
```

- Error display function:

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

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

Examples: Examples are written in assembler mnemonics and describe state 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's 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: The SH-series cross assembler version 1.0 does not support the conditional assembler functions.

Notes: 1. In the assembler descriptions in this manual for addressing modes that involve the following displacements (disp), the value prior to scaling (x1, x2, x4) according to the operand size is written. This is done to show clearly the operation of the LSI; see the assembler notation rules for the actural assembler descriptions.

@(disp:4, Rn):	Register indirect with displacement
@(disp:8, GBR):	GBR indirect with displacement
@(disp 8, PC):	PC relative with displacement
disp:8, disp:12:	PC relative

2. Among the 16 bits of the instruction code, a code not assigned as an instruction is treated as a general illegal instruction, and will result in illegal instruction exception processing, This includes the case where an instruction code for the SH7600 series only is executed on the SH7000 series.

   Example 1: H'FFF [General illegal instruction in both SH7000 and SH 7600]

   Example 2: H'3105 (=DMUL.L R0, R1)[Illegal instruction in SH7000]

3. If the instruction following a delayed branch instruction such as BRA, BT/S, etc., is a general illegal instruction or a branch instruction (known as a slot illegal instruction), illegal instruction exception processing will be performed.

Example 1 ....

```
BRA Label
. data. W H'FFFF   ← Slot illegal instruction
. . . .              [H'FFF is fundamentally a general illegal
                      instruction]
```

Example 2 `RTE`

```
BT/S Label ← Slot illegal instruction
```

6.2 ADD (ADD Binary): Arithmetic Instruction

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

Description: Adds general register Rn data to Rm data, and stores the result in Rn. The contents of Rn can also be added to 8-bit immediate 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

6.3 ADDC (ADD with Carry): Arithmetic Instruction

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

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

Operation:

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

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

Examples:

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

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

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

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

Operation:

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

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

Examples:

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

6.5 AND (AND Logical): Logic Operation Instruction

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

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

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

Operation:

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

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

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

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

Examples:

`AND`	`R0,R1`	Before execution	R0 = H'AAAAAAAA, R1 = H'55555555
		After execution	R1 = H'00000000
`AND`	`#H'0F,R0`	Before execution	R0 = H'FFFFFFFF
		After execution	R0 = H'0000000F
`AND.B`	`#H'80,@(R0,GBR)`	Before execution	@(R0,GBR) = H'A5
		After execution	@(R0,GBR) = H'80

6.6 BF (Branch if False): Branch Instruction

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

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

Note: When branching, three cycles; when not branching, one cycle.

Operation:

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

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

Example:

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

6.7 BF/S (Branch if False with Delay Slot): Branch Instruction (SH7600)

Class: Delayed branch instruction

Format	Abstract	Code	State	T Bit
`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 with delay slot. If T = 1, BF executes the next instruction. If T = 0, it branches after executing the next instruction. The branch destination is an address specified by PC + displacement. The PC points to the starting address of the second instruction after the branch instruction. The 8-bit displacement is sign-extended and doubled. Consequently, the relative interval from the branch destination is –256 to +254 bytes. If the displacement is too short to reach the branch destination, use BF/S with the BRA instruction or the like.

Note: Since this is a delayed branch instruction, the instruction immediately after is executed before the branch. Between the time this instruction and the instruction immediately after are executed, address errors or interrupts are not accepted. When the instruction immediately after 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)+4;
        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, 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
```

6.8 BRA (Branch): Branch Instruction

Class: Delayed branch instruction

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

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

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

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

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

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

6.9 BRAF (Branch Far): Branch Instruction (SH7600)

Class: Delayed branch instruction

Format	Abstract	Code	State	T Bit
BRAF Rn	Rn + PC → PC	0000nnnn00100011	2	—

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

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

```
BRAF(long n)   /* BRAF Rn */
{
    unsigned long temp;

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

Example:

```
        MOV.L   #(TRGET-BSRF_PC),R0     Sets displacement
        BRAF    @R0                     Branches to TRGET
        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
    TRGET:                              ← 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.

6.10 BSR (Branch to Subroutine): Branch Instruction

Class: Delayed branch instruction

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

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

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

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

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

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
```

6.11 BSRF (Branch to Subroutine Far): Branch Instruction (SH7600)

Class: Delayed branch instruction

Format	Abstract	Code	State	T Bit
BSRF Rn	PC → PR, Rn + PC → PC	0000nnnn00000011	2	—

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

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

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

Example:

```
        MOV.L   #(TRGET-BSRF_PC),R0     Sets displacement
        BRSF    @R0                     Branches to TRGET
        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
        .....
        .....
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.

6.12 BT (Branch if True): Branch Instruction

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

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

Note: When branching, requires three cycles; when not branching, one cycle.

Operation:

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

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

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

6.13 BT/S (Branch if True with Delay Slot): Branch Instruction (SH7600)

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

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

Note: Since this is a delay branch instruction, the instruction immediately after is executed before the branch. Between the time this instruction and the immediately after instruction are executed, address errors or interrupts are not accepted. When the immediately after 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)+4;
        Delay_Slot(temp+2);
    }
    else PC+=2;
}
```

Example:

```
        SETT              T is always 1
        BF/S   TRGET_F    Does not branch, because T = 1
        NOP
        BT/S   TRGET_T    Branches to TRGET, 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
TRGET_T:                  ← Branch destination of the BT/S instruction
```

6.14 CLRMAC (Clear MAC Register): System Control Instruction

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

Description: Clears the MACH and MACL registers.

Operation:

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

Example:

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

6.15 CLRT (Clear T Bit): System Control Instruction

Format	Abstract	Code	State	T Bit
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

6.16 CMP/cond (Compare Conditionally): Arithmetic Instruction

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

Description: Compares general register Rn data with Rm data, and sets the T bit to 1 if a specified condition (cond) is satisfied. The T bit is cleared to 0 if the condition is not satisfied. 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&0xFF000000)>>12;
    HL=(temp&0x00FF0000)>>8;
    LH=(temp&0x0000FF00)>>4;
    LL=temp&0x000000FF;
    HH=HH&&HL&&LH&&LL;
    if (HH==0) T=1;
    else T=0;
    PC+=2;
}

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

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

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

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

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

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

Operation:

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

Example: See DIV1.

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

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

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

Operation:

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

Example: See DIV1.

6.19 DIV1 (Divide Step 1): Arithmetic Instruction

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

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

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

(Dividend) – (divisor)] (quotient) = (remainder)

with the SH7600 series in which a divider is installed as a peripheral function, the remainder can be found as a function of the divider.

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

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

Operation:

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

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

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

Example 1:

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

Example 2:

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

Example 3:

```
                           R1 (16 bits)/R0 (16 bits) = R1 (16 bits):Signed
SHLL16      R0             Upper 16 bits = divisor, lower 16 bits = 0
EXTS.W      R1,R1          Sign-extends the dividend to 32 bits
XOR         R2,R2          R2 = 0
MOV         R1,R3
ROTCL       R3
SUBC        R2,R1          Decrements if the dividend is negative
DIV0S       R0,R1          Flag initialization
.arepeat    16
DIV1        R0,R1          Repeat 16 times
.aendr
EXTS.W      R1,R1
ROTCL       R1             R1 = quotient (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)
```

6.20 DMULS.L (Double-Length Multiply as Signed): Arithmetic Instruction (SH7600)

Format	Abstract	Code	State	T Bit
`DMULS.L Rm,Rn`	With signed, 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 registers. 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	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)	

6.21 DMULU.L (Double-Length Multiply as Unsigned): Arithmetic Instruction (SH7600)

Format		Abstract	Code	State	T Bit
DMULU.L	Rm,Rn	Without signed, 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 registers. 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	R0,R1	Before execution	R0 = H'FFFFFFFE, R1 = H'00005555
		After execution	MACH = H'00005554, MACL = H'FFFF5556
STS	MACH,R0	Operation result (top)	
STS	MACL,R0	Operation result (bottom)	

6.22 DT (Decrement and Test): Arithmetic Instruction (SH7600)

Format		Abstract	Code	State	T Bit
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 is decremented by 1 and the result is 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 if T=0. (In this example, loops 4 times.)
```

6.23 EXTS (Extend as Signed): Arithmetic Instruction

Format	Abstract	Code	State	T Bit
EXTS.B Rm,Rn	Sign-extended Rm from byte → Rn	0110nnnnmmmm1110	1	—
EXTS.W Rm,Rn	Sign-extended 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 transferred to bits 8 to 31 of Rn. If word length is specified, the bit 15 value of Rm is transferred to 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

6.24 EXTU (Extend as Unsigned): Arithmetic Instruction

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

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

6.25 JMP (Jump): Branch Instruction

Class: Delayed branch instruction

Format	Abstract	Code	State	T Bit
JMP @Rn	Rn → PC	0100nnnn00101011	2	—

Description: Delayed-branches unconditionally to the address specified with register indirect. The branch destination is an address specified by the 32-bit data in general register Rn.

Note: Since this is a delayed branch instruction, the instruction after JMP is executed before branching. No interrupts 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 n)    /* JMP @Rn */
{
    unsigned long temp;

    temp=PC;
    PC=R[n]+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
```

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.

6.26 JSR (Jump to Subroutine): Branch Instruction

Class: Delayed branch instruction

Format	Abstract	Code	State	T Bit
JSR @Rn	PC → PR, Rn → PC	0100nnnn00001011	2	—

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

Note: Since this is a delayed branch instruction, the instruction after JSR is executed before branching. No interrupts 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 n)    /* JSR @Rn */
{
    PR=PC;
    PC=R[n]+4;
    Delay_Slot(PR+2);
}
```

Example:

```
             MOV.L     JSR_TABLE,R0    R0 = Address of 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: 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.

6.27 LDC (Load to Control Register): System Control Instruction

Class: Interrupt disabled instruction

Format		Abstract	Code	State	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.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	—

Description: Stores the source operand into control registers SR, GBR, or VBR.

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]&0x000003F3;
    PC+=2;
}

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

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

```
LDCMSR(long m)      /* LDC.L @Rm+,SR */
{
    SR=Read_Long(R[m])&0x000003F3;
    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;
}
```

Examples:

`LDC`	`R0,SR`	Before execution	R0 = H'FFFFFFFF, SR = H'00000000
		After execution	SR = H'000003F3
`LDC.L`	`@R15+,GBR`	Before execution	R15 = H'10000000
		After execution	R15 = H'10000004, GBR = @H'10000000

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

Class: Interrupt disabled instruction

Format		Abstract	Code	State	T Bit
LDS	Rm,MACH	Rm → MACH	0100mmmm00001010	1	—
LDS	Rm,MACL	Rm → MACL	0100mmmm00011010	1	—
LDS	Rm,PR	Rm → PR	0100mmmm00101010	1	—
LDS.L	@Rm+,MACH	(Rm) → MACH, Rm + 4 → Rm	0100mmmm00000110	1	—
LDS.L	@Rm+,MACL	(Rm) → MACL, Rm + 4 → Rm	0100mmmm00010110	1	—
LDS.L	@Rm+,PR	(Rm) → PR, Rm + 4 → Rm	0100mmmm00100110	1	—

Description: Stores the source operand into the system registers MACH, MACL, or PR.

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

For the SH7000, the lower 10 bits are stored in MACH. For the SH7600, 32 bits are stored in MACH.

Operation:

```
LDSMACH(long m)              /* LDS Rm,MACH */
{
    MACH=R[m];
    if ((MACH&0x00000200)==0) MACH&=0x000003FF;
    else MACH|=0xFFFFFC00;
    PC+=2;
}
LDSMACL(long m)              /* LDS Rm,MACL */
{
    MACL=R[m];
    PC+=2;
}

LDSPR(long m)                /* LDS Rm,PR */
{
    PR=R[m];
    PC+=2;
}
```

For SH7000 (these 2 lines not needed for SH7600)

```
LDSMMACH(long m)          /* LDS.L @Rm+,MACH */
{
    MACH=Read_Long(R[m]);
    if ((MACH&0x00000200)==0) MACH&=0x000003FF;
    else MACH|=0xFFFFFC00;
    R[m]+=4;
    PC+=2;
}
LDSMMACL(long m)          /* LDS.L @Rm+,MACL */
{
    MACL=Read_Long(R[m]);
    R[m]+=4;
    PC+=2;
}

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

For SH7000 (these 2 lines not needed for SH7600)

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

6.29 MAC.L (Multiply and Accumulate Long): Arithmetic Instruction (SH7600)

Format		Abstract	Code	State	T Bit
MAC.L	@Rm+,@Rn+	Signed operation, (Rn) × (Rm) + MAC → MAC	0000nnnnmmmm1111	3/(2 to 4)	—

Description: Signed-multiplicates 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 at the 48th bit starting from the LSB. For the saturation operation, only the lower 48 bits of the MACL registers are enabled and the result is limited to a range of H'FFFF800000000000 (minimum) to 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,temp1,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
```

6.30 MAC (Multiply and Accumulate): Arithmetic Instruction (SH7000)

Format	Abstract	Code	State	T Bit
`MAC.W @Rm+,@Rn+`	With signed, (Rn) × (Rm) + MAC → MAC	`0100nnnnmmmm1111`	3/(2)	—

Description (SH7000): Multiplies 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. Everytime an operand is read, they increment Rm and Rn by two.

When the S bit is cleared to 0, the 42-bit result is stored in the coupled MACH and MACL registers. Bit 9 data is transferred to the upper 22 bits (bits 31 to 10) of the MACH register.

When the S bit is set to 1, 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) to 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, and 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: The normal number of cycles for execution is 3; however, this instruction can be executed in two cycles according to the succeeding instruction.

6.31 MAC.W (Multiply and Accumulate Word): Arithmetic Instruction (SH7600)

Format		Abstract	Code	State	T Bit
`MAC.W`	`@Rm+,@Rn+`	Signed operation, (Rn) × (Rm) +	`0100nnnnmmmm1111`	3/(2)	—
`MAC`	`@Rm+,@Rn+`	MAC → MAC			

Description (SH7600): Signed-multiplicates 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. Everytime an operand is read, they increment Rm and Rn by two.

When the S bit is cleared to 0, the operation is $16 \times 16 + 64 \rightarrow 64$-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 $16 \times 16 + 32 \rightarrow 32$-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) to 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, and 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 SH7600 series performs a $16 \times 16 + 64 \rightarrow 64$ bit multiply and accumulate operation and the SH7000 series performs a $16 \times 16 + 42 \rightarrow 42$ 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 SH7000 (these 2 lines
                MACH|=0x00000001;               not needed for SH7600)
            if (src==0) MACL=0x7FFFFFFF;
            if (src==2) MACL=0x80000000;
        }
    }
    else {
        MACH+=tempn;
        if (templ>MACL) MACH+=1;
        if ((MACH&0x00000200)==0)               For SH7000 (these 3 lines
             MACH&=0x000003FF;                  not needed for SH7600)
        else MACH|=0xFFFFFC00;
    }
    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
```

6.32 MOV (Move Data): Data Transfer Instruction

Format	Abstract	Code	State	T Bit
`MOV Rm,Rn`	Rm → Rn	0110nnnnmmmm0011	1	—
`MOV.B Rm,@Rn`	Rm → (Rn)	0010nnnnmmmm0000	1	—
`MOV.W Rm,@Rn`	Rm → (Rn)	0010nnnnmmmm0001	1	—
`MOV.L Rm,@Rn`	Rm → (Rn)	0010nnnnmmmm0010	1	—
`MOV.B @Rm,Rn`	(Rm) → sign extension → Rn	0110nnnnmmmm0000	1	—
`MOV.W @Rm,Rn`	(Rm) → sign extension → Rn	0110nnnnmmmm0001	1	—
`MOV.L @Rm,Rn`	(Rm) → Rn	0110nnnnmmmm0010	1	—
`MOV.B Rm,@-Rn`	Rn – 1 → Rn, Rm → (Rn)	0010nnnnmmmm0100	1	—
`MOV.W Rm,@-Rn`	Rn – 2 → Rn, Rm → (Rn)	0010nnnnmmmm0101	1	—
`MOV.L Rm,@-Rn`	Rn – 4 → Rn, Rm → (Rn)	0010nnnnmmmm0110	1	—
`MOV.B @Rm+,Rn`	(Rm) → sign extension → Rn, Rm + 1 → Rm	0110nnnnmmmm0100	1	—
`MOV.W @Rm+,Rn`	(Rm) → sign extension → Rn, Rm + 2 → Rm	0110nnnnmmmm0101	1	—
`MOV.L @Rm+,Rn`	(Rm) → Rn, Rm + 4 → Rm	0110nnnnmmmm0110	1	—
`MOV.B Rm,@(R0,Rn)`	Rm → (R0 + Rn)	0000nnnnmmmm0100	1	—
`MOV.W Rm,@(R0,Rn)`	Rm → (R0 + Rn)	0000nnnnmmmm0101	1	—
`MOV.L Rm,@(R0,Rn)`	Rm → (R0 + Rn)	0000nnnnmmmm0110	1	—
`MOV.B @(R0,Rm),Rn`	(R0 + Rm) → sign extension → Rn	0000nnnnmmmm1100	1	—
`MOV.W @(R0,Rm),Rn`	(R0 + Rm) → sign extension → Rn	0000nnnnmmmm1101	1	—
`MOV.L @(R0,Rm),Rn`	(R0 + Rm) → Rn	0000nnnnmmmm1110	1	—

Description: Transfers the source operand to the destination. When the operand is stored in memory, the transferred data can be a byte, word, or longword. When the source operand is in memory, 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

6.33 MOV (Move Immediate Data): Data Transfer Instruction

Format	Abstract	Code	State	T Bit
`MOV #imm,Rn`	imm → sign extension → Rn	`1110nnnniiiiiiii`	1	—
`MOV.W @(disp,PC),Rn`	(disp × 2 + PC) → sign extension → Rn	`1001nnnndddddddd`	1	—
`MOV.L @(disp,PC),Rn`	(disp × 4 + PC) → Rn	`1101nnnndddddddd`	1	—

Description: Stores immediate data, which has been sign-extended to a longword, into general register Rn.

If the data is a word or longword, table data stored in the address specified by PC + displacement is accessed. If the data is a word, the 8-bit displacement is zero-extended and doubled. Consequently, the relative interval from the table is up to PC + 510 bytes. The PC points to the starting address of the second instruction after this MOV instruction. If the data is a longword, the 8-bit displacement is zero-extended and quadrupled. Consequently, the relative interval from the table is up to PC + 1020 bytes. The PC points to the starting address of the second instruction after this MOV instruction, but the lowest two bits of the PC are corrected to B'00.

Note: The end address of the program area (module) or the second address after an unconditional branch instruction are suitable for the start address of the table. If suitable table assignment is impossible (for example, if there are no unconditional branch instructions within the area specified by PC + 510 bytes or PC + 1020 bytes), the BRA instruction must be used to jump past the table. When this MOV instruction is placed immediately after a delayed branch instruction, the PC points to an address specified by (the starting address of the branch destination) + 2.

Operation:

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

MOVWI(long d,long n)       /* MOV.W @(disp,PC),Rn */
{
    long disp;
```

```
    disp=(0x000000FF & (long)d);
    R[n]=(long)Read_Word(PC+(disp<<1));
    if ((R[n]&0x8000)==0) R[n]&=0x0000FFFF;
    else R[n]|=0xFFFF0000;
    PC+=2;
}

MOVLI(long d,long n)        /* MOV.L @(disp,PC),Rn */
{
    long disp;

    disp=(0x000000FF & (long)d);
    R[n]=Read_Long((PC&0xFFFFFFFC)+(disp<<2));
    PC+=2;
}
```

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	

6.34 MOV (Move Peripheral Data): Data Transfer Instruction

Format	Abstract	Code	State	T Bit
`MOV.B @(disp,GBR),R0`	(disp + GBR) → sign extension → R0	`11000100dddddddd`	1	—
`MOV.W @(disp,GBR),R0`	(disp × 2 + GBR) → sign extension → R0	`11000101dddddddd`	1	—
`MOV.L @(disp,GBR),R0`	(disp × 4+ GBR) → R0	`11000110dddddddd`	1	—
`MOV.B R0,@(disp,GBR)`	R0 → (disp + GBR)	`11000000dddddddd`	1	—
`MOV.W R0,@(disp,GBR)`	R0 → (disp × 2 + GBR)	`11000001dddddddd`	1	—
`MOV.L R0,@(disp,GBR)`	R0 → (disp × 4 + GBR)	`11000010dddddddd`	1	—

Description: Transfers the source operand to the destination. This instruction is suitable for accessing data in the peripheral module area. The data can be a byte, word, or longword, but the register is fixed to R0.

A peripheral module base address is set to the GBR. When the peripheral module data is a byte, the 8-bit displacement is zero-extended. 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. Changing 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'FFFF7F80

6.35 MOV (Move Structure Data): Data Transfer Instruction

Format		Abstract	Code	State	T Bit
`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 suitable for accessing data in a structure or a stack. The data can be a byte, word, or longword, but when a byte or word is selected, only the R0 register is fixed. When the data is a byte, the 4-bit displacement is zero-extend. 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. Changing 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

6.36 MOVA (Move Effective Address): Data Transfer Instruction

Format	Abstract	Code	State	T Bit
MOVA @(disp,PC),R0	disp ×4 + PC → R0	11000111dddddddd	1	—

Description: Stores the effective address of the source operand into general register R0. The 8-bit displacement is zero-extended and quadrupled. Consequently, the relative interval from the operand is PC + 1020 bytes. The PC points to the starting address of the second instruction after this MOVA instruction, but the lowest two bits of the PC are corrected to B'00.

Note: If this instruction is placed immediately after a delayed branch instruction, the PC must point to an address specified by (the starting address of the branch destination) + 2.

Operation:

```
MOVA(long d)    /* MOVA @(disp,PC),R0 */
{
    long disp;

    disp=(0x000000FF & (long)d);
    R[0]=(PC&0xFFFFFFFC)+(disp<<2);
    PC+=2;
}
```

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 Æ R0
2006		NOP		

6.37 MOVT (Move T Bit): Data Transfer Instruction

Format	Abstract	Code	State	T Bit
MOVT Rn	T → Rn	0000nnnn00101001	1	—

Description: Stores the T bit value into general register Rn. When T = 1, 1 is stored in Rn, and when T = 0, 0 is stored in Rn.

Operation:

```
MOVT(long n)   /* MOVT Rn */
{
    R[n]=(0x00000001 & SR);
    PC+=2;
}
```

Example:

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

6.38 MUL.L (Multiply Long): Arithmetic Instruction (SH7600)

Format	Abstract	Code	State	T Bit
`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 lower 32 bits of the result in the MACL register. The MACH register data does not change.

Operation:

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

Example:

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

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

Format		Abstract	Code	State	T Bit
`MULS.W` `MULS`	`Rm,Rn` `Rm,Rn`	Signed operation, Rn × Rm → MACL	`0010nnnnmmmm1111`	1 to 3	—

Description: Performs 16-bit multiplication of the contents of general registers Rn and Rm, and stores the 32-bit result in the MACL register. The operation is signed and the MACH register data does not change.

Operation:

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

Example:

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

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

Format	Abstract	Code	State	T Bit
`MULU.W Rm,Rn` `MULU Rm,Rn`	Unsigned, Rn × Rm → MAC	`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	

6.41 NEG (Negate): Arithmetic Instruction

Format	Abstract	Code	State	T Bit
NEG Rm,Rn	0 – Rm → Rn	0110nnnnmmmm1011	1	—

Description: Takes the two's complement of data in general register Rm, and stores the result in Rn. This effectively subtracts Rm data from 0, and stores the result in Rn.

Operation:

```
NEG(long m,long n)     /* NEG Rm,Rn */
{
    R[n]=0-R[m];
    PC+=2;
}
```

Example:

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

6.42 NEGC (Negate with Carry): Arithmetic Instruction

Format	Abstract	Code	State	T Bit
NEGC Rm,Rn	0 – Rm – T → Rn, Borrow → T	0110nnnnmmmm1010	1	Borrow

Description: Subtracts general register Rm data and the T bit from 0, and stores the result in Rn. If a borrow is generated, T bit changes accordingly. This instruction is used for inverting the sign of a value that has more than 32 bits.

Operation:

```
NEGC(long m,long n)    /* NEGC Rm,Rn */
{
    unsigned long temp;

    temp=0-R[m];
    R[n]=temp-T;
    if (0<temp)    T=1;
    else T=0;
    if (temp<R[n])    T=1;
    PC+=2;
}
```

Examples:

CLRT		Sign inversion of R1 and R0 (64 bits)	
NEGC	R1,R1	Before execution	R1 = H'00000001, T = 0
		After execution	R1 = H'FFFFFFFF, T = 1
NEGC	R0,R0	Before execution	R0 = H'00000000, T = 1
		After execution	R0 = H'FFFFFFFF, T = 1

6.43 NOP (No Operation): System Control Instruction

Format	Abstract	Code	State	T Bit
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

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

Format	Abstract	Code	State	T Bit
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

6.45 OR (OR Logical) Logic Operation Instruction

Format	Abstract	Code	State	T Bit
OR Rm,Rn	Rn | Rm → Rn	0010nnnnmmmm1011	1	—
OR #imm,R0	R0 | imm → R0	11001011iiiiiiii	1	—
OR.B #imm,@(R0,GBR)	(R0 + GBR) | imm → (R0 + GBR)	11001111iiiiiiii	3	—

Description: Logically ORs the contents of general registers Rn and Rm, and stores the result in Rn. The contents of general register R0 can also be ORed with zero-extended 8-bit immediate data, or 8-bit memory data accessed by using indirect indexed GBR addressing can be ORed with 8-bit immediate data.

Operation:

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

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

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

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

Examples:

`OR`	`R0,R1`	Before execution	R0 = H'AAAA5555, R1 = H'55550000
		After execution	R1 = H'FFFF5555
`OR`	`#H'F0,R0`	Before execution	R0 = H'00000008
		After execution	R0 = H'000000F8
`OR.B`	`#H'50,@(R0,GBR)`	Before execution	@(R0,GBR) = H'A5
		After execution	@(R0,GBR) = H'F5

6.46 ROTCL (Rotate with Carry Left): Shift Instruction

Format	Abstract	Code	State	T Bit
ROTCL Rn	T ← Rn ← T	0100nnnn00100100	1	MSB

Description: Rotates the contents of general register Rn and the T bit to the left by one bit, and stores the result in Rn. The bit that is shifted out of the operand is transferred to the T bit (figure 6.3).

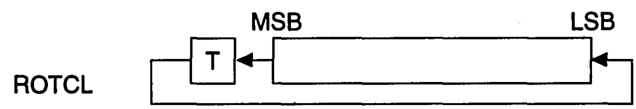


Figure 6.3 Rotate with Carry Left

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

6.47 ROTCR (Rotate with Carry Right): Shift Instruction

Format	Abstract	Code	State	T Bit
ROTCR Rn	T → Rn → T	0100nnnn00100101	1	LSB

Description: Rotates the contents of general register Rn and the T bit to the right by one bit, and stores the result in Rn. The bit that is shifted out of the operand is transferred to the T bit (figure 6.4).

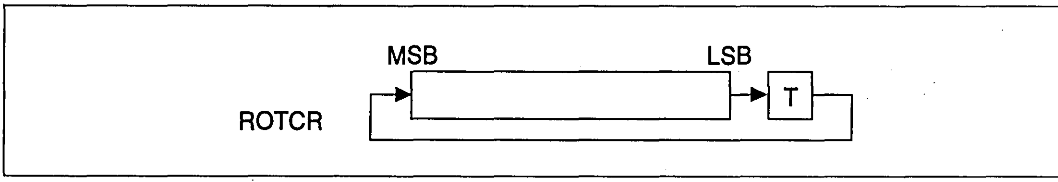


Figure 6.4 Rotate with Carry Right

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

6.48 ROTL (Rotate Left): Shift Instruction

Format	Abstract	Code	State	T Bit
ROTL Rn	T ← Rn ← MSB	0100nnnn00000100	1	MSB

Description: Rotates the contents of general register Rn to the left by one bit, and stores the result in Rn (figure 6.5). The bit that is shifted out of the operand is transferred to the T bit.

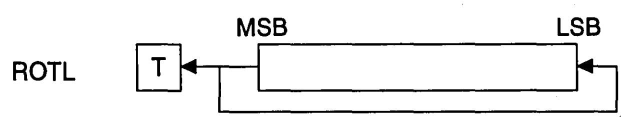


Figure 6.5 Rotate Left

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

6.49 ROTR (Rotate Right): Shift Instruction

Format	Abstract	Code	State	T Bit
ROTR Rn	LSB → Rn → T	0100nnnn00000101	1	LSB

Description: Rotates the contents of general register Rn to the right by one bit, and stores the result in Rn (figure 6.6). The bit that is shifted out of the operand is transferred to the T bit.

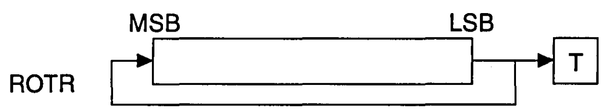


Figure 6.6 Rotate Right

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

6.50 RTE (Return from Exception): System Control Instruction

Class: Delayed branch instruction

Format	Abstract	Code	State	T Bit
RTE	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.

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])&0x000003F3;
    R[15]+=4;
    Delay_Slot(temp+2);
}
```

Example:

`RTE`	Returns to the original routine
`ADD #8,R14`	Executes ADD 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.

6.51 RTS (Return from Subroutine): Branch Instruction

Class: Delayed branch instruction

Format	Abstract	Code	State	T Bit
RTS	PR → PC	0000000000001011	2	—

Description: Returns from a subroutine procedure. The PC values are restored from the PR, and the program continues from the address specified by the restored PC value. This instruction is used to return to the program from a subroutine program called by a BSR or JSR instruction.

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

6.52 SETT (Set T Bit): System Control Instruction

Format	Abstract	Code	State	T Bit
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

6.53 SHAL (Shift Arithmetic Left): Shift Instruction

Format	Abstract	Code	State	T Bit
SHAL Rn	T ← Rn ← 0	0100nnnn00100000	1	MSB

Description: Arithmetically shifts the contents of general register Rn to the left by one bit, and stores the result in Rn. The bit that is shifted out of the operand is transferred to the T bit (figure 6.7).

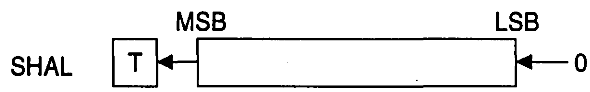


Figure 6.7 Shift Arithmetic Left

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

6.54 SHAR (Shift Arithmetic Right): Shift Instruction

Format	Abstract	Code	State	T Bit
SHAR Rn	MSB → Rn → T	0100nnnn00100001	1	LSB

Description: Arithmetically shifts the contents of general register Rn to the right by one bit, and stores the result in Rn. The bit that is shifted out of the operand is transferred to the T bit (figure 6.8).

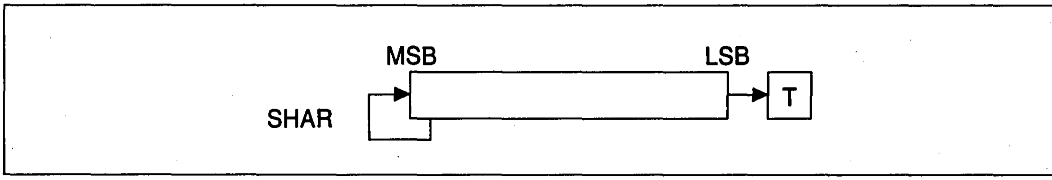


Figure 6.8 Shift Arithmetic Right

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

6.55 SHLL (Shift Logical Left): Shift Instruction

Format	Abstract	Code	State	T Bit
SHLL Rn	T ← Rn ← 0	0100nnnn00000000	1	MSB

Description: Logically shifts the contents of general register Rn to the left by one bit, and stores the result in Rn. The bit that is shifted out of the operand is transferred to the T bit (figure 6.9).

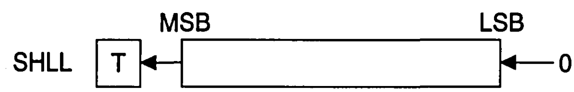


Figure 6.9 Shift Logical Left

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

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

Format		Abstract	Code	State	T Bit
SHLL2	Rn	Rn << 2 → Rn	0100nnnn00001000	1	—
SHLL8	Rn	Rn << 8 → Rn	0100nnnn00011000	1	—
SHLL16	Rn	Rn << 16 → Rn	0100nnnn00101000	1	—

Description: Logically shifts the contents of general register Rn to the left by 2, 8, or 16 bits, and stores the result in Rn. Bits that are shifted out of the operand are not stored (figure 6.10).

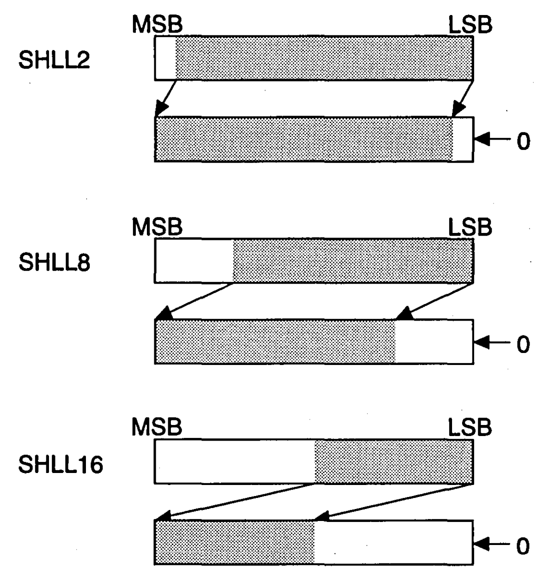


Figure 6.10 Shift Logical Left n Bits

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

6.57 SHLR (Shift Logical Right): Shift Instruction

Format	Abstract	Code	State	T Bit
SHLR Rn	0 → Rn → T	0100nnnn00000001	1	LSB

Description: Logically shifts the contents of general register Rn to the right by one bit, and stores the result in Rn. The bit that is shifted out of the operand is transferred to the T bit (figure 6.11).

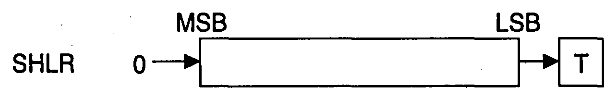


Figure 6.11 Shift Logical Right

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

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

Format		Abstract	Code	State	T Bit
SHLR2	Rn	Rn>>2 → Rn	0100nnnn00001001	1	—
SHLR8	Rn	Rn>>8 → Rn	0100nnnn00011001	1	—
SHLR16	Rn	Rn>>16 → Rn	0100nnnn00101001	1	—

Description: Logically shifts the contents of general register Rn to the right by 2, 8, or 16 bits, and stores the result in Rn. Bits that are shifted out of the operand are not stored (figure 6.12).

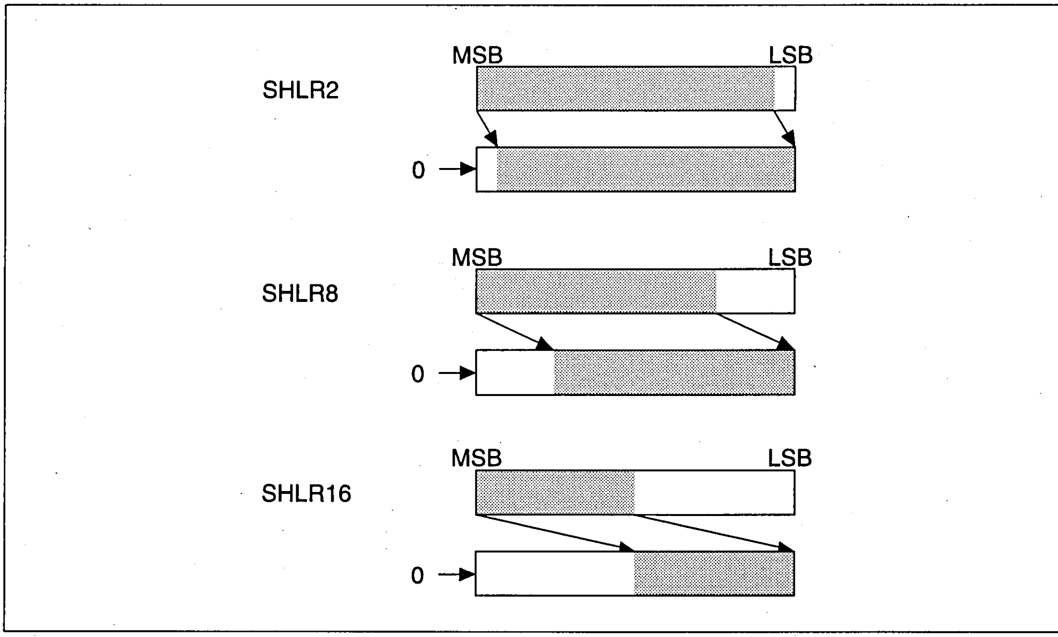


Figure 6.12 Shift Logical Right n Bits

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

6.59 SLEEP (Sleep): System Control Instruction

Format	Abstract	Code	State	T Bit
SLEEP	Sleep	0000000000011011	3	—

Description: Sets the CPU into power-down mode. In power-down mode, instruction execution stops, but the CPU module state 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;
    Error("Sleep Mode.");
}
```

Example:

```
SLEEP       Transits power-down mode
```

6.60 STC (Store Control Register): System Control Instruction

Class: Interrupt disabled instruction

Format		Abstract	Code	State	T Bit
STC	SR,Rn	SR → Rn	0000nnnn00000010	1	—
STC	GBR,Rn	GBR → Rn	0000nnnn00010010	1	—
STC	VBR,Rn	VBR → Rn	0000nnnn00100010	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	—

Description: Stores control registers SR, GBR, or VBR 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;
}
```

```
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;
}
```

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

6.61 STS (Store System Register): System Control Instruction

Class: Interrupt disabled instruction

Format		Abstract	Code	State	T Bit
STS	MACH,Rn	MACH → Rn	0000nnnn00001010	1	—
STS	MACL,Rn	MACL → Rn	0000nnnn00011010	1	—
STS	PR,Rn	PR → Rn	0000nnnn00101010	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	—

Description: Stores system registers MACH, MACL and PR data 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 SH7000 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 SH7600 series, 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 SH7000 (these 2 lines not needed for SH7600)

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

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

STSMMACH(long n)  /* STS.L MACH,@-Rn */
{
    R[n]-=4;
```

```
if ((MACH&0x00000200)==0)
Write_Long(R[n],MACH&0x000003FF);
else Write_Long
(R[n],MACH|0xFFFFFC00)
```

For SH7000

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

For SH7600

```
    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;
}
```

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

6.62 SUB (Subtract Binary): Arithmetic Instruction

Format		Abstract	Code	State	T Bit
SUB	Rm,Rn	Rn – Rm → Rn	0011nnnnmmmm1000	1	—

Description: Subtracts general register Rm data from Rn data, and stores the result in Rn. To subtract immediate data, use ADD #imm,Rn.

Operation:

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

Example:

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

6.63 SUBC (Subtract with Carry): Arithmetic Instruction

Format	Abstract	Code	State	T Bit
SUBC Rm,Rn	Rn – Rm– T → Rn, Borrow → T	0011nnnnmmmm1010	1	Borrow

Description: Subtracts Rm data and the T bit value from general register Rn, 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

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

Format	Abstract	Code	State	T Bit
SUBV Rm,Rn	Rn – Rm → Rn, Underflow → T	0011nnnnmmmm1011	1	Underflow

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

Operation:

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

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

Examples:

SUBV R0,R1	Before execution	R0 = H'00000002, R1 = H'80000001
	After execution	R1 = H'7FFFFFFF, T = 1
SUBV R2,R3	Before execution	R2 = H'FFFFFFFE, R3 = H'7FFFFFFE
	After execution	R3 = H'80000000, T = 1

6.65 SWAP (Swap Register Halves): Data Transfer Instruction

Format	Abstract	Code	State	T Bit
`SWAP.B Rm,Rn`	Rm → Swap upper and lower halves of lower 2 bytes → Rn	0110nnnnmmmm1000	1	—
`SWAP.W Rm,Rn`	Rm → Swap upper and lower word → Rn	0110nnnnmmmm1001	1	—

Description: Swaps the upper and lower bytes of the general register Rm data, and stores the result in Rn. If a byte is specified, bits 0 to 7 of Rm are swapped for bits 8 to 15. The upper 16 bits of Rm are transferred to the upper 16 bits of Rn. If a word is specified, bits 0 to 15 of Rm are swapped for bits 16 to 31.

Operation:

```
SWAPB(long m,long n)  /* SWAP.B Rm,Rn */
{
    unsigned long temp0,temp1;

    temp0=R[m]&0xffff0000;
    temp1=(R[m]&0x000000ff)<<8;
    R[n]=(R[m]&0x0000ff00)>>8;
    R[n]=R[n]|temp1|temp0;
    PC+=2;
}

SWAPW(long m,long n)  /* SWAP.W Rm,Rn */
{
    unsigned long temp;
    temp=(R[m]>>16)&0x0000FFFF;
    R[n]=R[m]<<16;
    R[n]|=temp;
    PC+=2;
}
```

Examples

`SWAP.B R0,R1`	Before execution	R0 = H'12345678
	After execution	R1 = H'12347856
`SWAP.W R0,R1`	Before execution	R0 = H'12345678
	After execution	R1 = H'56781234

6.66 TAS (Test and Set): Logic Operation Instruction

Format	Abstract	Code	State	T Bit
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
```

6.67 TRAPA (Trap Always): System Control Instruction

Format	Abstract	Code	State	T Bit
`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 points the starting address of the next instruction. TRAPA and RTE are both used 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
```

6.68 TST (Test Logical): Logic Operation Instruction

Format	Abstract	Code	State	T Bit
`TST Rm,Rn`	Rn & Rm, when result is 0, 1 →T	0010nnnnmmmm1000	1	Test results
`TST #imm,R0`	R0 & imm, when result is 0, 1 →T	11001000iiiiiiii	1	Test results
`TST.B #imm,@(R0,GBR)`	(R0 + GBR) & imm, when result is 0, 1 → T	11001100iiiiiiii	3	Test results

Description: Logically ANDs the contents of general registers Rn and Rm, and sets the T bit to 1 if the result is 0 or clears the T bit to 0 if the result is not 0. The Rn data does not change. The contents of general register R0 can also be ANDed with zero-extended 8-bit immediate data, or the contents of 8-bit memory accessed by indirect indexed GBR addressing can be ANDed with 8-bit immediate data. The R0 and memory data do not change.

Operation:

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

TSTI(long i)    /* TEST #imm,R0 */
{
    long temp;

    temp=R[0]&(0x000000FF & (long)i);
    if (temp==0) T=1;
    else T=0;
    PC+=2;
}

TSTM(long i)    /* TST.B #imm,@(R0,GBR) */
{
    long temp;
```

```
    temp=(long)Read_Byte(GBR+R[0]);
    temp&=(0x000000FF & (long)i);
    if (temp==0) T=1;
    else T=0;
    PC+=2;
}
```

Examples:

TST	R0,R0	Before execution	R0 = H'00000000
		After execution	T = 1
TST	#H'80,R0	Before execution	R0 = H'FFFFFF7F
		After execution	T = 1
TST.B	#H'A5,@(R0,GBR)	Before execution	@(R0,GBR) = H'A5
		After execution	T = 0

6.69 XOR (Exclusive OR Logical): Logic Operation Instruction

Format	Abstract	Code	State	T Bit
XOR Rm,Rn	Rn ^ Rm → Rn	0010nnnnmmmm1010	1	—
XOR #imm,R0	R0 ^ imm → R0	11001010iiiiiiii	1	—
XOR.B #imm,@(R0,GBR)	(R0 + GBR) ^ imm → (R0 + GBR)	11001110iiiiiiii	3	—

Description: Exclusive ORs the contents of general registers Rn and Rm, and stores the result in Rn. The contents of general register R0 can also be exclusive ORed with zero-extended 8-bit immediate data, or 8-bit memory accessed by indirect indexed GBR addressing can be exclusive ORed with 8-bit immediate data.

Operation:

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

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

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

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

Examples:

`XOR`	`R0,R1`	Before execution	R0 = H'AAAAAAAA, R1 = H'55555555
		After execution	R1 = H'FFFFFFFF
`XOR`	`#H'F0,R0`	Before execution	R0 = H'FFFFFFFF
		After execution	R0 = H'FFFFFF0F
`XOR.B`	`#H'A5,@(R0,GBR)`	Before execution	@(R0,GBR) = H'A5
		After execution	@(R0,GBR) = H'00

6.70 XTRCT (Extract): Data Transfer Instruction

Format	Abstract	Code	State	T Bit
XTRCT Rm,Rn	Center 32 bits of Rm and Rn → Rn	0010nnnnmmmm1101	1	—

Description: Extracts the middle 32 bits from the 64 bits of general registers Rm and Rn, and stores the 32 bits in Rn (figure 6.13).

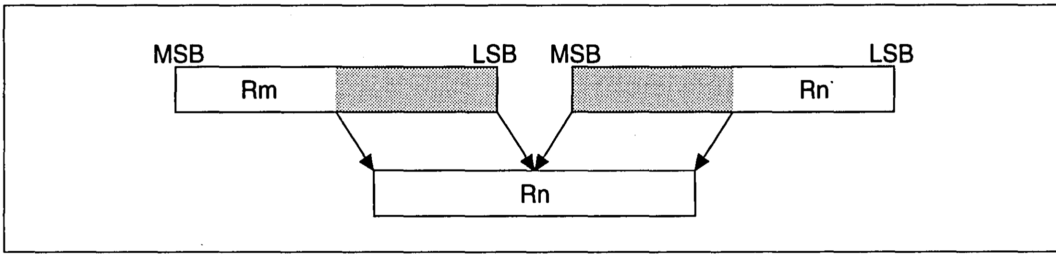


Figure 6.13 Extract

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

Section 7 Processing States

7.1 State Transitions

The CPU has five processing states: reset, exception processing, bus release, program execution and power-down. The transitions between the states are shown in figure 7.1. In the SH7600 series, the transitions in the bus release state are indicated for master mode. For more information, see the *SH Hardware Manual.*

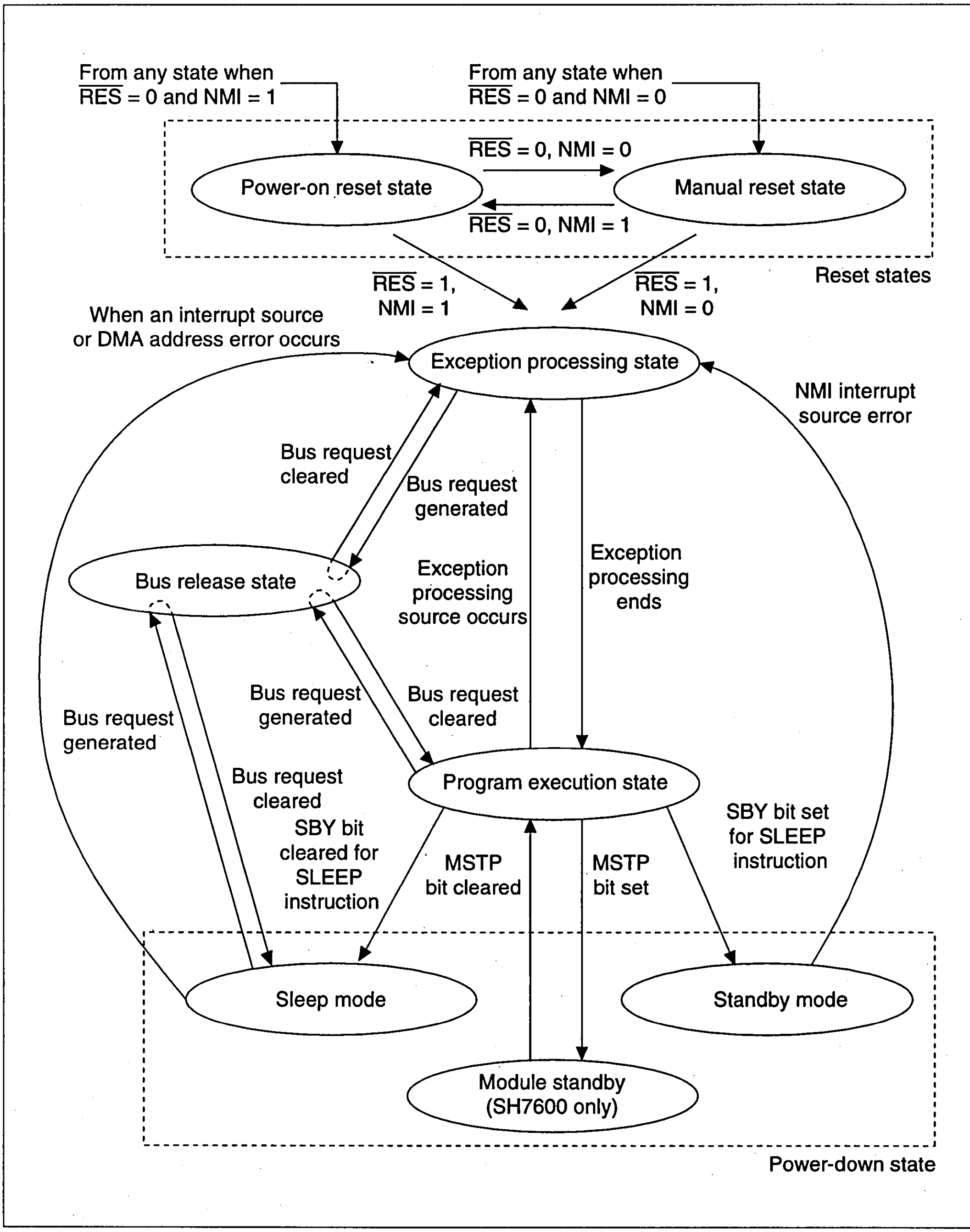


Figure 7.1 Transitions Between Processing States

7.1.1 Reset State

In the reset state, the CPU is reset. This occurs when the $\overline{RES}$ pin level goes low. When the NMI pin is high, the result is a power-on reset; when it is low, a manual reset will occur.

In the power-on reset, all CPU internal states and on-chip peripheral module registers are initialized. During manual reset, all on-chip peripheral module registers and CPU internal states, with the exception of the bus state controller (BSC) and pin function controller (PFC), are initialized. During manual reset the BSC is not initialized, allowing the refresh operation to continue.

7.1.2 Exception Processing State

The exception processing state is a transient state that occurs when the CPU's processing state flow is altered by exception processing sources such as resets or interrupts.

For a reset, the initial values of the program counter PC (execution start address) and stack pointer SP are fetched from the exception processing vector table and stored; the CPU then branches to the execution start address and execution of the program begins.

For an interrupt, the stack pointer (SP) is accessed and the program counter (PC) and status register (SR) are saved to the stack area. The exception service routine start address is fetched from the exception processing vector table; the CPU then branches to that address and the program starts executing, thereby entering the program execution state.

7.1.3 Program Execution State

In the program execution state, the CPU sequentially executes the program.

7.1.4 Power-Down State

In the power-down state, the CPU operation halts and power consumption declines. The SLEEP instruction places the CPU in the power-down state. This state has two modes: sleep mode and standby mode. See section 7.2 for more details. The SH7600 also has a module standby function.

7.1.5 Bus Release State

In the bus release state, the CPU releases access rights to the bus to the device that has requested them.

7.2 Power-Down State

In addition to the ordinary program execution states, the CPU also has a power-down state in which CPU operation halts and power consumption is lowered (table 7.1). There are two power-down state modes: sleep mode and standby mode.

7.2.1 Sleep Mode

When standby bit SBY (in the standby control register SBYCR) is cleared to 0 and a SLEEP instruction executed, the CPU moves from the program execution state to sleep mode. In the sleep mode, the CPU halts and the contents of its internal registers and the data in on-chip cache (RAM) are maintained. The on-chip peripheral modules other than the CPU do not halt in the sleep mode.

To return from sleep mode, use a reset, any interrupt, or a DMA address error; the CPU returns to the ordinary program execution state through the exception processing state.

7.2.2 Software Standby Mode

To enter the standby mode, set the standby bit SBY (in the standby control register SBYCR) to 1 and execute a SLEEP instruction. In standby mode, all CPU, on-chip peripheral module and oscillator functions are halted. CPU internal register contents and on-chip cache(RAM) data are held.

To return from standby mode, use a reset or an external NMI interrupt. For resets, the CPU returns to the ordinary program execution state through the exception processing state when placed in a reset state after the oscillator stabilization time has elapsed. For NMI interrupts, the CPU returns to the ordinary program execution state through the exception processing state after the oscillator stabilization time has elapsed. In this mode, power consumption declines markedly, since the oscillator stops.

7.2.3 Module Standby Function (SH7600 Only)

The module standby function is available for the multiplier (MULT), divider (DIVU), 16-bit free-running timer (FRT), serial communication interface (SCI), and the DMA controller (DMAC) for the on-chip peripheral modules.

The supply of the clock to these on-chip peripheral modules can be halted by setting the corresponding bits 4–0 (MSTP4–MSTP0) in the standby control register (SBYCR). Using this function can reduce the power consumption in sleep mode.

The external pins of the on-chip peripheral modules in module standby are reset and all registers except DMAC, MULT, and DIVU are initialized. (The master enable bit (bit 0) of the DMAC's DMA operation register (DMAOR) is initialized to 0.)

Module standby function is cleared by clearing the MSTP4–MSTP0 bits to 0.

Table 7.1 Power-Down State

		State						
Mode	**Condition**	**Clock**	**CPU**	**On-Chip Peripheral Module**	**CPU Register**	**RAM**	**I/O Port**	**Canceling**
Sleep mode	Executes SLEEP instruction with SBY bit cleared to 0 in SBYCR	Run	Halt	Run	Held	Held	Held	1. Interrupt 2. DMA address error 3. Power-on reset 4. Manual reset
Standby mode	Executes SLEEP instruction with SBY bit set to 1 in SBYCR	Halt	Halt	Halt and initialize*1	Held	Held	Held or high-Z*1	1. NMI 2. Power-on reset 3. Manual reset
Module standby function (SH7600 only)	Sets MSTP4–MSTP0 bits of SBYCR to 1	Run	Halt	Supply of clock to affected module is halted and module is initialized.*2	Held	Held	Held	Clears MSTP4–MSTP0 bits of SBYCR to 0

Notes: 1. Depends on the peripheral module and pin. For details, see the *Hardware Manual.*
2. Interrupt vectors maintain their settings.

7.3 Master Mode and Slave Mode (SH7600 Series Only)

The SH7600 series has two master modes and a slave mode for bus rights that can be selected with the MD5 pin. The master modes consist of a total master mode and a partial-share naster mode, which are specified using the MD5 pin and the partial-share space specification bit (PSHR) in bus control register 1 (BCR1). When the slave mode is selected with the MD5 pin, the device enters total slave mode. When the master mode is selected with the MD5 pin and partial space share is specified with the PSHR bit, the device enters the partial-share master mode. When partial space share is not specified with the PSHR bit, the device enters the total master mode.

The master mode has rights to bus use. External devices can be accessed freely. When a slave CPU requests the bus right, the master CPU can give the bus right to the slave CPU.

The total slave mode does not have rights to bus use. To access an external device, bus rights have to be requested to the master CPU, permission to use the bus gained, and then the external device accessed.

The partial-share master mode lacks bus rights only for CS2 space. To access the CS2 space, bus rights have to be requested to the master CPU, permission granted and then the CS2 space can be accessed. This mode has bus rights for all other space and does not need to request the bus when accessing them.

Table 7.2 Master Modes and Slave Mode (SH7600)

Mode	MD5 (Total Slave Mode Specification Pin)	PSHR (Partial-Share Bit)	Function
Total slave mode	1	(Not used)	Has no bus rights. To use a bus, requests the bus and receive permission from the master CPU to access.
Partial-share master mode	0	1	Has bus rights to CS0, CS1, and CS3 spaces. Lacks continuing bus rights only to CS2. To access CS2, first requests and be granted bus rights.
Total master mode	0	0	Always has bus rights. Gives bus rights to slave CPUs.

Section 8 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).

8.1 Basic Configuration of Pipelines

Pipelines are composed of the following five stages:

- IF (Instruction fetch) Fetches an instruction from the memory in which the program is stored.
- ID (Instruction decode) Decodes the instruction fetched.
- EX (Instruction execution) Performs data operations and address calculations according to the results of decoding.
- MA (Memory access) Accesses data in memory. Generated by instructions that involve memory access, with some exceptions.
- WB (Write back) Returns the results of the memory access (data) to a register. Generated by instructions that involve memory loads, with some exceptions.

As shown in figure 8.1, these stages flow with the execution of the instructions and thereby constitute a pipeline. At a given instant, five instructions are being executed simultaneously. All instructions have at least 3 stages: IF, ID, and EX. Most, but not all, have stages MA and WB as well. The way the pipeline flows also varies with the type of instruction. The basic pipeline flow is as shown in figure 8.1; some pipelines differ, however, because of contention between IF and MA. In figure 8.1, the period in which a single stage is operating is called a slot.

	↔	↔	↔	↔	↔	↔	↔	↔	↔	↔	: Slot
Instruction 1	IF	ID	EX	MA	WB						Instruction stream ↓
Instruction 2		IF	ID	EX	MA	WB					
Instruction 3			IF	ID	EX	MA	WB				
Instruction 4				IF	ID	EX	MA	WB			
Instruction 5					IF	ID	EX	MA	WB		
Instruction 6						IF	ID	EX	MA	WB	

Time →

Figure 8.1 Basic Structure of Pipeline Flow

8.2 Slot and Pipeline Flow

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

8.2.1 Instruction Execution

Each stage (IF, ID, EX, MA, and WB) of an instruction must be executed in one slot. Two or more stages cannot be executed within one slot (figure 8.2), 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   MA  WB
Instruction 2      IF        ID  EX  MA  WB

Note:  ID and EX of instruction 1 are being executed in the same slot.
```

Figure 8.2 Impossible Pipeline Flow 1

8.2.2 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 8.3).

```
               <-> <-> <-> <-> <-> <-> <-> <-> <-> <-> : Slot
Instruction 1  IF  ID  EX  MA  WB
Instruction 2  IF  ID  EX  MA  WB
Instruction 3      IF  ID  EX  MA  WB
Instruction 4          IF  ID  EX  MA  WB
Instruction 5          IF  ID  EX  MA  WB

Note:  Same stage of another instruction is being executed in same slot.
```

Figure 8.3 Impossible Pipeline Flow 2

8.2.3 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   Always one cycle

As an example, figure 8.4 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.

	(2)		(2)		(1)	(3)			(1)	(1) ← Number of cycles
Instruction 1	IF	IF	ID	—	EX	MA	MA	MA	WB	
Instruction 2			IF	IF	ID	EX	—	—	MA	WB

↔ : Slot

Figure 8.4 Slots Requiring Multiple Cycles

8.3 Number of Instruction Execution States

The number of instruction execution states is counted as the interval between execution of EX stages. The number of states 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 8.5, 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 state, the execution time of instruction 2 is one state.

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 an MOV Rm, Rn that follows instruction 3. (In the case of figure 8.5, 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 8.4, Contention Between Instruction Fetch (IF) and Memory Access (MA). The execution time between instructions 1 and 3 in figure 8.5 is seven states (5 + 1 + 1).

	←(2)→		←(2)→		←(2)→		←(4)→				↔(1)	↔(1) : Slot
Instruction 1	IF	IF	ID	—	[EX]	—	MA	MA	MA	WB		
Instruction 2			IF	IF	ID	—	—	—	—	[EX]		
Instruction 3					IF	IF	—	—	—	ID	[EX]	MA
(Instruction 4:	MOV Rm, Rn									IF	ID	[EX])

Figure 8.5 How Instruction Execution States Are Counted

8.4 Contention Between Instruction Fetch (IF) and Memory Access (MA)

8.4.1 Basic Operation When IF and MA are in Contention

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 8.6. When there is a WB, it is executed immediately after the MA ends.

: Slot	A	B	C	D	E	F	G		
Instruction 1	IF	ID	EX	$\boxed{\text{MA}}$	WB				
Instruction 2		IF	ID	EX	$\boxed{\text{MA}}$	WB			
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:

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

Split at D

Split at E

Figure 8.6 Operation When IF and MA Are in Contention

The slots in which MA and IF contend are split. 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 8.6 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 states 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.

8.4.2 The Relationship Between IF and the Location of Instructions in On-Chip ROM/RAM or On-Chip Memory

When the instruction is located in the on-chip memory (ROM or RAM) or on-chip cache of the SH microcomputer, the SH microcomputer accesses the on-chip memory in 32-bit units. The SH 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 8.7 illustrates these operations.

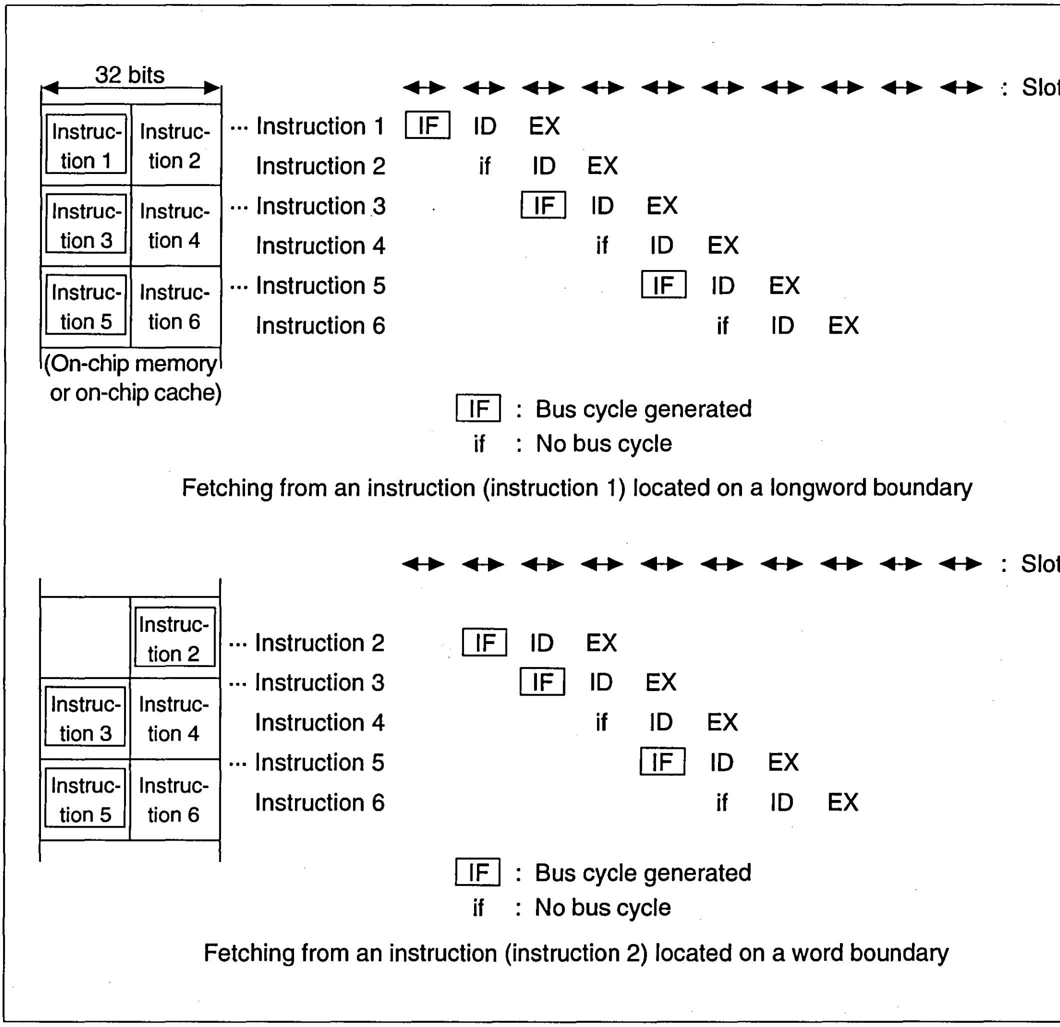


Figure 8.7 Relationship Between IF and Location of Instructions in On-Chip Memory

8.4.3 Relationship Between Position of Instructions Located in On-Chip ROM/RAM or On-Chip Memory and Contention Between IF and MA

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 8.4.2 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 8.8.

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.

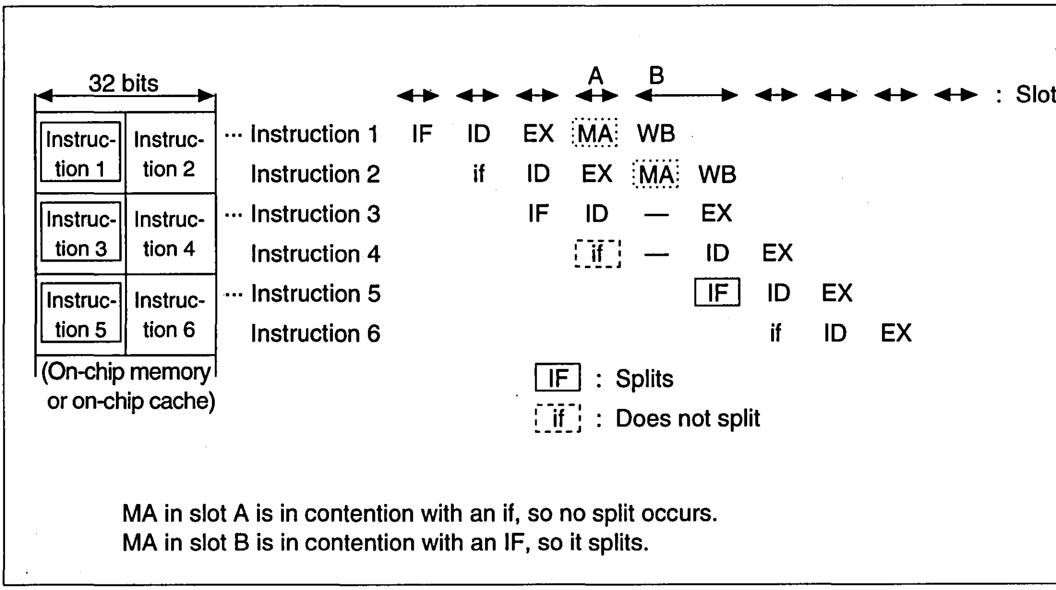


Figure 8.8 Relationship Between the Location of Instructions in On-Chip Memory and Contention Between IF and MA

8.5 Effects of Memory Load Instructions on Pipelines

Instructions that involve loading from memory return data to the destination register during the WB stage that comes at the end of the pipeline. The WB stage of such a load instruction (load instruction 1) will thus come after the EX stage of the instruction that immediately follows it (instruction 2).

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. The destination register of load instruction 1 is the same as the destination (not the source) of instruction 2, so it splits.

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 destination of load instruction 1 are the same.

The number of states in the slot generated by the split is the number of MA cycles plus the number of IF (or if) cycles, as illustrated in figure 8.9. 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.W @R0, R1)	IF	ID	EX	$\boxed{\text{MA}}$	WB		
Instruction 2 (ADD R1, R2)		IF	ID	—	$\boxed{\text{EX}}$		
Instruction 3			IF	—	ID	EX	.....
Instruction 4					IF	ID	.....

Figure 8.9 Effects of Memory Load Instructions on the Pipeline

8.6 Programming Guide

To improve instruction execution speed, consider the following when programming:

- To prevent contention between MA and IF, locate instructions that have MA stages so they start from the longword boundaries of on-chip memory (the position when the bottom two bits of the instruction address are 00 is A1 = 0 and A0 = 0) wherever possible.

- The instruction that immediately follows an instruction that loads from memory should not use the same destination register as the load instruction.

- Locate instructions that use the multiplier nonconsecutively.

8.7 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 states can be calculated.

In the following figures, "Instruction A" refers to the instruction being described. 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 described in the tables, with a few exceptions. When a slot has split, see section 8.4, Contention Between Instruction Fetch (IF) and Memory Access (MA). Base your response on the rules for pipeline operation given there.

Table 8.1 lists the format for number of instruction stages and execution states:

Table 8.1 Format for the Number of Stages and Execution States for Instructions

Type	Category	Stage	State	Contention	Instruction
Functional types	Instructions are categorized based on operations	Number of stages in an instruction	Number of execution states when no contention occurs	Contention that occurs	Corresponding instructions represented by mnemonic

Table 8.2 Number of Instruction Stages and Execution States

Type	Category	Stage	State	Contention	Instruction	
Data transfer instructions	Register-register transfer instructions	3	1	—	MOV	#imm,Rn
					MOV	Rm,Rn
					MOVA	@(disp,PC),R0
					MOVT	Rn
					SWAP.B	Rm,Rn
					SWAP.W	Rm,Rn
					XTRCT	Rm,Rn

Table 8.2 Number of Instruction Stages and Execution States (cont)

Type	Category	Stage	State	Contention	Instruction	
Data transfer instructions (cont)	Memory load instructions	5	1	• Contention occurs if the instruction placed immediately after this one uses the same destination register • MA contends with IF	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
	Memory store instructions	4	1	• MA contends with IF	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)

Table 8.2 Number of Instruction Stages and Execution States (cont)

Type	Category	Stage	State	Contention	Instruction	
Arithmetic instructions	Arithmetic instructions between registers (except multiplication instructions)	3	1	—	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*3
					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/accumulate instructions	7/8*1	3/(2)*2	• Multiplier contention occurs when an instruction that uses the multiplier follows a MAC instruction • MA contends with IF	MAC.W	@Rm+,@Rn+

Notes 1. In the SH7600, multiply/accumulate instructions are 7 stages, multiply instructions 6 stages; in the SH7000, multiply/accumulate instructions are 8 stages, multiply instructions 7 stages

2. The normal minimum number of execution states (The number in parentheses is the number of states when there is contention with preceding/following instructions)

3. SH7600 instructions

Table 8.2 Number of Instruction Stages and Execution States (cont)

Type	Category	Stage	State	Contention	Instruction	
Arithmetic instructions (cont)	Double-length multiply/accumulate instruction (SH7600 only)	9	3/(2 to 4)*2	• Multiplier contention occurs when an instruction that uses the multiplier follows a MAC instruction • MA contends with IF	MAC.L	@Rm+,@Rn+*3
	Multiplication instructions	6/7*1	1 to 3*2	• Multiplier contention occurs when an instruction that uses the multiplier follows a MUL instruction • MA contends with IF	MULS.W MULU.W	Rm,Rn Rm,Rn
	Double-length multiply/accumulate instruction (SH7600 only)	9	2 to 4*2	• Multiplier contention occurs when an instruction that uses the multiplier follows a MAC instruction • MA contends with IF	DMULS.L DMULU.L MUL.L	Rm,Rn*3 Rm,Rn*3 Rm,Rn*3
Logic operation instructions	Register-register logic operation instructions	3	1	—	AND AND NOT OR OR TST TST XOR XOR	Rm,Rn #imm,R0 Rm,Rn Rm,Rn #imm,R0 Rm,Rn #imm,R0 Rm,Rn #imm,R0

Notes 1. In the SH7600, multiply/accumulate instructions are 7 stages, multiply instructions 6 stages; in the SH7000, multiply/accumulate instructions are 8 stages, multiply instructions 7 stages

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

3. SH7600 instructions

Table 8.2 Number of Instruction Stages and Execution States (cont)

Type	Category	Stage	State	Contention	Instruction
Logic operation instructions (cont)	Memory logic operations instructions	6	3	• MA contends with IF	AND.B #imm,@(R0,GBR) OR.B #imm,@(R0,GBR) TST.B #imm,@(R0,GBR) XOR.B #imm,@(R0,GBR)
	TAS instruction	6	4	• MA contends with IF	TAS.B @Rn
Shift instructions	Shift instructions	3	1	—	ROTL Rn 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	3	3/1*4	—	BF label BT label
	Delayed conditional branch instructions (SH7600 only)	3	2/1*4	—	BF/S label*3 BT/S label*3
	Unconditional branch instructions	3	2	—	BRA label BRAF Rn*3 BSR label BSRF Rn*3 JMP @Rn JSR @Rn RTS

Notes 3. SH7600 instruction
4. One state when there is no branch

Table 8.2 Number of Instruction Stages and Execution States (cont)

Type	Category	Stage	State	Contention	Instruction	
System control instructions	System control ALU instructions	3	1	—	CLRT	
					LDC	Rm,SR
					LDC	Rm,GBR
					LDC	Rm,VBR
					LDS	Rm,PR
					NOP	
					SETT	
					STC	SR,Rn
					STC	GBR,Rn
					STC	VBR,Rn
					STS	PR,Rn
	STC.L instructions	4	2	• MA contends with IF	STC.L	SR,@-Rn
					STC.L	GBR,@-Rn
					STC.L	VBR,@-Rn
	LDS.L instructions (PR)	5	1	• Contention occurs when an instruction that uses the same destination register is placed immediately after this instruction • MA contends with IF	LDS.L	@Rm+,PR
	STS.L instruction (PR)	4	1	• MA contends with IF	STS.L	PR,@-Rn

Table 8.2 Number of Instruction Stages and Execution States (cont)

Type	Category	Stage	State	Contention	Instruction	
System control instructions (cont)	Register → MAC transfer instruction	4	1	• Contention occurs with multiplier • MA contends with IF	CLRMAC LDS LDS	 Rm,MACH Rm,MACL
	Memory → MAC transfer instructions	4	1	• Contention occurs with multiplier • MA contends with IF	LDS.L LDS.L	@Rm+,MACH @Rm+,MACL
	MAC → register transfer instruction	5	1	• Contention occurs with multiplier • Contention occurs when an instruction that uses the same destination register is placed immediately after this instruction • MA contends with IF	STS STS	MACH,Rn MACL,Rn
	MAC → memory transfer instruction	4	1	• Contention occurs with multiplier • MA contends with IF	STS.L STS.L	MACH,@-Rn MACL,@-Rn
	RTE instruction	5	4	—	RTE	
	TRAP instruction	9	8	—	TRAPA	#imm
	SLEEP instruction	3	3	—	SLEEP	

8.7.1 Data Transfer Instructions

Register-Register Transfer Instructions: Include 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			IF	ID	EX	......
......						

Figure 8.10 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: 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	MB	WB	
Next instruction		IF	ID	EX	.....	
Third instruction			IF	ID	EX	.....
......						

Figure 8.11 Memory Load Instruction Pipeline

Operation: The pipeline has five stages: IF, ID, EX, MA, and WB (figure 8.11). If an instruction that uses the same destination register as this instruction is placed immediately after it, contention will occur. (See Section 8.5, Effects of Memory Load Instructions on Pipelines.)

Memory Store Instructions: 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			IF	ID	EX	.....
......						

Figure 8.12 Memory Store Instruction Pipeline

Operation: The pipeline has four stages: IF, ID, EX, and MA (figure 8.12). Data is not returned to the register so there is no WB stage.

8.7.2 Arithmetic Instructions

Arithmetic Instructions between Registers (Except Multiplication Instructions): 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 (SH7600 only)
- 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          IF  ID  EX  .....
           ......
```

Figure 8.13 Pipeline for Arithmetic Instructions between Registers Except Multiplication Instructions

Operation: The pipeline has three stages: IF, ID, and EX (figure 8.13). The data operation is completed in the EX stage via the ALU.

Multiply/Accumulate Instruction (SH7000): Includes the following instruction type:

- MAC.W @Rm+, @Rn+

	Slot	Slot	Slot	Slot	Slot	Slot	Slot	Slot
MAC.W	IF	ID	EX	MA	MA	mm	mm	mm
Next instruction		IF	—	ID	EX	MA	WB	
Third instruction				IF	ID	EX	MA	WB
......								

Figure 8.14 Multiply/Accumulate Instruction Pipeline

Operation: The pipeline has eight stages: IF, ID, EX, MA, MA, mm, mm, and mm (figure 8.14). The second MA reads the memory and accesses the multiplier. The mm indicates that the multiplier is operating. The mm operates for three cycles after the final MA ends, regardless of slot. The ID of the instruction after the MAC.W instruction is stalled for one slot. The two MAs of the MAC.W instruction, when they contend with IF, split the slots as described in section 8.4, Contention Between Instruction Fetch (IF) and Memory Access (MA).

When an instruction that does not use the multiplier follows the MAC.W instruction, the MAC.W instruction may be considered to be 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.W instruction, 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 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 8.15).

```
                    <-> <-> <-> <-> <-> <-> <------> <-> <-> <-> : 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 8.15 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 8.16 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 8.16 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 8.17 illustrates a case of this type. This figure assumes MA and IF contention.

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 8.17 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 8.18) 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 8.18 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 8.19) to create a single slot. The MA of the STS contends with the IF. Figure 8.19 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 8.19 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 8.20) to create a single slot. The MA of the STS contends with the IF. Figure 8.20 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 8.20 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 8.21) to create a single slot. The MA of this LDS contends with IF. Figure 8.21 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 8.21 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 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 8.22) to create a single slot. The MA of the LDS contends with IF. Figure 8.22 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
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 8.22 LDS.L (Memory) Instruction Immediately After a MAC.W Instruction

Multiply/Accumulate Instruction (SH7600): Includes the following instruction type:

- MAC.W @Rm+, @Rn+

	Slot	Slot	Slot	Slot	Slot	Slot	Slot	Slot
MAC.W	IF	ID	EX	MA	MA	mm	mm	
Next instruction		IF	—	ID	EX	MA	WB	
Third instruction				IF	ID	EX	MA	WB
......								

Figure 8.23 Multiply/Accumulate Instruction Pipeline

Operation: The pipeline has seven stages: IF, ID, EX, MA, MA, mm and mm (figure 8.23). The second MA reads the memory and accesses the multiplier. The mm indicates that the multiplier is operating. The 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 one slot. The two MAs of the MAC.W instruction, when they contend with IF, split the slots as described in Section 8.4, Contention Between Instruction Fetch (IF) and Memory Access (MA).

When an instruction that does not use the multiplier follows the MAC.W instruction, the MAC.W instruction may be considered to be a five-stage pipeline 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.W instruction, 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 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 8.24 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 8.25 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 8.25 Consecutive MAC.Ws with Misalignment

When the second MA of the MAC.W instruction contends with IF, the slot will split as usual. Figure 8.26 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 8.26 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 8.27).

```
                      <-> <-> <-> <-> <-> <-> <-> <-> <-> <-> <-> : 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 8.27 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 8.28) 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 8.28 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 8.29).

```
                   ↔  ↔  ↔  ↔  ↔  ↔  ↔  ↔  ↔  ↔  ↔  ↔  ↔  ↔ : 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 8.29 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 8.30) to create a single slot. The MA of the STS contends with the IF. Figure 8.30 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 8.30 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 multiplier and writing to memory is added to the STS instruction, as described later. However, with the SH7600 series, unlike the SH7000 series, the MA of the STS does not contend with the multiplier operation (mm) when the cache is enabled. Figure 8.31 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 8.31 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 8.32) to create a single slot. The MA of this LDS contends with IF. Figure 8.32 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 8.32 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 8.33) to create a single slot. The MA of the LDS contends with IF. Figure 8.33 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 8.33 LDS.L (Memory) Instruction Immediately After a MAC.W Instruction

Double-Length Multiply/Accumulate Instruction (SH7600): Includes the following instruction type:

- MAC.L @Rm+, @Rn+ (SH7600 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 8.34 Multiply/Accumulate Instruction Pipeline

Operation: The pipeline has nine stages: IF, ID, EX, MA, MA, mm, mm, mm, and mm (figure 8.34). 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 8.4, 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 8.35) 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 8.35 MAC.L Instruction Immediately After Another MAC.L Instruction

Sometimes consecutive MACLs may have less multiplier contention even when there is misalignment of instruction execution caused by MA and IF contention. Figure 8.36 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 8.36 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 8.37 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 8.37 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 8.38) 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.

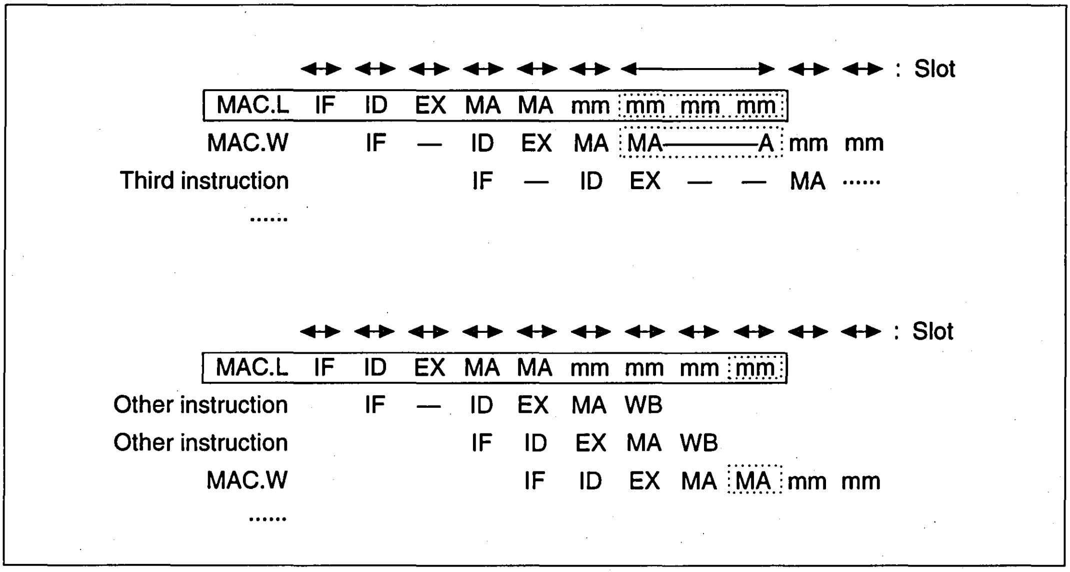


Figure 8.38 MAC.W Instruction Immediately After a MAC.L Instruction

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 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 8.39) 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 8.39 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 8.40) 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.

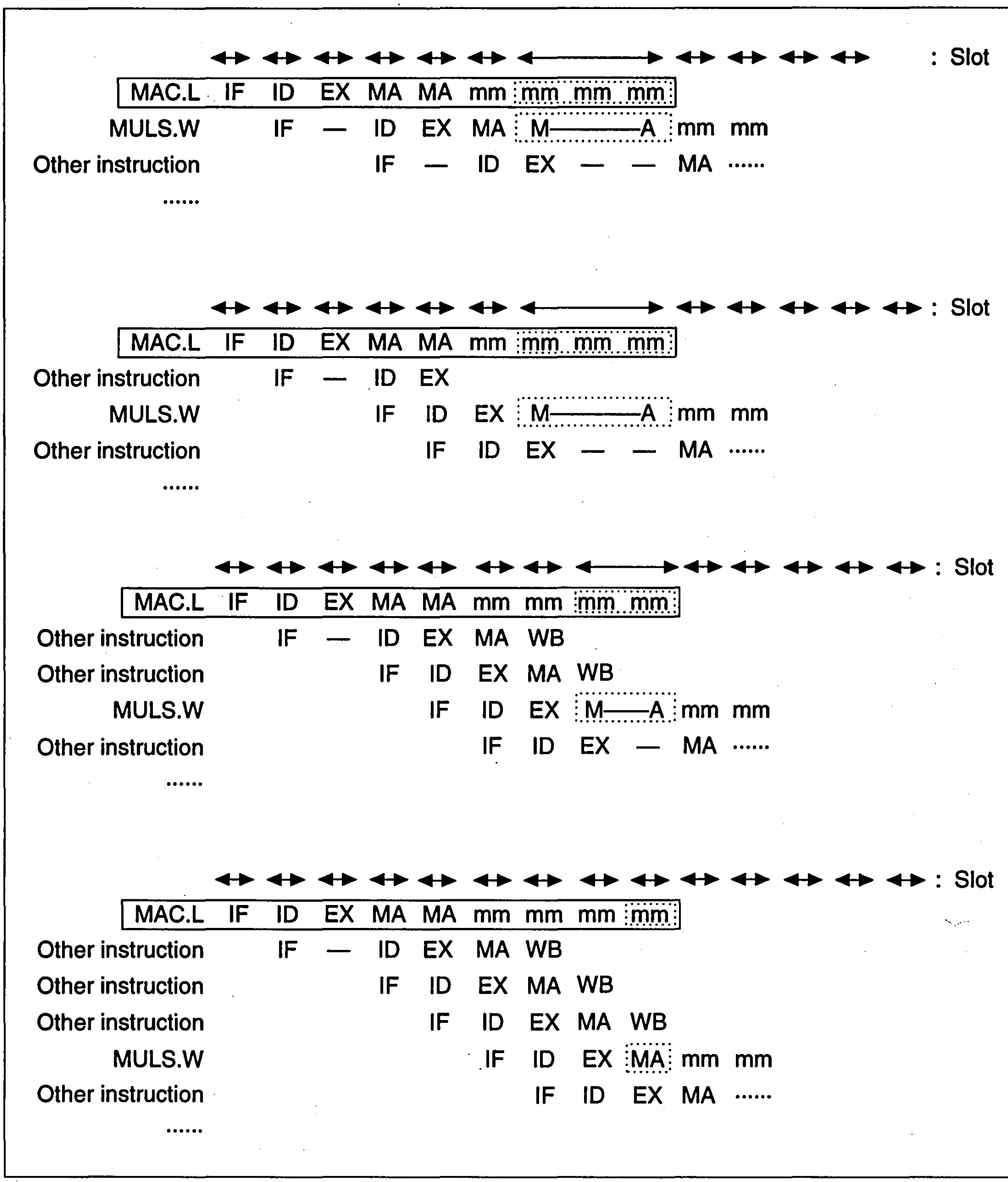


Figure 8.40 MULS.W Instruction Immediately After a MAC.L Instruction

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 8.41) to create a single slot. The MA of the STS contends with the IF. Figure 8.41 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 8.41 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. However, with the SH7600 series, unlike the SH7000 series, the MA of the STS does not contend with the multiplier operation (mm) when the cache is enabled. The MA of the STS contends with the IF. Figure 8.42 illustrates how this occurs, assuming MA and IF contention.

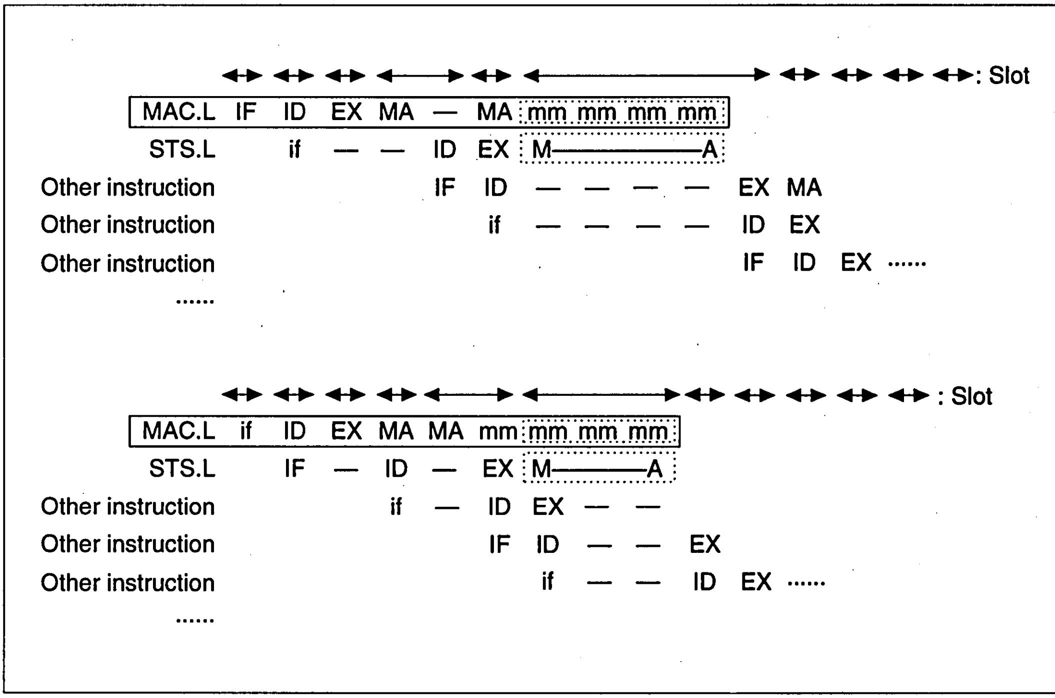


Figure 8.42 STS.L (Memory) Instruction Immediately After a MAC.L Instruction

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 8.43) to create a single slot. The MA of this LDS contends with IF. Figure 8.43 illustrates how this occurs, assuming MA and IF contention.

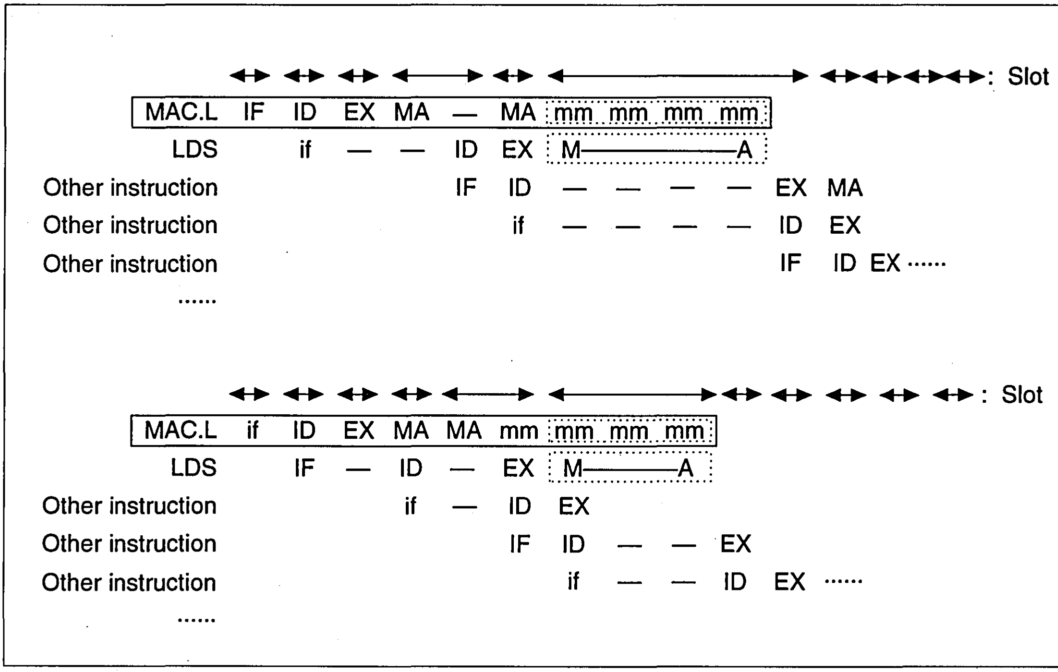


Figure 8.43 LDS (Register) Instruction Immediately After a MAC.L Instruction

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 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 8.44) to create a single slot. The MA of the LDS contends with IF. Figure 8.44 illustrates how this occurs, assuming MA and IF contention.

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 ......
......													

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 8.44 LDS.L (Memory) Instruction Immediately After a MAC.L Instruction

Multiplication Instructions (SH7000): Include the following instruction types:

- MULS.W Rm, Rn
- MULU.W Rm, Rn

	↔	↔	↔	↔	↔	↔	↔	↔	: Slot
MULS.W	IF	ID	EX	MA	mm	mm	mm		
Next instruction		IF	ID	EX	MA	WB			
Third instruction			IF	ID	EX	MA	WB		
......									

Figure 8.45 Multiplication Instruction Pipeline

Operation: The pipeline has seven stages: IF, ID, EX, MA, mm, mm, and mm (figure 8.45). The MA accesses the multiplier. The mm indicates that the multiplier is operating. The mm operates for three cycles after the MA ends, regardless of a slot. The MA of the MULS.W instruction, when it contends with IF, splits the slot as described in Section 8.4, 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 comes 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 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 8.46).

```
                    <-> <-> <-> <-> <-> <-------> <-> <-> <-> <-> <-> : 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 8.46 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 8.47) 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 8.47 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 8.48 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 8.48 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 8.49) to create a single slot. The MA of the STS contends with the IF. Figure 8.49 illustrates how this occurs, assuming MA and IF contention.

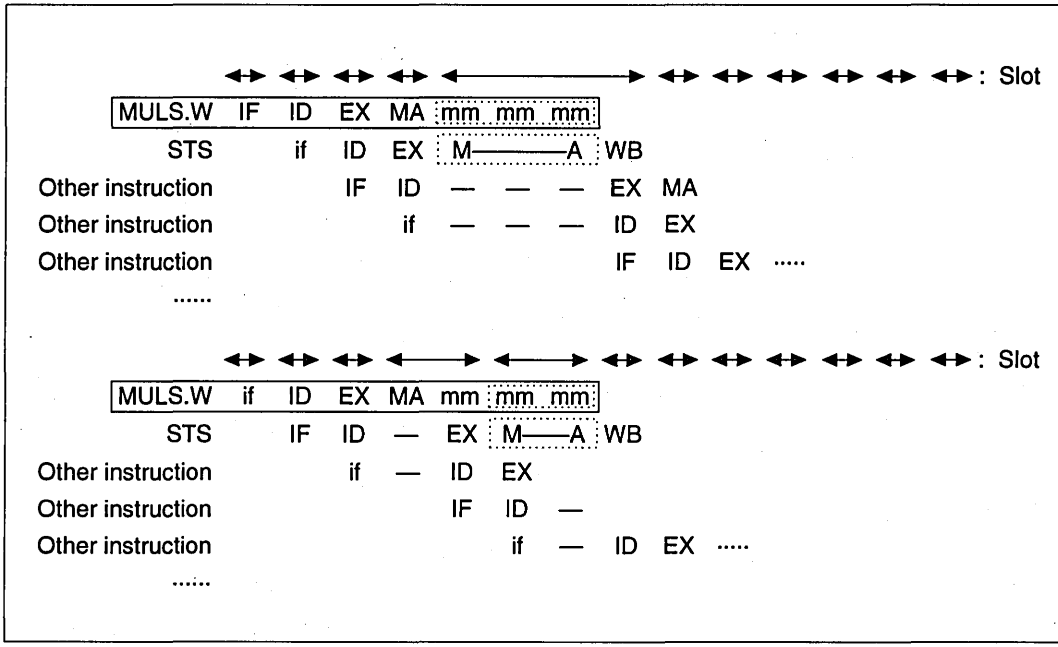


Figure 8.49 STS (Register) Instruction Immediately After a MULS.W Instruction

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 8.50) to create a single slot. The MA of the STS contends with the IF. Figure 8.50 illustrates how this occurs, assuming MA and IF contention.

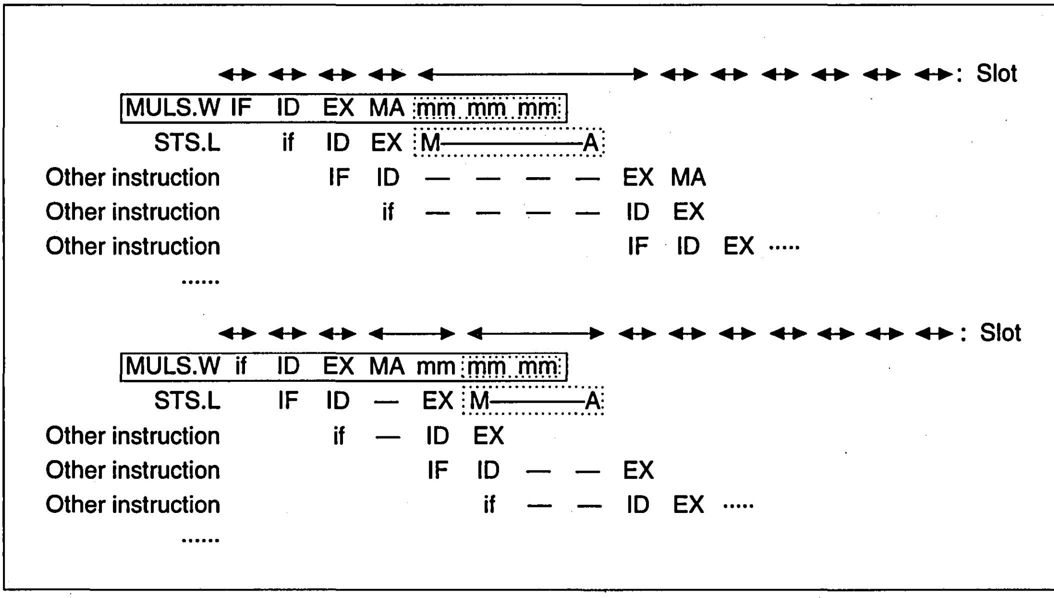


Figure 8.50 STS.L (Memory) Instruction Immediately After a MULS.W Instruction

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 8.51 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 8.51 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 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 8.52) to create a single slot. The MA of the LDS contends with IF. Figure 8.52 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 8.52 LDS.L (Memory) Instruction Immediately After a MULS.W Instruction

Multiplication Instructions (SH7600): 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 8.53 Multiplication Instruction Pipeline

Operation: The pipeline has six stages: IF, ID, EX, MA, mm, and mm (figure 8.53). 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 8.4, 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 8.54 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 8.55 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 8.56) 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 MA of MULS.W 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 8.56 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 the MA and IF will split the slot in the usual way. Figure 8.57 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 8.57 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

MULS.W instructions have an MA stage for accessing the multiplier. The MA of the MULS.W instruction does not contend with the operating multiplier (mm) of the DMULS.L 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 8.58 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 8.59) to create a single slot. The MA of the STS contends with the IF. Figure 8.59 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 8.59 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. However, with the SH7600 series, unlike the SH7000 series, the MA of the STS does not contend with the multiplier operation (mm) when the cache is enabled. The MA of the STS contends with the IF. Figure 8.60 illustrates how this occurs, assuming MA and IF contention.

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	......
......									

Slot

Figure 8.60 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 8.61 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 8.62) to create a single slot. The MA of the LDS contends with IF. Figure 8.62 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 8.62 LDS.L (Memory) Instruction Immediately After a MULS.W Instruction

Double-Length Multiplication Instructions (SH7600): Include the following instruction types:

- DMULS.L Rm, Rn (SH7600 only)
- DMULU.L Rm, Rn (SH7600 only)
- MUL.L Rm, Rn (SH7600 only)

```
                  <-> <-> <-> <-> <-> <-> <-> <-> <-> <-> <-> : Slot
         DMULS.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 8.63 Multiplication Instruction Pipeline

The pipeline has nine stages: IF, ID, EX, MA, MA, mm, mm, mm, and mm (figure 8.63). The MA accesses the multiplier. The mm indicates that the multiplier is operating. The mm operates for four cycles after the MA ends, regardless of a 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 8.4, 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 comes after the DMULS.L instruction, however, 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 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 8.64).

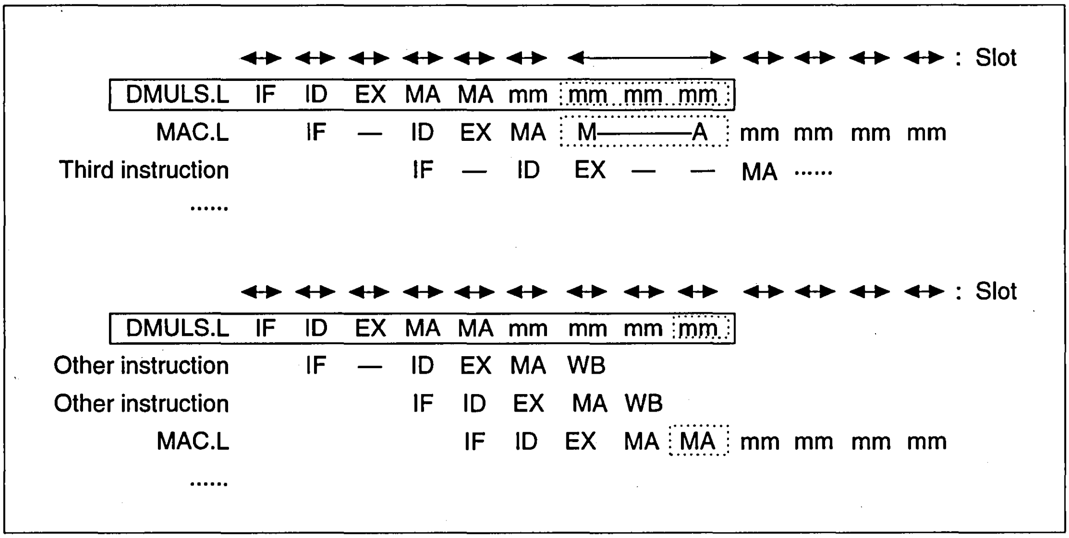


Figure 8.64 MAC.L Instruction Immediately After a DMULS.L Instruction

2. When a MAC.W instruction is located immediately after a DMULS.L 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 two or more instructions not related to the multiplier are located between the DMULS.L and MAC.W instructions, multiplier contention between the DMULS.L and MAC.W instructions does not cause stalls (figure 8.65).

```
                   <-> <-> <-> <-> <-> <-> <---------> <-> <-> <-> <-> : Slot
          DMULS.L  IF  ID  EX  MA  MA  mm  mm  mm  mm
            MAC.W      IF  —   ID  EX  MA  M---------A 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.W                  IF  ID  EX  MA  MA  mm  mm
           ......
```

Figure 8.65 MAC.W Instruction Immediately After a DMULS.L Instruction

3. When a DMULS.L instruction is located immediately after another DMULS.L instruction

DMULS.L instructions have an MA stage for accessing the multiplier. When the MA of the DMULS.L instruction contends with the operating multiplier (mm) of another DMULS.L instruction, the MA is extended until the mm ends (the M—A shown in the dotted line box in figure 8.66) to create a single slot. When two or more instructions not related to the multiplier are located between two DMULS.L instructions, contention between the DMULS.Ls does not cause stalling. When the DMULS.L MA and IF contend, the slot is split.

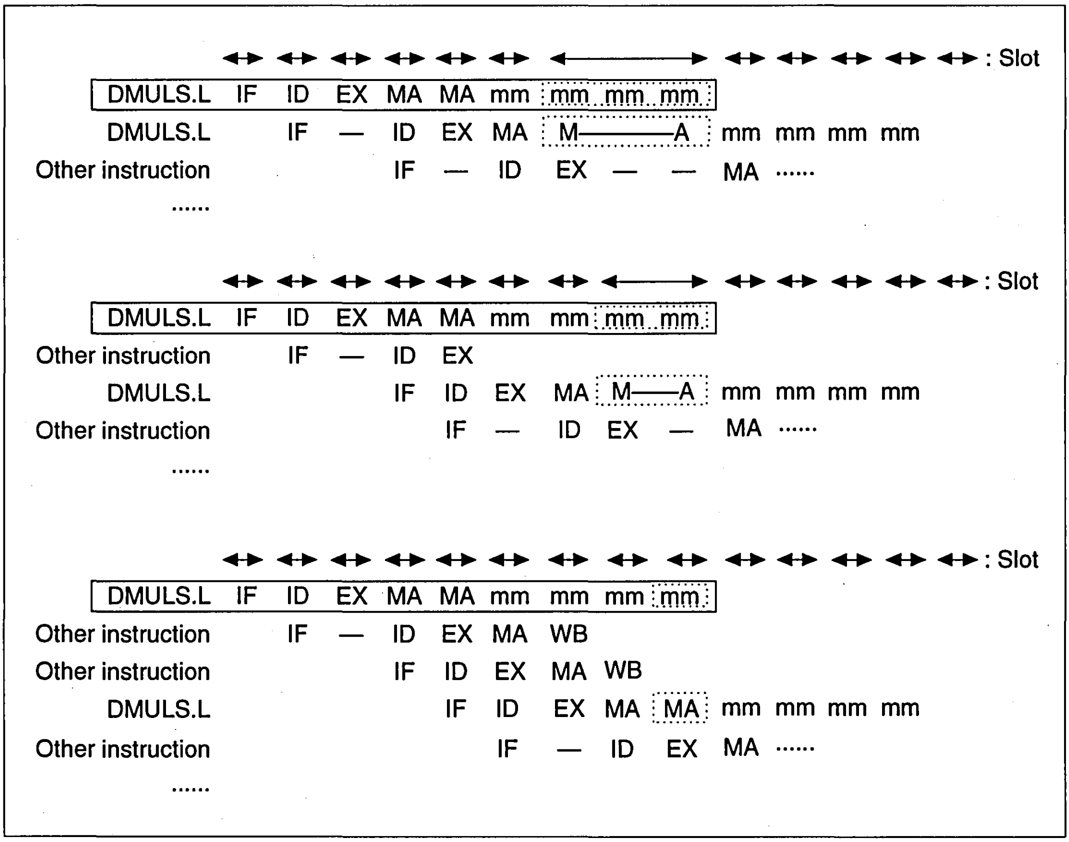


Figure 8.66 DMULS.L Instruction Immediately After Another DMULS.L Instruction

When the MA of the DMULS.L instruction is extended until the mm ends, contention between the MA and IF will split the slot in the usual way. Figure 8.67 illustrates a case of this type, assuming MA and IF contention.

														: Slot
DMULS.L	IF	ID	EX	MA	MA	—	mm	mm	mm	mm				
DMULS.L		if	—	EX	—	ID	MA	M	—	A	mm	mm	mm	mm
Other instruction						IF	ID	—	—	—	EX			
Other instruction							if	—	—	—	ID	EX	......	
Other instruction											IF	ID	......	
......														

Figure 8.67 DMULS.L Instruction Immediately After Another DMULS.L Instruction (IF and MA Contention)

4. When a MULS.W instruction is located immediately after a DMULS.L 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 a DMULS.L instruction, the MA is extended until the mm ends (the M—A shown in the dotted line box in figure 8.68) to create a single slot. When three or more instructions not related to the multiplier are located between the DMULS.L instruction and the MULS.W instruction, contention between the DMULS.L and MULS.W does not cause stalling. When the MULS.W MA and IF contend, the slot is split..

```
                  <-> <-> <-> <-> <-> <----------------> <-> <-> <-> <-> <-> : Slot
 DMULS.L          IF  ID  EX  MA  MA  mm  mm  mm  mm
 MULS.W               IF  —   ID  EX  M--------------A   mm  mm
 Other 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
 Other instruction                IF  ID  EX  MA  WB
 MULS.W                               IF  ID  EX  MA  MA  mm  mm
 Other instruction                        IF  ID  EX  MA  ......
 ......
```

Figure 8.68 MULS.W Instruction Immediately After a DMULS.L Instruction

When the MA of the DMULS.L instruction is extended until the mm ends, contention between the MA and IF will split the slot in the usual way. Figure 8.69 illustrates a case of this type, assuming MA and IF contention.

```
                  <-> <-> <-> <-----> <-----------------><-><-><-><-> : Slot
 DMULS.L          IF  ID  EX  MA  —   MA  mm  mm  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 8.69 MULS.W Instruction Immediately After a DMULS.L Instruction (IF and MA Contention)

5. When an STS (register) instruction is located immediately after a DMULS.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 8.70) to create a single slot. The MA of the STS contends with the IF. Figure 8.70 illustrates how this occurs, assuming MA and IF contention.

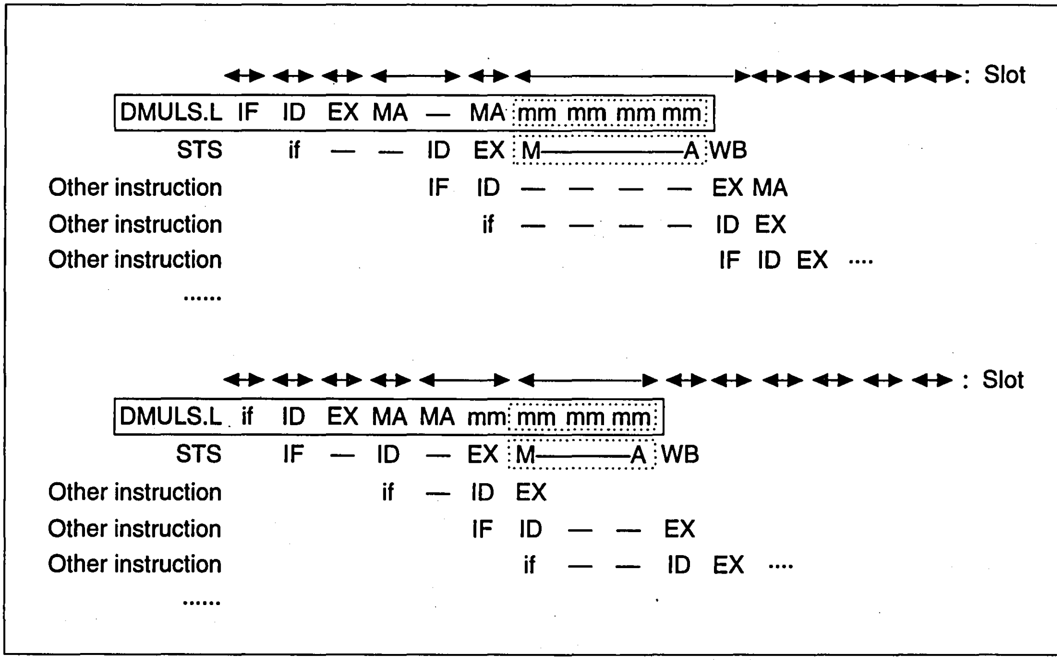


Figure 8.70 STS (Register) Instruction Immediately After a DMULS.L Instruction

6. When an STS.L (memory) instruction is located immediately after a DMULS.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. However, with the SH7600 series, unlike the SH7000 series, the MA of the STS does not contend with the multiplier operation (mm) when the cache is enabled. The MA of the STS contends with the IF. Figure 8.71 illustrates how this occurs, assuming MA and IF contention.

```
                    <-> <-> <-> <----> <-> <--------------> <-> <-> <-> <-> <->: Slot
          DMULS.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
          DMULS.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 8.71 STS.L (Memory) Instruction Immediately After a DMULS.L Instruction

7. When an LDS (register) instruction is located immediately after a DMULS.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 below) to create a single slot. The MA of this LDS contends with IF. The following figure illustrates how this occurs, assuming MA and IF contention.

```
                   <-> <-> <-> <----> <-> <--------------> <-> <-> <-> <-> <-> : Slot
          DMULS.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
          DMULS.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 8.72 LDS (Register) Instruction Immediately After a DMULS.L Instruction

8. When an LDS.L (memory) instruction is located immediately after a DMULS.L 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 8.73) to create a single slot. The MA of the LDS contends with IF. Figure 8.73 illustrates how this occurs, assuming MA and IF contention.

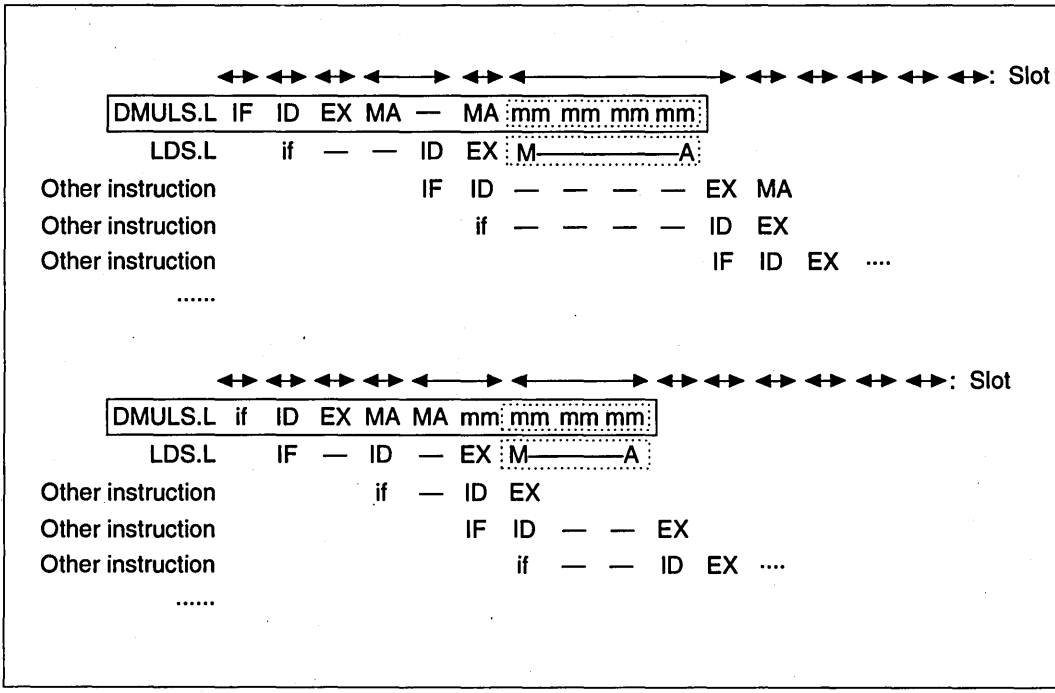


Figure 8.73 LDS.L (Memory) Instruction Immediately After a DMULS.L Instruction

8.7.3 Logic Operation Instructions

Register-Register Logic Operation Instructions: 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

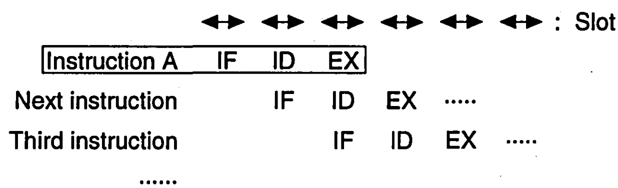


Figure 8.74 Register-Register Logic Operation Instruction Pipeline

Operation: The pipeline has three stages: IF, ID, and EX (figure 8.74). The data operation is completed in the EX stage via the ALU.

Memory Logic Operation Instructions: 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					IF	ID	EX	.....	
......									

Figure 8.75 Memory Logic Operation Instruction Pipeline

Operation: Operation: The pipeline has six stages: IF, ID, EX, MA, EX, and MA (figure 8.75). The ID of the next instruction stalls for 2 slots. The MAs of these instructions contend with IF.

TAS Instruction: Includes the following instruction type:

- TAS.B @Rn

	↔	↔	↔	↔	↔	↔	↔	↔	↔ : Slot
Instruction A	IF	ID	EX	MA	EX	MA			
Next instruction		IF	—	—	—	ID	EX	.....	
Third instruction						IF	ID	EX	.....
......									

Figure 8.76 TAS Instruction Pipeline

Operation: The pipeline has six stages: IF, ID, EX, MA, EX, and MA (figure 8.76). The ID of the next instruction stalls for 3 slots. The MA of the TAS instruction contends with IF.

8.7.4 Shift Instructions

Shift Instructions: Include the following instruction types:

- ROTL Rn
- 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

	↔	↔	↔	↔	↔	↔	↔	↔	↔ : Slot
Instruction A	IF	ID	EX						
Next instruction		IF	ID	EX	.....				
Third instruction			IF	ID	EX	.....			
......									

Figure 8.77 Shift Instruction Pipeline

Operation: The pipeline has three stages: IF, ID, and EX (figure 8.77). The data operation is completed in the EX stage via the ALU.

8.7.5 Branch Instructions

Conditional Branch Instructions: Include the following instruction types:

- BF label
- BT label

Operation: The pipeline has three stages: IF, ID, and EX. Condition verification is performed in the ID stage. Conditional branch instructions are not delayed branch.

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 8.78).

	↔	↔	↔	↔	↔	↔	↔	↔	↔ : Slot
Instruction A	IF	ID	EX						
Next instruction		IF	—		(Fetched but discarded)				
Third instruction			IF	—	(Fetched but discarded)				
Branch destination			—	IF	ID	EX	.....		
......					IF	ID	EX	.....	
......									

Figure 8.78 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 8.79).

	↔	↔	↔	↔	↔	↔	↔	↔	↔ : Slot
Instruction A	IF	ID	EX						
Next instruction		IF	ID	EX	.....				
Third instruction			IF	ID	EX	.....			
......				IF	ID	EX	.....		
......									

Figure 8.79 Branch Instruction When Condition is Not Satisfied

Delayed Conditional Branch Instructions (SH7600 only): Include the following instruction types:

- BF/S label (SH7600 only)
- BT/S label (SH7600 only)

Operation: 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 8.80).

	↔	↔	↔	↔	↔	↔	↔	↔	↔ : Slot
Instruction A	IF	ID	EX						
Next instruction		IF	ID	—	EX	MA	WB		
Third instruction			IF	—	(Fetched but discarded)				
Branch destination				IF	ID	EX	.....		
......					IF	ID	EX	.....	

Figure 8.80 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 8.81).

	↔	↔	↔	↔	↔	↔	↔	↔	↔ : Slot
Instruction A	IF	ID	EX						
Next instruction		IF	ID	EX	.....				
Third instruction			IF	ID	EX	.....			
......				IF	ID	EX	.....		
......									

Figure 8.81 Branch Instruction When Condition is Not Satisfied

Unconditional Branch Instructions: Include the following instruction types:

- BRA label
- BRAF Rn (SH7600 only)
- BSR label
- BSRF Rn (SH7600 only)
- JMP @Rn
- JSR @Rn
- RTS

	↔	↔	↔	↔	↔	↔	↔	↔	↔ : Slot
Instruction A	IF	ID	EX						
Delay slot		IF	—	ID	EX	MA	WB		
Branch destination				IF	ID	EX	.....		
.....					IF	ID	EX	.....	
.....									

Figure 8.82 Unconditional Branch Instruction Pipeline

Operation: The pipeline has three stages: IF, ID, and EX (figure 8.82). Unconditional branch instructions are delayed branch. 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 fetched and not discarded as the conditional branch instructions are, but is then executed. Note that the ID slot of the delay slot instruction does stall for one cycle. The branch destination instruction starts its fetch from the slot after the slot that has the EX stage of instruction A.

8.7.6 System Control Instructions

System Control ALU Instructions: Include the following instruction types:

- CLRT
- LDC Rm, SR
- LDC Rm, GBR
- LDC Rm, VBR
- LDS Rm, PR
- NOP
- SETT
- STC SR, Rn
- STC GBR, Rn
- STC VBR, Rn
- STS PR, Rn

	↔	↔	↔	↔	↔	↔	↔	↔	↔ : Slot
Instruction A	IF	ID	EX						
Next instruction		IF	ID	EX	.....				
Third instruction			IF	ID	EX	.....			
......									

Figure 8.83 System Control ALU Instruction Pipeline

Operation: The pipeline has three stages: IF, ID, and EX (figure 8.83). The data operation is completed in the EX stage via the ALU.

LDC.L Instructions: Include the following instruction types:

- LDC.L @Rm+, SR
- LDC.L @Rm+, GBR
- LDC.L @Rm+, VBR

	↔	↔	↔	↔	↔	↔	↔	↔	↔ : Slot
Instruction A	IF	ID	EX	MA	EX				
Next instruction		IF	—	—	ID	EX	.....		
Third instruction					IF	ID	EX	.....	
......									

Figure 8.84 LDC.L Instruction Pipeline

Operation: The pipeline has five stages: IF, ID, EX, MA, and EX (figure 8.84). The ID of the following instruction is stalled for two slots.

STC.L Instructions: Include the following instruction types:

- STC.L SR, @–Rn
- STC.L GBR, @–Rn
- STC.L VBR, @–Rn

	↔	↔	↔	↔	↔	↔	↔	↔	↔ : Slot
Instruction A	IF	ID	EX	MA					
Next instruction		IF	—	ID	EX	.....			
Third instruction				IF	ID	EX	.....		
......									

Figure 8.85 STC.L Instruction Pipeline

Operation: The pipeline has four stages: IF, ID, EX, and MA (figure 8.85). The ID of the next instruction is stalled for one slot.

LDS.L Instruction (PR): Includes the following instruction type:

- LDS.L @Rm+, PR

	↔	↔	↔	↔	↔	↔	↔	↔	↔	: Slot
Instruction A	IF	ID	EX	MA	WB					
Next instruction		IF	ID	EX	·····					
Third instruction			IF	ID	EX	·····				
······										

Figure 8.86 LDS.L Instruction (PR) Pipeline

Operation: The pipeline has five stages: IF, ID, EX, MA, and WB (figure 8.86). It is the same as an ordinary load instruction.

STS.L Instruction (PR): Includes the following instruction type:

- STS.L PR, @–Rn

	◄►	◄►	◄►	◄►	◄►	◄►	◄►	◄►	◄►	: Slot
Instruction A	IF	ID	EX	MA						
Next instruction		IF	ID	EX	·····					
Third instruction			IF	ID	EX	·····				
······										

Figure 8.87 STS.L Instruction (PR) Pipeline

Operation: The pipeline has four stages: IF, ID, EX, and MA (figure 8.87). It is the same as an ordinary store instruction.

Register → MAC Transfer Instructions: Include the following instruction types:

- CLRMAC
- LDS Rm, MACH
- LDS Rm, MACL

	↔	↔	↔	↔	↔	↔	↔	↔	↔ : Slot
Instruction A	IF	ID	EX	MA					
Next instruction		IF	ID	EX	·····				
Third instruction			IF	ID	EX	·····			
······									

Figure 8.88 Register → MAC Transfer Instruction Pipeline

Operation: The pipeline has four stages: IF, ID, EX, and MA (figure 8.88). The MA is a stage for accessing the multiplier. The MA contends with the IF. This makes it the same as ordinary store instructions. Since the multiplier contends with the MA, see the section for the MAC and MUL instructions.

Memory → MAC Transfer Instructions: Include the following instruction types:

- LDS.L @Rm+, MACH
- LDS.L @Rm+, MACL

	↔	↔	↔	↔	↔	↔	↔	↔	↔ : Slot
Instruction A	IF	ID	EX	MA					
Next instruction		IF	ID	EX	·····				
Third instruction			IF	ID	EX	·····			
······									

Figure 8.89 Memory → MAC Transfer Instruction Pipeline

Operation: The pipeline has four stages: IF, ID, EX, and MA (figure 8.89). The MA contends with the IF. The MA is a stage for memory access and multiplier access. This makes it the same as ordinary load instructions. Since the multiplier contends with the MA, see the section for the MAC and MUL instructions.

MAC → Register Transfer Instructions: Include the following instruction types:

- STS MACH, Rn
- STS MACL, Rn

	↔	↔	↔	↔	↔	↔	↔	↔	↔	: Slot
Instruction A	IF	ID	EX	MA	WB					
Next instruction		IF	ID	EX	.....					
Third instruction			IF	ID	EX	.....				
......										

Figure 8.90 MAC → Register Transfer Instruction Pipeline

Operation: The pipeline has five stages: IF, ID, EX, MA, and WB (figure 8.90). The MA is a stage for accessing the multiplier. The MA contends with the IF. This makes it the same as ordinary load instructions. Since the multiplier contends with the MA, see the section for the MAC and MUL instructions.

MAC → Memory Transfer Instructions: Include the following instruction types:

- STS.L MACH, @–Rn
- STS.L MACL, @–Rn

	↔	↔	↔	↔	↔	↔	↔	↔	↔ : Slot
Instruction A	IF	ID	EX	MA	WB				
Next instruction		IF	ID	EX	.....				
Third instruction			IF	ID	EX	.....			
......									

Figure 8.91 MAC → Memory Transfer Instruction Pipeline

Operation: The pipeline has four stages: IF, ID, EX, and MA (figure 8.91). The MA is a stage for accessing the multiplier. The MA contends with IF. This makes it the same as ordinary store instructions. Since the multiplier contends with the MA, see the section for the MAC and MUL instructions.

RTE Instruction: Includes the following instruction type:

- RTE

	↔	↔	↔	↔	↔	↔	↔	↔	↔	: Slot
RTE	IF	ID	EX	MA	MA					
Delay slot		IF	—	—	—	ID	EX	.....		
Branch destination						IF	ID	EX	.....	
.....										

Figure 8.92 RTE Instruction Pipeline

The pipeline has five stages: IF, ID, EX, MA, and MA (figure 8.92). The MAs contend with the IF. The RTE is a delayed branch instruction. The ID of the delay slot instruction is stalled for 3 slots. The IF of the branch destination instruction starts from the slot following the MA of the RTE.

TRAP Instruction: Includes the following instruction type:

- TRAPA #imm

	↔	↔	↔	↔	↔	↔	↔	↔	↔	↔	↔	↔	↔	: Slot
TRAPA	IF	ID	EX	EX	MA	MA	MA	EX	EX					
Next instruction		IF												
Third instruction			IF											
Branch destination									IF	ID	EX	.....		
......										IF	ID	EX	.....	

Figure 8.93 TRAP Instruction Pipeline

The pipeline has nine stages: IF, ID, EX, EX, MA, MA, MA, EX, and EX (figure 8.93). The MAs contend with the IF. The 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: Includes the following instruction type:

- SLEEP

```
                  ↔  ↔  ↔  ↔  ↔  ↔  ↔  ↔  ↔ : Slot
           |SLEEP  IF  ID  EX|
Next instruction       IF
              .....
```

Figure 8.94 SLEEP Instruction Pipeline

Operation: The pipeline has three stages: IF, ID and EX (figure 8.94). It is issued until the IF of the next instruction. After the SLEEP instruction is executed, the CPU enters sleep mode or standby mode.

8.7.7 Exception Processing

Interrupt Exception Processing: Includes the following instruction type:

- Interrupt exception processing

	↔	↔	↔	↔	↔	↔	↔	↔	↔	↔	↔	↔	↔ : Slot
Interrupt	IF	ID	EX	EX	MA	MA	EX	MA	EX	EX			
Next instruction		IF											
Branch destination										IF	ID	EX	.....
......											IF	ID	.....

Figure 8.95 Interrupt Exception Processing Pipeline

Operation: 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 8.95). 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, and on-chip peripheral module interrupts.

Address Error Exception Processing: Includes the following instruction type:

- Address error exception processing

														: Slot
Interrupt	IF	ID	EX	EX	MA	MA	EX	MA	EX	EX				
Next instruction		IF												
Branch destination										IF	ID	EX	.....	
....											IF	ID	.....	

Figure 8.96 Address Error Exception Processing Pipeline

Operation: 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 8.96). 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. Fetching an instruction from an odd address or fetching an instruction from an on-chip peripheral register causes an instruction fetch address error. Accessing word data from other than a word boundary, accessing longword data from other than a longword boundary, and accessing an on-chip peripheral register 8-bit space by longword cause a read or write address error.

Illegal Instruction Exception Processing: Includes the following instruction type:

- Illegal instruction exception processing

	Slot	Slot	Slot	Slot	Slot	Slot	Slot	Slot	Slot	Slot	Slot	Slot	Slot
Illegal instruction	IF	ID	EX	EX	MA	MA	MA	EX	EX				
Next instruction		IF											
(Third instruction			IF)										
Branch destination									IF	ID	EX	.....	
......										IF	ID	.....	

Figure 8.97 Illegal Instruction Exception Processing Pipeline

Operation: 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 8.97). Illegal instruction exception processing is not a delayed branch. In illegal instruction exception processing, an overrun fetch (IF) occurs. Whether there is an IF only in the next instruction or in the one after that as well depends on the instruction that was to be executed. In branch destination instructions, the IF starts from the slot that has the final EX in the illegal instruction exception processing.

Illegal instruction exception processing is caused by ordinary illegal instructions and by illegal slot instructions. 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.

Appendix A Instruction Code

See "6. Instruction Descriptions" for details.

A.1 Instruction Set by Addressing Mode

Table A.1 lists instruction codes and execution states by addressing modes.

Table A.1 Instruction Set by Addressing Mode

Addressing Mode	Category	Sample Instruction		Types SH 7600	Types SH 7000
No operand	—	NOP		8	8
Direct register addressing	Destination operand only	MOVT	Rn	18	17
	Source and destination operand	ADD	Rm,Rn	34	31
	Load and store with control register or system register	LDC STS	Rm,SR MACH,Rn	12	12
Indirect register addressing	Destination operand only	JMP	@Rn	3	3
	Data transfer with direct register addressing	MOV.L	Rm,@Rn	6	6
Post increment indirect register addressing	Multiply/accumulate operation	MAC.W	@Rm+,@Rn+	2	1
	Data transfer from direct register addressing	MOV.L	@Rm+,Rn	3	3
	Load to control register or system register	LDC.L	@Rm+,SR	6	6
Pre decrement indirect register addressing	Data transfer from direct register addressing	MOV.L	Rm,@-Rn	3	3
	Store from control register or system register	STC.L	SR,@-Rn	6	6
Indirect register addressing with displacement	Data transfer with direct register addressing	MOV.L	Rm,@(disp,Rn)	6	6
Indirect indexed register addressing	Data transfer with direct register addressing	MOV.L	Rm,@(R0,Rn)	6	6
Indirect GBR addressing with displacement	Data transfer with direct register addressing	MOV.L	R,@(disp,GBR)	6	6
Indirect indexed GBR addressing	Immediate data transfer	AND.B	#imm,@(R0,GBR)	4	4
PC relative addressing with displacement	Data transfer to direct register addressing	MOV.L	@(disp,PC),Rn	3	3
PC relative addressing with Rn	Branch instruction	BRAF	Rn	2	0
PC relative addressing	Branch instruction	BRA	label	6	4
Immediate addressing	Arithmetic logical operations with direct register addressing	ADD	#imm,Rn	7	7
	Specify exception processing vector	TRAPA	#imm	1	1

	Total:	142	133

A.1.1 No Operand

Table A.2 No Operand

Instruction	Code	Operation	State	T Bit
CLRT	0000000000001000	0 → T	1	0
CLRMAC	0000000000101000	0 → MACH, MACL	1	—
DIV0U	0000000000011001	0 → M/Q/T	1	0
NOP	0000000000001001	No operation	1	—
RTE	0000000000101011	Delayed branch, Stack area → PC/SR	4	LSB
RTS	0000000000001011	Delayed branch, PR → PC	2	—
SETT	0000000000011000	1 → T	1	1
SLEEP	0000000000011011	Sleep	3	—

A.1.2 Direct Register Addressing

Table A.3 Destination Operand Only

Instruction		Code	Operation	State	T Bit
CMP/PL	Rn	0100nnnn00010101	Rn > 0, 1 → T	1	Comparison result
CMP/PZ	Rn	0100nnnn00010001	Rn ≥ 0, 1 → T	1	Comparison result
DT	Rn*	0100nnnn00010000	Rn – 1 → Rn When Rn is 0, 1 → T, when Rn is nonzero, 0 → T	1	Comparison result
MOVT	Rn	0000nnnn00101001	T → Rn	1	—
ROTL	Rn	0100nnnn00000100	T ← Rn ← MSB	1	MSB
ROTR	Rn	0100nnnn00000101	LSB → Rn → T	1	LSB
ROTCL	Rn	0100nnnn00100100	T ← Rn ← T	1	MSB
ROTCR	Rn	0100nnnn00100101	T → Rn → T	1	LSB
SHAL	Rn	0100nnnn00100000	T ← Rn ← 0	1	MSB
SHAR	Rn	0100nnnn00100001	MSB → Rn → T	1	LSB
SHLL	Rn	0100nnnn00000000	T ← Rn ← 0	1	MSB
SHLR	Rn	0100nnnn00000001	0 → Rn → T	1	LSB
SHLL2	Rn	0100nnnn00001000	Rn<<2 → Rn	1	—
SHLR2	Rn	0100nnnn00001001	Rn>>2 → Rn	1	—
SHLL8	Rn	0100nnnn00011000	Rn<<8 → Rn	1	—
SHLR8	Rn	0100nnnn00011001	Rn>>8 → Rn	1	—
SHLL16	Rn	0100nnnn00101000	Rn<<16 → Rn	1	—
SHLR16	Rn	0100nnnn00101001	Rn>>16 → Rn	1	—

Note: SH7600 instruction

Table A.4 Source and Destination Operand

Instruction		Code	Operation	State	T Bit
ADD	Rm,Rn	0011nnnnmmmm1100	Rn + Rm → Rn	1	—
ADDC	Rm,Rn	0011nnnnmmmm1110	Rn + Rm + T → Rn, carry → T	1	Carry
ADDV	Rm,Rn	0011nnnnmmmm1111	Rn + Rm → Rn, overflow → T	1	Overflow
AND	Rm,Rn	0010nnnnmmmm1001	Rn & Rm → Rn	1	—

Table A.4 Source and Destination Operand (cont)

Instruction		Code	Operation	State	T Bit
CMP/EQ	Rm,Rn	0011nnnnmmmm0000	When Rn = Rm, 1 → T	1	Comparison result
CMP/HS	Rm,Rn	0011nnnnmmmm0010	When unsigned and Rn ≥ Rm, 1 → T	1	Comparison result
CMP/GE	Rm,Rn	0011nnnnmmmm0011	When signed and Rn ≥ Rm, 1 → T	1	Comparison result
CMP/HI	Rm,Rn	0011nnnnmmmm0110	When unsigned and Rn > Rm, 1 → T	1	Comparison result
CMP/GT	Rm,Rn	0011nnnnmmmm0111	When signed and Rn > Rm, 1 → T	1	Comparison result
CMP/STR	Rm,Rn	0010nnnnmmmm1100	When a byte in Rn equals bytes in Rm, 1 → T	1	Comparison result
DIV1	Rm,Rn	0011nnnnmmmm0100	1-step division (Rn ÷ Rm)	1	Calculation result
DIV0S	Rm,Rn	0010nnnnmmmm0111	MSB of Rn → Q, MSB of Rm → M, M ^ Q → T	1	Calculation result
DMULS.L	Rm,Rn*2	0011nnnnmmmm1101	Signed, Rn × Rm → MACH, MACL	2 to 4*1	—
DMULU.L	Rm,Rn*2	0011nnnnmmmm0101	Unsigned, Rn × Rm → MACH, MACL	2 to 4*1	—
EXTS.B	Rm,Rn	0110nnnnmmmm1110	Sign – extends Rm from byte → Rn	1	—
EXTS.W	Rm,Rn	0110nnnnmmmm1111	Sign – extends Rm from word → Rn	1	—
EXTU.B	Rm,Rn	0110nnnnmmmm1100	Zero – extends Rm from byte → Rn	1	—
EXTU.W	Rm,Rn	0110nnnnmmmm1101	Zero – extends Rm from word → Rn	1	—
MOV	Rm,Rn	0110nnnnmmmm0011	Rm → Rn	1	—
MUL.L	Rm,Rn*2	0000nnnnmmmm0111	Rn × Rm → MACL	2 to 4*1	—
MULS.W	Rm,Rn	0010nnnnmmmm1111	Signed, Rn × Rm → MAC	1 to 3*1	—
MULU.W	Rm,Rn	0010nnnnmmmm1110	Unsigned, Rn × Rm → MAC	1 to 3*1	—
NEG	Rm,Rn	0110nnnnmmmm1011	0 – Rm → Rn	1	—
NEGC	Rm,Rn	0110nnnnmmmm1010	0 – Rm – T → Rn, Borrow → T	1	Borrow

Notes: 1. The normal minimum number of execution states
2. SH7600 instruction

Table A.4 Source and Destination Operand (cont)

Instruction		Code	Operation	State	T Bit
NOT	Rm,Rn	0110nnnnmmmm0111	~Rm → Rn	1	—
OR	Rm,Rn	0010nnnnmmmm1011	Rn I Rm → Rn	1	—
SUB	Rm,Rn	0011nnnnmmmm1000	Rn – Rm → Rn	1	—
SUBC	Rm,Rn	0011nnnnmmmm1010	Rn – Rm – T → Rn, Borrow → T	1	Borrow
SUBV	Rm,Rn	0011nnnnmmmm1011	Rn – Rm → Rn, Underflow → T	1	Underflow
SWAP.B	Rm,Rn	0110nnnnmmmm1000	Rm → Swap upper and lower halves of lower 2 bytes → Rn	1	—
SWAP.W	Rm,Rn	0110nnnnmmmm1001	Rm → Swap upper and lower word → Rn	1	—
TST	Rm,Rn	0010nnnnmmmm1000	Rn & Rm, when result is 0, 1 →T	1	Test results
XOR	Rm,Rn	0010nnnnmmmm1010	Rn ^ Rm → Rn	1	—
XTRCT	Rm,Rn	0010nnnnmmmm1101	Center 32 bits of Rm and Rn → Rn	1	—

Table A.5 Load and Store with Control Register or System Register

Instruction		Code	Operation	State	T Bit
LDC	Rm,SR	0100mmmm00001110	Rm → SR	1	LSB
LDC	Rm,GBR	0100mmmm00011110	Rm → GBR	1	—
LDC	Rm,VBR	0100mmmm00101110	Rm → VBR	1	—
LDS	Rm,MACH	0100mmmm00001010	Rm → MACH	1	—
LDS	Rm,MACL	0100mmmm00011010	Rm → MACL	1	—
LDS	Rm,PR	0100mmmm00101010	Rm → PR	1	—
STC	SR,Rn	0000nnnn00000010	SR → Rn	1	—
STC	GBR,Rn	0000nnnn00010010	GBR → Rn	1	—
STC	VBR,Rn	0000nnnn00100010	VBR → Rn	1	—
STS	MACH,Rn	0000nnnn00001010	MACH → Rn	1	—
STS	MACL,Rn	0000nnnn00011010	MACL → Rn	1	—
STS	PR,Rn	0000nnnn00101010	PR → Rn	1	—

A.1.3 Indirect Register Addressing

Table A.6 Destination Operand Only

Instruction	Code	Operation	State	T Bit
JMP @Rn	0100nnnn00101011	Delayed branch, Rn → PC	2	—
JSR @Rn	0100nnnn00001011	Delayed branch, PC → PR, Rn → PC	2	—
TAS.B @Rn	0100nnnn00011011	When (Rn) is 0, 1 → T, 1 → MSB of (Rn)	4	Test results

Table A.7 Data Transfer with Direct Register Addressing

Instruction	Code	Operation	State	T Bit
MOV.B Rm,@Rn	0010nnnnmmmm0000	Rm → (Rn)	1	—
MOV.W Rm,@Rn	0010nnnnmmmm0001	Rm → (Rn)	1	—
MOV.L Rm,@Rn	0010nnnnmmmm0010	Rm → (Rn)	1	—
MOV.B @Rm,Rn	0110nnnnmmmm0000	(Rm) → sign extension → Rn	1	—
MOV.W @Rm,Rn	0110nnnnmmmm0001	(Rm) → sign extension → Rn	1	—
MOV.L @Rm,Rn	0110nnnnmmmm0010	(Rm) → Rn	1	—

A.1.4 Post Increment Indirect Register Addressing

Table A.8 Multiply/Accumulate Operation

Instruction	Code	Operation	State	T Bit
MAC.L @Rm+,@Rn+*2	0000nnnnmmmm1111	Signed, (Rn) × (Rm) + MAC → MAC	3/(2 to 4)*1	—
MAC.W @Rm+,@Rn+	0100nnnnmmmm1111	Signed, (Rn) × (Rm) + MAC → MAC	3/(2)*1	—

Notes: 1. The normal minimum number of execution states (The number in parentheses is the number of states when there is contention with preceding/following instructions).

2. SH7600 instruction

Table A.9 Data Transfer from Direct Register Addressing

Instruction	Code	Operation	State	T Bit
MOV.B @Rm+,Rn	0110nnnnmmmm0100	(Rm) → sign extension → Rn, Rm + 1 → Rm	1	—
MOV.W @Rm+,Rn	0110nnnnmmmm0101	(Rm) → sign extension → Rn, Rm + 2 → Rm	1	—
MOV.L @Rm+,Rn	0110nnnnmmmm0110	(Rm) → Rn, Rm + 4 → Rm	1	—

Table A.10 Load to Control Register or System Register

Instruction	Code	Operation	State	T Bit
LDC.L @Rm+,SR	0100mmmm00000111	(Rm) → SR, Rm + 4 → Rm	3	LSB
LDC.L @Rm+,GBR	0100mmmm00010111	(Rm) → GBR, Rm + 4 → Rm	3	—
LDC.L @Rm+,VBR	0100mmmm00100111	(Rm) → VBR, Rm + 4 → Rm	3	—
LDS.L @Rm+,MACH	0100mmmm00000110	(Rm) → MACH, Rm + 4 → Rm	1	—
LDS.L @Rm+,MACL	0100mmmm00010110	(Rm) → MACL, Rm + 4 → Rm	1	—
LDS.L @Rm+,PR	0100mmmm00100110	(Rm) → PR, Rm + 4 → Rm	1	—

A.1.5 Pre Decrement Indirect Register Addressing

Table A.11 Data Transfer from Direct Register Addressing

Instruction	Code	Operation	State	T Bit
MOV.B Rm,@-Rn	0010nnnnmmmm0100	Rn – 1 → Rn, Rm → (Rn)	1	—
MOV.W Rm,@-Rn	0010nnnnmmmm0101	Rn – 2 → Rn, Rm → (Rn)	1	—
MOV.L Rm,@-Rn	0010nnnnmmmm0110	Rn – 4 → Rn, Rm → (Rn)	1	—

Table A.12 Store from Control Register or System Register

Instruction	Code	Operation	State	T Bit
STC.L SR,@-Rn	0100nnnn00000011	Rn – 4 → Rn, SR → (Rn)	2	—
STC.L GBR,@-Rn	0100nnnn00010011	Rn – 4 → Rn, GBR → (Rn)	2	—
STC.L VBR,@-Rn	0100nnnn00100011	Rn – 4 → Rn, VBR → (Rn)	2	—
STS.L MACH,@-Rn	0100nnnn00000010	Rn – 4 → Rn, MACH → (Rn)	1	—
STS.L MACL,@-Rn	0100nnnn00010010	Rn – 4 → Rn, MACL → (Rn)	1	—
STS.L PR,@-Rn	0100nnnn00100010	Rn – 4 → Rn, PR → (Rn)	1	—

A.1.6 Indirect Register Addressing with Displacement

Table A.13 Indirect Register Addressing with Displacement

Instruction	Code	Operation	State	T Bit
MOV.B R0,@(disp,Rn)	10000000nnnndddd	R0 → (disp + Rn)	1	—
MOV.W R0,@(disp,Rn)	10000001nnnndddd	R0 → (disp × 2 + Rn)	1	—
MOV.L Rm,@(disp,Rn)	0001nnnnmmmmdddd	Rm → (disp × 4 + Rn)	1	—
MOV.B @(disp,Rm),R0	10000100mmmmdddd	(disp + Rm) → sign extension → R0	1	—
MOV.W @(disp,Rm),R0	10000101mmmmdddd	(disp × 2 + Rm) → sign extension → R0	1	—
MOV.L @(disp,Rm),Rn	0101nnnnmmmmdddd	(disp × 4 + Rm) → Rn	1	—

A.1.7 Indirect Indexed Register Addressing

Table A.14 Indirect Indexed Register Addressing

Instruction	Code	Operation	State	T Bit
MOV.B Rm,@(R0,Rn)	0000nnnnmmmm0100	Rm → (R0 + Rn)	1	—
MOV.W Rm,@(R0,Rn)	0000nnnnmmmm0101	Rm → (R0 + Rn)	1	—
MOV.L Rm,@(R0,Rn)	0000nnnnmmmm0110	Rm → (R0 + Rn)	1	—
MOV.B @(R0,Rm),Rn	0000nnnnmmmm1100	(R0 + Rm) → sign extension → Rn	1	—
MOV.W @(R0,Rm),Rn	0000nnnnmmmm1101	(R0 + Rm) → sign extension → Rn	1	—
MOV.L @(R0,Rm),Rn	0000nnnnmmmm1110	(R0 + Rm) → Rn	1	—

A.1.8 Indirect GBR Addressing with Displacement

Table A.15 Indirect GBR Addressing with Displacement

Instruction		Code	Operation	State	T Bit
MOV.B	R0,@(disp,GBR)	11000000dddddddd	R0 → (disp + GBR)	1	—
MOV.W	R0,@(disp,GBR)	11000001dddddddd	R0 → (disp × 2 + GBR)	1	—
MOV.L	R0,@(disp,GBR)	11000010dddddddd	R0 → (disp × 4 + GBR)	1	—
MOV.B	@(disp,GBR),R0	11000100dddddddd	(disp + GBR) → sign extension → R0	1	—
MOV.W	@(disp,GBR),R0	11000101dddddddd	(disp × 2 + GBR) → sign extension → R0	1	—
MOV.L	@(disp,GBR),R0	11000110dddddddd	(disp × 4 + GBR) → R0	1	—

A.1.9 Indirect Indexed GBR Addressing

Table A.16 Indirect Indexed GBR Addressing

Instruction		Code	Operation	State	T Bit
AND.B	#imm,@(R0,GBR)	11001101iiiiiiii	(R0 + GBR) & imm → (R0 + GBR)	3	—
OR.B	#imm,@(R0,GBR)	11001111iiiiiiii	(R0 + GBR) I imm → (R0 + GBR)	3	—
TST.B	#imm,@(R0,GBR)	11001100iiiiiiii	(R0 + GBR) & imm, when result is 0, 1 → T	3	Test results
XOR.B	#imm,@(R0,GBR)	11001110iiiiiiii	(R0 + GBR) ^ imm → (R0 + GBR)	3	—

A.1.10 PC Relative Addressing with Displacement

Table A.17 PC Relative Addressing with Displacement

Instruction		Code	Operation	State	T Bit
MOV.W	@(disp,PC),Rn	1001nnnndddddddd	(disp × 2 + PC) → sign extension → Rn	1	—
MOV.L	@(disp,PC),Rn	1101nnnndddddddd	(disp × 4 + PC) → Rn	1	—
MOVA	@(disp,PC),R0	11000111dddddddd	disp ×4 + PC → R0	1	—

A.1.11 PC Relative Addressing with Rn

Table A.18 PC Relative Addressing with Rn

Instruction		Code	Operation	State	T Bit
BRAF	Rn*2	0000nnnn00100011	Delayed branch, Rn + PC → PC	2	—
BSRF	Rn*2	0000nnnn00000011	Delayed branch, PC → PR, Rn + PC → PC	2	—

Notes: 2. SH7600 instruction

A.1.12 PC Relative Addressing

Table A.19 PC Relative Addressing

Instruction		Code	Operation	State	T Bit
BF	label	10001011dddddddd	When T = 0, disp × 2 + PC → PC; When T = 1, nop	3/1*3	—
BF/S	label*2	10001111dddddddd	When T = 0, disp × 2 + PC → PC; When T = 1, nop	2/1*3	—
BT	label	10001001dddddddd	When T = 1, disp × 2+ PC → PC; When T = 0, nop	3/1*3	—
BT/S	label*2	10001101dddddddd	When T = 1, disp × 2 + PC → PC; When T = 0, nop	2/1*3	—
BRA	label	1010dddddddddddd	Delayed branch, disp × 2 + PC → PC	2	—
BSR	label	1011dddddddddddd	Delayed branch, PC → PR, disp × 2 + PC → PC	2	—

Notes: 2. SH7600 instruction
3. One state when it does not branch

A.1.13 Immediate

Table A.20 Arithmetic Logical Operation with Direct Register Addressing

Instruction		Code	Operation	State	T Bit
ADD	#imm,Rn	0111nnnniiiiiiii	Rn + imm → Rn	1	—
AND	#imm,R0	11001001iiiiiiii	R0 & imm → R0	1	—
CMP/EQ	#imm,R0	10001000iiiiiiii	When R0 = imm, 1 → T	1	Comparison result
MOV	#imm,Rn	1110nnnniiiiiiii	imm → sign extension → Rn	1	—
OR	#imm,R0	11001011iiiiiiii	R0 I imm → R0	1	—
TST	#imm,R0	11001000iiiiiiii	R0 & imm, when result is 0, 1 →T	1	Test results
XOR	#imm,R0	11001010iiiiiiii	R0 ^ imm → R0	1	—

Table A.21 Specify Exception Processing Vector

Instruction		Code	Operation	State	T Bit
TRAPA	#imm	11000011iiiiiiii	PC/SR → Stack area, (imm × 4 + VBR) → PC	8	—

A.2 Instruction Sets by Instruction Format

Tables A.22 to A.48 list instruction codes and execution states by instruction formats.

Table A.22 Instruction Sets by Format

Format	Category	Sample Instruction		Types SH 7600	Types SH 7000
0	—	NOP		8	8
n	Direct register addressing	MOVT	Rn	18	17
	Direct register addressing (store with control or system registers)	STS	MACH,Rn	6	6
	Direct register addressing	JMP	@Rn	3	3
	Pre decrement indirect register addressing	STC.L	SR,@-Rn	6	6
	PC relative addressing with Rn	BRAF	Rn	2	0
m	Direct register addressing (load with control or system registers)	LDC	Rm,SR	6	6
	Post increment indirect register addressing	LDC.L	@Rm+,SR	6	6
nm	Direct register addressing	ADD	Rm,Rn	34	31
	Indirect register addressing	MOV.L	Rm,@Rn	6	6
	Post increment indirect register addressing (multiply/accumulate operation)	MAC.W	@Rm+,@Rn+	2	1
	Post increment indirect register addressing	MOV.L	@Rm+,Rn	3	3
	Pre decrement indirect register addressing	MOV.L	Rm,@-Rn	3	3
	Indirect indexed register addressing	MOV.L	Rm,@(R0,Rn)	6	6
md	Indirect register addressing with displacement	MOV.B	@(disp,Rm),R0	2	2
nd4	Indirect register addressing with displacement	MOV.B	R0,@(disp,Rn)	2	2
nmd	Indirect register addressing with displacement	MOV.L	Rm,@(disp,Rn)	2	2
d	Indirect GBR addressing with displacement	MOV.L	R0,@(disp,GBR)	6	6
	Indirect PC addressing with displacement	MOVA	@(disp,PC),R0	1	1
	PC relative addressing	BF	label	4	2
d12	PC relative addressing	BRA	label	2	2
nd8	PC relative addressing with displacement	MOV.L	@(disp,PC),Rn	2	2
i	Indirect indexed GBR addressing	AND.B	#imm,@(R0,GBR)	4	4
	Immediate addressing (arithmetic and logical operations with direct register)	AND	#imm,R0	5	5
	Immediate addressing (specify exception processing vector)	TRAPA	#imm	1	1
ni	Immediate addressing (direct register arithmetic operations and data transfers)	ADD	#imm,Rn	2	2
			Total:	142	133

A.2.1 0 Format

Table A.23 0 Format

Instruction	Code	Operation	State	T Bit
CLRT	0000000000001000	0 →T	1	0
CLRMAC	0000000000101000	0 → MACH, MACL	1	—
DIV0U	0000000000011001	0 → M/Q/T	1	0
NOP	0000000000001001	No operation	1	—
RTE	0000000000101011	Delayed branching, stack area → PC/SR	4	LSB
RTS	0000000000001011	Delayed branching, PR → PC	2	—
SETT	0000000000011000	1 →T	1	1
SLEEP	0000000000011011	Sleep	3*4	—

Notes: 4. This is the number of states until a transition is made to the Sleep state.

A.2.2 n Format

Table A.24 Direct Register Addressing

Instruction		Code	Operation	State	T Bit
CMP/PL	Rn	0100nnnn00010101	Rn > 0, 1 → T	1	Comparison result
CMP/PZ	Rn	0100nnnn00010001	Rn ≥ 0, 1 → T	1	Comparison result
DT	Rn*2	0100nnnn00010000	Rn - 1 → Rn; If Rn is 0, 1 → T, if Rn is nonzero, 0 → T	1	Comparison result
MOVT	Rn	0000nnnn00101001	T → Rn	1	—
ROTL	Rn	0100nnnn00000100	T ← Rn ← MSB	1	MSB
ROTR	Rn	0100nnnn00000101	LSB → Rn → T	1	LSB
ROTCL	Rn	0100nnnn00100100	T ← Rn ← T	1	MSB
ROTCR	Rn	0100nnnn00100101	T → Rn → T	1	LSB
SHAL	Rn	0100nnnn00100000	T ← Rn ← 0	1	MSB
SHAR	Rn	0100nnnn00100001	MSB → Rn → T	1	LSB
SHLL	Rn	0100nnnn00000000	T ← Rn ← 0	1	MSB
SHLR	Rn	0100nnnn00000001	0 → Rn → T	1	LSB
SHLL2	Rn	0100nnnn00001000	Rn<<2 → Rn	1	—
SHLR2	Rn	0100nnnn00001001	Rn>>2 → Rn	1	—
SHLL8	Rn	0100nnnn00011000	Rn<<8 → Rn	1	—
SHLR8	Rn	0100nnnn00011001	Rn>>8 → Rn	1	—
SHLL16	Rn	0100nnnn00101000	Rn<<16 → Rn	1	—
SHLR16	Rn	0100nnnn00101001	Rn>>16 → Rn	1	—

Notes: 2. SH7600 instruction.

Table A.25 Direct Register Addressing (Store with Control and System Registers)

Instruction		Code	Operation	State	T Bit
STC	SR,Rn	0000nnnn00000010	SR → Rn	1	—
STC	GBR,Rn	0000nnnn00010010	GBR → Rn	1	—
STC	VBR,Rn	0000nnnn00100010	VBR → Rn	1	—
STS	MACH,Rn	0000nnnn00001010	MACH → Rn	1	—
STS	MACL,Rn	0000nnnn00011010	MACL → Rn	1	—
STS	PR,Rn	0000nnnn00101010	PR → Rn	1	—

Table A.26 Indirect Register Addressing

Instruction	Code	Operation	State	T Bit
JMP @Rn	0100nnnn00101011	Delayed branch, Rn → PC	2	—
JSR @Rn	0100nnnn00001011	Delayed branch, PC → PR, Rn → PC	2	—
TAS.B @Rn	0100nnnn00011011	When (Rn) is 0, 1 → T, 1 → MSB of (Rn)	4	Test results

Table A.27 Pre Decrement Indirect Register

Instruction	Code	Operation	State	T Bit
STC.L SR,@-Rn	0100nnnn00000011	Rn – 4 → Rn, SR → (Rn)	2	—
STC.L GBR,@-Rn	0100nnnn00010011	Rn – 4 → Rn, GBR → (Rn)	2	—
STC.L VBR,@-Rn	0100nnnn00100011	Rn – 4 → Rn, VBR → (Rn)	2	—
STS.L MACH,@-Rn	0100nnnn00000010	Rn – 4 → Rn, MACH → (Rn)	1	—
STS.L MACL,@-Rn	0100nnnn00010010	Rn – 4 → Rn, MACL → (Rn)	1	—
STS.L PR,@-Rn	0100nnnn00100010	Rn – 4 → Rn, PR → (Rn)	1	—

Table A.28 PC Relative Addressing With Rn

Instruction	Code	Operation	State	T Bit
BRAF Rn*2	0000nnnn00100011	Delayed branch, Rn + PC → PC	2	—
BSRF Rn*2	0000nnnn00000011	Delayed branch, PC → PR, Rn + PC → PC	2	—

Notes: 2. SH7600 instruction

A.2.3 m Format

Table A.29 Direct Register Addressing (Load with Control and System Registers)

Instruction		Code	Operation	State	T Bit
LDC	Rm, SR	0100mmmm00001110	Rm → SR	1	LSB
LDC	Rm, GBR	0100mmmm00011110	Rm → GBR	1	—
LDC	Rm, VBR	0100mmmm00101110	Rm → VBR	1	—
LDS	Rm, MACH	0100mmmm00001010	Rm → MACH	1	—
LDS	Rm, MACL	0100mmmm00011010	Rm → MACL	1	—
LDS	Rm, PR	0100mmmm00101010	Rm → PR	1	—

Table A.30 Post Increment Indirect Register

Instruction		Code	Operation	State	T Bit
LDC.L	@Rm+, SR	0100mmmm00000111	(Rm) → SR, Rm + 4 → Rm	3	LSB
LDC.L	@Rm+, GBR	0100mmmm00010111	(Rm) → GBR, Rm + 4 → Rm	3	—
LDC.L	@Rm+, VBR	0100mmmm00100111	(Rm) → VBR, Rm + 4 → Rm	3	—
LDS.L	@Rm+, MACH	0100mmmm00000110	(Rm) → MACH, Rm + 4 → Rm	1	—
LDS.L	@Rm+, MACL	0100mmmm00010110	(Rm) → MACL, Rm + 4 → Rm	1	—
LDS.L	@Rm+, PR	0100mmmm00100110	(Rm) → PR, Rm + 4 → Rm	1	—

A.2.4 nm Format

Table A.31 Direct Register Addressing

Instruction		Code	Operation	State	T Bit
ADD	Rm,Rn	0011nnnnmmmm1100	Rn + Rm → Rn	1	—
ADDC	Rm,Rn	0011nnnnmmmm1110	Rn + Rm + T → Rn, carry →T	1	Carry
ADDV	Rm,Rn	0011nnnnmmmm1111	Rn + Rm → Rn, overflow →T	1	Overflow
AND	Rm,Rn	0010nnnnmmmm1001	Rn & Rm → Rn	1	—
CMP/EQ	Rm,Rn	0011nnnnmmmm0000	When Rn = Rm, 1 → T	1	Comparison result
CMP/HS	Rm,Rn	0011nnnnmmmm0010	When unsigned and Rn ≥ Rm, 1 → T	1	Comparison result
CMP/GE	Rm,Rn	0011nnnnmmmm0011	When signed and Rn ≥ Rm, 1 → T	1	Comparison result
CMP/HI	Rm,Rn	0011nnnnmmmm0110	When unsigned and Rn > Rm, 1 → T	1	Comparison result
CMP/GT	Rm,Rn	0011nnnnmmmm0111	When signed and Rn > Rm, 1 → T	1	Comparison result
CMP/STR	Rm,Rn	0010nnnnmmmm1100	When a byte in Rn equals a byte in Rm, 1 → T	1	Comparison result
DIV1	Rm,Rn	0011nnnnmmmm0100	1-step division (Rn ÷ Rm)	1	Calculation result
DIV0S	Rm,Rn	0010nnnnmmmm0111	MSB of Rn → Q, MSB of Rm → M, M ^ Q →T	1	Calculation result
DMULS.L	Rm,Rn*2	0011nnnnmmmm1101	Signed, Rn x Rm → MACH, MACL	2 to 4*1	—
DMULU.L	Rm,Rn*2	0011nnnnmmmm0101	Unsigned, Rn x Rm → MACH, MACL	2 to 4*1	—
EXTS.B	Rm,Rn	0110nnnnmmmm1110	Sign-extends Rm from byte → Rn	1	—
EXTS.W	Rm,Rn	0110nnnnmmmm1111	Sign-extends Rm from word → Rn	1	—
EXTU.B	Rm,Rn	0110nnnnmmmm1100	Zero-extends Rm from byte → Rn	1	—
EXTU.W	Rm,Rn	0110nnnnmmmm1101	Zero-extends Rm from word → Rn	1	—
MOV	Rm,Rn	0110nnnnmmmm0011	Rm → Rn	1	—

Notes: 1. The normal minimum number of execution states
2. SH7600 instruction

Table A.31 Direct Register Addressing (cont)

Instruction		Code	Operation	State	T Bit
MUL.L	Rm,Rn*2	0000nnnnmmmm0111	Rn × Rm → MACL	2 to 4*1	—
MULS.W	Rm,Rn	0010nnnnmmmm1111	Signed, Rn × Rm → MAC	1 to 3*1	—
MULU.W	Rm,Rn	0010nnnnmmmm1110	Unsigned, Rn × Rm → MAC	1 to 3*1	—
NEG	Rm,Rn	0110nnnnmmmm1011	0 – Rm → Rn	1	—
NEGC	Rm,Rn	0110nnnnmmmm1010	0 – Rm – T → Rn, borrow → T	1	Borrow
NOT	Rm,Rn	0110nnnnmmmm0111	~Rm → Rn	1	—
OR	Rm,Rn	0010nnnnmmmm1011	Rn | Rm → Rn	1	—
SUB	Rm,Rn	0011nnnnmmmm1000	Rn – Rm → Rn	1	—
SUBC	Rm,Rn	0011nnnnmmmm1010	Rn – Rm – T → Rn, borrow → T	1	Borrow
SUBV	Rm,Rn	0011nnnnmmmm1011	Rn – Rm → Rn, underflow → T	1	Underflow
SWAP.B	Rm,Rn	0110nnnnmmmm1000	Rm → Swap upper and lower halves of lower 2 bytes → Rn	1	—
SWAP.W	Rm,Rn	0110nnnnmmmm1001	Rm → Swap upper and lower word → Rn	1	—
TST	Rm,Rn	0010nnnnmmmm1000	Rn & Rm, when result is 0, 1 → T	1	Test results
XOR	Rm,Rn	0010nnnnmmmm1010	Rn ^ Rm → Rn	1	—
XTRCT	Rm,Rn	0010nnnnmmmm1101	Center 32 bits of Rm and Rn → Rn	1	—

Notes: 1. The normal minimum number of execution cycles.
2. SH7600 instructions

Table A.32 Indirect Register Addressing

Instruction		Code	Operation	State	T Bit
MOV.B	Rm,@Rn	0010nnnnmmmm0000	Rm → (Rn)	1	—
MOV.W	Rm,@Rn	0010nnnnmmmm0001	Rm → (Rn)	1	—
MOV.L	Rm,@Rn	0010nnnnmmmm0010	Rm → (Rn)	1	—
MOV.B	@Rm,Rn	0110nnnnmmmm0000	(Rm) → sign extension → Rn	1	—
MOV.W	@Rm,Rn	0110nnnnmmmm0001	(Rm) → sign extension → Rn	1	—
MOV.L	@Rm,Rn	0110nnnnmmmm0010	(Rm) → Rn	1	—

Table A.33 Post Increment Indirect Register (Multiply/Accumulate Operation)

Instruction		Code	Operation	State	T Bit
MAC.L	@Rm+,@Rn+*2	0000nnnnmmmm1111	Signed, (Rn) × (Rm) + MAC → MAC	3/(2 to 4)*1	—
MAC.W	@Rm+,@Rn+	0100nnnnmmmm1111	Signed, (Rn) × (Rm) + MAC → MAC	3/(2)*1	—

Notes: 1. The normal minimum number of execution cycles.(The number in parentheses in the number of cycles when there is contention with preceding/following instructions).
2. SH7600 instruction.

Table A.34 Post Increment Indirect Register

Instruction		Code	Operation	State	T Bit
MOV.B	@Rm+,Rn	0110nnnnmmmm0100	(Rm) → sign extension → Rn, Rm + 1 → Rm	1	—
MOV.W	@Rm+,Rn	0110nnnnmmmm0101	(Rm) → sign extension → Rn, Rm + 2 → Rm	1	—
MOV.L	@Rm+,Rn	0110nnnnmmmm0110	(Rm) → Rn, Rm + 4 → Rm	1	—

Table A.35 Pre Decrement Indirect Register

Instruction		Code	Operation	State	T Bit
MOV.B	Rm,@–Rn	0010nnnnmmmm0100	Rn – 1 → Rn, Rm → (Rn)	1	—
MOV.W	Rm,@–Rn	0010nnnnmmmm0101	Rn – 2 → Rn, Rm → (Rn)	1	—
MOV.L	Rm,@–Rn	0010nnnnmmmm0110	Rn – 4 → Rn, Rm → (Rn)	1	—

Table A.36 Indirect Indexed Register

Instruction		Code	Operation	Cycles	T Bit
MOV.B	Rm,@(R0,Rn)	0000nnnnmmmm0100	Rm → (R0 + Rn)	1	—
MOV.W	Rm,@(R0,Rn)	0000nnnnmmmm0101	Rm → (R0 + Rn)	1	—
MOV.L	Rm,@(R0,Rn)	0000nnnnmmmm0110	Rm → (R0 + Rn)	1	—
MOV.B	@(R0,Rm),Rn	0000nnnnmmmm1100	(R0 + Rm) → sign extension → Rn	1	—
MOV.W	@(R0,Rm),Rn	0000nnnnmmmm1101	(R0 + Rm) → sign extension → Rn	1	—
MOV.L	@(R0,Rm),Rn	0000nnnnmmmm1110	(R0 + Rm) → Rn	1	—

A.2.5 md Format

Table A.37 md Format

Instruction	Code	Operation	State	T Bit
`MOV.B @(disp,Rm),R0`	10000100mmmmdddd	(disp + Rm) → sign extension → R0	1	—
`MOV.W @(disp,Rm),R0`	10000101mmmmdddd	(disp × 2 + Rm) → sign extension → R0	1	—

A.2.6 nd4 Format

Table A.38 nd4 Format

Instruction	Code	Operation	State	T Bit
`MOV.B R0,@(disp,Rn)`	10000000nnnndddd	R0 → (disp + Rn)	1	—
`MOV.W R0,@(disp,Rn)`	10000001nnnndddd	R0 → (disp × 2+ Rn)	1	—

A.2.7 nmd Format

Table A.39 nmd Format

Instruction	Code	Operation	State	T Bit
`MOV.L Rm,@(disp,Rn)`	0001nnnnmmmmdddd	Rm → (disp × 4 + Rn)	1	—
`MOV.L @(disp,Rm),Rn`	0101nnnnmmmmdddd	(disp × 4+ Rm) → Rn	1	—

A.2.8 d Format

Table A.40 Indirect GBR with Displacement

Instruction	Code	Operation	State	T Bit
MOV.B R0,@(disp,GBR)	11000000dddddddd	R0 → (disp + GBR)	1	—
MOV.W R0,@(disp,GBR)	11000001dddddddd	R0 → (disp × 2 + GBR)	1	—
MOV.L R0,@(disp,GBR)	11000010dddddddd	R0 → (disp × 4 + GBR)	1	—
MOV.B @(disp,GBR),R0	11000100dddddddd	(disp + GBR) → sign extension → R0	1	—
MOV.W @(disp,GBR),R0	11000101dddddddd	(disp × 2 + GBR) → sign extension → R0	1	—
MOV.L @(disp,GBR),R0	11000110dddddddd	(disp × 4 + GBR) → R0	1	—

Table A.41 PC Relative with Displacement

Instruction	Code	Operation	State	T Bit
MOVA @(disp,PC),R0	11000111dddddddd	disp × 4 + PC → R0	1	—

Table A.42 PC Relative Addressing

Instruction	Code	Operation	State	T Bit
BF label	10001011dddddddd	When T = 0, disp × 2 + PC → PC; When T = 1, nop	3/1*3	—
BF/S label*2	10001111dddddddd	When T = 0, disp × 2 + PC → PC; When T = 1, nop	2/1*3	—
BT label	10001001dddddddd	When T = 1, disp × 2 + PC → PC; When T = 0, nop	3/1*3	—
BT/S label*2	10001101dddddddd	When T = 1, disp × 2 + PC → PC; When T = 0, nop	2/1*3	—

Notes: 2. SH7600 instruction

3. One state when it does not branch

A.2.9 d12 Format

Table A.43 d12 Format

Instruction	Code	Operation	State	T Bit
`BRA label`	`1010dddddddddddd`	Delayed branch, disp $\times 2+$ PC $\rightarrow$ PC	2	—
`BSR label`	`1011dddddddddddd`	Delayed branching, PC $\rightarrow$ PR, disp $\times 2$ + PC $\rightarrow$ PC	2	—

A.2.10 nd8 Format

Table A.44 nd8 Format

Instruction	Code	Operation	State	T Bit
`MOV.W @(disp,PC),Rn`	`1001nnnndddddddd`	(disp $\times$ 2 + PC) $\rightarrow$ sign extension $\rightarrow$ Rn	1	—
`MOV.L @(disp,PC),Rn`	`1101nnnndddddddd`	(disp $\times$ 4 + PC) $\rightarrow$ Rn	1	—

A.2.11 i Format

Table A.45 Indirect Indexed GBR Addressing

Instruction	Code	Operation	State	T Bit
`AND.B #imm,@(R0,GBR)`	`11001101iiiiiiii`	(R0 + GBR) & imm $\rightarrow$ (R0 + GBR)	3	—
`OR.B #imm,@(R0,GBR)`	`11001111iiiiiiii`	(R0 + GBR) | imm $\rightarrow$ (R0 + GBR)	3	—
`TST.B #imm,@(R0,GBR)`	`11001100iiiiiiii`	(R0 + GBR) & imm, when result is 0, 1 $\rightarrow$ T	3	Test results
`XOR.B #imm,@(R0,GBR)`	`11001110iiiiiiii`	(R0 + GBR) ^ imm $\rightarrow$ (R0 + GBR)	3	—

Table A.46 Immediate Addressing (Arithmetic Logical Operation with Direct Register)

Instruction		Code	Operation	State	T Bit
AND	#imm,R0	11001001iiiiiiii	R0 & imm → R0	1	—
CMP/EQ	#imm,R0	10001000iiiiiiii	When R0 = imm, 1 → T	1	Comparison results
OR	#imm,R0	11001011iiiiiiii	R0 | imm → R0	1	—
TST	#imm,R0	11001000iiiiiiii	R0 & imm, when result is 0, 1 →T	1	Test results
XOR	#imm,R0	11001010iiiiiiii	R0 ^ imm → R0	1	—

Table A.47 Immediate Addressing (Specify Exception Processing Vector)

Instruction		Code	Operation	State	T Bit
TRAPA	#imm	11000011iiiiiiii	PC/SR → Stack area, (imm × 4 + VBR) → PC	8	—

A.2.12 ni Format

Table A.48 ni Format

Instruction		Code	Operation	State	T Bit
ADD	#imm,Rn	0111nnnniiiiiiii	Rn + imm → Rn	1	—
MOV	#imm,Rn	1110nnnniiiiiiii	imm → sign extension → Rn	1	—

A.3 Instruction Set in Order by Instruction Code

Table A.49 lists instruction codes and execution states in order by instruction code.

Table A.49 Instruction Set by Instruction Code

Instruction	Code	Operation	State	T Bit
CLRT	0000000000001000	0 →T	1	0
NOP	0000000000001001	No operation	1	—
RTS	0000000000001011	Delayed branch, PR → PC	2	—
SETT	0000000000011000	1 →T	1	1
DIV0U	0000000000011001	0 → M/Q/T	1	0

Table A.49 Instruction Set by Instruction Code (cont)

Instruction		Code	Operation	State	T Bit
SLEEP		0000000000011011	Sleep	3	—
CLRMAC		0000000000101000	0 → MACH, MACL	1	—
RTE		0000000000101011	Delayed branch, stack area → PC/SR	4	LSB
STC	SR,Rn	0000nnnn00000010	SR → Rn	1	—
BSRF	Rn*2	0000nnnn00000011	Delayed branch, PC → PR, Rn + PC → PC	2	—
STS	MACH,Rn	0000nnnn00001010	MACH → Rn	1	—
STC	GBR,Rn	0000nnnn00010010	GBR → Rn	1	—
STS	MACL,Rn	0000nnnn00011010	MACL → Rn	1	—
STC	VBR,Rn	0000nnnn00100010	VBR → Rn	1	—
BRAF	Rn*2	0000nnnn00100011	Delayed branch, Rn + PC → PC	2	—
MOVT	Rn	0000nnnn00101001	T → Rn	1	—
STS	PR,Rn	0000nnnn00101010	PR → Rn	1	—
MOV.B	Rm,@(R0,Rn)	0000nnnnmmmm0100	Rm → (R0 + Rn)	1	—
MOV.W	Rm,@(R0,Rn)	0000nnnnmmmm0101	Rm → (R0 + Rn)	1	—
MOV.L	Rm,@(R0,Rn)	0000nnnnmmmm0110	Rm → (R0 + Rn)	1	—
MUL.L	Rm,Rn*2	0000nnnnmmmm0111	Rn x Rm → MACL	2 (to 4)*1	—
MOV.B	@(R0,Rm),Rn	0000nnnnmmmm1100	(R0 + Rm) → sign extension → Rn	1	—
MOV.W	@(R0,Rm),Rn	0000nnnnmmmm1101	(R0 + Rm) → sign extension → Rn	1	—
MOV.L	@(R0,Rm),Rn	0000nnnnmmmm1110	(R0 + Rm) → Rn	1	—
MAC.L	@Rm+,@Rn+*2	0000nnnnmmmm1111	Signed, (Rn) x (Rm) + MAC → MAC	3/ (2 to 4)*1	—
MOV.L	Rm,@(disp,Rn)	0001nnnnmmmmdddd	Rm → (disp × 4 + Rn)	1	—
MOV.B	Rm,@Rn	0010nnnnmmmm0000	Rm → (Rn)	1	—
MOV.W	Rm,@Rn	0010nnnnmmmm0001	Rm → (Rn)	1	—

Notes: 1. The normal minimum number of execution states (The number in parentheses is the number of states when there is contention with preceding/following instructions)
2. SH7600 instruction

Table A.49 Instruction Set by Instruction Code (cont)

Instruction		Code	Operation	State	T Bit
MOV.L	Rm,@Rn	0010nnnnmmmm0010	Rm → (Rn)	1	—
MOV.B	Rm,@-Rn	0010nnnnmmmm0100	Rn – 1 → Rn, Rm → (Rn)	1	—
MOV.W	Rm,@-Rn	0010nnnnmmmm0101	Rn – 2 → Rn, Rm → (Rn)	1	—
MOV.L	Rm,@-Rn	0010nnnnmmmm0110	Rn – 4 → Rn, Rm → (Rn)	1	—
DIV0S	Rm,Rn	0010nnnnmmmm0111	MSB of Rn → Q, MSB of Rm → M, M ^ Q → T	1	Calculation result
TST	Rm,Rn	0010nnnnmmmm1000	Rn & Rm, when result is 0, 1 →T	1	Test results
AND	Rm,Rn	0010nnnnmmmm1001	Rn & Rm → Rn	1	—
XOR	Rm,Rn	0010nnnnmmmm1010	Rn ^ Rm → Rn	1	—
OR	Rm,Rn	0010nnnnmmmm1011	Rn I Rm → Rn	1	—
CMP/STR	Rm,Rn	0010nnnnmmmm1100	When a byte in Rn equals a byte in Rm, 1 →T	1	Comparison result
XTRCT	Rm,Rn	0010nnnnmmmm1101	Center 32 bits of Rm and Rn → Rn	1	—
MULU.W	Rm,Rn	0010nnnnmmmm1110	Unsigned, Rn × Rm → MAC	1 to 3*1	—
MULS.W	Rm,Rn	0010nnnnmmmm1111	Signed, Rn × Rm → MAC	1 to 3*1	—
CMP/EQ	Rm,Rn	0011nnnnmmmm0000	When Rn = Rm, 1 → T	1	Comparison result
CMP/HS	Rm,Rn	0011nnnnmmmm0010	When unsigned and Rn ≥ Rm, 1 →T	1	Comparison result
CMP/GE	Rm,Rn	0011nnnnmmmm0011	When signed and Rn ≥ Rm, 1 → T	1	Comparison result
DIV1	Rm,Rn	0011nnnnmmmm0100	1-step division (Rn ÷ Rm)	1	Calculation result
DMULU.L	Rm,Rn*2	0011nnnnmmmm0101	Unsigned, Rn x Rm → MACH, MACL	2 to 4*1	—

Notes: 1. The normal minimum number of execution states
2. SH7600 instruction

Table A.49 Instruction Set by Instruction Code (cont)

Instruction		Code	Operation	State	T Bit
CMP/HI	Rm,Rn	0011nnnnmmmm0110	When unsigned and Rn > Rm, 1 → T	1	Comparison result
CMP/GT	Rm,Rn	0011nnnnmmmm0111	When signed and Rn > Rm, 1 → T	1	Comparison result
SUB	Rm,Rn	0011nnnnmmmm1000	Rn – Rm → Rn	1	—
SUBC	Rm,Rn	0011nnnnmmmm1010	Rn – Rm – T → Rn, borrow → T	1	Borrow
SUBV	Rm,Rn	0011nnnnmmmm1011	Rn – Rm → Rn, underflow → T	1	Underflow
ADD	Rm,Rn	0011nnnnmmmm1100	Rm + Rn → Rn	1	—
DMULS.L	Rm,Rn*2	0011nnnnmmmm1101	Signed, Rn x Rm → MACH, MACL	2 to 4*1	—
ADDC	Rm,Rn	0011nnnnmmmm1110	Rn + Rm + T → Rn, carry → T	1	Carry
ADDV	Rm,Rn	0011nnnnmmmm1111	Rn + Rm → Rn, overflow → T	1	Overflow
SHLL	Rn	0100nnnn00000000	T ← Rn ← 0	1	MSB
SHLR	Rn	0100nnnn00000001	0 → Rn → T	1	LSB
STS.L	MACH,@-Rn	0100nnnn00000010	Rn – 4 → Rn, MACH → (Rn)	1	—
STC.L	SR,@-Rn	0100nnnn00000011	Rn – 4 → Rn, SR → (Rn)	2	—
ROTL	Rn	0100nnnn00000100	T ← Rn ← MSB	1	MSB
ROTR	Rn	0100nnnn00000101	LSB → Rn → T	1	LSB
LDS.L	@Rm+,MACH	0100mmmm00000110	(Rm) → MACH, Rm + 4 → Rm	1	—
LDC.L	@Rm+,SR	0100mmmm00000111	(Rm) → SR, Rm + 4 → Rm	3	LSB
SHLL2	Rn	0100nnnn00001000	Rn<<2 → Rn	1	—
SHLR2	Rn	0100nnnn00001001	Rn>>2 → Rn	1	—
LDS	Rm,MACH	0100mmmm00001010	Rm → MACH	1	—

Notes: 1. The normal minimum number of execution states

2. SH7600 instruction

Table A.49 Instruction Set by Instruction Code (cont)

Instruction		Code	Operation	State	T Bit
JSR	@Rn	0100nnnn00001011	Delayed branch, PC → PR, Rn → PC	2	—
LDC	Rm, SR	0100mmmm00001110	Rm → SR	1	LSB
DT	Rn*2	0100nnnn00010000	Rn - 1 → Rn; if Rn is 0, 1 → T, if Rn is nonzero, 0 → T	1	Comparison result
CMP/PZ	Rn	0100nnnn00010001	Rn ≥ 0, 1 → T	1	Comparison result
STS.L	MACL, @-Rn	0100nnnn00010010	Rn – 4 → Rn, MACL → (Rn)	1	—
STC.L	GBR, @-Rn	0100nnnn00010011	Rn – 4 → Rn, GBR → (Rn)	2	—
CMP/PL	Rn	0100nnnn00010101	Rn > 0, 1 → T	1	Comparison result
LDS.L	@Rm+, MACL	0100mmmm00010110	(Rm) → MACL, Rm + 4 → Rm	1	—
LDC.L	@Rm+, GBR	0100mmmm00010111	(Rm) → GBR, Rm + 4 → Rm	3	—
SHLL8	Rn	0100nnnn00011000	Rn<<8 → Rn	1	—
SHLR8	Rn	0100nnnn00011001	Rn>>8 → Rn	1	—
LDS	Rm, MACL	0100mmmm00011010	Rm → MACL	1	—
TAS.B	@Rn	0100nnnn00011011	When (Rn) is 0, 1 → T, 1 → MSB of (Rn)	4	Test results
LDC	Rm, GBR	0100mmmm00011110	Rm → GBR	1	—
SHAL	Rn	0100nnnn00100000	T ← Rn ← 0	1	MSB
SHAR	Rn	0100nnnn00100001	MSB → Rn → T	1	LSB
STS.L	PR, @-Rn	0100nnnn00100010	Rn – 4 → Rn, PR → (Rn)	1	—
STC.L	VBR, @-Rn	0100nnnn00100011	Rn – 4 → Rn, VBR → (Rn)	2	—
ROTCL	Rn	0100nnnn00100100	T ← Rn ← T	1	MSB
ROTCR	Rn	0100nnnn00100101	T → Rn → T	1	LSB
LDS.L	@Rm+, PR	0100mmmm00100110	(Rm) → PR, Rm + 4 → Rm	1	—
LDC.L	@Rm+, VBR	0100mmmm00100111	(Rm) → VBR, Rm + 4 → Rm	3	—

Notes: 2. SH7600 instruction

Table A.49 Instruction Set by Instruction Code (cont)

Instruction		Code	Operation	State	T Bit
SHLL16	Rn	0100nnnn00101000	Rn<<16 → Rn	1	—
SHLR16	Rn	0100nnnn00101001	Rn>>16 → Rn	1	—
LDS	Rm, PR	0100mmmm00101010	Rm → PR	1	—
JMP	@Rn	0100nnnn00101011	Delayed branch, Rn → PC	2	—
LDC	Rm, VBR	0100mmmm00101110	Rm → VBR	1	—
MAC.W	@Rm+, @Rn+	0100nnnnmmmm1111	Signed, (Rn) × (Rm) + MAC → MAC	3/(2)*1	—
MOV.L	@(disp, Rm), Rn	0101nnnnmmmmdddd	(disp + Rm) → Rn	1	—
MOV.B	@Rm, Rn	0110nnnnmmmm0000	(Rm) → sign extension → Rn	1	—
MOV.W	@Rm, Rn	0110nnnnmmmm0001	(Rm) → sign extension → Rn	1	—
MOV.L	@Rm, Rn	0110nnnnmmmm0010	(Rm) → Rn	1	—
MOV	Rm, Rn	0110nnnnmmmm0011	Rm → Rn	1	—
MOV.B	@Rm+, Rn	0110nnnnmmmm0100	(Rm) → sign extension → Rn, Rm + 1 → Rm	1	—
MOV.W	@Rm+, Rn	0110nnnnmmmm0101	(Rm) → sign extension → Rn, Rm + 2 → Rm	1	—
MOV.L	@Rm+, Rn	0110nnnnmmmm0110	(Rm) → Rn, Rm + 4 → Rm	1	—
NOT	Rm, Rn	0110nnnnmmmm0111	~Rm → Rn	1	—
SWAP.B	Rm, Rn	0110nnnnmmmm1000	Rm → Swap upper and lower halves of lower 2 bytes → Rn	1	—
SWAP.W	Rm, Rn	0110nnnnmmmm1001	Rm → Swap upper and lower word → Rn	1	—
NEGC	Rm, Rn	0110nnnnmmmm1010	0 – Rm – T → Rn, borrow → T	1	Borrow
NEG	Rm, Rn	0110nnnnmmmm1011	0 – Rm → Rn	1	—

Notes: 1 The normal minimum number of execution states (The number in parentheses is the number in contention with preceding/following instructions)

Table A.49 Instruction Set by Instruction Code (cont)

Instruction		Code	Operation	State	T Bit
EXTU.B	Rm,Rn	0110nnnnmmmm1100	Zero-extends Rm from byte → Rn	1	—
EXTU.W	Rm,Rn	0110nnnnmmmm1101	Zero-extends Rm from word → Rn	1	—
EXTS.B	Rm,Rn	0110nnnnmmmm1110	Sign-extends Rm from byte → Rn	1	—
EXTS.W	Rm,Rn	0110nnnnmmmm1111	Sign-extends Rm from word → Rn	1	—
ADD	#imm,Rn	0111nnnniiiiiiii	Rn + imm → Rn	1	—
MOV.B	R0,@(disp,Rn)	10000000nnnndddd	R0 → (disp + Rn)	1	—
MOV.W	R0,@(disp,Rn)	10000001nnnndddd	R0 → (disp × 2 + Rn)	1	—
MOV.B	@(disp,Rm),R0	10000100mmmmdddd	(disp + Rm) → sign extension → R0	1	—
MOV.W	@(disp,Rm),R0	10000101mmmmdddd	(disp × 2 + Rm) → sign extension → R0	1	—
CMP/EQ	#imm,R0	10001000iiiiiiii	When R0 = imm, 1 → T	1	Comparison results
BT	label	10001001dddddddd	When T = 1, disp × 2 + PC → PC; When T = 0, nop.	3/1*3	—
BT/S	label*	10001101dddddddd	When T = 1, disp × 2 + PC → PC; When T = 1, nop.	2/1*3	—
BF	label	10001011dddddddd	When T = 0, disp × 2 + PC → PC; When T = 0, nop	3/1*3	—
BF/S	label*	10001111dddddddd	When T = 0, disp × 2 + PC → PC; When T = 1, nop	2/1*3	—
MOV.W	@(disp,PC),Rn	1001nnnndddddddd	(disp × 2 + PC) → sign extension → Rn	1	—
BRA	label	1010dddddddddddd	Delayed branch, disp × 2 + PC → PC	2	—

Notes: 2. SH7600 instruction
3. One state when it does not branch

Table A.49 Instruction Set by Instruction Code (cont)

Instruction		Code	Operation	State	T Bit
BSR	label	1011dddddddddddd	Delayed branch, PC → PR, disp × 2 + PC → PC	2	—
MOV.B	R0,@(disp,GBR)	11000000dddddddd	R0 → (disp + GBR)	1	—
MOV.W	R0,@(disp,GBR)	11000001dddddddd	R0 → (disp × 2 + GBR)	1	—
MOV.L	R0,@(disp,GBR)	11000010dddddddd	R0 → (disp × 4 + GBR)	1	—
TRAPA	#imm	11000011iiiiiiii	PC/SR → Stack area, (imm × 4 + VBR) → PC	8	—
MOV.B	@(disp,GBR),R0	11000100dddddddd	(disp + GBR) → sign extension → R0	1	—
MOV.W	@(disp,GBR),R0	11000101dddddddd	(disp × 2 + GBR) → sign extension → R0	1	—
MOV.L	@(disp,GBR),R0	11000110dddddddd	(disp × 4 + GBR) → R0	1	—
MOVA	@(disp,PC),R0	11000111dddddddd	disp × 4 + PC → R0	1	—
TST	#imm,R0	11001000iiiiiiii	R0 & imm, when result is 0, 1 → T	1	Test results
AND	#imm,R0	11001001iiiiiiii	R0 & imm → R0	1	—
XOR	#imm,R0	11001010iiiiiiii	R0 ^ imm → R0	1	—
OR	#imm,R0	11001011iiiiiiii	R0 | imm → R0	1	—
TST.B	#imm,@(R0,GBR)	11001100iiiiiiii	(R0 + GBR) & imm, when result is 0, 1 → T	3	Test results
AND.B	#imm,@(R0,GBR)	11001101iiiiiiii	(R0 + GBR) & imm → (R0 + GBR)	3	—
XOR.B	#imm,@(R0,GBR)	11001110iiiiiiii	(R0 + GBR) ^ imm → (R0 + GBR)	3	—
OR.B	#imm,@(R0,GBR)	11001111iiiiiiii	(R0 + GBR) | imm → (R0 + GBR)	3	—
MOV.L	@(disp,PC),Rn	1101nnnndddddddd	(disp × 4 + PC) → Rn	1	—
MOV	#imm,Rn	1110nnnniiiiiiii	imm → sign extension → Rn	1	—

A.4 Operation Code Map

Table A.50 is an operation code map.

Table A.50 Operation Code Map

Instruction Code				Fx: 0000	Fx: 0001	Fx: 0010	Fx: 0011–1111
MSB			LSB	MD: 00	MD: 01	MD: 10	MD: 11
0000	Rn	Fx	0000				
0000	Rn	Fx	0001				
0000	Rn	Fx	0010	STC SR,Rn*	STC GBR,Rn	STC VBR,Rn	
0000	Rn	Fx	0011	BSRF Rn*		BRAF Rn*	
0000	Rn	Rm	01MD	MOV.B Rm,@(R0,Rn)	MOV.W Rm,@(R0,Rn)	MOV.L Rm,@(R0,Rn)	MUL.L Rm,Rn*
0000	0000	Fx	1000	CLRT	SETT	CLRMAC	
0000	0000	Fx	1001	NOP	DIV0U		
0000	0000	Fx	1010				
0000	0000	Fx	1011	RTS	SLEEP	RTE	
0000	Rn	Fx	1000				
0000	Rn	Fx	1001			MOVT Rn	
0000	Rn	Fx	1010	STS MACH,Rn	STS MACL,Rn	STS PR,Rn	
0000	Rn	Fx	1011				
0000	Rn	Fx	11MD	MOV.B @(R0,Rm),Rn	MOV.W @(R0,Rm),Rn	MOV.L @(R0,Rm),Rn	MAC.L @Rm+,@Rn+*
0001	Rn	Rm	disp	MOV.L Rm,@(disp:4,Rn)			
0010	Rn	Rm	00MD	MOV.B Rm,@Rn	MOV.W Rm,@Rn	MOV.L Rm,@Rn	
0010	Rn	Rm	01MD	MOV.B Rm,@-Rn	MOV.W Rm,@-Rn	MOV.L Rm,@-Rn	DIV0S Rm,Rn
0010	Rn	Rm	10MD	TST Rm,Rn	AND Rm,Rn	XOR Rm,Rn	OR Rm,Rn
0010	Rn	Rm	11MD	CMP/STR Rm,Rn	XTRCT Rm,Rn	MULU.W Rm,Rn	MULS.W Rm,Rn
0011	Rn	Rm	00MD	CMP/EQ Rm,Rn		CMP/HS Rm,Rn	CMP/GE Rm,Rn
0011	Rn	Rm	01MD	DIV1 Rm,Rn	DMULU.L Rm,Rn*	CMP/HI Rm,Rn	CMP/GT Rm,Rn
0011	Rn	Rm	10MD	SUB Rm,Rn		SUBC Rm,Rn	SUBV Rm,Rn
0011	Rn	Rm	11MD	ADD Rm,Rn	DMULS.L Rm,Rn*	ADDC Rm,Rn	ADDV Rm,Rn
0100	Rn	Fx	0000	SHLL Rn	DT Rn*	SHAL Rn	
0100	Rn	Fx	0001	SHLR Rn	CMP/PZ Rn	SHAR Rn	

Table A.50 Operation Code Map (cont)

Instruction Code				Fx: 0000	Fx: 0001	Fx: 0010	Fx: 0011–1111
MSB			LSB	MD: 00	MD: 01	MD: 10	MD: 11
0100	Rn	Fx	0010	STS.L MACH,@-Rn	STS.L MACL,@-Rn	STS.L PR,@-Rn	
0100	Rn	Fx	0011	STC.L SR,@-Rn	STC.L GBR,@-Rn	STC.L VBR,@-Rn	
0100	Rn	Fx	0100	ROTL Rn		ROTCL Rn	
0100	Rn	Fx	0101	ROTR Rn	CMP/PL Rn	ROTCR Rn	
0100	Rm	Fx	0110	LDS.L @Rm+,MACH	LDS.L @Rm+,MACL	LDS.L @Rm+,PR	
0100	Rm	Fx	0111	LDC.L @Rm+,SR	LDC.L @Rm+,GBR	LDC.L @Rm+,VBR	
0100	Rn	Fx	1000	SHLL2 Rn	SHLL8 Rn	SHLL16 Rn	
0100	Rn	Fx	1001	SHLR2 Rn	SHLR8 Rn	SHLR16 Rn	
0100	Rm	Fx	1010	LDS Rm,MACH	LDS Rm,MACL	LDS Rm,PR	
0100	Rn	Fx	1011	JSR @Rn	TAS.B @Rn	JMP @Rn	
0100	Rm	Fx	1100				
0100	Rm	Fx	1101				
0100	Rn	Fx	1110	LDC Rm,SR	LDC Rm,GBR	LDC Rm,VBR	
0100	Rn	Rm	1111	MAC.W @Rm+,@Rn+			
0101	Rn	Rm	disp	MOV.L @(disp:4,Rm),Rn			
0110	Rn	Rm	00MD	MOV.B Rm,Rn	MOV.W @Rm,Rn	MOV.L @Rm,Rn	MOV Rm,Rn
0110	Rn	Rm	01MD	MOV.B Rm+,Rn	MOV.W @Rm+,Rn	MOV.L @Rm+,Rn	NOT Rm,Rn
0110	Rn	Rm	10MD	SWAP.B Rm,Rn	SWAP.W Rm,Rn	NEGC Rm,Rn	NEG Rm,Rn
0110	Rn	Rm	11MD	EXTU.B Rm,Rn	EXTU.W Rm,Rn	EXTS.B Rm,Rn	EXTS.W Rm,Rn
0111	Rn	imm		ADD #imm:8,Rn			
1000	00MD	Rn	disp	MOV.B R0,@(disp:4,Rn)	MOV.W R0,@(disp:4,Rn)		
1000	01MD	Rm	disp	MOV.B @(disp:4,Rm),R0	MOV.W @(disp:4,Rm),R0		
1000	10MD	imm/disp		CMP/EQ #imm:8,R0	BT label:8		BF label:8
1000	11MD	imm/disp			BT/S label:8*		BF/S label:8*

Table A.50 Operation Code Map (cont)

Instruction Code			Fx: 0000	Fx: 0001	Fx: 0010	Fx: 0011–1111
MSB		LSB	MD: 00	MD: 01	MD: 10	MD: 11
1001	Rn	disp	MOV.W @(disp:8,PC),Rn			
1010	disp		BRA label:12			
1011	disp		BSR label:12			
1100	00MD	imm/disp	MOV.B R0, @(disp:8, GBR)	MOV.W R0, @(disp:8, GBR)	MOV.L R0, @(disp:8, GBR)	TRAPA #imm:8
1100	01MD	disp	MOV.B @(disp:8, GBR),R0	MOV.W @(disp:8, GBR),R0	MOV.L @(disp:8, GBR),R0	MOVA @(disp:8, PC),R0
1100	10MD	imm	TST #imm:8,R0	AND #imm:8,R0	XOR #imm:8,R0	OR #imm:8,R0
1100	11MD	imm	TST.B #imm:8, @(R0,GBR)	AND.B #imm:8, @(R0,GBR)	XOR.B #imm:8, @(R0,GBR)	OR.B #imm:8, @(R0,GBR)
1101	Rn	disp	MOV.L @(disp:8,PC),R0			
1110	Rn	imm	MOV #imm:8,Rn			
1111	...					

Note: SH7600 instructions

Appendix B Pipeline Operation and Contention

The SH7000 series is designed so that basic instructions are executed in one state. Two or more states are required for instructions when, for example, the branch destination address is changed by a branch instruction or when the number of states is increased by contention between MA and IF. Table B.1 gives the number of execution states and stages for different types of contention and their instructions. Instructions without contention and instructions that require 2 or more cycles even without contention are also shown.

Instructions experience contention in the following ways:

- Operations and transfers between registers are executed in one state with no contention.
- No contention occurs, but the instruction still requires 2 or more cycles.
- Contention occurs, increasing the number of execution states. Contention combinations are as follows:
  - MA contends with IF
  - MA contends with IF and sometimes with memory loads as well
  - MA contends with IF and sometimes with the multiplier as well
  - MA contends with IF and sometimes with memory loads and sometimes with the multiplier

Table B.1 Instructions and Their Contention Patterns

Contention	State	Stage	Instruction
None	1	3	Transfer between registers
			Operation between registers (except multiplication instruction)
			Logical operation between registers
			Shift instruction
			System control ALU instruction
	2	3	Unconditional branche
	3/1*3	3	Conditional branche
	3	3	SLEEP instruction
	4	5	RTE instruction
	8	9	TRAP instruction
MA contends with IF	1	4	Memory store instruction and STS.L instruction (PR)
	2	4	STC.L instruction
	3	6	Memory logic operations
	4	6	TAS instruction
MA contends with IF and sometimes with memory loads as well	1	5	Memory load instructions and LDS.L instruction (PR)
	3	5	LDC.L instruction
MA contends with IF and sometimes with the multiplier as well	1	4	Register to MAC transfer instruction, memory to MAC transfer instruction and MAC to memory transfer instruction
	1 to 3 *2	6/7*1	Multiplication instruction
	3/(2)*2	7/8*1	Multiply/accumulate instruction
	3/(2 to 4)*2	9	Double-length multiply/accumulate instruction (SH7600 only)
	2 to 4*2	9	Double-length multiplication instruction (SH7600 only)
MA contends with IF and sometimes with memory loads and sometimes with the multiplier	1	5	MAC to register transfer instruction

Notes: 1. With the SH7600, multiply/accumulate instructions are 7 stages and multiplication instructions are 6 stages, while with the SH7000, multiply/accumulate instructions are 8 stages and multiplication instructions are 7 stages.

2. The normal minimum number of execution states (The number in parentheses is the number in contention with preceding/following instructions).

3. One stage when it does not branch.

HITACHI®

HITACHI AMERICA, LTD.

Semiconductor & I.C. Division
San Francisco Center
2000 Sierra Point Parkway
Brisbane, CA 94005-1835
(415) 589-8300
Fax: (415) 583-4207

Engineering Facility
Hitachi Micro Systems, Inc.
179 East Tasman Drive
San Jose, CA 95134

Manufacturing Facility
Hitachi Semiconductor (America) Inc.
6431 Longhorn Drive
Irving, TX 75063-2712

AREA OFFICES

Eastern
Hitachi America, Ltd.
5511 Capital Center Drive
Suite 204
Raleigh, NC 27606
(919) 233-0800

Central
Hitachi America, Ltd.
Two Lincoln Centre
5420 LBJ Freeway
Suite 1446
Dallas, TX 75240
(214) 991-4510

Western
Hitachi America, Ltd.
1740 Technology Drive
Suite 500
San Jose, CA 95110
(408) 451-9570

REGIONAL OFFICES

Northeast
Hitachi America, Ltd.
77 S. Bedford Street
Burlington, MA 01803
(617) 229-2150

North Central
Hitachi America, Ltd.
500 Park Boulevard
Suite 415
Itasca, IL 60143
(708) 773-4864

Southwest
Hitachi America, Ltd.
2030 Main Street
Suite 450
Irvine, CA 92714
(714) 553-8500

Mountain Pacific
Hitachi America, Ltd.
4600 S. Ulster Street
Suite 690
Denver, CO 80237
(303) 779-5535

AT&T
Hitachi America, Ltd.
325 Columbia Turnpike
Suite 203
Florham Park, NJ 07932
(201) 514-2100

Automotive
Hitachi America, Ltd.
Fairlane Plaza South
Suite 311
290 Town Center Drive
Dearborn, MI 48126
(313) 271-4410

IBM
Hitachi America, Ltd.
21 Old Main Street
Suite 104
Fishkill, NY 12524
(914) 897-3000

DISTRICT OFFICES

Southeast
Hitachi America, Ltd.
4901 N.W. 17th Way
Suite 302
Ft. Lauderdale, FL 33309
(305) 491-6154

Mid-Atlantic
Hitachi America, Ltd.
325 Columbia Turnpike
Suite 203
Florham Park, NJ 07932
(201) 514-2100

Bloomington
Hitachi America, Ltd.
3800 W. 80th Street
Suite 1050
Bloomington, MN 55431
(612) 896-3444

South Central
Hitachi America, Ltd.
One Westchase Center
Suite 1040
10777 Westheimer Road
Houston, TX 77042
(713) 974-0534

IBM Engineering
Hitachi America, Ltd.
9600 Great Hills Trail
Suite 150W
Austin, TX 78759
(512) 502-3033

Ottawa
Hitachi (Canadian) Ltd.
320 March Road
Suite 602
Kanata, Ontario,
Canada K2K2E3
(613) 591-1990

DISTRIBUTORS

Cronin Electronics, Inc. • Marsh Electronics, Inc. •
Marshall Industries • Milgray Electronics, Inc •
Reptron Electronics • Sterling Electronics •
Vantage Components Inc. •

REPRESENTATIVE OFFICES

Electri-Rep • Electronic Sales & Engineering • EIR, Inc. •
Jay Marketing Associates • Longman Sales, Inc. •
M. Gottlieb Associates, Inc. • Mycros Electronica •
The Novus Group, Inc. • Parker-Webster Company • QuadRep I
QuadRep/Crown, Inc. • QuadRep Southern, Inc. •
Robert Electronic Sales • Strategic Sales, Inc. • Sumer Inc. •
System Sales of Arizona • System Sales of New Mexico •
Technology Sales, Inc. • TekRep, Inc. •
Thompson & Associates, Inc. • West Associates •
Wes Tech Associates

Printed on Recycled Paper

295/1500/Banta/N
Order Number: M27T

HITACHI®

Hitachi America, Ltd.

Semiconductor & I.C. Division
San Francisco Center
2000 Sierra Point Parkway
Brisbane, CA 94005-1835
(415) 583-4207

Engineering Facility
Hitachi Micro Systems, Inc.
179 East Tasman Drive
San Jose, CA 95134

Manufacturing Facility
Hitachi Semiconductor (America) Inc.
6321 Longhorn Drive
Irving, TX 75063-2712

Area Offices

Western
Hitachi America, Ltd.
1740 Technology Drive
Suite 500
San Jose, CA 95110
(408) 451-9570

Central
Hitachi America, Ltd.
Two Lincoln Centre
5420 LBJ Freeway
Suite 1446
Dallas, TX 75240
(214) 991-4510

Eastern
Hitachi America, Ltd.
5511 Capital Center Drive
Suite 204
Raleigh, NC 27606
(919) 233-0800

Regional Offices

Western
Hitachi America, Ltd.
2030 Main Street
Suite 450
Irvine, CA 92714
(714) 553-8500

Hitachi America, Ltd.
4600 S. Ulster Street
Suite 690
Denver, CO 80237
(303) 779-5535

Central
Hitachi America, Ltd.
500 Park Boulevard
Suite 415
Itasca, IL 60143
(708) 773-4864

Hitachi America, Ltd.
Fairlane Plaza South
Suite 311
290 Town Center Drive
Dearborn, MI 48126
(313) 271-4410

Eastern
Hitachi America, Ltd.
77 S. Bedford Street
Burlington, MA 01803
(617) 229-2150

Hitachi America, Ltd.
325 Columbia Turnpike
Suite 203
Florham Park, NJ 07932
(201) 514-2100

Hitachi America, Ltd.
21 Old Main Street
Suite 104
Fishkill, NY 12524
(914) 897-3000

District Offices

Central
Hitachi America, Ltd.
3800 W. 80th Street
Suite 1050
Bloomington, MN 55431
(612) 896-3444

Hitachi America, Ltd.
One Westchase Center
1077 Westheimer Drive
Suite 1040
Houston, TX 77042
(713) 974-0534

Hitachi America, Ltd.
9600 Great Hills Trail
Suite 150 W
Austin, TX 78754
(512) 502-3033

Eastern
Hitachi America, Ltd.
4901 N.W. 17th Way
Suite 302
Ft. Lauderdale, FL 33309
(305) 491-6154

Hitachi America, Ltd.
325 Columbia Turnpike
Suite 203
Florham Park, NJ 07932
(201) 514-2100

Hitachi (Canadian) Ltd.
320 March Road
Suite 602
Kanata, Ontario,
Canada K2K2E3
(613) 591-1990

For literature fulfillment request contact Hitachi Literature Fulfillment at 1-800-285-1601

Printed on Recycled Paper

295/1.5M/Banta/MFM
Order Number: M27T021