REJ09B0003-0150Z

Everywhere you imagine. RENESAS

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

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

32

SH-4A

Software Manual

Renesas 32-Bit RISC Microcomputer
SuperH™ RISC engine Family

Software Manual

Rev.1.50
Revision Date: Oct. 29, 2004

Renesas Technology
www.renesas.com

General Precautions on Handling of Product

1. Treatment of NC Pins

Note: Do not connect anything to the NC pins.
The NC (not connected) pins are either not connected to any of the internal circuitry or are they are used as test pins or to reduce noise. If something is connected to the NC pins, the operation of the LSI is not guaranteed.

2. Treatment of Unused Input Pins

Note: Fix all unused input pins to high or low level.
Generally, the input pins of CMOS products are high-impedance input pins. If unused pins are in their open states, intermediate levels are induced by noise in the vicinity, a pass-through current flows internally, and a malfunction may occur.

3. Processing before Initialization

Note: When power is first supplied, the product's state is undefined.
The states of internal circuits are undefined until full power is supplied throughout the chip and a low level is input on the reset pin. During the period where the states are undefined, the register settings and the output state of each pin are also undefined. Design your system so that it does not malfunction because of processing while it is in this undefined state. For those products which have a reset function, reset the LSI immediately after the power supply has been turned on.

4. Prohibition of Access to Undefined or Reserved Addresses

Note: Access to undefined or reserved addresses is prohibited.
The undefined or reserved addresses may be used to expand functions, or test registers may have been be allocated to these addresses. Do not access these registers; the system's operation is not guaranteed if they are accessed.

5. Reading from/Writing to Reserved Bit of Each Register

Note: Treat the reserved bit of register used in each module as follows except in cases where the specifications for values which are read from or written to the bit are provided in the description.
The bit is always read as 0. The write value should be 0 or one, which has been read immediately before writing.
Writing the value, which has been read immediately before writing has the advantage of preventing the bit from being affected on its extended function when the function is assigned.

Configuration of This Manual

This manual comprises the following items:

1. General Precautions on Handling of Product
2. Configuration of This Manual
3. Preface
4. Contents
5. Overview
6. Description of Functional Modules
    - CPU and System-Control Modules
    - On-Chip Peripheral Modules

      The configuration of the functional description of each module differs according to the module. However, the generic style includes the following items:

    i) Feature

    ii) Input/Output Pin

    iii) Register Description

    iv) Operation

    v) Usage Note

When designing an application system that includes this LSI, take notes into account. Each section includes notes in relation to the descriptions given, and usage notes are given, as required, as the final part of each section.

7. List of Registers
8. Appendix
9. Index

Preface

The SH-4A is a RISC (Reduced Instruction Set Computer) microcomputer which includes a Renesas Technology-original RISC CPU as its core.

Target Users: This manual was written for users who will be using the SH-4A in the design of application systems. Users of this manual are expected to understand the fundamentals of electrical circuits, logical circuits, microcomputers, and assembly/C languages programming.

Objective: This manual was written to understand the instructions of the SH4A. For the hardware functions, refer to corresponding hardware manual.

Notes on reading this manual:

- In order to understand the overall functions of the chip

  Read the manual according to the contents. This manual can be roughly categorized into parts on the CPU, system control functions, and instructions.
- In order to understand the instructions

  The instruction format and basic operation are explained in section 3, Instruction Set. For details on each instruction operation, read section 10, Instruction Descriptions.

Rules:	Register name:	The following notation is used for cases when the same or a similar function, e.g. serial communication, is implemented on more than one channel: XXX_N (XXX is the register name and N is the channel number)
	Bit order:	The MSB is on the left and the LSB is on the right.
	Number notation:	Binary is B'xxxx, hexadecimal is H'xxxx, decimal is xxxx.
	Signal notation:	An overbar is added to a low-active signal: $\overline{\text{xxxx}}$

Related Manuals: The latest versions of all related manuals are available from our web site. Please ensure you have the latest versions of all documents you require.
http://www.renesas.com/

Abbreviations

ALU	Arithmetic Logic Unit
ASID	Address Space Identifier
CPU	Central Processing Unit
FPU	Floating Point Unit
LRU	Least Recently Used
LSB	Least Significant Bit
MMU	Memory Management Unit
MSB	Most Significant Bit
PC	Program Counter
RISC	Reduced Instruction Set Computer
TLB	Translation Lookaside Buffer

Contents

Figures

Tables

Section 1 Overview

1.1 Features

The SH-4A is a 32-bit RISC (reduced instruction set computer) microprocessor that is upward compatible with the SH-1, SH-2, SH-3, and SH-4 microcomputers at instruction set code level. Its 16-bit fixed-length instruction set enables program code size to be reduced by almost 50% compared with 32-bit instructions. The features of the SH-4A are listed in table 1.1.

Table 1.1 Features

Item	Features
CPU	• Renesas Technology original architecture • 32-bit internal data bus • General-register files: — Sixteen 32-bit general registers (eight 32-bit shadow registers) — Seven 32-bit control registers — Four 32-bit system registers • RISC-type instruction set (upward compatible with the SH-1, SH-2, SH-3, and SH-4 microcomputers) — Instruction length: 16-bit fixed length for improved code efficiency — Load/store architecture — Delayed branch instructions — Instructions executed with conditions — Instruction set based on the C language • Super scalar which executes two instructions simultaneously including the FPU • Instruction execution time: Two instructions per cycle (max) • Virtual address space: 4 Gbytes • Space identifier ASID: 8 bits, 256 virtual address spaces • On-chip multiplier • Seven-stage pipeline

Item	Features
Floatingpoint unit (FPU)	• On-chip floating-point coprocessor • Supports single-precision (32 bits) and double-precision (64 bits) • Supports IEEE754-compliant data types and exceptions • Two rounding modes: Round to Nearest and Round to Zero • Handling of denormalized numbers: Truncation to zero or interrupt generation for IEEE754 compliance • Floating-point registers: 32 bits × 16 words × 2 banks (single-precision × 16 words or double-precision × 8 words) × 2 banks • 32-bit CPU-FPU floating-point communication register (FPUL) • Supports FMAC (multiply-and-accumulate) instruction • Supports FDIV (divide) and FSQRT (square root) instructions • Supports FLDI0/FLDI1 (load constant 0/1) instructions • Instruction execution times — Latency (FADD/FSUB): 3 cycles (single-precision), 5 cycles (double-precision) — Latency (FMAC/ FMUL): 5 cycles (single-precision), 7 cycles (double-precision) — Pitch (FADD/FSUB): 1 cycle (single-precision/double-precision) — Pitch (FMAC/FMUL): 1 cycle (single-precision), 3 cycles (double-precision) Note: FMAC is supported for single-precision only. • 3-D graphics instructions (single-precision only): — 4-dimensional vector conversion and matrix operations (FTRV): 4 cycles (pitch), 8 cycles (latency) — 4-dimensional vector (FIPR) inner product: 1 cycle (pitch), 5 cycles (latency) • Ten-stage pipeline
Memory management unit (MMU)	• 4 Gbytes of physical address space, 256 address space identifiers (address space identifier ASID: 8 bits) • Supports single virtual memory mode and multiple virtual memory mode • Supports multiple page sizes: 1 Kbyte, 4 Kbytes, 64 Kbytes, or 1 Mbyte • 4-entry full associative TLB for instructions • 64-entry full associative TLB for instructions and operands • Supports software selection of replacement method and random-counter replacement algorithms • Contents of TLB are directly accessible through address mapping

Item	Features
Cache memory	• Instruction cache (IC) — 4-way set associative — 32-byte block length • Operand cache (OC) — 4-way set associative — 32-byte block length — Selectable write method (copy-back or write-through) • Storage queue (32 bytes × 2 entries) Note: For the size of instruction cash and operand cash, see corresponding hardware manual on the product.
L memory	• Two independent read/write ports — 8-/16-/32-/64-bit access from the CPU — 8-/16-/32-/64-bit and 16-/32-byte access from the external devices Note: For the size of L memory, see the hardware manual of the target product.

1.2 Changes from SH-4 to SH-4A

Table 1.2 summarizes the changes from SH-4 to SH-4A based on the sections and sub-sections in this manual.

Table 1.2 Changes from SH-4 to SH-4A

Section No. and Name	Sub-section	Sub-section Name	Changes
1. Overview	—	—	Modified entirely (Detailed differences are described in the following sections).
2. Programming Model	2.2	Register Descriptions	The operations in SZ=1 and PR=1 are added to the floating point status/control register (FPSCR).
3. Instruction Set	3.3	Instruction Set	9 instructions are added as CPU instructions.
			3 instructions are added as FPU instructions.
4. Pipelining	4.1	Pipelines	The number of stages in the pipeline is changed from five to seven.
	4.2	Parallel-Executability	9 instructions are added as CPU instructions.
			3 instructions are added as FPU instructions.
			Instruction group and parallel execution combinations are modified.
	4.3	Execution Cycles	The number of execution cycles is modified.
5. Exception Handling	—	—	—
6. FPU	6.3.2	Floating-Point Status/Control Register (FPSCR)	Operations in SZ = 1 and PR = 1 and each endian are added
	6.5	Floating-Point Exceptions	Specification of FPU exception detection condition with FPU exception enabled is changed.

Section No. and Name	Sub-section	Sub-section Name	Changes
7. Memory Management Unit	7.1.1	Address Spaces	Area P4 configuration is modified.
			On-chip RAM space is deleted.
	7.2	Register Descriptions	The page table entry assist register (PTEA) is deleted.
			A physical address space control register is added.
	7.2.6	Physical Address Space Control Register (PASCR)	Newly added
	7.2.7	Instruction Re-Fetch Inhibit Control Register (IRMCR)	Newly added.
	7.3	TLB Functions	Space attribute bits (SA [2:0]) and timing control bit (TC) are deleted from the TLB.
	7.4.5	Avoiding Synonym Problems	The corresponding bits are modified according to the cache size change and the index mode deletion.
	7.5.1, 7.5.4	Instruction TLB Multiple Hit Exception and Data TLB Multiple Hit Exception	Multiple hits during the UTLB search caused by ITLB mishandling are changed to be handled as a TLB multiple hit instruction exception.
	7.6	Memory-Mapped TLB Configuration	Data array 2 in the ITLB and UTLB is deleted.
	7.6.3	UTLB Address Array	Associative writes to the UTLB address array are changed to not generate data TLB multiple hit exceptions.
			Memory allocated addresses are changed from H'F6000000–H'F6FFFFFF to H'F6000000–H'F60FFFFF.
	7.6.4	UTLB Data Array	Memory allocated addresses are changed from H'F7000000–H'F77FFFFF to H'F7000000–H'F70FFFFF.
	7.7	32-Bit Address Extended Mode	Newly added.

Section No. and Name	Sub-section	Sub-section Name	Changes
8. Caches	8.1	Features	Instruction cache capacity is changed to 32 Kbytes.
			The caching method is changed to a 4-way set-associative method.
	8.2	Register Descriptions	An on-chip memory control register is added.
	8.2.1	Cache Control Register (CCR)	Modified. (Descriptions in CCR are modified.)
	8.2.4	On-Chip Memory Control Register (RAMCR)	Newly added.
	8.3	Operand Cache Operation	RAM mode and OC index mode are deleted.
	8.3.6	OC Two-Way Mode	Newly added.
	8.4	Instruction Cache Operation	IC index mode is deleted.
	8.4.3	IC Two-Way Mode	Newly added.
	8.5.1	Coherency between Cache and External Memory	The ICBI, PREFI, and SYNCO instructions are added.
	8.6	Memory-Mapped Cache Configuration	The entry bits and the way bits are modified according to the size modification and changed into 4-way set associative cache.
	8.8	Notes on Using 32-Bit Address Extended Mode	Newly added.
9. L Memory	—	—	Newly added.
10. Instruction Descriptions	—	—	9 instructions are added as CPU instructions.
			3 instructions are added as FPU instructions.

Section 2 Programming Model

The programming model of the SH-4A is explained in this section. The SH-4A has registers and data formats as shown below.

2.1 Data Formats

The data formats supported in the SH-4A are shown in figure 2.1.

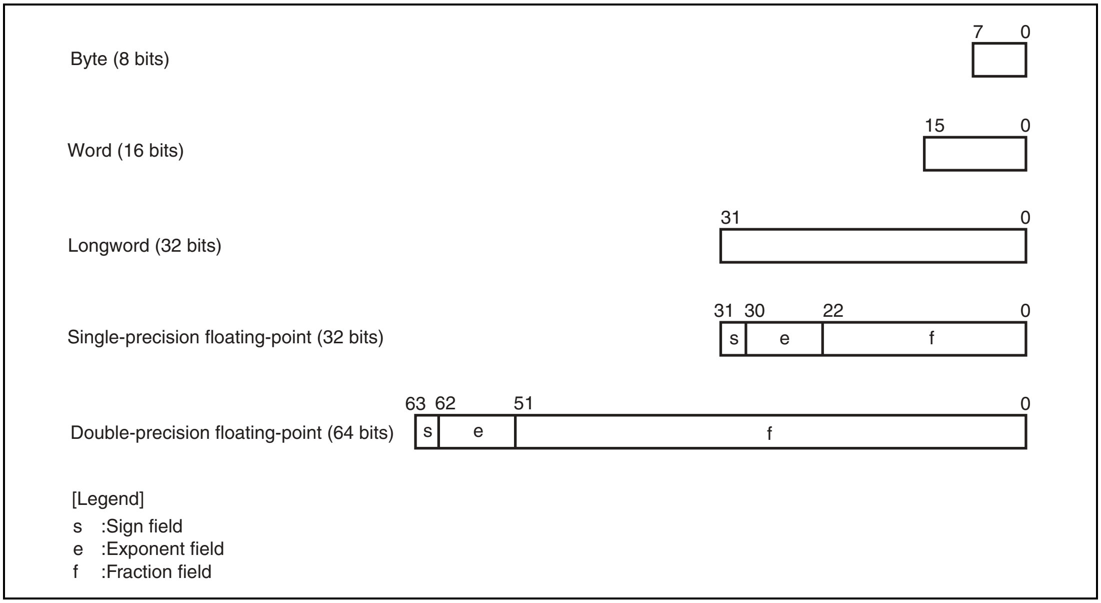


Figure 2.1 Data Formats

2.2 Register Descriptions

2.2.1 Privileged Mode and Banks

Processing Modes: This LSI has two processing modes, user mode and privileged mode. This LSI normally operates in user mode, and switches to privileged mode when an exception occurs or an interrupt is accepted. There are four kinds of registers—general registers, system registers, control registers, and floating-point registers—and the registers that can be accessed differ in the two processing modes.

General Registers: There are 16 general registers, designated R0 to R15. General registers R0 to R7 are banked registers which are switched by a processing mode change.

- Privileged mode

  In privileged mode, the register bank bit (RB) in the status register (SR) defines which banked register set is accessed as general registers, and which set is accessed only through the load control register (LDC) and store control register (STC) instructions.

  When the RB bit is 1 (that is, when bank 1 is selected), the 16 registers comprising bank 1 general registers R0_BANK1 to R7_BANK1 and non-banked general registers R8 to R15 can be accessed as general registers R0 to R15. In this case, the eight registers comprising bank 0 general registers R0_BANK0 to R7_BANK0 are accessed by the LDC/STC instructions. When the RB bit is 0 (that is, when bank 0 is selected), the 16 registers comprising bank 0 general registers R0_BANK0 to R7_BANK0 and non-banked general registers R8 to R15 can be accessed as general registers R0 to R15. In this case, the eight registers comprising bank 1 general registers R0_BANK1 to R7_BANK1 are accessed by the LDC/STC instructions.
- User mode

  In user mode, the 16 registers comprising bank 0 general registers R0_BANK0 to R7_BANK0 and non-banked general registers R8 to R15 can be accessed as general registers R0 to R15. The eight registers comprising bank 1 general registers R0_BANK1 to R7_BANK1 cannot be accessed.

Control Registers: Control registers comprise the global base register (GBR) and status register (SR), which can be accessed in both processing modes, and the saved status register (SSR), saved program counter (SPC), vector base register (VBR), saved general register 15 (SGR), and debug base register (DBR), which can only be accessed in privileged mode. Some bits of the status register (such as the RB bit) can only be accessed in privileged mode.

System Registers: System registers comprise the multiply-and-accumulate registers (MACH/MACL), the procedure register (PR), and the program counter (PC). Access to these registers does not depend on the processing mode.

Floating-Point Registers and System Registers Related to FPU: There are thirty-two floating-point registers, FR0–FR15 and XF0–XF15. FR0–FR15 and XF0–XF15 can be assigned to either of two banks (FPR0_BANK0–FPR15_BANK0 or FPR0_BANK1–FPR15_BANK1).

FR0–FR15 can be used as the eight registers DR0/2/4/6/8/10/12/14 (double-precision floating-point registers, or pair registers) or the four registers FV0/4/8/12 (register vectors), while XF0–XF15 can be used as the eight registers XD0/2/4/6/8/10/12/14 (register pairs) or register matrix XMTRX.

System registers related to the FPU comprise the floating-point communication register (FPUL) and the floating-point status/control register (FPSCR). These registers are used for communication between the FPU and the CPU, and the exception handling setting.

Register values after a reset are shown in table 2.1.

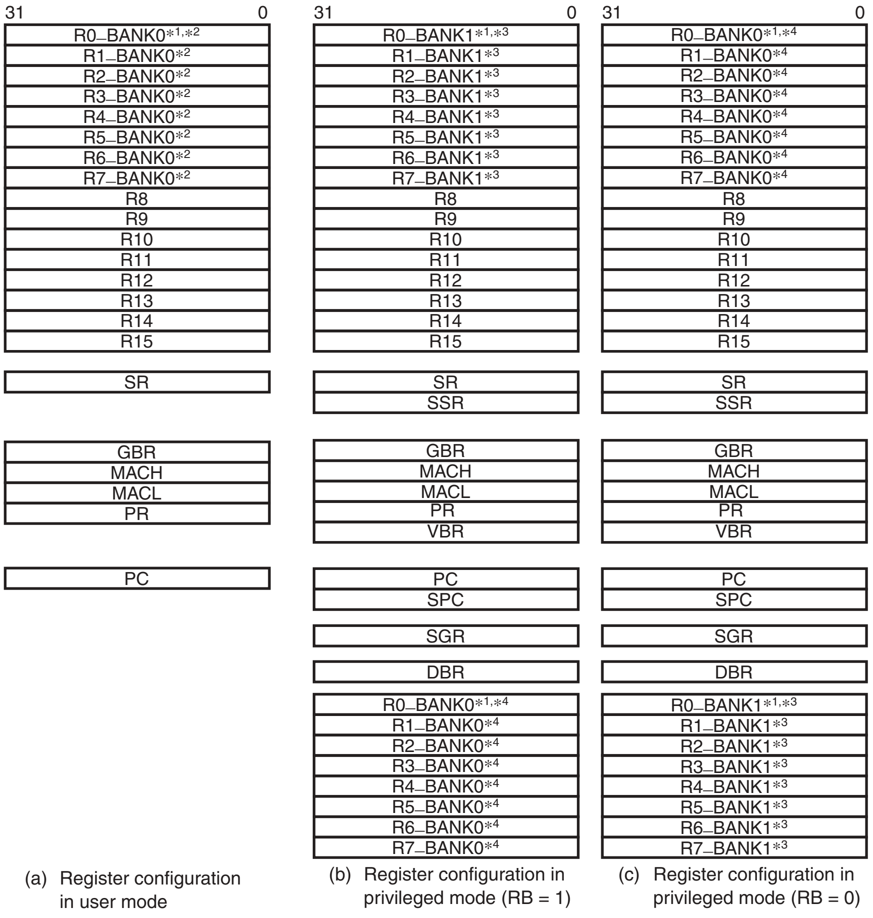
Table 2.1 Initial Register Values

Type	Registers	Initial Value*
General registers	R0_BANK0 to R7_BANK0, R0_BANK1 to R7_BANK1, R8 to R15	Undefined
Control registers	SR	MD bit = 1, RB bit = 1, BL bit = 1, FD bit = 0, IMASK = B'1111, reserved bits = 0, others = undefined
	GBR, SSR, SPC, SGR, DBR	Undefined
	VBR	H'00000000
System registers	MACH, MACL, PR	Undefined
	PC	H'A0000000
Floating-point registers	FR0 to FR15, XF0 to XF15, FPUL	Undefined
	FPSCR	H'00040001

Note: * Initialized by a power-on reset and manual reset.

The CPU register configuration in each processing mode is shown in figure 2.2.

User mode and privileged mode are switched by the processing mode bit (MD) in the status register.



Notes: 1. R0 is used as the index register in indexed register-indirect addressing mode and indexed GBR indirect addressing mode.
2. Banked registers
3. Banked registers
Accessed as general registers when the RB bit is set to 1 in SR. Accessed only by LDC/STC instructions when the RB bit is cleared to 0.
4. Banked registers
Accessed as general registers when the RB bit is cleared to 0 in SR. Accessed only by LDC/STC instructions when the RB bit is set to 1.

Figure 2.2 CPU Register Configuration in Each Processing Mode

2.2.2 General Registers

Figure 2.3 shows the relationship between the processing modes and general registers. The SH-4A has twenty-four 32-bit general registers (R0_BANK0 to R7_BANK0, R0_BANK1 to R7_BANK1, and R8 to R15). However, only 16 of these can be accessed as general registers R0 to R15 in one processing mode. The SH-4A has two processing modes, user mode and privileged mode.

- R0_BANK0 to R7_BANK0

  Allocated to R0 to R7 in user mode (SR.MD = 0)

  Allocated to R0 to R7 when SR.RB = 0 in privileged mode (SR.MD = 1).
- R0_BANK1 to R7_BANK1

  Cannot be accessed in user mode.

  Allocated to R0 to R7 when SR.RB = 1 in privileged mode.

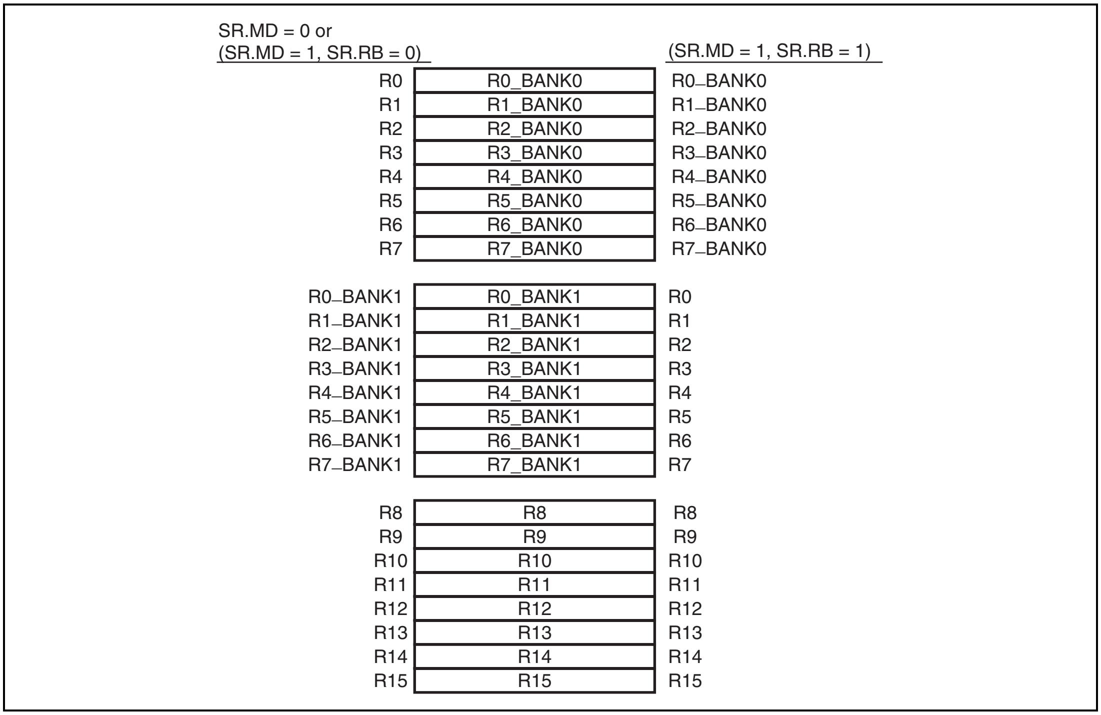


Figure 2.3 General Registers

Note on Programming: As the user's R0 to R7 are assigned to R0_BANK0 to R7_BANK0, and after an exception or interrupt R0 to R7 are assigned to R0_BANK1 to R7_BANK1, it is not necessary for the interrupt handler to save and restore the user's R0 to R7 (R0_BANK0 to R7_BANK0).

2.2.3 Floating-Point Registers

Figure 2.4 shows the floating-point register configuration. There are thirty-two 32-bit floating-point registers, FPR0_BANK0 to FPR15_BANK0, AND FPR0_BANK1 to FPR15_BANK1, comprising two banks. These registers are referenced as FR0 to FR15, DR0/2/4/6/8/10/12/14, FV0/4/8/12, XF0 to XF15, XD0/2/4/6/8/10/12/14, or XMTRX. Reference names of each register are defined depending on the state of the FR bit in FPSCR (see figure 2.4).

1. Floating-point registers, FPRn_BANKj (32 registers)
   FPR0_BANK0 to FPR15_BANK0
   FPR0_BANK1 to FPR15_BANK1

2. Single-precision floating-point registers, FRi (16 registers)
   When FPSCR.FR = 0, FR0 to FR15 are assigned to FPR0_BANK0 to FPR15_BANK0;
   when FPSCR.FR = 1, FR0 to FR15 are assigned to FPR0_BANK1 to FPR15_BANK1.

3. Double-precision floating-point registers or single-precision floating-point registers, DRi (8 registers): A DR register comprises two FR registers.
   DR0 = {FR0, FR1}, DR2 = {FR2, FR3}, DR4 = {FR4, FR5}, DR6 = {FR6, FR7},
   DR8 = {FR8, FR9}, DR10 = {FR10, FR11}, DR12 = {FR12, FR13}, DR14 = {FR14, FR15}

4. Single-precision floating-point vector registers, FVi (4 registers): An FV register comprises four FR registers.
   FV0 = {FR0, FR1, FR2, FR3}, FV4 = {FR4, FR5, FR6, FR7},
   FV8 = {FR8, FR9, FR10, FR11}, FV12 = {FR12, FR13, FR14, FR15}

5. Single-precision floating-point extended registers, XFi (16 registers)
   When FPSCR.FR = 0, XF0 to XF15 are assigned to FPR0_BANK1 to FPR15_BANK1;
   when FPSCR.FR = 1, XF0 to XF15 are assigned to FPR0_BANK0 to FPR15_BANK0.

6. Double-precision floating-point extended registers, XDi (8 registers): An XD register comprises two XF registers.
   XD0 = {XF0, XF1}, XD2 = {XF2, XF3}, XD4 = {XF4, XF5}, XD6 = {XF6, XF7},
   XD8 = {XF8, XF9}, XD10 = {XF10, XF11}, XD12 = {XF12, XF13}, XD14 = {XF14, XF15}

7. Single-precision floating-point extended register matrix, XMTRX: XMTRX comprises all 16 XF registers.

$$\mathrm{XMTRX} = \begin{bmatrix} \mathrm{XF0} & \mathrm{XF4} & \mathrm{XF8} & \mathrm{XF12} \\ \mathrm{XF1} & \mathrm{XF5} & \mathrm{XF9} & \mathrm{XF13} \\ \mathrm{XF2} & \mathrm{XF6} & \mathrm{XF10} & \mathrm{XF14} \\ \mathrm{XF3} & \mathrm{XF7} & \mathrm{XF11} & \mathrm{XF15} \end{bmatrix}$$

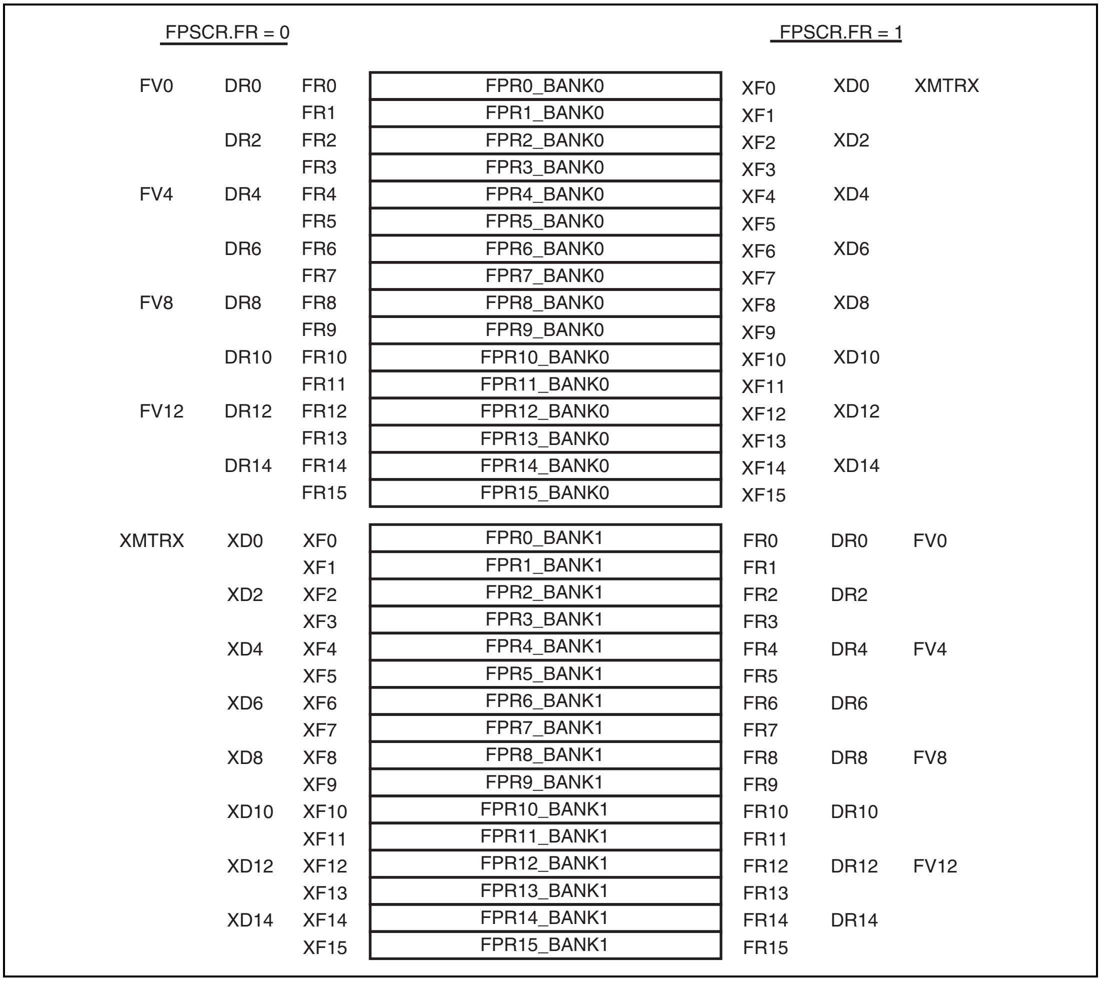

FPSCR.FR = 0
FPSCR.FR = 1
FV0 DR0 FR0 FPR0_BANK0 XF0 XD0 XMTRX
FR1 FPR1_BANK0 XF1
DR2 FR2 FPR2_BANK0 XF2 XD2
FR3 FPR3_BANK0 XF3
FV4 DR4 FR4 FPR4_BANK0 XF4 XD4
FR5 FPR5_BANK0 XF5
DR6 FR6 FPR6_BANK0 XF6 XD6
FR7 FPR7_BANK0 XF7
FV8 DR8 FR8 FPR8_BANK0 XF8 XD8
FR9 FPR9_BANK0 XF9
DR10 FR10 FPR10_BANK0 XF10 XD10
FR11 FPR11_BANK0 XF11
FV12 DR12 FR12 FPR12_BANK0 XF12 XD12
FR13 FPR13_BANK0 XF13
DR14 FR14 FPR14_BANK0 XF14 XD14
FR15 FPR15_BANK0 XF15
XMTRX XD0 XF0 FPR0_BANK1 FR0 DR0 FV0
XF1 FPR1_BANK1 FR1
XD2 XF2 FPR2_BANK1 FR2 DR2
XF3 FPR3_BANK1 FR3
XD4 XF4 FPR4_BANK1 FR4 DR4 FV4
XF5 FPR5_BANK1 FR5
XD6 XF6 FPR6_BANK1 FR6 DR6
XF7 FPR7_BANK1 FR7
XD8 XF8 FPR8_BANK1 FR8 DR8 FV8
XF9 FPR9_BANK1 FR9
XD10 XF10 FPR10_BANK1 FR10 DR10
XF11 FPR11_BANK1 FR11
XD12 XF12 FPR12_BANK1 FR12 DR12 FV12
XF13 FPR13_BANK1 FR13
XD14 XF14 FPR14_BANK1 FR14 DR14
XF15 FPR15_BANK1 FR15

Figure 2.4 Floating-Point Registers

2.2.4 Control Registers

Status Register (SR)

BIt:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	—	MD	RB	BL	—	—	—	—	—	—	—	—	—	—	—	—
Initial value:	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0
R/W:	R	R/W	R/W	R/W	R	R	R	R	R	R	R	R	R	R	R	R

BIt:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	FD	—	—	—	—	—	M	Q	IMASK				—	—	S	T
Initial value:	0	0	0	0	0	0	0	0	1	1	1	1	0	0	0	0
R/W:	R/W	R	R	R	R	R	R/W	R/W	R/W	R/W	R/W	R/W	R	R	R/W	R/W

Bit	Bit Name	Initial Value	R/W	Description
31	—	0	R	Reserved For details on reading/writing this bit, see General Precautions on Handling of Product.
30	MD	1	R/W	Processing Mode Selects the processing mode. 0: User mode (Some instructions cannot be executed and some resources cannot be accessed.) 1: Privileged mode This bit is set to 1 by an exception or interrupt.
29	RB	1	R/W	Privileged Mode General Register Bank Specification Bit 0: R0_BANK0 to R7_BANK0 are accessed as general registers R0 to R7 and R0_BANK1 to R7_BANK1 can be accessed using LDC/STC instructions 1: R0_BANK1 to R7_BANK1 are accessed as general registers R0 to R7 and R0_BANK0–R7_BANK0 can be accessed using LDC/STC instructions This bit is set to 1 by an exception or interrupt.
28	BL	1	R/W	Exception/Interrupt Block Bit This bit is set to 1 by a reset, an exception, or an interrupt. While this bit is set to 1, an interrupt request is masked. In this case, this processor enters the reset state when a general exception other than a user break occurs.
27 to 16	—	All 0	R	Reserved For details on reading/writing this bit, see General Precautions on Handling of Product.

Bit	Bit Name	Initial Value	R/W	Description
15	FD	0	R/W	FPU Disable Bit When this bit is set to 1 and an FPU instruction is not in a delay slot, a general FPU disable exception occurs. When this bit is set to 1 and an FPU instruction is in a delay slot, a slot FPU disable exception occurs. (FPU instructions: H'F*** instructions and LDS (.L)/STS(.L) instructions using FPUL/FPSCR)
14 to 10	—	All 0	R	Reserved For details on reading/writing this bit, see General Precautions on Handling of Product.
9	M	0	R/W	M Bit Used by the DIV0S, DIV0U, and DIV1 instructions.
8	Q	0	R/W	Q Bit Used by the DIV0S, DIV0U, and DIV1 instructions.
7 to 4	IMASK	All 1	R/W	Interrupt Mask Level Bits An interrupt whose priority is equal to or less than the value of the IMASK bits is masked. It can be chosen by CPU operation mode register (CPUOPM) whether the level of IMASK is changed to accept an interrupt or not when an interrupt is occurred. For details, see Appendix A, CPU Operation Mode Register (CPUOPM).
3, 2	—	All 0	R	Reserved For details on reading/writing this bit, see General Precautions on Handling of Product.
1	S	0	R/W	S Bit Used by the MAC instruction.
0	T	0	R/W	T Bit Indicates true/false condition, carry/borrow, or overflow/underflow. For details, see section 3, Instruction Set.

Saved Status Register (SSR) (32 bits, Privileged Mode, Initial Value = Undefined): The contents of SR are saved to SSR in the event of an exception or interrupt.

Saved Program Counter (SPC) (32 bits, Privileged Mode, Initial Value = Undefined): The address of an instruction at which an interrupt or exception occurs is saved to SPC.

Global Base Register (GBR) (32 bits, Initial Value = Undefined): GBR is referenced as the base address of addressing @(disp,GBR) and @(R0,GBR).

Vector Base Register (VBR) (32 bits, Privileged Mode, Initial Value = H'00000000): VBR is referenced as the branch destination base address in the event of an exception or interrupt. For details, see section 5, Exception Handling.

Saved General Register 15 (SGR) (32 bits, Privileged Mode, Initial Value = Undefined): The contents of R15 are saved to SGR in the event of an exception or interrupt.

Debug Base Register (DBR) (32 bits, Privileged Mode, Initial Value = Undefined): When the user break debugging function is enabled (CBCR.UBDE = 1), DBR is referenced as the branch destination address of the user break handler instead of VBR.

2.2.5 System Registers

Multiply-and-Accumulate Registers (MACH and MACL) (32 bits, Initial Value = Undefined): MACH and MACL are used for the added value in a MAC instruction, and to store the operation result of a MAC or MUL instruction.

Procedure Register (PR) (32 bits, Initial Value = Undefined): The return address is stored in PR in a subroutine call using a BSR, BSRF, or JSR instruction. PR is referenced by the subroutine return instruction (RTS).

Program Counter (PC) (32 bits, Initial Value = H'A0000000): PC indicates the address of the instruction currently being executed.

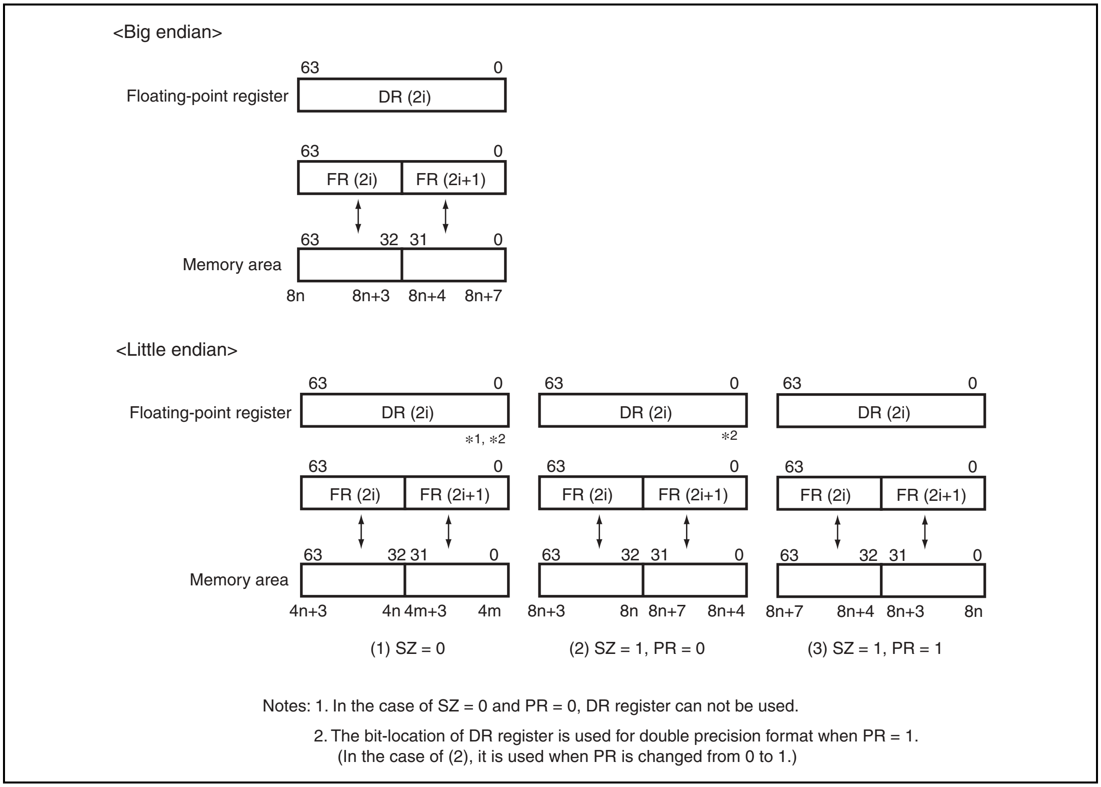
Floating-Point Status/Control Register (FPSCR)

Bit:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	—	—	—	—	—	—	—	—	—	—	FR	SZ	PR	DN	Cause	
Initial value:	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
R/W:	R	R	R	R	R	R	R	R	R	R	R/W	R/W	R/W	R/W	R/W	R/W

Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	Cause				Enable (EN)					Flag					RM	
Initial value:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
R/W:	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

Bit	Bit Name	Initial Value	R/W	Description
31 to 22	—	All 0	R	Reserved For details on reading/writing this bit, see General Precautions on Handling of Product.
21	FR	0	R/W	Floating-Point Register Bank 0: FPR0_BANK0 to FPR15_BANK0 are assigned to FR0 to FR15 and FPR0_BANK1 to FPR15_BANK1 are assigned to XF0 to XF15 1: FPR0_BANK0 to FPR15_BANK0 are assigned to XF0 to XF15 and FPR0_BANK1 to FPR15_BANK1 are assigned to FR0 to FR15
20	SZ	0	R/W	Transfer Size Mode 0: Data size of FMOV instruction is 32-bits 1: Data size of FMOV instruction is a 32-bit register pair (64 bits) For relationship between the SZ bit, PR bit, and endian, see figure 2.5.
19	PR	0	R/W	Precision Mode 0: Floating-point instructions are executed as single-precision operations 1: Floating-point instructions are executed as double-precision operations (graphics support instructions are undefined) For relationship between the SZ bit, PR bit, and endian, see figure 2.5
18	DN	1	R/W	Denormalization Mode 0: Denormalized number is treated as such 1: Denormalized number is treated as zero

Bit	Bit Name	Initial Value	R/W	Description
17 to 12	Cause	All 0	R/W	FPU Exception Cause Field
11 to 7	Enable (EN)	All 0	R/W	FPU Exception Enable Field
6 to 2	Flag	All 0	R/W	FPU Exception Flag Field Each time an FPU operation instruction is executed, the FPU exception cause field is cleared to 0. When an FPU exception occurs, the bits corresponding to FPU exception cause field and flag field are set to 1. The FPU exception flag field remains set to 1 until it is cleared to 0 by software. For bit allocations of each field, see table 2.2.
1, 0	RM	01	R/W	Rounding Mode These bits select the rounding mode. 00: Round to Nearest 01: Round to Zero 10: Reserved 11: Reserved



Notes: 1. In the case of SZ = 0 and PR = 0, DR register can not be used.
2. The bit-location of DR register is used for double precision format when PR = 1. (In the case of (2), it is used when PR is changed from 0 to 1.)

Figure 2.5 Relationship between SZ bit and Endian

Table 2.2 Bit Allocation for FPU Exception Handling

Field Name		FPU Error (E)	Invalid Operation (V)	Division by Zero (Z)	Overflow (O)	Underflow (U)	Inexact (I)
Cause	FPU exception cause field	Bit 17	Bit 16	Bit 15	Bit 14	Bit 13	Bit 12
Enable	FPU exception enable field	None	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7
Flag	FPU exception flag field	None	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2

Floating-Point Communication Register (FPUL) (32 bits, Initial Value = Undefined): Information is transferred between the FPU and CPU via FPUL.

2.3 Memory-Mapped Registers

Some control registers are mapped to the following memory areas. Each of the mapped registers has two addresses.

H'1C00 0000 to H'1FFF FFFF
H'FC00 0000 to H'FFFF FFFF

These two areas are used as follows.

- H'1C00 0000 to H'1FFF FFFF

  This area must be accessed using the address translation function of the MMU.

  Setting the page number of this area to the corresponding field of the TLB enables access to a memory-mapped register.

  The operation of an access to this area without using the address translation function of the MMU is not guaranteed.
- H'FC00 0000 to H'FFFF FFFF

  Access to area H'FC00 0000 to H'FFFF FFFF in user mode will cause an address error. Memory-mapped registers can be referenced in user mode by means of access that involves address translation.

Note: Do not access addresses to which registers are not mapped in either area. The operation of an access to an address with no register mapped is undefined. Also, memory-mapped registers must be accessed using a fixed data size. The operation of an access using an invalid data size is undefined.

2.4 Data Formats in Registers

Register operands are always longwords (32 bits). When a 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.

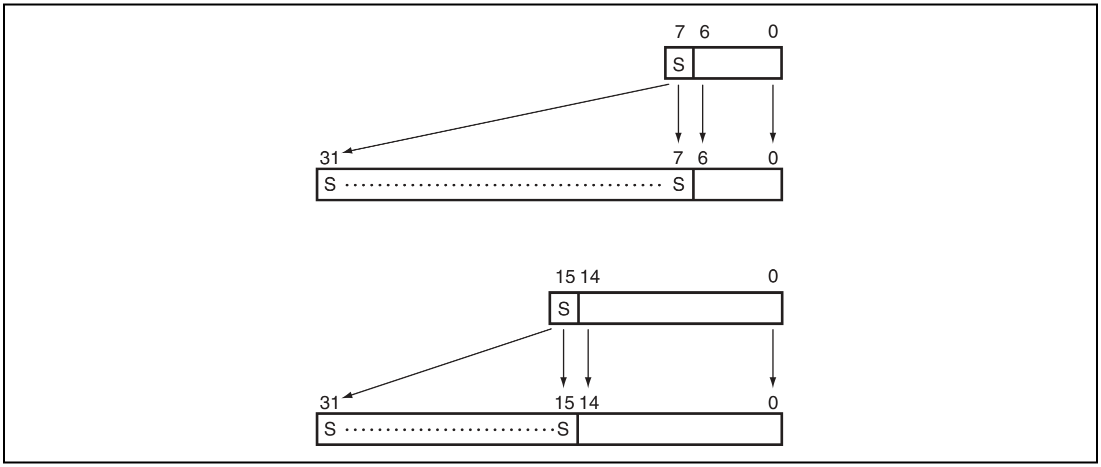

7
6
0
S
31
7
6
0
S ······································· S
15
14
0
S
31
15
14
0
S ··························S

Figure 2.6 Formats of Byte Data and Word Data in Register

2.5 Data Formats in Memory

Memory data formats are classified into bytes, words, and longwords. Memory can be accessed in an 8-bit byte, 16-bit word, or 32-bit longword form. A memory operand less than 32 bits in length is sign-extended before being loaded into a register.

A word operand must be accessed starting from a word boundary (even address of a 2-byte unit: address 2n), and a longword operand starting from a longword boundary (even address of a 4-byte unit: address 4n). An address error will result if this rule is not observed. A byte operand can be accessed from any address.

Big endian or little endian byte order can be selected for the data format. The endian should be set with the external pin after a power-on reset. The endian cannot be changed dynamically. Bit positions are numbered left to right from most-significant to least-significant. Thus, in a 32-bit longword, the leftmost bit, bit 31, is the most significant bit and the rightmost bit, bit 0, is the least significant bit.

The data format in memory is shown in figure 2.7.

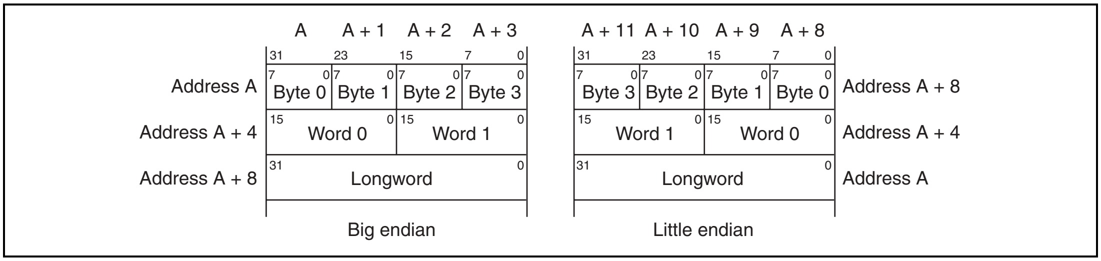


Figure 2.7 Data Formats in Memory

For the 64-bit data format, see figure 2.5.

2.6 Processing States

This LSI has major three processing states: the reset state, instruction execution state, and power-down state.

Reset State: In this state the CPU is reset. The reset state is divided into the power-on reset state and the manual reset.

In the power-on reset state, the internal state of the CPU and the on-chip peripheral module registers are initialized. In the manual reset state, the internal state of the CPU and some registers of on-chip peripheral modules are initialized. For details, see register descriptions for each section.

Instruction Execution State: In this state, the CPU executes program instructions in sequence. The Instruction execution state has the normal program execution state and the exception handling state.

Power-Down State: In a power-down state, CPU halts operation and power consumption is reduced. The power-down state is entered by executing a SLEEP instruction. There are two modes in the power-down state: sleep mode and standby mode.

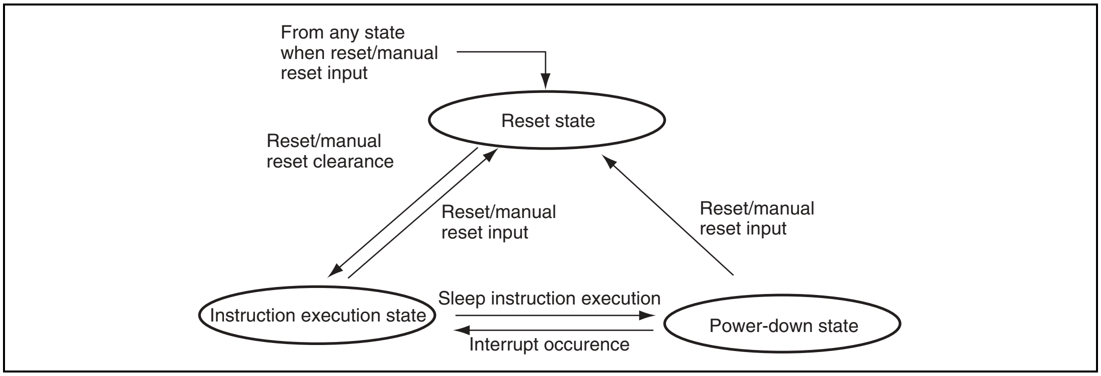

From any state when reset/manual reset input
Reset state
Reset/manual reset clearance
Reset/manual reset input
Reset/manual reset input
Instruction execution state
Sleep instruction execution
Interrupt occurence
Power-down state

Figure 2.8 Processing State Transitions

2.7 Usage Notes

2.7.1 Notes on Self-Modified Codes

The SH-4A prefetches instructions to accelerate the processing speed. Therefore if the instruction in the memory is modified and it is executed immediately, then the pre-modified code in the prefetch buffer may be executed. And the SH4AL-DSP supports each instruction and operand cache, the coherency should be considered. In order to reflect the modified code definitely, one of the following sequences should be executed.

In Case the Modified Codes are in Non-Cacheable Area:

```
SYNCO
ICBI @Rn
```

The target for the ICBI instruction can be any address within the range where no address error exception occurs.

In Case the Modified Codes are in Cacheable Area (Write-Through):

```
SYNCO
ICBI @Rn
```

All instruction cache areas corresponding to the modified codes should be invalidated by the ICBI instruction. The ICBI instruction should be issued to each cache line. One cache line is 32 bytes.

In Case the Modified Codes are in Cacheable Area (Copy-Back):

```
OCBP @Rm or OCBWB @Rm
SYNCO
ICBI @Rn
```

All operand cache areas corresponding to the modified codes should be written back to the main memory by the OCBP or OCBWB instruction. Then all instruction cache areas corresponding to the modified codes should be invalidated by the ICBI instruction. The OCBP, OCBWB, and ICBI instruction should be issued to each cache line. One cache line is 32 bytes.

Section 3 Instruction Set

The SH-4A's instruction set is implemented with 16-bit fixed-length instructions. The SH-4A can use byte (8-bit), word (16-bit), longword (32-bit), and quadword (64-bit) data sizes for memory access. Single-precision floating-point data (32 bits) can be moved to and from memory using longword or quadword size. Double-precision floating-point data (64 bits) can be moved to and from memory using longword size. When the SH-4A moves byte-size or word-size data from memory to a register, the data is sign-extended.

3.1 Execution Environment

PC: At the start of instruction execution, the PC indicates the address of the instruction itself.

Load-Store Architecture: The SH-4A has a load-store architecture in which operations are basically executed using registers. Except for bit-manipulation operations such as logical AND that are executed directly in memory, operands in an operation that requires memory access are loaded into registers and the operation is executed between the registers.

Delayed Branches: Except for the two branch instructions BF and BT, the SH-4A's branch instructions and RTE are delayed branches. In a delayed branch, the instruction following the branch is executed before the branch destination instruction.

Delay Slot: This execution slot following a delayed branch is called a delay slot. For example, the BRA execution sequence is as follows:

Table 3.1 Execution Order of Delayed Branch Instructions

	Instructions			Execution Order
	BRA	TARGET	(Delayed branch instruction)	BRA
	ADD		(Delay slot)	↓
	:			ADD
	:			↓
TARGET	target-inst		(Branch destination instruction)	target-inst

A slot illegal instruction exception may occur when a specific instruction is executed in a delay slot. For details, see section 5, Exception Handling. The instruction following BF/S or BT/S for which the branch is not taken is also a delay slot instruction.

T Bit: The T bit in SR is used to show the result of a compare operation, and is referenced by a conditional branch instruction. An example of the use of a conditional branch instruction is shown below.

```
ADD       #1, R0     ; T bit is not changed by ADD operation
CMP/EQ    R1, R0     ; If R0 = R1, T bit is set to 1
BT        TARGET     ; Branches to TARGET if T bit = 1 (R0 = R1)
```

In an RTE delay slot, the SR bits are referenced as follows. In instruction access, the MD bit is used before modification, and in data access, the MD bit is accessed after modification. The other bits—S, T, M, Q, FD, BL, and RB—after modification are used for delay slot instruction execution. The STC and STC.L SR instructions access all SR bits after modification.

Constant Values: An 8-bit constant value can be specified by the instruction code and an immediate value. 16-bit and 32-bit constant values can be defined as literal constant values in memory, and can be referenced by a PC-relative load instruction.

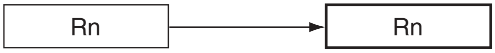
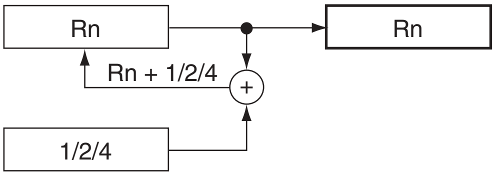
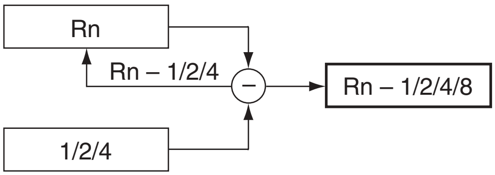
```
MOV.W    @(disp, PC), Rn
MOV.L    @(disp, PC), Rn
```

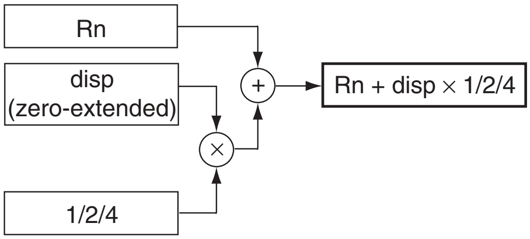
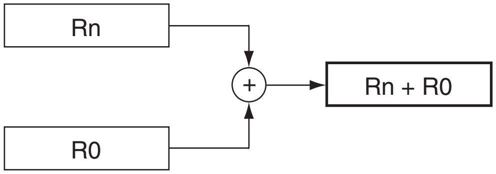
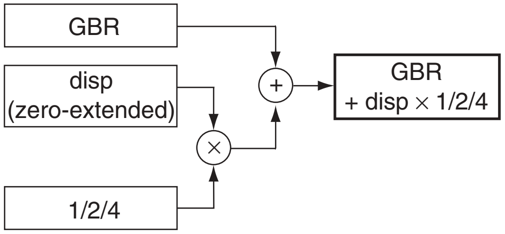
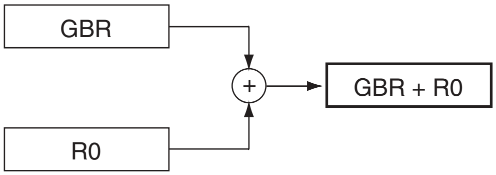
There are no PC-relative load instructions for floating-point operations. However, it is possible to set 0.0 or 1.0 by using the FLDI0 or FLDI1 instruction on a single-precision floating-point register.

3.2 Addressing Modes

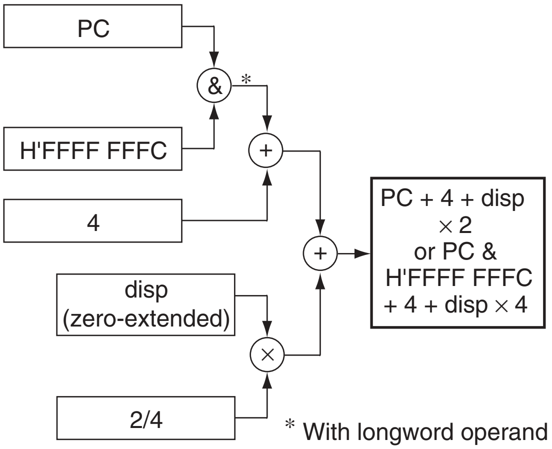
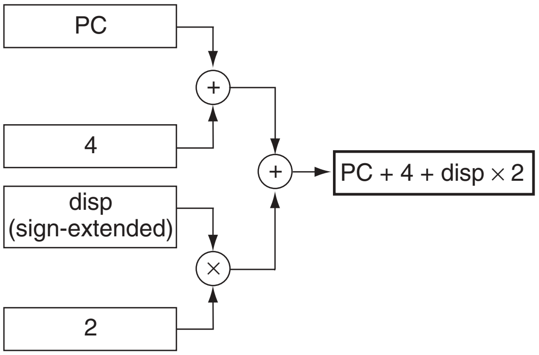
Addressing modes and effective address calculation methods are shown in table 3.2. When a location in virtual memory space is accessed (AT in MMUCR = 1), the effective address is translated into a physical memory address. If multiple virtual memory space systems are selected (SV in MMUCR = 0), the least significant bit of PTEH is also referenced as the access ASID. For details, see section 7, Memory Management Unit (MMU).

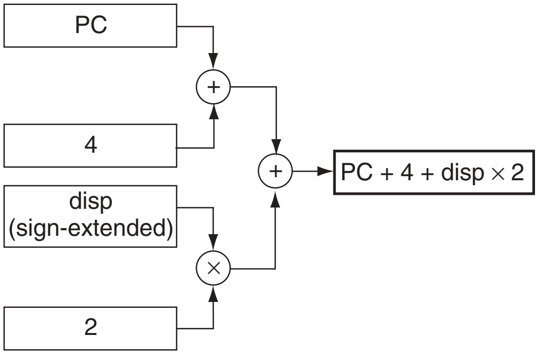
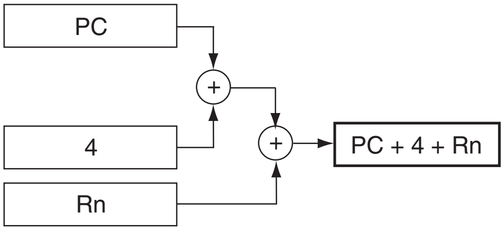
Table 3.2 Addressing Modes and Effective Addresses

Addressing Mode	Instruction Format	Effective Address Calculation Method	Calculation Formula
Register direct	Rn	Effective address is register Rn. (Operand is register Rn contents.)	—
Register indirect	@Rn	Effective address is register Rn contents. 	Rn → EA (EA: effective address)
Register indirect with post-increment	@Rn+	Effective address is register Rn contents. A constant is added to Rn after instruction execution: 1 for a byte operand, 2 for a word operand, 4 for a longword operand, 8 for a quadword operand. 	Rn → EA After instruction execution Byte: Rn + 1 → Rn Word: Rn + 2 → Rn Longword: Rn + 4 → Rn Quadword: Rn + 8 → Rn
Register indirect with pre-decrement	@–Rn	Effective address is register Rn contents, decremented by a constant beforehand: 1 for a byte operand, 2 for a word operand, 4 for a longword operand, 8 for a quadword operand. 	Byte: Rn – 1 → Rn Word: Rn – 2 → Rn Longword: Rn – 4 → Rn Quadword: Rn – 8 → Rn Rn → EA (Instruction executed with Rn after calculation)

Addressing Mode	Instruction Format	Effective Address Calculation Method	Calculation Formula
Register indirect with displacement	@(disp:4, Rn)	Effective address is register Rn contents with 4-bit displacement disp added. After disp is zero-extended, it is multiplied by 1 (byte), 2 (word), or 4 (longword), according to the operand size.  Rn disp (zero-extended) 1/2/4 + × Rn + disp × 1/2/4	Byte: Rn + disp → EA Word: Rn + disp × 2 → EA Longword: Rn + disp × 4 → EA
Indexed register indirect	@(R0, Rn)	Effective address is sum of register Rn and R0 contents. 	Rn + R0 → EA
GBR indirect with displace-ment	@(disp:8, GBR)	Effective address is register GBR contents with 8-bit displacement disp added. After disp is zero-extended, it is multiplied by 1 (byte), 2 (word), or 4 (longword), according to the operand size.  GBR disp (zero-extended) 1/2/4 + × GBR + disp × 1/2/4	Byte: GBR + disp → EA Word: GBR + disp × 2 → EA Longword: GBR + disp × 4 → EA
Indexed GBR indirect	@(R0, GBR)	Effective address is sum of register GBR and R0 contents. 	GBR + R0 → EA

RENESAS

Addressing Mode	Instruction Format	Effective Address Calculation Method	Calculation Formula
PC-relative with displacement	@(disp:8, PC)	Effective address is PC + 4 with 8-bit displacement disp added. After disp is zero-extended, it is multiplied by 2 (word), or 4 (longword), according to the operand size. With a longword operand, the lower 2 bits of PC are masked.  PC & * H'FFFF FFFC + 4 + PC + 4 + disp × 2 or PC & H'FFFF FFFC + 4 + disp × 4 disp (zero-extended) × 2/4 * With longword operand	Word: PC + 4 + disp × 2 → EA Longword: PC & H'FFFF FFFC + 4 + disp × 4 → EA
PC-relative	disp:8	Effective address is PC + 4 with 8-bit displacement disp added after being sign-extended and multiplied by 2.  PC + 4 + PC + 4 + disp × 2 disp (sign-extended) × 2	PC + 4 + disp × 2 → Branch-Target

Addressing Mode	Instruction Format	Effective Address Calculation Method	Calculation Formula
PC-relative	disp:12	Effective address is PC + 4 with 12-bit displacement disp added after being sign-extended and multiplied by 2.  PC + 4 + PC + 4 + disp × 2 disp (sign-extended) × 2	PC + 4 + disp × 2 → Branch-Target
	Rn	Effective address is sum of PC + 4 and Rn. 	PC + 4 + Rn → Branch-Target
Immediate	#imm:8	8-bit immediate data imm of TST, AND, OR, or XOR instruction is zero-extended.	—
	#imm:8	8-bit immediate data imm of MOV, ADD, or CMP/EQ instruction is sign-extended.	—
	#imm:8	8-bit immediate data imm of TRAPA instruction is zero-extended and multiplied by 4.	—

Note: For the addressing modes below that use a displacement (disp), the assembler descriptions in this manual show the value before scaling (×1, ×2, or ×4) is performed according to the operand size. This is done to clarify the operation of the LSI. Refer to the relevant assembler notation rules for the actual 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

3.3 Instruction Set

Table 3.3 shows the notation used in the SH instruction lists shown in tables 3.4 to 3.13.

Table 3.3 Notation Used in Instruction List

Item	Format	Description	
Instruction mnemonic	OP.Sz SRC, DEST	OP:	Operation code
		Sz:	Size
		SRC:	Source operand
		DEST:	Source and/or destination operand
		Rm:	Source register
		Rn:	Destination register
		imm:	Immediate data
		disp:	Displacement
Operation notation		→, ←	Transfer direction
		(xx)	Memory operand
		M/Q/T	SR flag bits
		&	Logical AND of individual bits
		|	Logical OR of individual bits
		^	Logical exclusive-OR of individual bits
		~	Logical NOT of individual bits
		<<n, >>n	n-bit shift
Instruction code	MSB ↔ LSB	mmmm:	Register number (Rm, FRm)
		nnnn:	Register number (Rn, FRn)
		0000:	R0, FR0
		0001:	R1, FR1
		:	
		1111:	R15, FR15
		mmm:	Register number (DRm, XDm, Rm_BANK)
		nnn:	Register number (DRn, XDn, Rn_BANK)
		000:	DR0, XD0, R0_BANK
		001:	DR2, XD2, R1_BANK
		:	
		111:	DR14, XD14, R7_BANK
		mm:	Register number (FVm)
		nn:	Register number (FVn)
		00:	FV0
		01:	FV4
		10:	FV8
		11:	FV12
		iiii:	Immediate data
		dddd:	Displacement
Privileged mode		"Privileged" means the instruction can only be executed in privileged mode.	

Item	Format	Description
T bit	Value of T bit after instruction execution	—: No change
New	—	"New" means the instruction which is newly added in this LSI.

Note: Scaling (×1, ×2, ×4, or ×8) is executed according to the size of the instruction operand.

Table 3.4 Fixed-Point Transfer Instructions

Instruction		Operation	Instruction Code	Privileged	T Bit	New
MOV	#imm,Rn	imm → sign extension → Rn	1110nnnniiiiiiii	—	—	—
MOV.W	@(disp*,PC), Rn	(disp × 2 + PC + 4) → sign extension → Rn	1001nnnndddddddd	—	—	—
MOV.L	@(disp*,PC), Rn	(disp × 4 + PC & H'FFFF FFFC + 4) → Rn	1101nnnndddddddd	—	—	—
MOV	Rm,Rn	Rm → Rn	0110nnnnmmmm0011	—	—	—
MOV.B	Rm,@Rn	Rm → (Rn)	0010nnnnmmmm0000	—	—	—
MOV.W	Rm,@Rn	Rm → (Rn)	0010nnnnmmmm0001	—	—	—
MOV.L	Rm,@Rn	Rm → (Rn)	0010nnnnmmmm0010	—	—	—
MOV.B	@Rm,Rn	(Rm) → sign extension → Rn	0110nnnnmmmm0000	—	—	—
MOV.W	@Rm,Rn	(Rm) → sign extension → Rn	0110nnnnmmmm0001	—	—	—
MOV.L	@Rm,Rn	(Rm) → Rn	0110nnnnmmmm0010	—	—	—
MOV.B	Rm,@-Rn	Rn-1 → Rn, Rm → (Rn)	0010nnnnmmmm0100	—	—	—
MOV.W	Rm,@-Rn	Rn-2 → Rn, Rm → (Rn)	0010nnnnmmmm0101	—	—	—
MOV.L	Rm,@-Rn	Rn-4 → Rn, Rm → (Rn)	0010nnnnmmmm0110	—	—	—
MOV.B	@Rm+,Rn	(Rm)→ sign extension → Rn, Rm + 1 → Rm	0110nnnnmmmm0100	—	—	—
MOV.W	@Rm+,Rn	(Rm) → sign extension → Rn, Rm + 2 → Rm	0110nnnnmmmm0101	—	—	—
MOV.L	@Rm+,Rn	(Rm) → Rn, Rm + 4 → Rm	0110nnnnmmmm0110	—	—	—
MOV.B	R0,@(disp*,Rn)	R0 → (disp + Rn)	10000000nnnndddd	—	—	—
MOV.W	R0,@(disp*,Rn)	R0 → (disp × 2 + Rn)	10000001nnnndddd	—	—	—
MOV.L	Rm,@(disp*,Rn)	Rm → (disp × 4 + Rn)	0001nnnnmmmmdddd	—	—	—
MOV.B	@(disp*,Rm),R0	(disp + Rm) → sign extension → R0	10000100mmmmdddd	—	—	—
MOV.W	@(disp*,Rm),R0	(disp × 2 + Rm) → sign extension → R0	10000101mmmmdddd	—	—	—
MOV.L	@(disp*,Rm),Rn	(disp × 4 + Rm) → Rn	0101nnnnmmmmdddd	—	—	—
MOV.B	Rm,@(R0,Rn)	Rm → (R0 + Rn)	0000nnnnmmmm0100	—	—	—
MOV.W	Rm,@(R0,Rn)	Rm → (R0 + Rn)	0000nnnnmmmm0101	—	—	—
MOV.L	Rm,@(R0,Rn)	Rm → (R0 + Rn)	0000nnnnmmmm0110	—	—	—
MOV.B	@(R0,Rm),Rn	(R0 + Rm) → sign extension → Rn	0000nnnnmmmm1100	—	—	—
MOV.W	@(R0,Rm),Rn	(R0 + Rm) → sign extension → Rn	0000nnnnmmmm1101	—	—	—
MOV.L	@(R0,Rm),Rn	(R0 + Rm) → Rn	0000nnnnmmmm1110	—	—	—

Instruction		Operation	Instruction Code	Privileged	T Bit	New
MOV.B	R0,@(disp*,GBR)	R0 → (disp + GBR)	11000000dddddddd	—	—	—
MOV.W	R0,@(disp*,GBR)	R0 → (disp × 2 + GBR)	11000001dddddddd	—	—	—
MOV.L	R0,@(disp*,GBR)	R0 → (disp × 4 + GBR)	11000010dddddddd	—	—	—
MOV.B	@(disp*,GBR),R0	(disp + GBR) → sign extension → R0	11000100dddddddd	—	—	—
MOV.W	@(disp*,GBR),R0	(disp × 2 + GBR) → sign extension → R0	11000101dddddddd	—	—	—
MOV.L	@(disp*,GBR),R0	(disp × 4 + GBR) → R0	11000110dddddddd	—	—	—
MOVA	@(disp*,PC),R0	disp × 4 + PC & H'FFFF FFFC + 4 → R0	11000111dddddddd	—	—	—
MOVCO.L	R0,@Rn	LDST → T If (T == 1) R0 → (Rn) 0 → LDST	0000nnnn01110011	—	LDST	New
MOVLI.L	@Rm,R0	1 → LDST (Rm) → R0 When interrupt/exception occurred 0 → LDST	0000mmmm01100011	—	—	New
MOVUA.L	@Rm,R0	(Rm) → R0 Load non-boundary alignment data	0100mmmm10101001	—	—	New
MOVUA.L	@Rm+,R0	(Rm) → R0, Rm + 4 → Rm Load non-boundary alignment data	0100mmmm11101001	—	—	New
MOVT	Rn	T → Rn	0000nnnn00101001	—	—	—
SWAP.B	Rm,Rn	Rm → swap lower 2 bytes → Rn	0110nnnnmmmm1000	—	—	—
SWAP.W	Rm,Rn	Rm → swap upper/lower words → Rn	0110nnnnmmmm1001	—	—	—
XTRCT	Rm,Rn	Rm:Rn middle 32 bits → Rn	0010nnnnmmmm1101	—	—	—

Note: * The assembler of Renesas uses the value after scaling (×1, ×2, or ×4) as the displacement (disp).

Table 3.5 Arithmetic Operation Instructions

Instruction		Operation	Instruction Code	Privileged	T Bit	New
ADD	Rm,Rn	Rn + Rm → Rn	`0011nnnnmmmm1100`	—	—	—
ADD	#imm,Rn	Rn + imm → Rn	`0111nnnniiiiiiii`	—	—	—
ADDC	Rm,Rn	Rn + Rm + T → Rn, carry → T	`0011nnnnmmmm1110`	—	Carry	—
ADDV	Rm,Rn	Rn + Rm → Rn, overflow → T	`0011nnnnmmmm1111`	—	Overflow	—
CMP/EQ	#imm,R0	When R0 = imm, 1 → T Otherwise, 0 → T	`10001000iiiiiiii`	—	Comparison result	—
CMP/EQ	Rm,Rn	When Rn = Rm, 1 → T Otherwise, 0 → T	`0011nnnnmmmm0000`	—	Comparison result	—
CMP/HS	Rm,Rn	When Rn ≥ Rm (unsigned), 1 → T Otherwise, 0 → T	`0011nnnnmmmm0010`	—	Comparison result	—
CMP/GE	Rm,Rn	When Rn ≥ Rm (signed), 1 → T Otherwise, 0 → T	`0011nnnnmmmm0011`	—	Comparison result	—
CMP/HI	Rm,Rn	When Rn > Rm (unsigned), 1 → T Otherwise, 0 → T	`0011nnnnmmmm0110`	—	Comparison result	—
CMP/GT	Rm,Rn	When Rn > Rm (signed), 1 → T Otherwise, 0 → T	`0011nnnnmmmm0111`	—	Comparison result	—
CMP/PZ	Rn	When Rn ≥ 0, 1 → T Otherwise, 0 → T	`0100nnnn00010001`	—	Comparison result	—
CMP/PL	Rn	When Rn > 0, 1 → T Otherwise, 0 → T	`0100nnnn00010101`	—	Comparison result	—
CMP/STR	Rm,Rn	When any bytes are equal, 1 → T Otherwise, 0 → T	`0010nnnnmmmm1100`	—	Comparison result	—
DIV1	Rm,Rn	1-step division (Rn ÷ Rm)	`0011nnnnmmmm0100`	—	Calculation result	—
DIV0S	Rm,Rn	MSB of Rn → Q, MSB of Rm → M, M^Q → T	`0010nnnnmmmm0111`	—	Calculation result	—
DIV0U		0 → M/Q/T	`0000000000011001`	—	0	—
DMULS.L	Rm,Rn	Signed, Rn × Rm → MAC, 32 × 32 → 64 bits	`0011nnnnmmmm1101`	—	—	—
DMULU.L	Rm,Rn	Unsigned, Rn × Rm → MAC, 32 × 32 → 64 bits	`0011nnnnmmmm0101`	—	—	—

Instruction		Operation	Instruction Code	Privileged	T Bit	New
DT	Rn	Rn – 1 → Rn; when Rn = 0, 1 → T When Rn ≠ 0, 0 → T	0100nnnn00010000	—	Comparison result	—
EXTS.B	Rm,Rn	Rm sign-extended from byte → Rn	0110nnnnmmmm1110	—	—	—
EXTS.W	Rm,Rn	Rm sign-extended from word → Rn	0110nnnnmmmm1111	—	—	—
EXTU.B	Rm,Rn	Rm zero-extended from byte → Rn	0110nnnnmmmm1100	—	—	—
EXTU.W	Rm,Rn	Rm zero-extended from word → Rn	0110nnnnmmmm1101	—	—	—
MAC.L	@Rm+,@Rn+	Signed, (Rn) × (Rm) + MAC → MAC Rn + 4 → Rn, Rm + 4 → Rm 32 × 32 + 64 → 64 bits	0000nnnnmmmm1111	—	—	—
MAC.W	@Rm+,@Rn+	Signed, (Rn) × (Rm) + MAC → MAC Rn + 2 → Rn, Rm + 2 → Rm 16 × 16 + 64 → 64 bits	0100nnnnmmmm1111	—	—	—
MUL.L	Rm,Rn	Rn × Rm → MACL 32 × 32 → 32 bits	0000nnnnmmmm0111	—	—	—
MULS.W	Rm,Rn	Signed, Rn × Rm → MACL 16 × 16 → 32 bits	0010nnnnmmmm1111	—	—	—
MULU.W	Rm,Rn	Unsigned, Rn × Rm → MACL 16 × 16 → 32 bits	0010nnnnmmmm1110	—	—	—
NEG	Rm,Rn	0 – Rm → Rn	0110nnnnmmmm1011	—	—	—
NEGC	Rm,Rn	0 – Rm – T → Rn, borrow → T	0110nnnnmmmm1010	—	Borrow	—
SUB	Rm,Rn	Rn – Rm → Rn	0011nnnnmmmm1000	—	—	—
SUBC	Rm,Rn	Rn – Rm – T → Rn, borrow → T	0011nnnnmmmm1010	—	Borrow	—
SUBV	Rm,Rn	Rn – Rm → Rn, underflow → T	0011nnnnmmmm1011	—	Underflow	—

Table 3.6 Logic Operation Instructions

Instruction		Operation	Instruction Code	Privileged	T Bit	New
AND	Rm,Rn	Rn & Rm → Rn	0010nnnnmmmm1001	—	—	—
AND	#imm,R0	R0 & imm → R0	11001001iiiiiiii	—	—	—
AND.B	#imm, @(R0,GBR)	(R0 + GBR) & imm → (R0 + GBR)	11001101iiiiiiii	—	—	—
NOT	Rm,Rn	~Rm → Rn	0110nnnnmmmm0111	—	—	—
OR	Rm,Rn	Rn | Rm → Rn	0010nnnnmmmm1011	—	—	—
OR	#imm,R0	R0 | imm → R0	11001011iiiiiiii	—	—	—
OR.B	#imm, @(R0,GBR)	(R0 + GBR) | imm → (R0 + GBR)	11001111iiiiiiii	—	—	—
TAS.B	@Rn	When (Rn) = 0, 1 → T Otherwise, 0 → T In both cases, 1 → MSB of (Rn)	0100nnnn00011011	—	Test result	—
TST	Rm,Rn	Rn & Rm; when result = 0, 1 → T Otherwise, 0 → T	0010nnnnmmmm1000	—	Test result	—
TST	#imm,R0	R0 & imm; when result = 0, 1 → T Otherwise, 0 → T	11001000iiiiiiii	—	Test result	—
TST.B	#imm, @(R0,GBR)	(R0 + GBR) & imm; when result = 0, 1 → T Otherwise, 0 → T	11001100iiiiiiii	—	Test result	—
XOR	Rm,Rn	Rn ∧ Rm → Rn	0010nnnnmmmm1010	—	—	—
XOR	#imm,R0	R0 ∧ imm → R0	11001010iiiiiiii	—	—	—
XOR.B	#imm, @(R0,GBR)	(R0 + GBR) ∧ imm → (R0 + GBR)	11001110iiiiiiii	—	—	—

Table 3.7 Shift Instructions

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

Table 3.8 Branch Instructions

Instruction		Operation	Instruction Code	Privileged	T Bit	New
BF	label	When T = 0, disp × 2 + PC + 4 → PC When T = 1, nop	10001011dddddddd	—	—	—
BF/S	label	Delayed branch; when T = 0, disp × 2 + PC + 4 → PC When T = 1, nop	10001111dddddddd	—	—	—
BT	label	When T = 1, disp × 2 + PC + 4 → PC When T = 0, nop	10001001dddddddd	—	—	—
BT/S	label	Delayed branch; when T = 1, disp × 2 + PC + 4 → PC When T = 0, nop	10001101dddddddd	—	—	—
BRA	label	Delayed branch, disp × 2 + PC + 4 → PC	1010dddddddddddd	—	—	—
BRAF	Rn	Delayed branch, Rn + PC + 4 → PC	0000nnnn00100011	—	—	—
BSR	label	Delayed branch, PC + 4 → PR, disp × 2 + PC + 4 → PC	1011dddddddddddd	—	—	—
BSRF	Rn	Delayed branch, PC + 4 → PR, Rn + PC + 4 → PC	0000nnnn00000011	—	—	—
JMP	@Rn	Delayed branch, Rn → PC	0100nnnn00101011	—	—	—
JSR	@Rn	Delayed branch, PC + 4 → PR, Rn → PC	0100nnnn00001011	—	—	—
RTS		Delayed branch, PR → PC	0000000000001011	—	—	—

Table 3.9 System Control Instructions

Instruction		Operation	Instruction Code	Privileged	T Bit	New
CLRMAC		0 → MACH, MACL	0000000000101000	—	—	—
CLRS		0 → S	0000000001001000	—	—	—
CLRT		0 → T	0000000000001000	—	0	—
ICBI	@Rn	Invalidates instruction cache block indicated by logical address	0000nnnn11100011	—	—	New
LDC	Rm,SR	Rm → SR	0100mmmm00001110	Privileged	LSB	—
LDC	Rm,GBR	Rm → GBR	0100mmmm00011110	—	—	—
LDC	Rm,VBR	Rm → VBR	0100mmmm00101110	Privileged	—	—
LDC	Rm,SGR	Rm → SGR	0100mmmm00111010	Privileged	—	—
LDC	Rm,SSR	Rm → SSR	0100mmmm00111110	Privileged	—	—

Instruction		Operation	Instruction Code	Privileged	T Bit	New
LDC	Rm,SPC	Rm → SPC	0100mmmm01001110	Privileged	—	—
LDC	Rm,DBR	Rm → DBR	0100mmmm11111010	Privileged	—	—
LDC	Rm,Rn_BANK	Rm → Rn_BANK (n = 0 to 7)	0100mmmm1nnn1110	Privileged	—	—
LDC.L	@Rm+,SR	(Rm) → SR, Rm + 4 → Rm	0100mmmm00000111	Privileged	LSB	—
LDC.L	@Rm+,GBR	(Rm) → GBR, Rm + 4 → Rm	0100mmmm00010111	—	—	—
LDC.L	@Rm+,VBR	(Rm) → VBR, Rm + 4 → Rm	0100mmmm00100111	Privileged	—	—
LDC.L	@Rm+,SGR	(Rm) → SGR, Rm + 4 → Rm	0100mmmm00110110	Privileged	—	—
LDC.L	@Rm+,SSR	(Rm) → SSR, Rm + 4 → Rm	0100mmmm00110111	Privileged	—	—
LDC.L	@Rm+,SPC	(Rm) → SPC, Rm + 4 → Rm	0100mmmm01000111	Privileged	—	—
LDC.L	@Rm+,DBR	(Rm) → DBR, Rm + 4 → Rm	0100mmmm11110110	Privileged	—	—
LDC.L	@Rm+,Rn_BANK	(Rm) → Rn_BANK, Rm + 4 → Rm	0100mmmm1nnn0111	Privileged	—	—
LDS	Rm,MACH	Rm → MACH	0100mmmm00001010	—	—	—
LDS	Rm,MACL	Rm → MACL	0100mmmm00011010	—	—	—
LDS	Rm,PR	Rm → PR	0100mmmm00101010	—	—	—
LDS.L	@Rm+,MACH	(Rm) → MACH, Rm + 4 → Rm	0100mmmm00000110	—	—	—
LDS.L	@Rm+,MACL	(Rm) → MACL, Rm + 4 → Rm	0100mmmm00010110	—	—	—
LDS.L	@Rm+,PR	(Rm) → PR, Rm + 4 → Rm	0100mmmm00100110	—	—	—
LDTLB		PTEH/PTEL → TLB	0000000000111000	Privileged	—	—
MOVCA.L	R0,@Rn	R0 → (Rn) (without fetching cache block)	0000nnnn11000011	—	—	—
NOP		No operation	0000000000001001	—	—	—
OCBI	@Rn	Invalidates operand cache block	0000nnnn10010011	—	—	—
OCBP	@Rn	Writes back and invalidates operand cache block	0000nnnn10100011	—	—	—
OCBWB	@Rn	Writes back operand cache block	0000nnnn10110011	—	—	—
PREF	@Rn	(Rn) → operand cache	0000nnnn10000011	—	—	—
PREFI	@Rn	Reads 32-byte instruction block into instruction cache	0000nnnn11010011	—	—	New
RTE		Delayed branch, SSR/SPC → SR/PC	0000000000101011	Privileged	—	—
SETS		1 → S	0000000001011000	—	—	—
SETT		1 → T	0000000000011000	—	1	—
SLEEP		Sleep or standby	0000000000011011	Privileged	—	—
STC	SR,Rn	SR → Rn	0000nnnn00000010	Privileged	—	—
STC	GBR,Rn	GBR → Rn	0000nnnn00010010	—	—	—

Instruction		Operation	Instruction Code	Privileged	T Bit	New
STC	VBR,Rn	VBR → Rn	0000nnnn00100010	Privileged	—	—
STC	SSR,Rn	SSR → Rn	0000nnnn00110010	Privileged	—	—
STC	SPC,Rn	SPC → Rn	0000nnnn01000010	Privileged	—	—
STC	SGR,Rn	SGR → Rn	0000nnnn00111010	Privileged	—	—
STC	DBR,Rn	DBR → Rn	0000nnnn11111010	Privileged	—	—
STC	Rm_BANK,Rn	Rm_BANK → Rn (m = 0 to 7)	0000nnnn1mmm0010	Privileged	—	—
STC.L	SR,@-Rn	Rn – 4 → Rn, SR → (Rn)	0100nnnn00000011	Privileged	—	—
STC.L	GBR,@-Rn	Rn – 4 → Rn, GBR → (Rn)	0100nnnn00010011	—	—	—
STC.L	VBR,@-Rn	Rn – 4 → Rn, VBR → (Rn)	0100nnnn00100011	Privileged	—	—
STC.L	SSR,@-Rn	Rn – 4 → Rn, SSR → (Rn)	0100nnnn00110011	Privileged	—	—
STC.L	SPC,@-Rn	Rn – 4 → Rn, SPC → (Rn)	0100nnnn01000011	Privileged	—	—
STC.L	SGR,@-Rn	Rn – 4 → Rn, SGR → (Rn)	0100nnnn00110010	Privileged	—	—
STC.L	DBR,@-Rn	Rn – 4 → Rn, DBR → (Rn)	0100nnnn11110010	Privileged	—	—
STC.L	Rm_BANK,@-Rn	Rn – 4 → Rn, Rm_BANK → (Rn) (m = 0 to 7)	0100nnnn1mmm0011	Privileged	—	—
STS	MACH,Rn	MACH → Rn	0000nnnn00001010	—	—	—
STS	MACL,Rn	MACL → Rn	0000nnnn00011010	—	—	—
STS	PR,Rn	PR → Rn	0000nnnn00101010	—	—	—
STS.L	MACH,@-Rn	Rn – 4 → Rn, MACH → (Rn)	0100nnnn00000010	—	—	—
STS.L	MACL,@-Rn	Rn – 4 → Rn, MACL → (Rn)	0100nnnn00010010	—	—	—
STS.L	PR,@-Rn	Rn – 4 → Rn, PR → (Rn)	0100nnnn00100010	—	—	—
SYNCO		Prevents the next instruction from being issued until instructions issued before this instruction have been completed.	0000000010101011	—	—	New
TRAPA	#imm	PC + 2 → SPC, SR → SSR, #imm << 2 → TRA, H'160 → EXPEVT, VBR + H'0100 → PC	11000011iiiiiiii	—	—	—

Table 3.10 Floating-Point Single-Precision Instructions

Instruction		Operation	Instruction Code	Privileged	T Bit	New
FLDI0	FRn	H'0000 0000 → FRn	1111nnnn10001101	—	—	—
FLDI1	FRn	H'3F80 0000 → FRn	1111nnnn10011101	—	—	—
FMOV	FRm,FRn	FRm → FRn	1111nnnnmmmm1100	—	—	—
FMOV.S	@Rm,FRn	(Rm) → FRn	1111nnnnmmmm1000	—	—	—
FMOV.S	@(R0,Rm),FRn	(R0 + Rm) → FRn	1111nnnnmmmm0110	—	—	—
FMOV.S	@Rm+,FRn	(Rm) → FRn, Rm + 4 → Rm	1111nnnnmmmm1001	—	—	—
FMOV.S	FRm,@Rn	FRm → (Rn)	1111nnnnmmmm1010	—	—	—
FMOV.S	FRm,@-Rn	Rn-4 → Rn, FRm → (Rn)	1111nnnnmmmm1011	—	—	—
FMOV.S	FRm,@(R0,Rn)	FRm → (R0 + Rn)	1111nnnnmmmm0111	—	—	—
FMOV	DRm,DRn	DRm → DRn	1111nnn0mmm01100	—	—	—
FMOV	@Rm,DRn	(Rm) → DRn	1111nnn0mmmm1000	—	—	—
FMOV	@(R0,Rm),DRn	(R0 + Rm) → DRn	1111nnn0mmmm0110	—	—	—
FMOV	@Rm+,DRn	(Rm) → DRn, Rm + 8 → Rm	1111nnn0mmmm1001	—	—	—
FMOV	DRm,@Rn	DRm → (Rn)	1111nnnnmmm01010	—	—	—
FMOV	DRm,@-Rn	Rn-8 → Rn, DRm → (Rn)	1111nnnnmmm01011	—	—	—
FMOV	DRm,@(R0,Rn)	DRm → (R0 + Rn)	1111nnnnmmm00111	—	—	—
FLDS	FRm,FPUL	FRm → FPUL	1111mmmm00011101	—	—	—
FSTS	FPUL,FRn	FPUL → FRn	1111nnnn00001101	—	—	—
FABS	FRn	FRn & H'7FFF FFFF → FRn	1111nnnn01011101	—	—	—
FADD	FRm,FRn	FRn + FRm → FRn	1111nnnnmmmm0000	—	—	—
FCMP/EQ	FRm,FRn	When FRn = FRm, 1 → T Otherwise, 0 → T	1111nnnnmmmm0100	—	Comparis on result	—
FCMP/GT	FRm,FRn	When FRn > FRm, 1 → T Otherwise, 0 → T	1111nnnnmmmm0101	—	Comparis on result	—
FDIV	FRm,FRn	FRn/FRm → FRn	1111nnnnmmmm0011	—	—	—
FLOAT	FPUL,FRn	(float) FPUL → FRn	1111nnnn00101101	—	—	—
FMAC	FR0,FRm,FRn	FR0*FRm + FRn → FRn	1111nnnnmmmm1110	—	—	—
FMUL	FRm,FRn	FRn*FRm → FRn	1111nnnnmmmm0010	—	—	—
FNEG	FRn	FRn ∧ H'8000 0000 → FRn	1111nnnn01001101	—	—	—
FSQRT	FRn	√FRn → FRn	1111nnnn01101101	—	—	—
FSUB	FRm,FRn	FRn – FRm → FRn	1111nnnnmmmm0001	—	—	—
FTRC	FRm,FPUL	(long) FRm → FPUL	1111mmmm00111101	—	—	—

Table 3.11 Floating-Point Double-Precision Instructions

Instruction		Operation	Instruction Code	Privileged	T Bit	New
FABS	DRn	DRn & H'7FFF FFFF FFFF FFFF → DRn	1111nnn001011101	—	—	—
FADD	DRm,DRn	DRn + DRm → DRn	1111nnn0mmm00000	—	—	—
FCMP/EQ	DRm,DRn	When DRn = DRm, 1 → T Otherwise, 0 → T	1111nnn0mmm00100	—	Comparison result	—
FCMP/GT	DRm,DRn	When DRn > DRm, 1 → T Otherwise, 0 → T	1111nnn0mmm00101	—	Comparison result	—
FDIV	DRm,DRn	DRn /DRm → DRn	1111nnn0mmm00011	—	—	—
FCNVDS	DRm,FPUL	double_to_ float(DRm) → FPUL	1111mmm010111101	—	—	—
FCNVSD	FPUL,DRn	float_to_ double (FPUL) → DRn	1111nnn010101101	—	—	—
FLOAT	FPUL,DRn	(float)FPUL → DRn	1111nnn000101101	—	—	—
FMUL	DRm,DRn	DRn *DRm → DRn	1111nnn0mmm00010	—	—	—
FNEG	DRn	DRn ^ H'8000 0000 0000 0000 → DRn	1111nnn001001101	—	—	—
FSQRT	DRn	√DRn → DRn	1111nnn001101101	—	—	—
FSUB	DRm,DRn	DRn – DRm → DRn	1111nnn0mmm00001	—	—	—
FTRC	DRm,FPUL	(long) DRm → FPUL	1111mmm000111101	—	—	—

Table 3.12 Floating-Point Control Instructions

Instruction		Operation	Instruction Code	Privileged	T Bit	New
LDS	Rm,FPSCR	Rm → FPSCR	0100mmmm01101010	—	—	—
LDS	Rm,FPUL	Rm → FPUL	0100mmmm01011010	—	—	—
LDS.L	@Rm+,FPSCR	(Rm) → FPSCR, Rm+4 → Rm	0100mmmm01100110	—	—	—
LDS.L	@Rm+,FPUL	(Rm) → FPUL, Rm+4 → Rm	0100mmmm01010110	—	—	—
STS	FPSCR,Rn	FPSCR → Rn	0000nnnn01101010	—	—	—
STS	FPUL,Rn	FPUL → Rn	0000nnnn01011010	—	—	—
STS.L	FPSCR,@-Rn	Rn – 4 → Rn, FPSCR → (Rn)	0100nnnn01100010	—	—	—
STS.L	FPUL,@-Rn	Rn – 4 → Rn, FPUL → (Rn)	0100nnnn01010010	—	—	—

Table 3.13 Floating-Point Graphics Acceleration Instructions

Instruction		Operation	Instruction Code	Privileged	T Bit	New
FMOV	DRm,XDn	DRm → XDn	1111nnn1mmm01100	—	—	—
FMOV	XDm,DRn	XDm → DRn	1111nnn0mmm11100	—	—	—
FMOV	XDm,XDn	XDm → XDn	1111nnn1mmm11100	—	—	—
FMOV	@Rm,XDn	(Rm) → XDn	1111nnn1mmmm1000	—	—	—
FMOV	@Rm+,XDn	(Rm) → XDn, Rm + 8 → Rm	1111nnn1mmmm1001	—	—	—
FMOV	@(R0,Rm),XDn	(R0 + Rm) → XDn	1111nnn1mmmm0110	—	—	—
FMOV	XDm,@Rn	XDm → (Rn)	1111nnnnmmm11010	—	—	—
FMOV	XDm,@-Rn	Rn – 8 → Rn, XDm → (Rn)	1111nnnnmmm11011	—	—	—
FMOV	XDm,@(R0,Rn)	XDm → (R0 + Rn)	1111nnnnmmm10111	—	—	—
FIPR	FVm,FVn	inner_product (FVm, FVn) → FR[n+3]	1111nnmm11101101	—	—	—
FTRV	XMTRX,FVn	transform_vector (XMTRX, FVn) → FVn	1111nn0111111101	—	—	—
FRCHG		~FPSCR.FR → FPSCR.FR	1111101111111101	—	—	—
FSCHG		~FPSCR.SZ → FPSCR.SZ	1111001111111101	—	—	—
FPCHG		~FPSCR.PR → FPSCR.PR	1111011111111101	—	—	New
FSRRA	FRn	1/sqrt(FRn) → FRn	1111nnnn01111101	—	—	New
FSCA	FPUL,DRn	sin(FPUL) → FRn cos(FPUL) → FR[n + 1]	1111nnn011111101	—	—	New

Note: * sqrt(FRn) is the square root of FRn.

Section 4 Pipelining

The SH-4A is a 2-ILP (instruction-level-parallelism) superscalar pipelining microprocessor. Instruction execution is pipelined, and two instructions can be executed in parallel.

4.1 Pipelines

Figure 4.1 shows the basic pipelines. Normally, a pipeline consists of seven stages: instruction fetch (I1/I2), decode and register read (ID), execution (E1/E2/E3), and write-back (WB). An instruction is executed as a combination of basic pipelines.

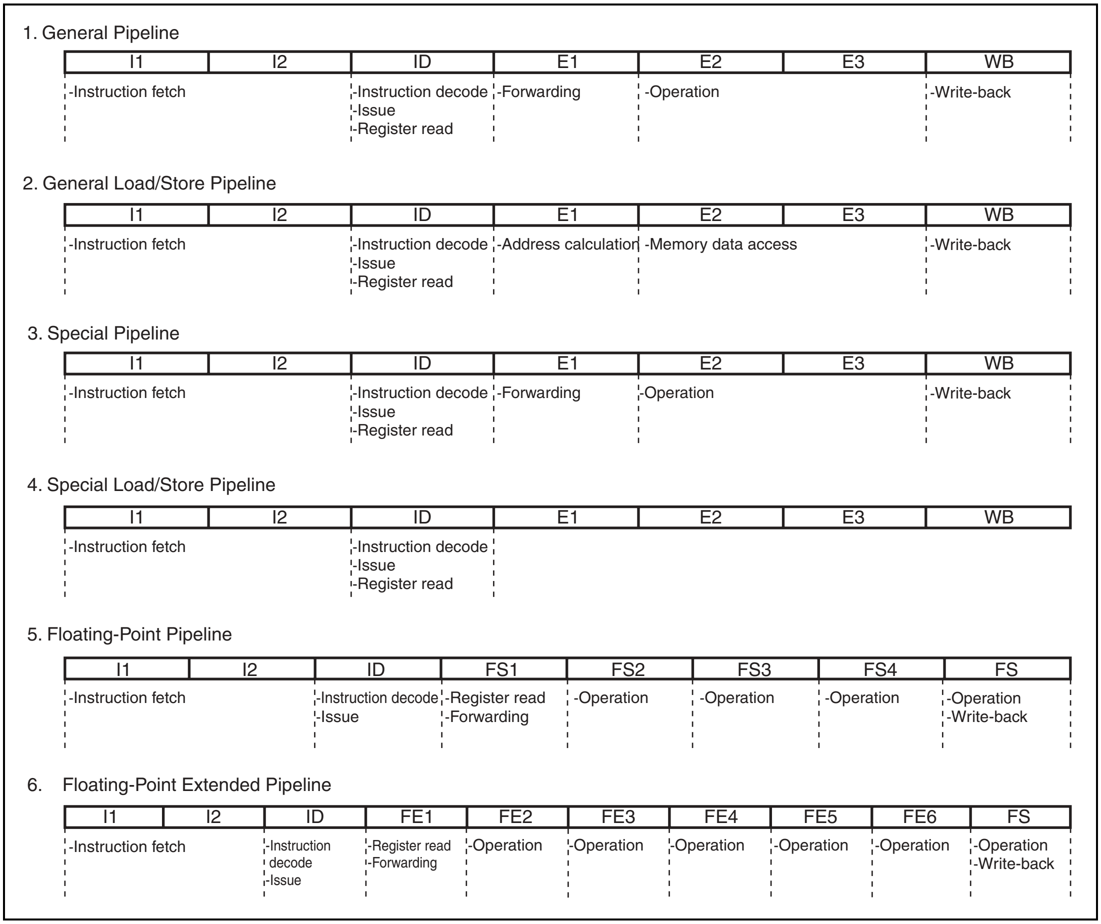


Figure 4.1 Basic Pipelines

Figure 4.2 shows the instruction execution patterns. Representations in figure 4.2 and their descriptions are listed in table 4.1.

Table 4.1 Representations of Instruction Execution Patterns

Representation	Description
$\boxed{\text{E1}}\boxed{\text{E2}}\boxed{\text{E3}}\boxed{\text{WB}}$	CPU EX pipe is occupied
$\boxed{\text{S1}}\boxed{\text{S2}}\boxed{\text{S3}}\boxed{\text{WB}}$	CPU LS pipe is occupied (with memory access)
$\boxed{\text{s1}}\boxed{\text{s2}}\boxed{\text{s3}}\boxed{\text{WB}}$	CPU LS pipe is occupied (without memory access)
$\boxed{\text{E1/S1}}$	Either CPU EX pipe or CPU LS pipe is occupied
$\boxed{\text{E1S1}}$, $\boxed{\text{E1s1}}$	Both CPU EX pipe and CPU LS pipe are occupied
$\boxed{\text{M2}}\boxed{\text{M3}}\boxed{\text{MS}}$	CPU MULT operation unit is occupied
$\boxed{\text{FE1}}\boxed{\text{FE2}}\boxed{\text{FE3}}\boxed{\text{FE4}}\boxed{\text{FE5}}\boxed{\text{FE6}}\boxed{\text{FS}}$	FPU-EX pipe is occupied
$\boxed{\text{FS1}}\boxed{\text{FS2}}\boxed{\text{FS3}}\boxed{\text{FS4}}\boxed{\text{FS}}$	FPU-LS pipe is occupied
$\boxed{\text{ID}}$	ID stage is locked
└──	Both CPU and FPU pipes are occupied

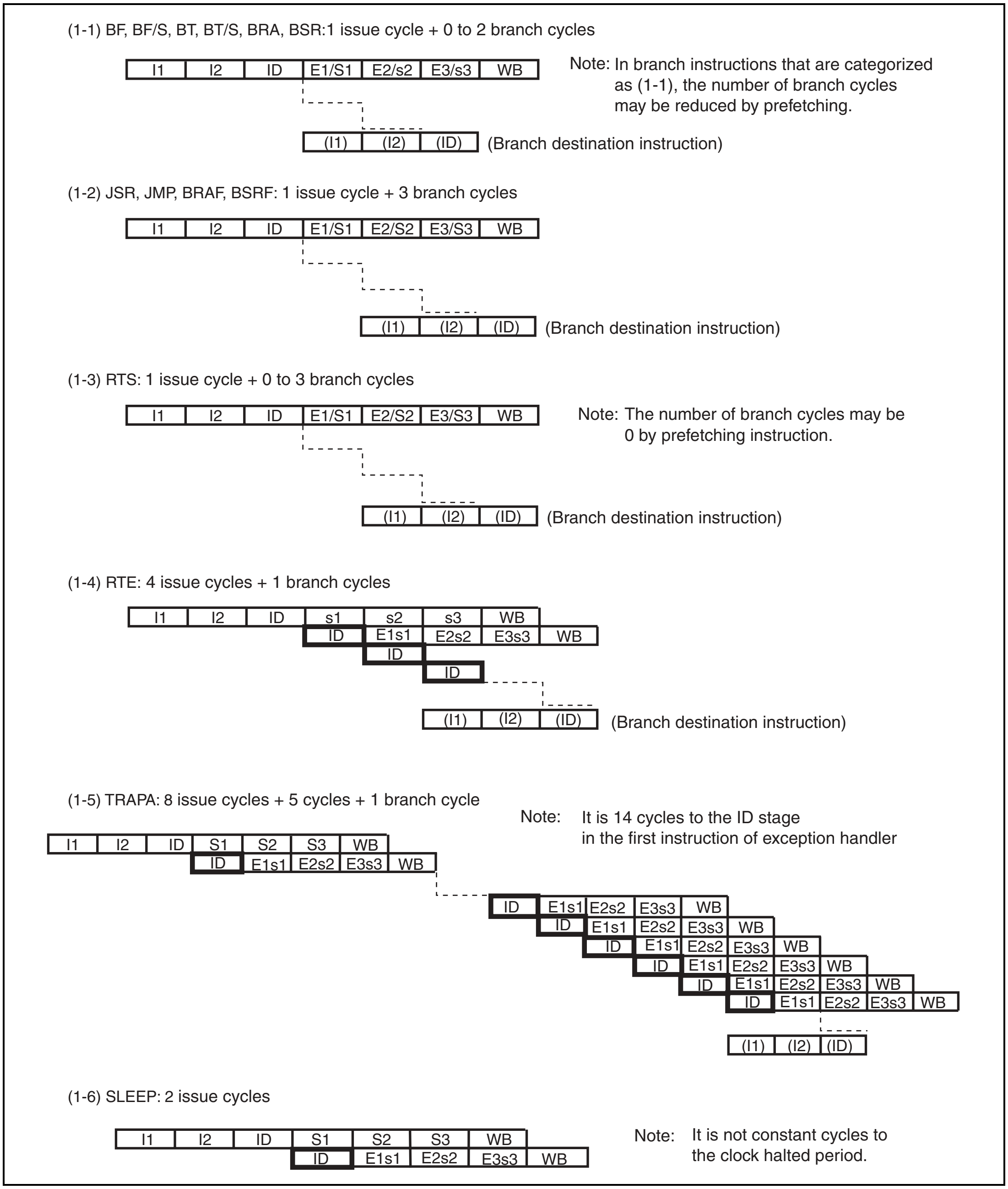

(1-1) BF, BF/S, BT, BT/S, BRA, BSR:1 issue cycle + 0 to 2 branch cycles
I1 | I2 | ID | E1/S1 | E2/s2 | E3/s3 | WB
(I1) | (I2) | (ID) (Branch destination instruction)
Note: In branch instructions that are categorized as (1-1), the number of branch cycles may be reduced by prefetching.
(1-2) JSR, JMP, BRAF, BSRF: 1 issue cycle + 3 branch cycles
I1 | I2 | ID | E1/S1 | E2/S2 | E3/S3 | WB
(I1) | (I2) | (ID) (Branch destination instruction)
(1-3) RTS: 1 issue cycle + 0 to 3 branch cycles
I1 | I2 | ID | E1/S1 | E2/S2 | E3/S3 | WB
Note: The number of branch cycles may be 0 by prefetching instruction.
(I1) | (I2) | (ID) (Branch destination instruction)
(1-4) RTE: 4 issue cycles + 1 branch cycles
I1 | I2 | ID | s1 | s2 | s3 | WB
ID | E1s1 | E2s2 | E3s3 | WB
ID
ID
(I1) | (I2) | (ID) (Branch destination instruction)
(1-5) TRAPA: 8 issue cycles + 5 cycles + 1 branch cycle
Note: It is 14 cycles to the ID stage in the first instruction of exception handler
I1 | I2 | ID | S1 | S2 | S3 | WB
ID | E1s1 | E2s2 | E3s3 | WB
ID | E1s1 | E2s2 | E3s3 | WB
ID | E1s1 | E2s2 | E3s3 | WB
ID | E1s1 | E2s2 | E3s3 | WB
ID | E1s1 | E2s2 | E3s3 | WB
ID | E1s1 | E2s2 | E3s3 | WB
ID | E1s1 | E2s2 | E3s3 | WB
(I1) | (I2) | (ID)
(1-6) SLEEP: 2 issue cycles
I1 | I2 | ID | S1 | S2 | S3 | WB
ID | E1s1 | E2s2 | E3s3 | WB
Note: It is not constant cycles to the clock halted period.

Figure 4.2 Instruction Execution Patterns (1)

(2-1) 1-step operation (EX type): 1 issue cycle

EXT[SU].[BW], MOVT, SWAP, XTRCT, ADD*, CMP*, DIV*, DT, NEG*, SUB*, AND, AND#, NOT, OR, OR#, TST, TST#, XOR, XOR#, ROT*, SHA*, SHL*, CLRS, CLRT, SETS, SETT

Note: Except for AND#, OR#, TST#, and XOR# instructions using GBR relative addressing mode

I1	I2	ID	E1	E2	E3	WB

(2-2) 1-step operation (LS type): 1 issue cycle

MOVA

I1	I2	ID	s1	s2	s3	WB

(2-3) 1-step operation (MT type): 1 issue cycle

MOV#, NOP

I1	I2	ID	E1/S1	E2/s2	E3/s3	WB

(2-4) MOV (MT type): 1 issue cycle

MOV

I1	I2	ID	E1/s1	E2/s2	E3/S3	WB

Figure 4.2 Instruction Execution Patterns (2)

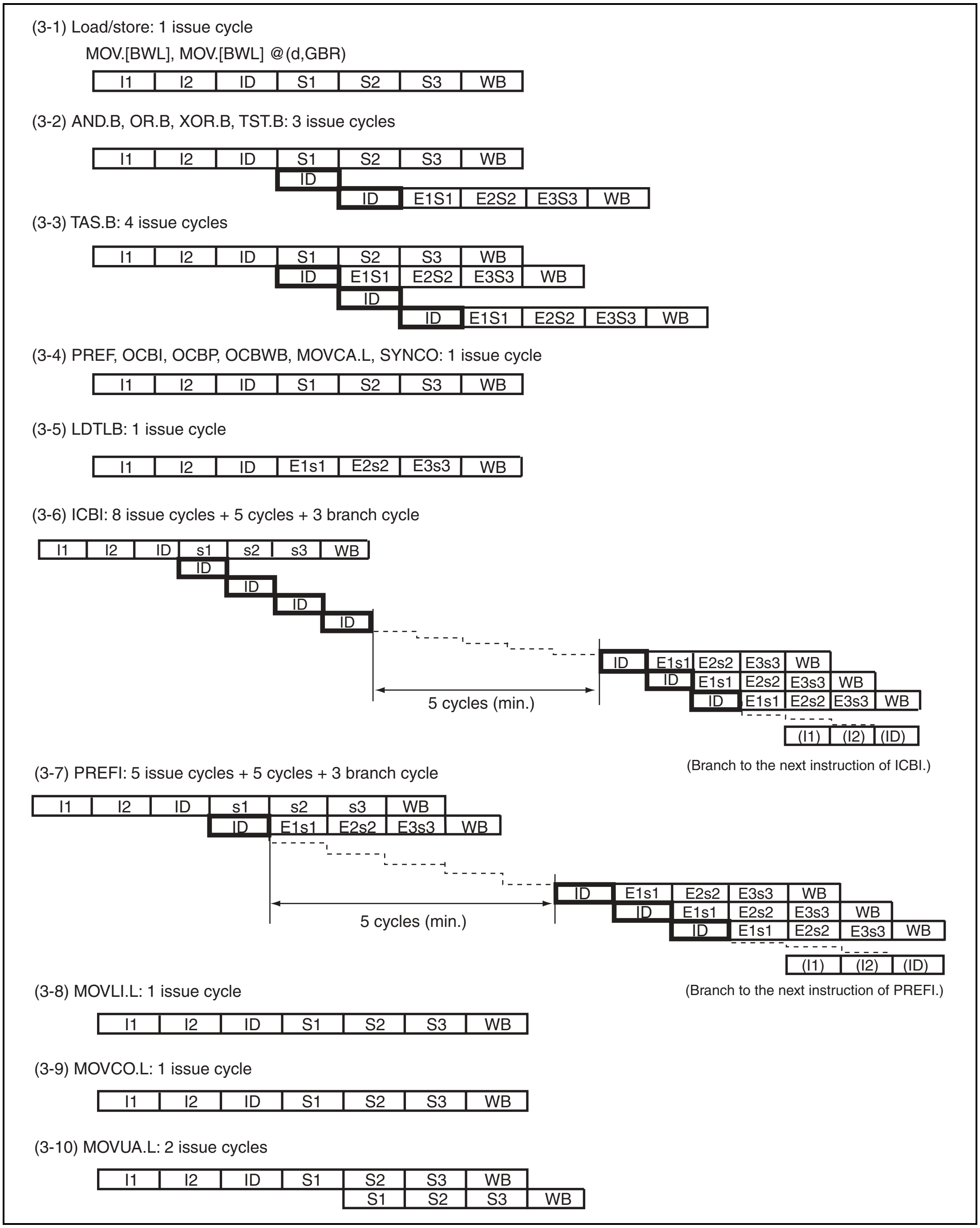

(3-1) Load/store: 1 issue cycle
MOV.[BWL], MOV.[BWL] @(d,GBR)
I1 I2 ID S1 S2 S3 WB
(3-2) AND.B, OR.B, XOR.B, TST.B: 3 issue cycles
I1 I2 ID S1 S2 S3 WB
ID
ID E1S1 E2S2 E3S3 WB
(3-3) TAS.B: 4 issue cycles
I1 I2 ID S1 S2 S3 WB
ID E1S1 E2S2 E3S3 WB
ID
ID E1S1 E2S2 E3S3 WB
(3-4) PREF, OCBI, OCBP, OCBWB, MOVCA.L, SYNCO: 1 issue cycle
I1 I2 ID S1 S2 S3 WB
(3-5) LDTLB: 1 issue cycle
I1 I2 ID E1s1 E2s2 E3s3 WB
(3-6) ICBI: 8 issue cycles + 5 cycles + 3 branch cycle
I1 I2 ID s1 s2 s3 WB
ID
ID
ID
ID
5 cycles (min.)
ID E1s1 E2s2 E3s3 WB
ID E1s1 E2s2 E3s3 WB
ID E1s1 E2s2 E3s3 WB
(I1) (I2) (ID)
(Branch to the next instruction of ICBI.)
(3-7) PREFI: 5 issue cycles + 5 cycles + 3 branch cycle
I1 I2 ID s1 s2 s3 WB
ID E1s1 E2s2 E3s3 WB
5 cycles (min.)
ID E1s1 E2s2 E3s3 WB
ID E1s1 E2s2 E3s3 WB
ID E1s1 E2s2 E3s3 WB
(I1) (I2) (ID)
(Branch to the next instruction of PREFI.)
(3-8) MOVLI.L: 1 issue cycle
I1 I2 ID S1 S2 S3 WB
(3-9) MOVCO.L: 1 issue cycle
I1 I2 ID S1 S2 S3 WB
(3-10) MOVUA.L: 2 issue cycles
I1 I2 ID S1 S2 S3 WB
S1 S2 S3 WB

Figure 4.2 Instruction Execution Patterns (3)

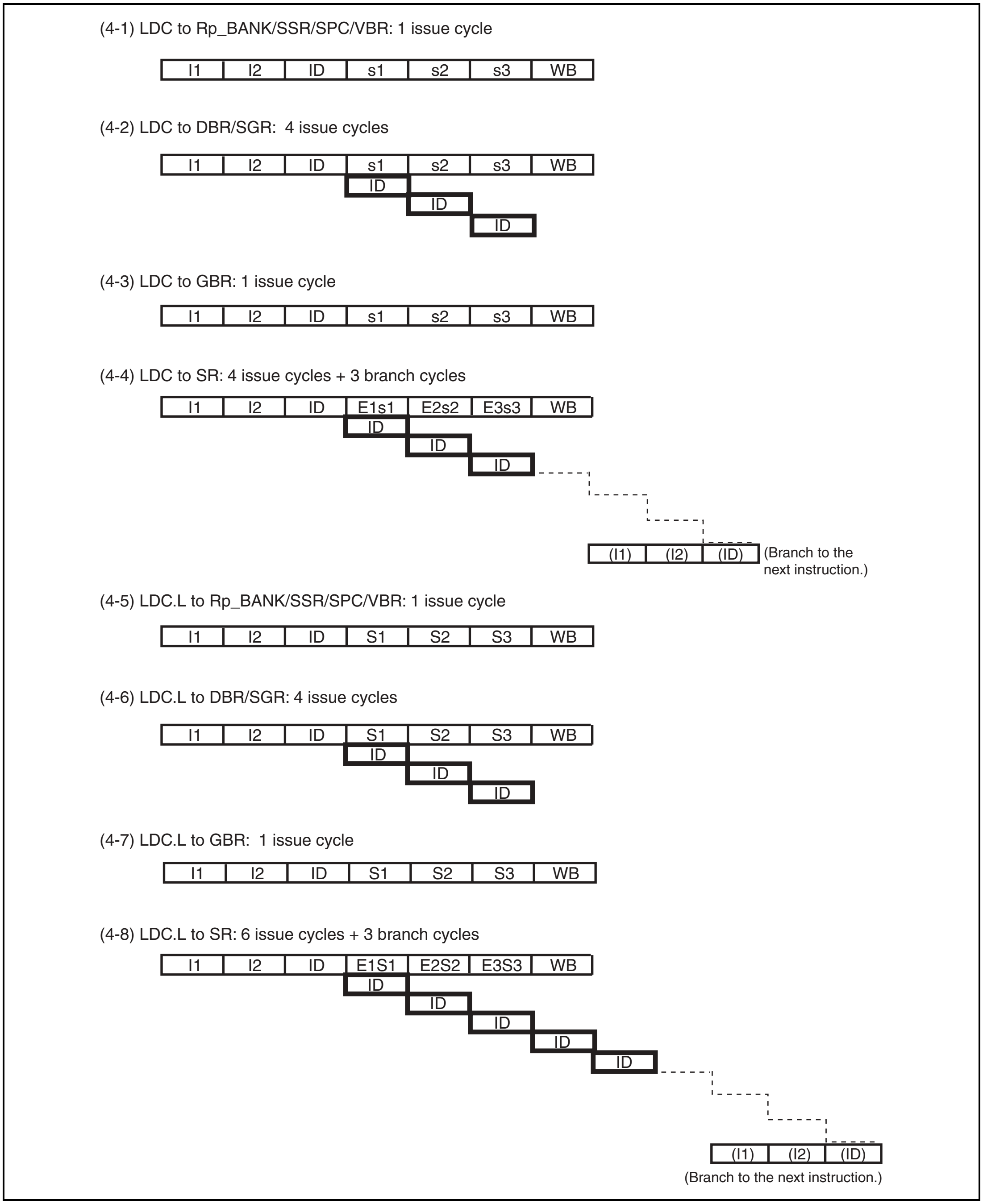

(4-1) LDC to Rp_BANK/SSR/SPC/VBR: 1 issue cycle
I1 I2 ID s1 s2 s3 WB
(4-2) LDC to DBR/SGR: 4 issue cycles
I1 I2 ID s1 s2 s3 WB
ID
ID
ID
(4-3) LDC to GBR: 1 issue cycle
I1 I2 ID s1 s2 s3 WB
(4-4) LDC to SR: 4 issue cycles + 3 branch cycles
I1 I2 ID E1s1 E2s2 E3s3 WB
ID
ID
ID
(I1) (I2) (ID)
(Branch to the next instruction.)
(4-5) LDC.L to Rp_BANK/SSR/SPC/VBR: 1 issue cycle
I1 I2 ID S1 S2 S3 WB
(4-6) LDC.L to DBR/SGR: 4 issue cycles
I1 I2 ID S1 S2 S3 WB
ID
ID
ID
(4-7) LDC.L to GBR: 1 issue cycle
I1 I2 ID S1 S2 S3 WB
(4-8) LDC.L to SR: 6 issue cycles + 3 branch cycles
I1 I2 ID E1S1 E2S2 E3S3 WB
ID
ID
ID
ID
ID
(I1) (I2) (ID)
(Branch to the next instruction.)

Figure 4.2 Instruction Execution Patterns (4)

(4-9) STC from DBR/GBR/Rp_BANK/SSR/SPC/VBR/SGR: 1 issue cycle

I1	I2	ID	s1	s2	s3	WB

(4-10) STC from SR: 1 issue cycle

I1	I2	ID	E1s1	E2s2	E3s3	WB

(4-11) STC.L from DBR/GBR/Rp_BANK/SSR/SPC/VBR/SGR: 1 issue cycle

I1	I2	ID	S1	S2	S3	WB

(4-12) STC.L from SR: 1 issue cycle

I1	I2	ID	E1S1	E2S2	E3S3	WB

(4-13) LDS to PR: 1 issue cycle

I1	I2	ID	s1	s2	s3	WB

(4-14) LDS.L to PR: 1 issue cycle

I1	I2	ID	S1	S2	S3	WB

(4-15) STS from PR: 1 issue cycle

I1	I2	ID	s1	s2	s3	WB

(4-16) STS.L from PR: 1 issue cycle

I1	I2	ID	S1	S2	S3	WB

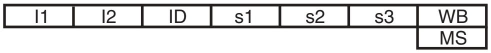
(4-17) BSRF, BSR, JSR delay slot instructions (PR set): 0 issue cycle

(I1)	(I2)	(ID)	(??1)	(??2)	(??3)	(WB)

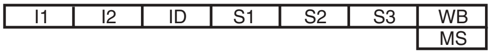
Notes: The value of PR is changed in the E3 stage of delay slot instruction.
When the STS and STS.L instructions from PR are used as delay slot instructions, changed PR value is used.

Figure 4.2 Instruction Execution Patterns (5)

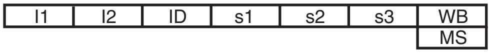
(5-1) LDS to MACH/L: 1 issue cycle



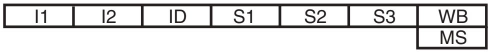
(5-2) LDS.L to MACH/L: 1 issue cycle



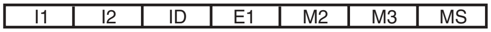
(5-3) STS from MACH/L: 1 issue cycle



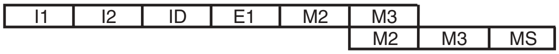
(5-4) STS.L from MACH/L: 1 issue cycle



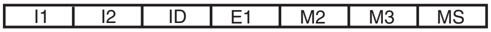
(5-5) MULS.W, MULU.W: 1 issue cycle



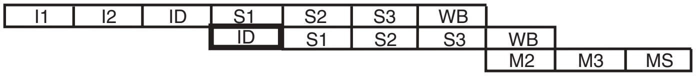
(5-6) DMULS.L, DMULU.L, MUL.L: 1 issue cycle



(5-7) CLRMAC: 1 issue cycle



(5-8) MAC.W: 2 issue cycle



(5-9) MAC.L: 2 issue cycle

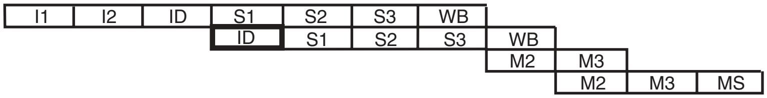


Figure 4.2 Instruction Execution Patterns (6)

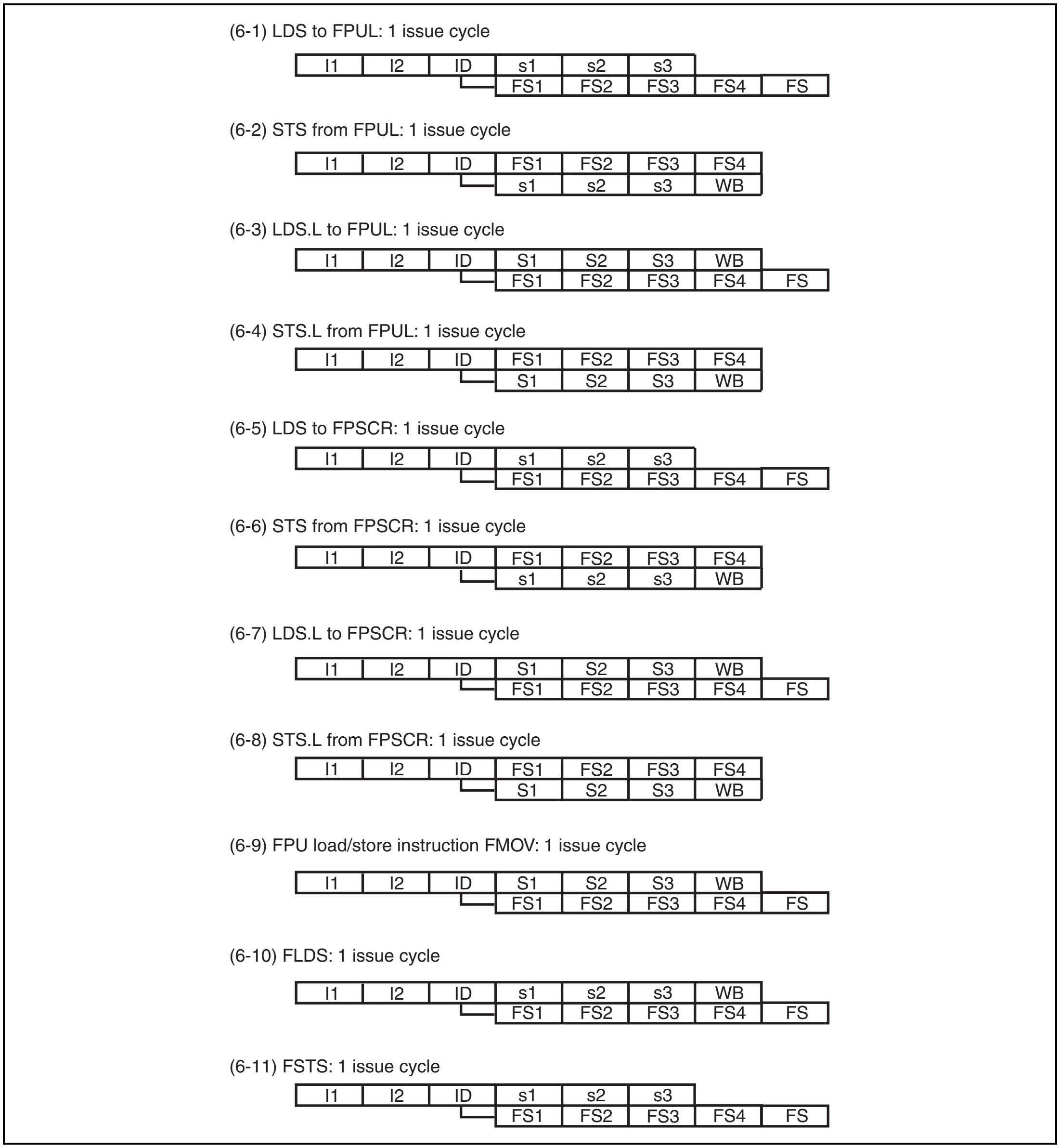
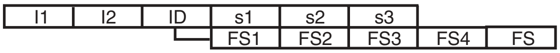
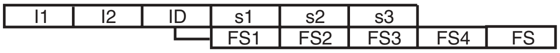


Figure 4.2 Instruction Execution Patterns (7)

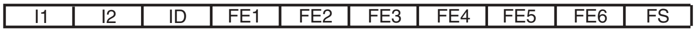
(6-12) Single-precision FABS, FNEG/double-precision FABS, FNEG: 1 issue cycle



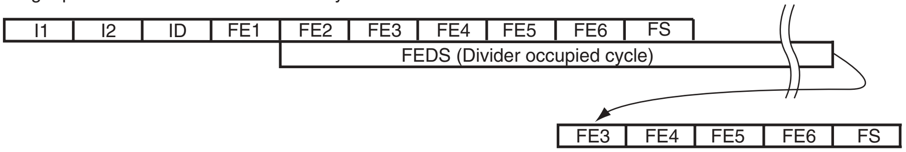
(6-13) FLDI0, FLDI1: 1 issue cycle



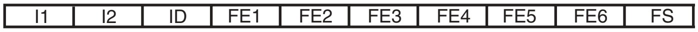
(6-14) Single-precision floating-point computation: 1 issue cycle
FCMP/EQ, FCMP/GT, FADD, FLOAT, FMAC, FMUL, FSUB, FTRC, FRCHG, FSCHG, FPCHG



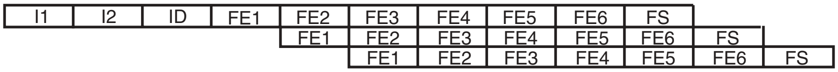
(6-15) Single-precision FDIV/FSQRT: 1 issue cycle


I1 I2 ID FE1 FE2 FE3 FE4 FE5 FE6 FS
FEDS (Divider occupied cycle)
FE3 FE4 FE5 FE6 FS

(6-16) Double-precision floating-point computation: 1 issue cycle
FCMP/EQ, FCMP/GT, FADD, FLOAT, FSUB, FTRC, FCNVSD, FCNVDS



(6-17) Double-precision floating-point computation: 1 issue cycle
FMUL


I1 I2 ID FE1 FE2 FE3 FE4 FE5 FE6 FS
FE1 FE2 FE3 FE4 FE5 FE6 FS
FE1 FE2 FE3 FE4 FE5 FE6 FS

(6-18) Double-precision FDIV/FSQRT: 1 issue cycle

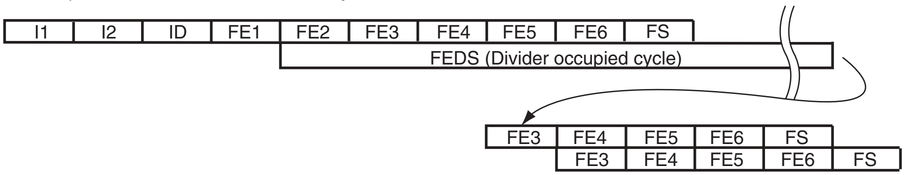

I1 I2 ID FE1 FE2 FE3 FE4 FE5 FE6 FS
FEDS (Divider occupied cycle)
FE3 FE4 FE5 FE6 FS
FE3 FE4 FE5 FE6 FS

Figure 4.2 Instruction Execution Patterns (8)

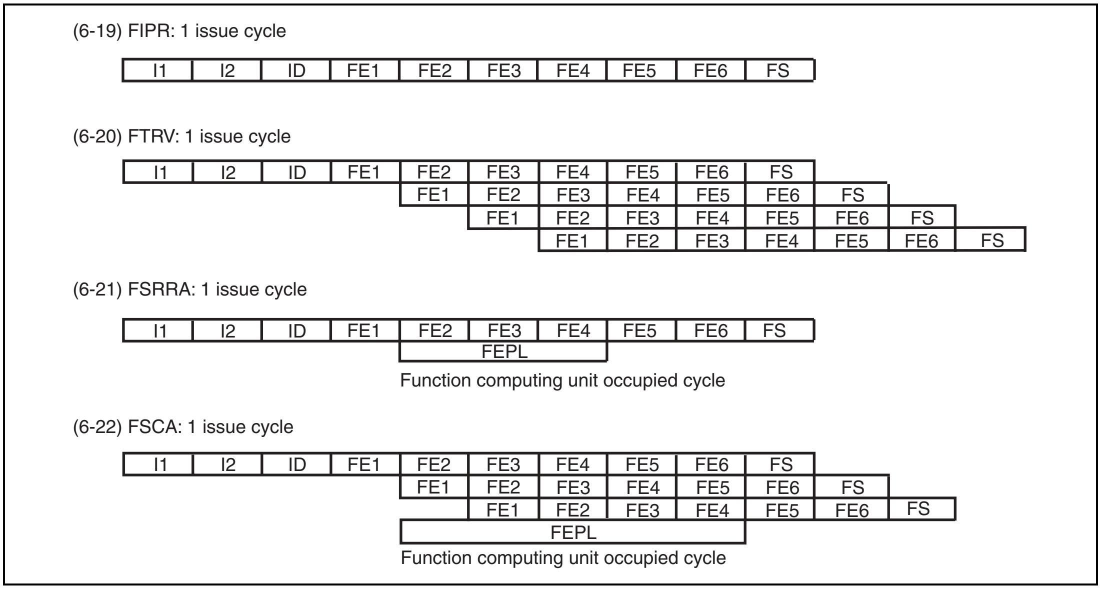

(6-19) FIPR: 1 issue cycle
I1 I2 ID FE1 FE2 FE3 FE4 FE5 FE6 FS
(6-20) FTRV: 1 issue cycle
I1 I2 ID FE1 FE2 FE3 FE4 FE5 FE6 FS
FE1 FE2 FE3 FE4 FE5 FE6 FS
FE1 FE2 FE3 FE4 FE5 FE6 FS
FE1 FE2 FE3 FE4 FE5 FE6 FS
(6-21) FSRRA: 1 issue cycle
I1 I2 ID FE1 FE2 FE3 FE4 FE5 FE6 FS
FEPL
Function computing unit occupied cycle
(6-22) FSCA: 1 issue cycle
I1 I2 ID FE1 FE2 FE3 FE4 FE5 FE6 FS
FE1 FE2 FE3 FE4 FE5 FE6 FS
FE1 FE2 FE3 FE4 FE5 FE6 FS
FEPL
Function computing unit occupied cycle

Figure 4.2 Instruction Execution Patterns (9)

4.2 Parallel-Executability

Instructions are categorized into six groups according to the internal function blocks used, as shown in table 4.2. Table 4.3 shows the parallel-executability of pairs of instructions in terms of groups. For example, ADD in the EX group and BRA in the BR group can be executed in parallel.

Table 4.2 Instruction Groups

Instruction Group	Instruction			
EX	ADD	DT	ROTL	SHLR8
	ADDC	EXTS	ROTR	SHLR16
	ADDV	EXTU	SETS	SUB
	AND #imm,R0	MOVT	SETT	SUBC
	AND Rm,Rn	MUL.L	SHAD	SUBV
	CLRMAC	MULS.W	SHAL	SWAP
	CLRS	MULU.W	SHAR	TST #imm,R0
	CLRT	NEG	SHLD	TST Rm,Rn
	CMP	NEGC	SHLL	XOR #imm,R0
	DIV0S	NOT	SHLL2	XOR Rm,Rn
	DIV0U	OR #imm,R0	SHLL8	XTRCT
	DIV1	OR Rm,Rn	SHLL16	
	DMUS.L	ROTCL	SHLR	
	DMULU.L	ROTCR	SHLR2	
MT	MOV #imm,Rn	MOV Rm,Rn	NOP	
BR	BF	BRAF	BT	JSR
	BF/S	BSR	BT/S	RTS
	BRA	BSRF	JMP	
LS	FABS	FMOV.S FR,@adr	MOV.[BWL] @adr,R	STC CR2,Rn
	FNEG	FSTS	MOV.[BWL] R,@adr	STC.L CR2,@-Rn
	FLDI0	LDC Rm,CR1	MOVA	STS SR2,Rn
	FLDI1	LDC.L @Rm+,CR1	MOVCA.L	STS.L SR2,@-Rn
	FLDS	LDS Rm,SR1	MOVUA	STS SR1,Rn
	FMOV @adr,FR	LDS Rm,SR2	OCBI	STS.L SR1,@-Rn
	FMOV FR,@adr	LDS.L @adr,SR2	OCBP	
	FMOV FR,FR	LDS.L @Rm+,SR1	OCBWB	
	FMOV.S @adr,FR	LDS.L @Rm+,SR2	PREF	
FE	FADD	FDIV	FRCHG	FSCA
	FSUB	FIPR	FSCHG	FSRRA
	FCMP (S/D)	FLOAT	FSQRT	FPCHG
	FCNVDS	FMAC	FTRC	
	FCNVSD	FMUL	FTRV	

Instruction Group	Instruction			
CO	AND.B #imm,@(R0,GBR)	LDC.L @Rm+,SR	PREFI	TRAPA
	ICBI	LDTLB	RTE	TST.B #imm,@(R0,GBR)
	LDC Rm,DBR	MAC.L	SLEEP	XOR.B #imm,@(R0,GBR)
	LDC Rm, SGR	MAC.W	STC SR,Rn	
	LDC Rm,SR	MOVCO	STC.L SR,@-Rn	
	LDC.L @Rm+,DBR	MOVLI	SYNCO	
	LDC.L @Rm+,SGR	OR.B #imm,@(R0,GBR)	TAS.B	

[Legend]

R:	Rm/Rn
@adr:	Address
SR1:	MACH/MACL/PR
SR2:	FPUL/FPSCR
CR1:	GBR/Rp_BANK/SPC/SSR/VBR
CR2:	CR1/DBR/SGR
FR:	FRm/FRn/DRm/DRn/XDm/XDn

The parallel execution of two instructions can be carried out under following conditions.

1. Both addr (preceding instruction) and addr+2 (following instruction) are specified within the minimum page size (1 Kbyte).
2. The execution of these two instructions is supported in table 4.3, Combination of Preceding and Following Instructions.
3. Data used by an instruction of addr does not conflict with data used by a previous instruction
4. Data used by an instruction of addr+2 does not conflict with data used by a previous instruction
5. Both instructions are valid

Table 4.3 Combination of Preceding and Following Instructions

		Preceding Instruction (addr)					
		EX	MT	BR	LS	FE	CO
Following Instruction (addr+2)	EX	No	Yes	Yes	Yes	Yes	
	MT	Yes	Yes	Yes	Yes	Yes	
	BR	Yes	Yes	No	Yes	Yes	
	LS	Yes	Yes	Yes	No	Yes	
	FE	Yes	Yes	Yes	Yes	No	
	CO						No

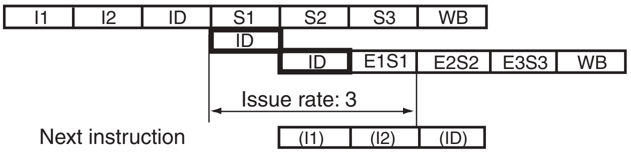
4.3 Issue Rates and Execution Cycles

Instruction execution cycles are summarized in table 4.4. Instruction Group in the table 4.4 corresponds to the category in the table 4.2. Penalty cycles due to a pipeline stall are not considered in the issue rates and execution cycles in this section.

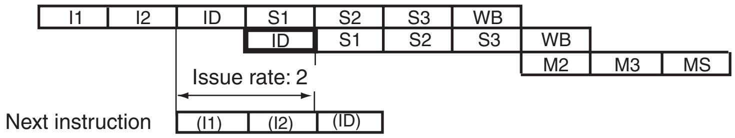
1. Issue Rate

Issue rates indicates the issue period between one instruction and next instruction.

E.g. AND.B instruction



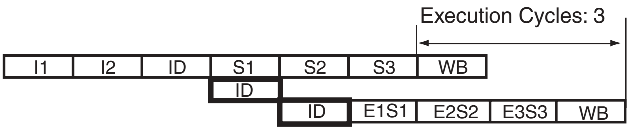
E.g. MAC.W instruction



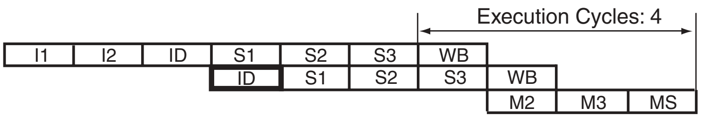
2. Execution Cycles

Execution cycles indicates the cycle counts an instruction occupied the pipeline based on the next rules.

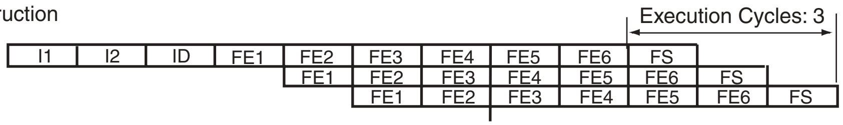
CPU instruction
E.g. AND.B instruction



E.g. MAC.W instruction



FPU instruction
E.g. FMUL instruction



E.g. FDIV instruction

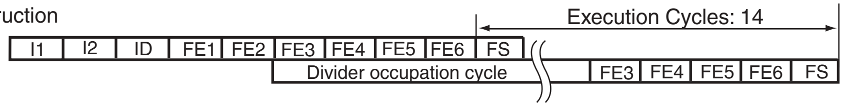


Table 4.4 Issue Rates and Execution Cycles

Functional Category	No.	Instruction		Instruction Group	Issue Rate	Execution Cycles	Execution Pattern
Data transfer instructions	1	EXTS.B	Rm,Rn	EX	1	1	2-1
	2	EXTS.W	Rm,Rn	EX	1	1	2-1
	3	EXTU.B	Rm,Rn	EX	1	1	2-1
	4	EXTU.W	Rm,Rn	EX	1	1	2-1
	5	MOV	Rm,Rn	MT	1	1	2-4
	6	MOV	#imm,Rn	MT	1	1	2-3
	7	MOVA	@(disp,PC),R0	LS	1	1	2-2
	8	MOV.W	@(disp,PC),Rn	LS	1	1	3-1
	9	MOV.L	@(disp,PC),Rn	LS	1	1	3-1
	10	MOV.B	@Rm,Rn	LS	1	1	3-1
	11	MOV.W	@Rm,Rn	LS	1	1	3-1
	12	MOV.L	@Rm,Rn	LS	1	1	3-1
	13	MOV.B	@Rm+,Rn	LS	1	1	3-1
	14	MOV.W	@Rm+,Rn	LS	1	1	3-1
	15	MOV.L	@Rm+,Rn	LS	1	1	3-1
	16	MOV.B	@(disp,Rm),R0	LS	1	1	3-1
	17	MOV.W	@(disp,Rm),R0	LS	1	1	3-1
	18	MOV.L	@(disp,Rm),Rn	LS	1	1	3-1
	19	MOV.B	@(R0,Rm),Rn	LS	1	1	3-1
	20	MOV.W	@(R0,Rm),Rn	LS	1	1	3-1
	21	MOV.L	@(R0,Rm),Rn	LS	1	1	3-1
	22	MOV.B	@(disp,GBR),R0	LS	1	1	3-1
	23	MOV.W	@(disp, GBR),R0	LS	1	1	3-1
	24	MOV.L	@(disp, GBR),R0	LS	1	1	3-1
	25	MOV.B	Rm,@Rn	LS	1	1	3-1
	26	MOV.W	Rm,@Rn	LS	1	1	3-1
	27	MOV.L	Rm,@Rn	LS	1	1	3-1
	28	MOV.B	Rm,@-Rn	LS	1	1	3-1
	29	MOV.W	Rm,@-Rn	LS	1	1	3-1
	30	MOV.L	Rm,@-Rn	LS	1	1	3-1
	31	MOV.B	R0,@(disp,Rn)	LS	1	1	3-1

Functional Category	No.	Instruction		Instruction Group	Issue Rate	Execution Cycles	Execution Pattern
Data transfer instructions	32	MOV.W	R0,@(disp,Rn)	LS	1	1	3-1
	33	MOV.L	Rm,@(disp,Rn)	LS	1	1	3-1
	34	MOV.B	Rm,@(R0,Rn)	LS	1	1	3-1
	35	MOV.W	Rm,@(R0,Rn)	LS	1	1	3-1
	36	MOV.L	Rm,@(R0,Rn)	LS	1	1	3-1
	37	MOV.B	R0,@(disp,GBR)	LS	1	1	3-1
	38	MOV.W	R0,@(disp,GBR)	LS	1	1	3-1
	39	MOV.L	R0,@(disp,GBR)	LS	1	1	3-1
	40	MOVCA.L	R0,@Rn	LS	1	1	3-4
	41	MOVCO.L	R0,@Rn	CO	1	1	3-9
	42	MOVLI.L	@Rm,R0	CO	1	1	3-8
	43	MOVUA.L	@Rm,R0	LS	2	2	3-10
	44	MOVUA.L	@Rm+,R0	LS	2	2	3-10
	45	MOVT	Rn	EX	1	1	2-1
	46	OCBI	@Rn	LS	1	1	3-4
	47	OCBP	@Rn	LS	1	1	3-4
	48	OCBWB	@Rn	LS	1	1	3-4
	49	PREF	@Rn	LS	1	1	3-4
	50	SWAP.B	Rm,Rn	EX	1	1	2-1
	51	SWAP.W	Rm,Rn	EX	1	1	2-1
	52	XTRCT	Rm,Rn	EX	1	1	2-1
Fixed-point arithmetic instructions	53	ADD	Rm,Rn	EX	1	1	2-1
	54	ADD	#imm,Rn	EX	1	1	2-1
	55	ADDC	Rm,Rn	EX	1	1	2-1
	56	ADDV	Rm,Rn	EX	1	1	2-1
	57	CMP/EQ	#imm,R0	EX	1	1	2-1
	58	CMP/EQ	Rm,Rn	EX	1	1	2-1
	59	CMP/GE	Rm,Rn	EX	1	1	2-1
	60	CMP/GT	Rm,Rn	EX	1	1	2-1
	61	CMP/HI	Rm,Rn	EX	1	1	2-1
	62	CMP/HS	Rm,Rn	EX	1	1	2-1

Functional Category	No.	Instruction		Instruction Group	Issue Rate	Execution Cycles	Execution Pattern
Fixed-point arithmetic instructions	63	CMP/PL	Rn	EX	1	1	2-1
	64	CMP/PZ	Rn	EX	1	1	2-1
	65	CMP/STR	Rm,Rn	EX	1	1	2-1
	66	DIV0S	Rm,Rn	EX	1	1	2-1
	67	DIV0U		EX	1	1	2-1
	68	DIV1	Rm,Rn	EX	1	1	2-1
	69	DMULS.L	Rm,Rn	EX	1	2	5-6
	70	DMULU.L	Rm,Rn	EX	1	2	5-6
	71	DT	Rn	EX	1	1	2-1
	72	MAC.L	@Rm+,@Rn+	CO	2	5	5-9
	73	MAC.W	@Rm+,@Rn+	CO	2	4	5-8
	74	MUL.L	Rm,Rn	EX	1	2	5-6
	75	MULS.W	Rm,Rn	EX	1	1	5-5
	76	MULU.W	Rm,Rn	EX	1	1	5-5
	77	NEG	Rm,Rn	EX	1	1	2-1
	78	NEGC	Rm,Rn	EX	1	1	2-1
	79	SUB	Rm,Rn	EX	1	1	2-1
	80	SUBC	Rm,Rn	EX	1	1	2-1
	81	SUBV	Rm,Rn	EX	1	1	2-1
Logical instructions	82	AND	Rm,Rn	EX	1	1	2-1
	83	AND	#imm,R0	EX	1	1	2-1
	84	AND.B	#imm,@(R0,GBR)	CO	3	3	3-2
	85	NOT	Rm,Rn	EX	1	1	2-1
	86	OR	Rm,Rn	EX	1	1	2-1
	87	OR	#imm,R0	EX	1	1	2-1
	88	OR.B	#imm,@(R0,GBR)	CO	3	3	3-2
	89	TAS.B	@Rn	CO	4	4	3-3
	90	TST	Rm,Rn	EX	1	1	2-1
	91	TST	#imm,R0	EX	1	1	2-1
	92	TST.B	#imm,@(R0,GBR)	CO	3	3	3-2
	93	XOR	Rm,Rn	EX	1	1	2-1
	94	XOR	#imm,R0	EX	1	1	2-1

Functional Category	No.	Instruction		Instruction Group	Issue Rate	Execution Cycles	Execution Pattern
Logical instructions	95	XOR.B	#imm,@(R0,GBR)	CO	3	3	3-2
Shift instructions	96	ROTL	Rn	EX	1	1	2-1
	97	ROTR	Rn	EX	1	1	2-1
	98	ROTCL	Rn	EX	1	1	2-1
	99	ROTCR	Rn	EX	1	1	2-1
	100	SHAD	Rm,Rn	EX	1	1	2-1
	101	SHAL	Rn	EX	1	1	2-1
	102	SHAR	Rn	EX	1	1	2-1
	103	SHLD	Rm,Rn	EX	1	1	2-1
	104	SHLL	Rn	EX	1	1	2-1
	105	SHLL2	Rn	EX	1	1	2-1
	106	SHLL8	Rn	EX	1	1	2-1
	107	SHLL16	Rn	EX	1	1	2-1
	108	SHLR	Rn	EX	1	1	2-1
	109	SHLR2	Rn	EX	1	1	2-1
	110	SHLR8	Rn	EX	1	1	2-1
	111	SHLR16	Rn	EX	1	1	2-1
Branch instructions	112	BF	disp	BR	1+0 to 2	1	1-1
	113	BF/S	disp	BR	1+0 to 2	1	1-1
	114	BT	disp	BR	1+0 to 2	1	1-1
	115	BT/S	disp	BR	1+0 to 2	1	1-1
	116	BRA	disp	BR	1+0 to 2	1	1-1
	117	BRAF	Rm	BR	1+3	1	1-2
	118	BSR	disp	BR	1+0 to 2	1	1-1
	119	BSRF	Rm	BR	1+3	1	1-2
	120	JMP	@Rn	BR	1+3	1	1-2
	121	JSR	@Rn	BR	1+3	1	1-2
	122	RTS		BR	1+0 to 3	1	1-3
System control instructions	123	NOP		MT	1	1	2-3
	124	CLRMAC		EX	1	1	5-7
	125	CLRS		EX	1	1	2-1

Functional Category	No.	Instruction		Instruction Group	Issue Rate	Execution Cycles	Execution Pattern
System control instructions	126	CLRT		EX	1	1	2-1
	127	ICBI	@Rn	CO	8+5+3	13	3-6
	128	SETS		EX	1	1	2-1
	129	SETT		EX	1	1	2-1
	130	PREFI		CO	5+5+3	10	3-7
	131	SYNCO	@Rn	CO	Undefined	Undefined	3-4
	132	TRAPA	#imm	CO	8+5+1	13	1-5
	133	RTE		CO	4+1	4	1-4
	134	SLEEP		CO	Undefined	Undefined	1-6
	135	LDTLB		CO	1	1	3-5
	136	LDC	Rm,DBR	CO	4	4	4-2
	137	LDC	Rm,SGR	CO	4	4	4-2
	138	LDC	Rm,GBR	LS	1	1	4-3
	139	LDC	Rm,Rp_BANK	LS	1	1	4-1
	140	LDC	Rm,SR	CO	4+3	4	4-4
	141	LDC	Rm,SSR	LS	1	1	4-1
	142	LDC	Rm,SPC	LS	1	1	4-1
	143	LDC	Rm,VBR	LS	1	1	4-1
	144	LDC.L	@Rm+,DBR	CO	4	4	4-6
	145	LDC.L	@Rm+,SGR	CO	4	4	4-6
	146	LDC.L	@Rm+,GBR	LS	1	1	4-7
	147	LDC.L	@Rm+,Rp_BANK	LS	1	1	4-5
	148	LDC.L	@Rm+,SR	CO	6+3	4	4-8
	149	LDC.L	@Rm+,SSR	LS	1	1	4-5
	150	LDC.L	@Rm+,SPC	LS	1	1	4-5
	151	LDC.L	@Rm+,VBR	LS	1	1	4-5
	152	LDS	Rm,MACH	LS	1	1	5-1
	153	LDS	Rm,MACL	LS	1	1	5-1
	154	LDS	Rm,PR	LS	1	1	4-13
	155	LDS.L	@Rm+,MACH	LS	1	1	5-2
	156	LDS.L	@Rm+,MACL	LS	1	1	5-2
	157	LDS.L	@Rm+,PR	LS	1	1	4-14

Functional Category	No.	Instruction		Instruction Group	Issue Rate	Execution Cycles	Execution Pattern
System control instructions	158	STC	DBR,Rn	LS	1	1	4-9
	159	STC	SGR,Rn	LS	1	1	4-9
	160	STC	GBR,Rn	LS	1	1	4-9
	161	STC	Rp_BANK,Rn	LS	1	1	4-9
	162	STC	SR,Rn	CO	1	1	4-10
	163	STC	SSR,Rn	LS	1	1	4-9
	164	STC	SPC,Rn	LS	1	1	4-9
	165	STC	VBR,Rn	LS	1	1	4-9
	166	STC.L	DBR,@-Rn	LS	1	1	4-11
	167	STC.L	SGR,@-Rn	LS	1	1	4-11
	168	STC.L	GBR,@-Rn	LS	1	1	4-11
	169	STC.L	Rp_BANK,@-Rn	LS	1	1	4-11
	170	STC.L	SR,@-Rn	CO	1	1	4-12
	171	STC.L	SSR,@-Rn	LS	1	1	4-11
	172	STC.L	SPC,@-Rn	LS	1	1	4-11
	173	STC.L	VBR,@-Rn	LS	1	1	4-11
	174	STS	MACH,Rn	LS	1	1	5-3
	175	STS	MACL,Rn	LS	1	1	5-3
	176	STS	PR,Rn	LS	1	1	4-15
	177	STS.L	MACH,@-Rn	LS	1	1	5-4
	178	STS.L	MACL,@-Rn	LS	1	1	5-4
	179	STS.L	PR,@-Rn	LS	1	1	4-16
Single-precision floating-point instructions	180	FLDI0	FRn	LS	1	1	6-13
	181	FLDI1	FRn	LS	1	1	6-13
	182	FMOV	FRm,FRn	LS	1	1	6-9
	183	FMOV.S	@Rm,FRn	LS	1	1	6-9
	184	FMOV.S	@Rm+,FRn	LS	1	1	6-9
	185	FMOV.S	@(R0,Rm),FRn	LS	1	1	6-9
	186	FMOV.S	FRm,@Rn	LS	1	1	6-9
	187	FMOV.S	FRm,@-Rn	LS	1	1	6-9
	188	FMOV.S	FRm,@(R0,Rn)	LS	1	1	6-9

Functional Category	No.	Instruction		Instruction Group	Issue Rate	Execution Cycles	Execution Pattern
Single-precision floating-point instructions	189	FLDS	FRm,FPUL	LS	1	1	6-10
	190	FSTS	FPUL,FRn	LS	1	1	6-11
	191	FABS	FRn	LS	1	1	6-12
	192	FADD	FRm,FRn	FE	1	1	6-14
	193	FCMP/EQ	FRm,FRn	FE	1	1	6-14
	194	FCMP/GT	FRm,FRn	FE	1	1	6-14
	195	FDIV	FRm,FRn	FE	1	14	6-15
	196	FLOAT	FPUL,FRn	FE	1	1	6-14
	197	FMAC	FR0,FRm,FRn	FE	1	1	6-14
	198	FMUL	FRm,FRn	FE	1	1	6-14
	199	FNEG	FRn	LS	1	1	6-12
	200	FSQRT	FRn	FE	1	30	6-15
	201	FSUB	FRm,FRn	FE	1	1	6-14
	202	FTRC	FRm,FPUL	FE	1	1	6-14
	203	FMOV	DRm,DRn	LS	1	1	6-9
	204	FMOV	@Rm,DRn	LS	1	1	6-9
	205	FMOV	@Rm+,DRn	LS	1	1	6-9
	206	FMOV	@(R0,Rm),DRn	LS	1	1	6-9
	207	FMOV	DRm,@Rn	LS	1	1	6-9
	208	FMOV	DRm,@-Rn	LS	1	1	6-9
	209	FMOV	DRm,@(R0,Rn)	LS	1	1	6-9
Double-precision floating-point instructions	210	FABS	DRn	LS	1	1	6-12
	211	FADD	DRm,DRn	FE	1	1	6-16
	212	FCMP/EQ	DRm,DRn	FE	1	1	6-16
	213	FCMP/GT	DRm,DRn	FE	1	1	6-16
	214	FCNVDS	DRm,FPUL	FE	1	1	6-16
	215	FCNVSD	FPUL,DRn	FE	1	1	6-16
	216	FDIV	DRm,DRn	FE	1	14	6-18
	217	FLOAT	FPUL,DRn	FE	1	1	6-16
	218	FMUL	DRm,DRn	FE	1	3	6-17
	219	FNEG	DRn	LS	1	1	6-12

Functional Category	No.	Instruction		Instruction Group	Issue Rate	Execution Cycles	Execution Pattern
Double-precision floating-point instructions	220	FSQRT	DRn	FE	1	30	6-18
	221	FSUB	DRm,DRn	FE	1	1	6-16
	222	FTRC	DRm,FPUL	FE	1	1	6-16
FPU system control instructions	223	LDS	Rm,FPUL	LS	1	1	6-1
	224	LDS	Rm,FPSCR	LS	1	1	6-5
	225	LDS.L	@Rm+,FPUL	LS	1	1	6-3
	226	LDS.L	@Rm+,FPSCR	LS	1	1	6-7
	227	STS	FPUL,Rn	LS	1	1	6-2
	228	STS	FPSCR,Rn	LS	1	1	6-6
	229	STS.L	FPUL,@-Rn	LS	1	1	6-4
	230	STS.L	FPSCR,@-Rn	LS	1	1	6-8
Graphics acceleration instructions	231	FMOV	DRm,XDn	LS	1	1	6-9
	232	FMOV	XDm,DRn	LS	1	1	6-9
	233	FMOV	XDm,XDn	LS	1	1	6-9
	234	FMOV	@Rm,XDn	LS	1	1	6-9
	235	FMOV	@Rm+,XDn	LS	1	1	6-9
	236	FMOV	@(R0,Rm),XDn	LS	1	1	6-9
	237	FMOV	XDm,@Rn	LS	1	1	6-9
	238	FMOV	XDm,@-Rn	LS	1	1	6-9
	239	FMOV	XDm,@(R0,Rn)	LS	1	1	6-9
	240	FIPR	FVm,FVn	FE	1	1	6-19
	241	FRCHG		FE	1	1	6-14
	242	FSCHG		FE	1	1	6-14
	243	FPCHG		FE	1	1	6-14
	244	FSRRA	FRn	FE	1	1	6-21
	245	FSCA	FPUL,DRn	FE	1	3	6-22
	246	FTRV	XMTRX,FVn	FE	1	4	6-20

Section 5 Exception Handling

5.1 Summary of Exception Handling

Exception handling processing is handled by a special routine which is executed by a reset, general exception handling, or interrupt. For example, if the executing instruction ends abnormally, appropriate action must be taken in order to return to the original program sequence, or report the abnormality before terminating the processing. The process of generating an exception handling request in response to abnormal termination, and passing control to a user-written exception handling routine, in order to support such functions, is given the generic name of exception handling.

The exception handling in the SH-4A is of three kinds: resets, general exceptions, and interrupts.

5.2 Register Descriptions

Table 5.1 lists the configuration of registers related exception handling.

Table 5.1 Register Configuration

Register Name	Abbr.	R/W	P4 Address*	Area 7 Address*	Access Size
TRAPA exception register	TRA	R/W	H'FF00 0020	H'1F00 0020	32
Exception event register	EXPEVT	R/W	H'FF00 0024	H'1F00 0024	32
Interrupt event register	INTEVT	R/W	H'FF00 0028	H'1F00 0028	32

Note: * P4 is the address when virtual address space P4 area is used. Area 7 is the address when physical address space area 7 is accessed by using the TLB.

Table 5.2 States of Register in Each Operating Mode

Register Name	Abbr.	Power-on Reset	Manual Reset	Sleep	Standby
TRAPA exception register	TRA	Undefined	Undefined	Retained	Retained
Exception event register	EXPEVT	H'0000 0000	H'0000 0020	Retained	Retained
Interrupt event register	INTEVT	Undefined	Undefined	Retained	Retained

5.2.1 TRAPA Exception Register (TRA)

The TRAPA exception register (TRA) consists of 8-bit immediate data (imm) for the TRAPA instruction. TRA is set automatically by hardware when a TRAPA instruction is executed. TRA can also be modified by software.

Bit:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
Initial value:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
R/W:	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R

Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	—	—	—	—	—	—	TRACODE								—	—
Initial value:	0	0	0	0	0	0	—	—	—	—	—	—	—	—	0	0
R/W:	R	R	R	R	R	R	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R	R

Bit	Bit Name	Initial Value	R/W	Description
31 to 10	—	All 0	R	Reserved For details on reading/writing this bit, see General Precautions on Handling of Product.
9 to 2	TRACODE	Undefined	R/W	TRAPA Code 8-bit immediate data of TRAPA instruction is set
1, 0	—	All 0	R	Reserved For details on reading/writing this bit, see General Precautions on Handling of Product.

5.2.2 Exception Event Register (EXPEVT)

The exception event register (EXPEVT) consists of a 12-bit exception code. The exception code set in EXPEVT is that for a reset or general exception event. The exception code is set automatically by hardware when an exception occurs. EXPEVT can also be modified by software.

Bit:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
Initial value:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
R/W:	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R

Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	—	—	—	—	EXPCODE											
Initial value:	0	0	0	0	0	0	0	0	0	0	0/1	0	0	0	0	0
R/W:	R	R	R	R	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

Bit	Bit Name	Initial Value	R/W	Description
31 to 12	—	All 0	R	Reserved For details on reading/writing this bit, see General Precautions on Handling of Product.
11 to 0	EXPCODE	H'000 or H'020	R/W	Exception Code The exception code for a reset or general exception is set. For details, see table 5.3.

5.2.3 Interrupt Event Register (INTEVT)

The interrupt event register (INTEVT) consists of a 14-bit exception code. The exception code is set automatically by hardware when an exception occurs. INTEVT can also be modified by software.

Bit:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
Initial value:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
R/W:	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R

Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	—	—	INTCODE													
Initial value:	0	0	—	—	—	—	—	—	—	—	—	—	—	—	—	—
R/W:	R	R	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

Bit	Bit Name	Initial Value	R/W	Description
31 to 14	—	All 0	R	Reserved For details on reading/writing this bit, see General Precautions on Handling of Product.
13 to 0	INTCODE	Undefined	R/W	Exception Code The exception code for an interrupt is set. For details, see table 5.3.

5.3 Exception Handling Functions

5.3.1 Exception Handling Flow

In exception handling, the contents of the program counter (PC), status register (SR), and R15 are saved in the saved program counter (SPC), saved status register (SSR), and saved general register15 (SGR), and the CPU starts execution of the appropriate exception handling routine according to the vector address. An exception handling routine is a program written by the user to handle a specific exception. The exception handling routine is terminated and control returned to the original program by executing a return-from-exception instruction (RTE). This instruction restores the PC and SR contents and returns control to the normal processing routine at the point at which the exception occurred. The SGR contents are not written back to R15 with an RTE instruction.

The basic processing flow is as follows. For the meaning of the SR bits, see section 2, Programming Model.

1. The PC, SR, and R15 contents are saved in SPC, SSR, and SGR, respectively.
2. The block bit (BL) in SR is set to 1.
3. The mode bit (MD) in SR is set to 1.
4. The register bank bit (RB) in SR is set to 1.
5. In a reset, the FPU disable bit (FD) in SR is cleared to 0.
6. The exception code is written to bits 11 to 0 of the exception event register (EXPEVT) or interrupt event register (INTEVT).
7. The CPU branches to the determined exception handling vector address, and the exception handling routine begins.

5.3.2 Exception Handling Vector Addresses

The reset vector address is fixed at H'A0000000. Exception and interrupt vector addresses are determined by adding the offset for the specific event to the vector base address, which is set by software in the vector base register (VBR). In the case of the TLB miss exception, for example, the offset is H'00000400, so if H'9C080000 is set in VBR, the exception handling vector address will be H'9C080400. If a further exception occurs at the exception handling vector address, a duplicate exception will result, and recovery will be difficult; therefore, addresses that are not to be converted (in P1 and P2 areas) should be specified for vector addresses.

5.4 Exception Types and Priorities

Table 5.3 shows the types of exceptions, with their relative priorities, vector addresses, and exception/interrupt codes.

Table 5.3 Exceptions

					Exception Transition Direction*3		
Exception Category	**Execution Mode**	**Exception**	**Priority Level*2**	**Priority Order*2**	**Vector Address**	**Offset**	**Exception Code*4**
Reset	Abort type	Power-on reset	1	1	H'A000 0000	—	H'000
		Manual reset	1	2	H'A000 0000	—	H'020
		H-UDI reset	1	1	H'A000 0000	—	H'000
		Instruction TLB multiple-hit exception	1	3	H'A000 0000	—	H'140
		Data TLB multiple-hit exception	1	4	H'A000 0000	—	H'140
General exception	Re-execution type	User break before instruction execution*	2	0	(VBR/DBR)	H'100/—	H'1E0
		Instruction address error	2	1	(VBR)	H'100	H'0E0
		Instruction TLB miss exception	2	2	(VBR)	H'400	H'040
		Instruction TLB protection violation exception	2	3	(VBR)	H'100	H'0A0
		General illegal instruction exception	2	4	(VBR)	H'100	H'180
		Slot illegal instruction exception	2	4	(VBR)	H'100	H'1A0
		General FPU disable exception	2	4	(VBR)	H'100	H'800
		Slot FPU disable exception	2	4	(VBR)	H'100	H'820
		Data address error (read)	2	5	(VBR)	H'100	H'0E0
		Data address error (write)	2	5	(VBR)	H'100	H'100
		Data TLB miss exception (read)	2	6	(VBR)	H'400	H'040
		Data TLB miss exception (write)	2	6	(VBR)	H'400	H'060
		Data TLB protection violation exception (read)	2	7	(VBR)	H'100	H'0A0
		Data TLB protection violation exception (write)	2	7	(VBR)	H'100	H'0C0
		FPU exception	2	8	(VBR)	H'100	H'120
		Initial page write exception	2	9	(VBR)	H'100	H'080
	Completion type	Unconditional trap (TRAPA)	2	4	(VBR)	H'100	H'160
		User break after instruction execution*	2	10	(VBR/DBR)	H'100/—	H'1E0

					Exception Transition Direction*[3]		
Exception Category	**Execution Mode**	**Exception**	**Priority Level*[2]**	**Priority Order*[2]**	**Vector Address**	**Offset**	**Exception Code*[4]**
Interrupt	Completion type	Nonmaskable interrupt	3	—	(VBR)	H'600	H'1C0
		General interrupt request	4	—	(VBR)	H'600	—

Note: 1. When UBDE in CBCR = 1, PC = DBR. In other cases, PC = VBR + H'100.
2. Priority is first assigned by priority level, then by priority order within each level (the lowest number represents the highest priority).
3. Control passes to H'A000 0000 in a reset, and to [VBR + offset] in other cases.
4. Stored in EXPEVT for a reset or general exception, and in INTEVT for an interrupt.

5.5 Exception Flow

5.5.1 Exception Flow

Figure 5.1 shows an outline flowchart of the basic operations in instruction execution and exception handling. For the sake of clarity, the following description assumes that instructions are executed sequentially, one by one. Figure 5.1 shows the relative priority order of the different kinds of exceptions (reset, general exception, and interrupt). Register settings in the event of an exception are shown only for SSR, SPC, SGR, EXPEVT/INTEVT, SR, and PC. However, other registers may be set automatically by hardware, depending on the exception. For details, see section 5.6, Description of Exceptions. Also, see section 5.6.4, Priority Order with Multiple Exceptions, for exception handling during execution of a delayed branch instruction and a delay slot instruction, or in the case of instructions in which two data accesses are performed.

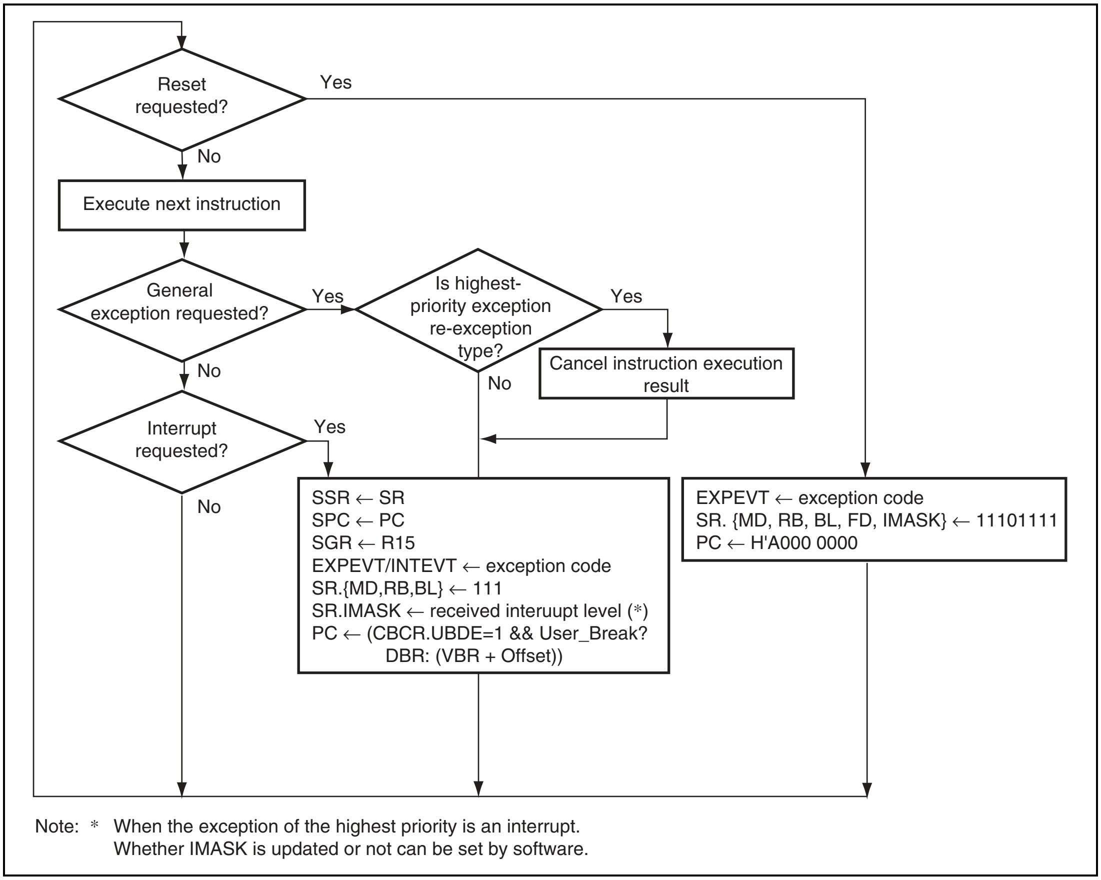

Reset requested?
Yes
No
Execute next instruction
General exception requested?
Yes
Is highest-priority exception re-exception type?
Yes
Cancel instruction execution result
No
No
Interrupt requested?
Yes
No
SSR ← SR
SPC ← PC
SGR ← R15
EXPEVT/INTEVT ← exception code
SR.{MD,RB,BL} ← 111
SR.IMASK ← received interuupt level (*)
PC ← (CBCR.UBDE=1 && User_Break? DBR: (VBR + Offset))
EXPEVT ← exception code
SR. {MD, RB, BL, FD, IMASK} ← 11101111
PC ← H'A000 0000
Note: * When the exception of the highest priority is an interrupt. Whether IMASK is updated or not can be set by software.

Figure 5.1 Instruction Execution and Exception Handling

5.5.2 Exception Source Acceptance

A priority ranking is provided for all exceptions for use in determining which of two or more simultaneously generated exceptions should be accepted. Five of the general exceptions—general illegal instruction exception, slot illegal instruction exception, general FPU disable exception, slot FPU disable exception, and unconditional trap exception—are detected in the process of instruction decoding, and do not occur simultaneously in the instruction pipeline. These exceptions therefore all have the same priority. General exceptions are detected in the order of instruction execution. However, exception handling is performed in the order of instruction flow (program order). Thus, an exception for an earlier instruction is accepted before that for a later instruction. An example of the order of acceptance for general exceptions is shown in figure 5.2.

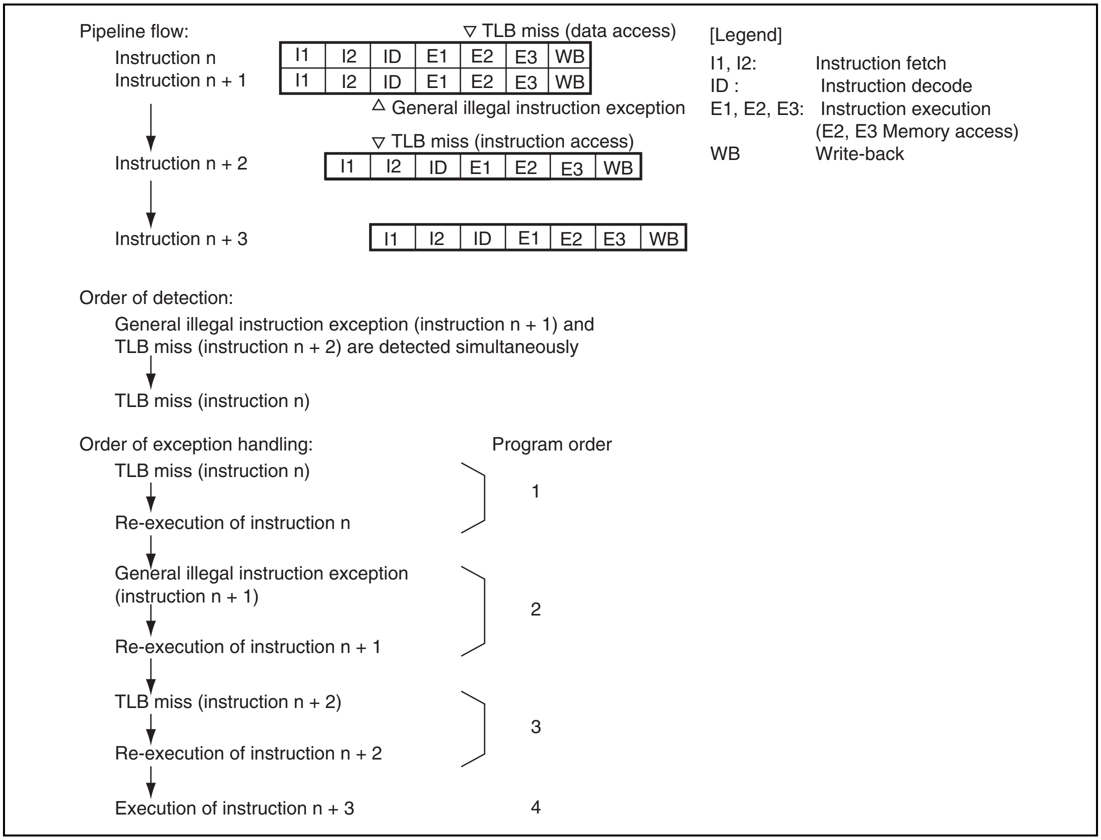

Pipeline flow:
Instruction n
Instruction n + 1
Instruction n + 2
Instruction n + 3
I1 I2 ID E1 E2 E3 WB
▽ TLB miss (data access)
△ General illegal instruction exception
▽ TLB miss (instruction access)
[Legend]
I1, I2: Instruction fetch
ID : Instruction decode
E1, E2, E3: Instruction execution (E2, E3 Memory access)
WB Write-back
Order of detection:
General illegal instruction exception (instruction n + 1) and TLB miss (instruction n + 2) are detected simultaneously
TLB miss (instruction n)
Order of exception handling:
Program order
TLB miss (instruction n)
Re-execution of instruction n
1
General illegal instruction exception (instruction n + 1)
Re-execution of instruction n + 1
2
TLB miss (instruction n + 2)
Re-execution of instruction n + 2
3
Execution of instruction n + 3
4

Figure 5.2 Example of General Exception Acceptance Order

5.5.3 Exception Requests and BL Bit

When the BL bit in SR is 0, exceptions and interrupts are accepted.

When the BL bit in SR is 1 and an exception other than a user break is generated, the CPU's internal registers and the registers of the other modules are set to their states following a manual reset, and the CPU branches to the same address as in a reset (H'A0000000). For the operation in the event of a user break, see the User Break Controller (UBC) section of the hardware manual of the target product. If an ordinary interrupt occurs, the interrupt request is held pending and is accepted after the BL bit has been cleared to 0 by software. If a nonmaskable interrupt (NMI) occurs, it can be held pending or accepted according to the setting made by software.

Thus, normally, SPC and SSR are saved and then the BL bit in SR is cleared to 0, to enable multiple exception state acceptance.

5.5.4 Return from Exception Handling

The RTE instruction is used to return from exception handling. When the RTE instruction is executed, the SPC contents are restored to PC and the SSR contents to SR, and the CPU returns from the exception handling routine by branching to the SPC address. If SPC and SSR were saved to external memory, set the BL bit in SR to 1 before restoring the SPC and SSR contents and issuing the RTE instruction.

5.6 Description of Exceptions

The various exception handling operations explained here are exception sources, transition address on the occurrence of exception, and processor operation when a transition is made.

5.6.1 Resets

Power-On Reset:

- Condition:

  Power-on reset request
- Operations:

  Exception code H'000 is set in EXPEVT, initialization of the CPU and on-chip peripheral module is carried out, and then a branch is made to the reset vector (H'A0000000). For details, see the register descriptions in the relevant sections. A power-on reset should be executed when power is supplied.

Manual Reset:

- Condition:

  Manual reset request
- Operations:

  Exception code H'020 is set in EXPEVT, initialization of the CPU and on-chip peripheral module is carried out, and then a branch is made to the branch vector (H'A0000000). The registers initialized by a power-on reset and manual reset are different. For details, see the register descriptions in the relevant sections.

H-UDI Reset:

- Source: SDIR.TI[7:4] = B'0110 (negation) or B'0111 (assertion)
- Transition address: H'A0000000
- Transition operations:

  Exception code H'000 is set in EXPEVT, initialization of VBR and SR is performed, and a branch is made to PC = H'A0000000.

  CPU and on-chip peripheral module initialization is performed. For details, see the register descriptions in the relevant sections of the hardware manual of the target product.

Instruction TLB Multiple Hit Exception:

- Source: Multiple ITLB address matches
- Transition address: H'A0000000
- Transition operations:

  The virtual address (32 bits) at which this exception occurred is set in TEA, and the corresponding virtual page number (22 bits) is set in PTEH [31:10]. ASID in PTEH indicates the ASID when this exception occurred.

  Exception code H'140 is set in EXPEVT, initialization of VBR and SR is performed, and a branch is made to PC = H'A0000000.

  CPU and on-chip peripheral module initialization is performed in the same way as in a manual reset. For details, see the register descriptions in the relevant sections of the hardware manual of the target product.

Data TLB Multiple-Hit Exception:

- Source: Multiple UTLB address matches
- Transition address: H'A0000000
- Transition operations:

  The virtual address (32 bits) at which this exception occurred is set in TEA, and the corresponding virtual page number (22 bits) is set in PTEH [31:10]. ASID in PTEH indicates the ASID when this exception occurred.

  Exception code H'140 is set in EXPEVT, initialization of VBR and SR is performed, and a branch is made to PC = H'A0000000.

  CPU and on-chip peripheral module initialization is performed in the same way as in a manual reset. For details, see the register descriptions in the relevant sections of the hardware manual of the target product.

5.6.2 General Exceptions

Data TLB Miss Exception:

- Source: Address mismatch in UTLB address comparison
- Transition address: VBR + H'00000400
- Transition operations:

  The virtual address (32 bits) at which this exception occurred is set in TEA, and the corresponding virtual page number (22 bits) is set in PTEH [31:10]. ASID in PTEH indicates the ASID when this exception occurred.

  The PC and SR contents for the instruction at which this exception occurred are saved in SPC and SSR. The R15 contents at this time are saved in SGR.

  Exception code H'040 (for a read access) or H'060 (for a write access) is set in EXPEVT. The BL, MD, and RB bits are set to 1 in SR, and a branch is made to PC = VBR + H'0400.

  To speed up TLB miss processing, the offset is separate from that of other exceptions.

```
Data_TLB_miss_exception()
{
    TEA = EXCEPTION_ADDRESS;
    PTEH.VPN = PAGE_NUMBER;
    SPC = PC;
    SSR = SR;
    SGR = R15;
    EXPEVT = read_access ? H'0000 0040 : H'0000 0060;
    SR.MD = 1;
    SR.RB = 1;
    SR.BL = 1;
    PC = VBR + H'0000 0400;
}
```

Instruction TLB Miss Exception:

- Source: Address mismatch in ITLB address comparison
- Transition address: VBR + H'00000400
- Transition operations:

  The virtual address (32 bits) at which this exception occurred is set in TEA, and the corresponding virtual page number (22 bits) is set in PTEH [31:10]. ASID in PTEH indicates the ASID when this exception occurred.

  The PC and SR contents for the instruction at which this exception occurred are saved in SPC and SSR. The R15 contents at this time are saved in SGR.

  Exception code H'40 is set in EXPEVT. The BL, MD, and RB bits are set to 1 in SR, and a branch is made to PC = VBR + H'0400.

  To speed up TLB miss processing, the offset is separate from that of other exceptions.

```
ITLB_miss_exception()
{
    TEA = EXCEPTION_ADDRESS;
    PTEH.VPN = PAGE_NUMBER;
    SPC = PC;
    SSR = SR;
    SGR = R15;
    EXPEVT = H'0000 0040;
    SR.MD = 1;
    SR.RB = 1;
    SR.BL = 1;
    PC = VBR + H'0000 0400;
}
```

Initial Page Write Exception:

- Source: TLB is hit in a store access, but dirty bit D = 0
- Transition address: VBR + H'00000100
- Transition operations:

  The virtual address (32 bits) at which this exception occurred is set in TEA, and the corresponding virtual page number (22 bits) is set in PTEH [31:10]. ASID in PTEH indicates the ASID when this exception occurred.

  The PC and SR contents for the instruction at which this exception occurred are saved in SPC and SSR. The R15 contents at this time are saved in SGR.

  Exception code H'080 is set in EXPEVT. The BL, MD, and RB bits are set to 1 in SR, and a branch is made to PC = VBR + H'0100.

```
Initial_write_exception()
{
    TEA = EXCEPTION_ADDRESS;
    PTEH.VPN = PAGE_NUMBER;
    SPC = PC;
    SSR = SR;
    SGR = R15;
    EXPEVT = H'0000 0080;
    SR.MD = 1;
    SR.RB = 1;
    SR.BL = 1;
    PC = VBR + H'0000 0100;
}
```

RENESAS

Data TLB Protection Violation Exception:

- Source: The access does not accord with the UTLB protection information (PR bits) shown below.

PR	Privileged Mode	User Mode
00	Only read access possible	Access not possible
01	Read/write access possible	Access not possible
10	Only read access possible	Only read access possible
11	Read/write access possible	Read/write access possible

- Transition address: VBR + H'00000100
- Transition operations:

  The virtual address (32 bits) at which this exception occurred is set in TEA, and the corresponding virtual page number (22 bits) is set in PTEH [31:10]. ASID in PTEH indicates the ASID when this exception occurred.

  The PC and SR contents for the instruction at which this exception occurred are saved in SPC and SSR. The R15 contents at this time are saved in SGR.

  Exception code H'0A0 (for a read access) or H'0C0 (for a write access) is set in EXPEVT. The BL, MD, and RB bits are set to 1 in SR, and a branch is made to PC = VBR + H'0100.

```
Data_TLB_protection_violation_exception()
{
    TEA = EXCEPTION_ADDRESS;
    PTEH.VPN = PAGE_NUMBER;
    SPC = PC;
    SSR = SR;
    SGR = R15;
    EXPEVT = read_access ? H'0000 00A0 : H'0000 00C0;
    SR.MD = 1;
    SR.RB = 1;
    SR.BL = 1;
    PC = VBR + H'0000 0100;
}
```

Instruction TLB Protection Violation Exception:

- Source: The access does not accord with the ITLB protection information (PR bits) shown below.

PR	Privileged Mode	User Mode
0	Access possible	Access not possible
1	Access possible	Access possible

- Transition address: VBR + H'00000100
- Transition operations:

  The virtual address (32 bits) at which this exception occurred is set in TEA, and the corresponding virtual page number (22 bits) is set in PTEH [31:10]. ASID in PTEH indicates the ASID when this exception occurred.

  The PC and SR contents for the instruction at which this exception occurred are saved in SPC and SSR. The R15 contents at this time are saved in SGR.

  Exception code H'0A0 is set in EXPEVT. The BL, MD, and RB bits are set to 1 in SR, and a branch is made to PC = VBR + H'0100.

```
ITLB_protection_violation_exception()
{
    TEA = EXCEPTION_ADDRESS;
    PTEH.VPN = PAGE_NUMBER;
    SPC = PC;
    SSR = SR;
    SGR = R15;
    EXPEVT = H'0000 00A0;
    SR.MD = 1;
    SR.RB = 1;
    SR.BL = 1;
    PC = VBR + H'0000 0100;
}
```

Data Address Error:

- Sources:
  - Word data access from other than a word boundary (2n +1)
  - Longword data access from other than a longword data boundary (4n +1, 4n + 2, or 4n +3)
  - Quadword data access from other than a quadword data boundary (8n +1, 8n + 2, 8n +3, 8n + 4, 8n + 5, 8n + 6, or 8n + 7)
  - Access to area H'80000000 to H'FFFFFFFF in user mode

    Areas H'E0000000 to H'E3FFFFFF and H'E5000000 to H'E5FFFFFF can be accessed in user mode. For details, see section 7, Memory Management Unit (MMU) and section 9, L Memory.
- Transition address: VBR + H'0000100
- Transition operations:

  The virtual address (32 bits) at which this exception occurred is set in TEA, and the corresponding virtual page number (22 bits) is set in PTEH [31:10]. ASID in PTEH indicates the ASID when this exception occurred.

  The PC and SR contents for the instruction at which this exception occurred are saved in SPC and SSR. The R15 contents at this time are saved in SGR.

  Exception code H'0E0 (for a read access) or H'100 (for a write access) is set in EXPEVT. The BL, MD, and RB bits are set to 1 in SR, and a branch is made to PC = VBR + H'0100. For details, see section 7, Memory Management Unit (MMU).

```
Data_address_error()
{
    TEA = EXCEPTION_ADDRESS;
    PTEH.VPN = PAGE_NUMBER;
    SPC = PC;
    SSR = SR;
    SGR = R15;
    EXPEVT = read_access? H'0000 00E0: H'0000 0100;
    SR.MD = 1;
    SR.RB = 1;
    SR.BL = 1;
    PC = VBR + H'0000 0100;
}
```

Instruction Address Error:

- Sources:
  - Instruction fetch from other than a word boundary (2n +1)
  - Instruction fetch from area H'80000000 to H'FFFFFFFF in user mode

    Area H'E5000000 to H'E5FFFFFF can be accessed in user mode. For details, see section 9, L Memory.
- Transition address: VBR + H'00000100
- Transition operations:

  The virtual address (32 bits) at which this exception occurred is set in TEA, and the corresponding virtual page number (22 bits) is set in PTEH [31:10]. ASID in PTEH indicates the ASID when this exception occurred.

  The PC and SR contents for the instruction at which this exception occurred are saved in the SPC and SSR. The R15 contents at this time are saved in SGR.

  Exception code H'0E0 is set in EXPEVT. The BL, MD, and RB bits are set to 1 in SR, and a branch is made to PC = VBR + H'0100. For details, see section 7, Memory Management Unit (MMU).

```
Instruction_address_error()
{
   TEA = EXCEPTION_ADDRESS;
   PTEH.VPN = PAGE_NUMBER;
   SPC = PC;
   SSR = SR;
   SGR = R15;
   EXPEVT = H'0000 00E0;
   SR.MD = 1;
   SR.RB = 1;
   SR.BL = 1;
   PC = VBR + H'0000 0100;
}
```

Unconditional Trap:

- Source: Execution of TRAPA instruction
- Transition address: VBR + H'00000100
- Transition operations:

  As this is a processing-completion-type exception, the PC contents for the instruction following the TRAPA instruction are saved in SPC. The value of SR and R15 when the TRAPA instruction is executed are saved in SSR and SGR. The 8-bit immediate value in the TRAPA instruction is multiplied by 4, and the result is set in TRA [9:0]. Exception code H'160 is set in EXPEVT. The BL, MD, and RB bits are set to 1 in SR, and a branch is made to PC = VBR + H'0100.

```
TRAPA_exception()
{
    SPC = PC + 2;
    SSR = SR;
    SGR = R15;
    TRA = imm << 2;
    EXPEVT = H'0000 0160;
    SR.MD = 1;
    SR.RB = 1;
    SR.BL = 1;
    PC = VBR + H'0000 0100;
}
```

General Illegal Instruction Exception:

- Sources:
  - Decoding of an undefined instruction not in a delay slot

    Delayed branch instructions: JMP, JSR, BRA, BRAF, BSR, BSRF, RTS, RTE, BT/S, BF/S

    Undefined instruction: H'FFFD
  - Decoding in user mode of a privileged instruction not in a delay slot

    Privileged instructions: LDC, STC, RTE, LDTLB, SLEEP, but excluding LDC/STC instructions that access GBR
- Transition address: VBR + H'00000100
- Transition operations:

  The PC and SR contents for the instruction at which this exception occurred are saved in SPC and SSR. The R15 contents at this time are saved in SGR.

  Exception code H'180 is set in EXPEVT. The BL, MD, and RB bits are set to 1 in SR, and a branch is made to PC = VBR + H'0100. Operation is not guaranteed if an undefined code other than H'FFFD is decoded.

```
General_illegal_instruction_exception()
{
   SPC = PC;
   SSR = SR;
   SGR = R15;
   EXPEVT = H'0000 0180;
   SR.MD = 1;
   SR.RB = 1;
   SR.BL = 1;
   PC = VBR + H'0000 0100;
}
```

Slot Illegal Instruction Exception:

- Sources:
  - Decoding of an undefined instruction in a delay slot

    Delayed branch instructions: JMP, JSR, BRA, BRAF, BSR, BSRF, RTS, RTE, BT/S, BF/S

    Undefined instruction: H'FFFD
  - Decoding of an instruction that modifies PC in a delay slot

    Instructions that modify PC: JMP, JSR, BRA, BRAF, BSR, BSRF, RTS, RTE, BT, BF, BT/S, BF/S, TRAPA, LDC Rm,SR, LDC.L @Rm+,SR, ICBI, PREFI
  - Decoding in user mode of a privileged instruction in a delay slot

    Privileged instructions: LDC, STC, RTE, LDTLB, SLEEP, but excluding LDC/STC instructions that access GBR
  - Decoding of a PC-relative MOV instruction or MOVA instruction in a delay slot
- Transition address: VBR + H'000 0100
- Transition operations:

  The PC contents for the preceding delayed branch instruction are saved in SPC. The SR and R15 contents when this exception occurred are saved in SSR and SGR.

  Exception code H'1A0 is set in EXPEVT. The BL, MD, and RB bits are set to 1 in SR, and a branch is made to PC = VBR + H'0100. Operation is not guaranteed if an undefined code other than H'FFFD is decoded.

```
Slot_illegal_instruction_exception()
{
   SPC = PC - 2;
   SSR = SR;
   SGR = R15;
   EXPEVT = H'0000 01A0;
   SR.MD = 1;
   SR.RB = 1;
   SR.BL = 1;
   PC = VBR + H'0000 0100;
}
```

General FPU Disable Exception:

- Source: Decoding of an FPU instruction* not in a delay slot with SR.FD =1
- Transition address: VBR + H'00000100
- Transition operations:

  The PC and SR contents for the instruction at which this exception occurred are saved in SPC and SSR. The R15 contents at this time are saved in SGR.

  Exception code H'800 is set in EXPEVT. The BL, MD, and RB bits are set to 1 in SR, and a branch is made to PC = VBR + H'0100.

Note: * FPU instructions are instructions in which the first 4 bits of the instruction code are F (but excluding undefined instruction H'FFFD), and the LDS, STS, LDS.L, and STS.L instructions corresponding to FPUL and FPSCR.

```
General_fpu_disable_exception()
{
    SPC = PC;
    SSR = SR;
    SGR = R15;
    EXPEVT = H'0000 0800;
    SR.MD = 1;
    SR.RB = 1;
    SR.BL = 1;
    PC = VBR + H'0000 0100;
}
```

Slot FPU Disable Exception:

- Source: Decoding of an FPU instruction in a delay slot with SR.FD =1
- Transition address: VBR + H'00000100
- Transition operations:

  The PC contents for the preceding delayed branch instruction are saved in SPC. The SR and R15 contents when this exception occurred are saved in SSR and SGR.

  Exception code H'820 is set in EXPEVT. The BL, MD, and RB bits are set to 1 in SR, and a branch is made to PC = VBR + H'0100.

```
Slot_fpu_disable_exception()
{
    SPC = PC - 2;
    SSR = SR;
    SGR = R15;
    EXPEVT = H'0000 0820;
    SR.MD = 1;
    SR.RB = 1;
    SR.BL = 1;
    PC = VBR + H'0000 0100;
}
```

Pre-Execution User Break/Post-Execution User Break:

- Source: Fulfilling of a break condition set in the user break controller
- Transition address: VBR + H'00000100, or DBR
- Transition operations:

  In the case of a post-execution break, the PC contents for the instruction following the instruction at which the breakpoint is set are set in SPC. In the case of a pre-execution break, the PC contents for the instruction at which the breakpoint is set are set in SPC.

  The SR and R15 contents when the break occurred are saved in SSR and SGR. Exception code H'1E0 is set in EXPEVT.

  The BL, MD, and RB bits are set to 1 in SR, and a branch is made to PC = VBR + H'0100. It is also possible to branch to PC = DBR.

  For details of PC, etc., when a data break is set, see the User Break Controller (UBC) section of the hardware manual of the target product.

```
User_break_exception()
{
   SPC = (pre_execution break? PC : PC + 2);
   SSR = SR;
   SGR = R15;
   EXPEVT = H'0000 01E0;
   SR.MD = 1;
   SR.RB = 1;
   SR.BL = 1;
   PC = (BRCR.UBDE==1 ? DBR : VBR + H'0000 0100);
}
```

FPU Exception:

- Source: Exception due to execution of a floating-point operation
- Transition address: VBR + H'00000100
- Transition operations:

  The PC and SR contents for the instruction at which this exception occurred are saved in SPC and SSR . The R15 contents at this time are saved in SGR. Exception code H'120 is set in EXPEVT. The BL, MD, and RB bits are set to 1 in SR, and a branch is made to PC = VBR + H'0100.

```
FPU_exception()
{
    SPC = PC;
    SSR = SR;
    SGR = R15;
    EXPEVT = H'0000 0120;
    SR.MD = 1;
    SR.RB = 1;
    SR.BL = 1;
    PC = VBR + H'0000 0100;
}
```

5.6.3 Interrupts

NMI (Nonmaskable Interrupt):

- Source: NMI pin edge detection
- Transition address: VBR + H'00000600
- Transition operations:

  The PC and SR contents for the instruction immediately after this exception is accepted are saved in SPC and SSR. The R15 contents at this time are saved in SGR.

  Exception code H'1C0 is set in INTEVT. The BL, MD, and RB bits are set to 1 in SR, and a branch is made to PC = VBR + H'0600. When the BL bit in SR is 0, this interrupt is not masked by the interrupt mask bits in SR, and is accepted at the highest priority level. When the BL bit in SR is 1, a software setting can specify whether this interrupt is to be masked or accepted. For details, see the Interrupt Controller section of the hardware manual of the target product.

```
NMI()
{
    SPC = PC;
    SSR = SR;
    SGR = R15;
    INTEVT = H'0000 01C0;
    SR.MD = 1;
    SR.RB = 1;
    SR.BL = 1;
    PC = VBR + H'0000 0600;
}
```

General Interrupt Request:

- Source: The interrupt mask level bits setting in SR is smaller than the interrupt level of interrupt request, and the BL bit in SR is 0 (accepted at instruction boundary).
- Transition address: VBR + H'00000600
- Transition operations:

  The PC contents immediately after the instruction at which the interrupt is accepted are set in SPC. The SR and R15 contents at the time of acceptance are set in SSR and SGR.

  The code corresponding to the each interrupt source is set in INTEVT. The BL, MD, and RB bits are set to 1 in SR, and a branch is made to VBR + H'0600. For details, see the Interrupt Controller section of the hardware manual of the target product.

```
Module_interruption()
{
    SPC = PC;
    SSR = SR;
    SGR = R15;
    INTEVT = H'0000 0400 ~ H'0000 3FE0;
    SR.MD = 1;
    SR.RB = 1;
    SR.BL = 1;
    if (cond) SR.IMASK = level_of accepted_interrupt ();
    PC = VBR + H'0000 0600;
}
```

5.6.4 Priority Order with Multiple Exceptions

With some instructions, such as instructions that make two accesses to memory, and the indivisible pair comprising a delayed branch instruction and delay slot instruction, multiple exceptions occur. Care is required in these cases, as the exception priority order differs from the normal order.

- Instructions that make two accesses to memory

  With MAC instructions, memory-to-memory arithmetic/logic instructions, TAS instructions, and MOVUA instructions, two data transfers are performed by a single instruction, and an exception will be detected for each of these data transfers. In these cases, therefore, the following order is used to determine priority.

  1. Data address error in first data transfer
  2. TLB miss in first data transfer
  3. TLB protection violation in first data transfer
  4. Initial page write exception in first data transfer
  5. Data address error in second data transfer
  6. TLB miss in second data transfer
  7. TLB protection violation in second data transfer
  8. Initial page write exception in second data transfer

- Indivisible delayed branch instruction and delay slot instruction

  As a delayed branch instruction and its associated delay slot instruction are indivisible, they are treated as a single instruction. Consequently, the priority order for exceptions that occur in these instructions differs from the usual priority order. The priority order shown below is for the case where the delay slot instruction has only one data transfer.

  1. A check is performed for the interrupt type and re-execution type exceptions of priority levels 1 and 2 in the delayed branch instruction.
  2. A check is performed for the interrupt type and re-execution type exceptions of priority levels 1 and 2 in the delay slot instruction.
  3. A check is performed for the completion type exception of priority level 2 in the delayed branch instruction.
  4. A check is performed for the completion type exception of priority level 2 in the delay slot instruction.
  5. A check is performed for priority level 3 in the delayed branch instruction and priority level 3 in the delay slot instruction. (There is no priority ranking between these two.)
  6. A check is performed for priority level 4 in the delayed branch instruction and priority level 4 in the delay slot instruction. (There is no priority ranking between these two.)

  If the delay slot instruction has a second data transfer, two checks are performed in step 2, as in the above case (Instructions that make two accesses to memory).

  If the accepted exception (the highest-priority exception) is a delay slot instruction re-execution type exception, the branch instruction PR register write operation (PC $\rightarrow$ PR operation performed in a BSR, BSRF, or JSR instruction) is not disabled. Note that in this case, the contents of PR register are not guaranteed.

5.7 Usage Notes

1. Return from exception handling
   - A. Check the BL bit in SR with software. If SPC and SSR have been saved to memory, set the BL bit in SR to 1 before restoring them.
   - B. Issue an RTE instruction. When RTE is executed, the SPC contents are saved in PC, the SSR contents are saved in SR, and branch is made to the SPC address to return from the exception handling routine.
2. If an exception or interrupt occurs when BL bit in SR = 1
   - A. Exception

     When an exception other than a user break occurs, a manual reset is executed. The value in EXPEVT at this time is H'00000020; the SPC and SSR contents are undefined.
   - B. Interrupt

     If an ordinary interrupt occurs, the interrupt request is held pending and is accepted after the BL bit in SR has been cleared to 0 by software. If a nonmaskable interrupt (NMI) occurs, it can be held pending or accepted according to the setting made by software.

     In sleep or standby mode, however, an interrupt is accepted even if the BL bit in SR is set to 1.
3. SPC when an exception occurs
   - A. Re-execution type exception

     The PC value for the instruction at which the exception occurred is set in SPC, and the instruction is re-executed after returning from the exception handling routine. If an exception occurs in a delay slot instruction, however, the PC value for the delayed branch instruction is saved in SPC regardless of whether or not the preceding delay slot instruction condition is satisfied.
   - B. Completion type exception or interrupt

     The PC value for the instruction following that at which the exception occurred is set in SPC. If an exception occurs in a branch instruction with delay slot, however, the PC value for the branch destination is saved in SPC.
4. RTE instruction delay slot
   - A. The instruction in the delay slot of the RTE instruction is executed only after the value saved in SSR has been restored to SR. The acceptance of the exception related to the instruction access is determined depending on SR before restoring, while the acceptance of other exceptions is determined depending on the processing mode by SR after restoring or the BL bit. The completion type exception is accepted before branching to the destination of RTE instruction. However, if the re-execution type exception is occurred, the operation cannot be guaranteed.
   - B. The user break is not accepted by the instruction in the delay slot of the RTE instruction.

5. Changing the SR register value and accepting exception
    A. When the MD or BL bit in the SR register is changed by the LDC instruction, the acceptance of the exception is determined by the changed SR value, starting from the next instruction.* In the completion type exception, an exception is accepted after the next instruction has been executed. However, an interrupt of completion type exception is accepted before the next instruction is executed.

Note: * When the LDC instruction for SR is executed, following instructions are fetched again and the instruction fetch exception is evaluated again by the changed SR.

Section 6 Floating-Point Unit (FPU)

6.1 Features

The FPU has the following features.

- Conforms to IEEE754 standard
- 32 single-precision floating-point registers (can also be referenced as 16 double-precision registers)
- Two rounding modes: Round to Nearest and Round to Zero
- Two denormalization modes: Flush to Zero and Treat Denormalized Number
- Six exception sources: FPU Error, Invalid Operation, Divide By Zero, Overflow, Underflow, and Inexact
- Comprehensive instructions: Single-precision, double-precision, graphics support, and system control
- Following three instructions are added in the SH-4A
  FSRRA, FSCA, and FPCHG

When the FD bit in SR is set to 1, the FPU cannot be used, and an attempt to execute an FPU instruction will cause an FPU disable exception (general FPU disable exception or slot FPU disable exception).

6.2 Data Formats

6.2.1 Floating-Point Format

A floating-point number consists of the following three fields:

- Sign bit (s)
- Exponent field (e)
- Fraction field (f)

The SH-4A can handle single-precision and double-precision floating-point numbers, using the formats shown in figures 6.1 and 6.2.

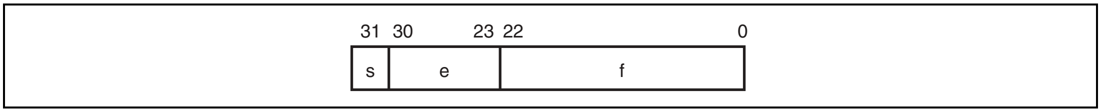


Figure 6.1 Format of Single-Precision Floating-Point Number

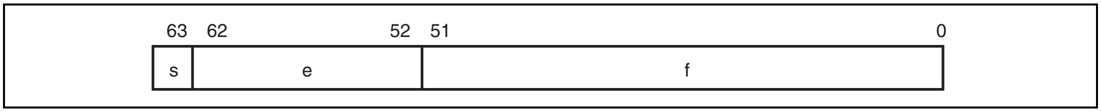


Figure 6.2 Format of Double-Precision Floating-Point Number

The exponent is expressed in biased form, as follows:

$$e = E + \text{bias}$$

The range of unbiased exponent E is $E_{min} - 1$ to $E_{max} + 1$. The two values $E_{min} - 1$ and $E_{max} + 1$ are distinguished as follows. $E_{min} - 1$ indicates zero (both positive and negative sign) and a denormalized number, and $E_{max} + 1$ indicates positive or negative infinity or a non-number (NaN). Table 6.1 shows floating-point formats and parameters.

Table 6.1 Floating-Point Number Formats and Parameters

Parameter	Single-Precision	Double-Precision
Total bit width	32 bits	64 bits
Sign bit	1 bit	1 bit
Exponent field	8 bits	11 bits
Fraction field	23 bits	52 bits
Precision	24 bits	53 bits
Bias	+127	+1023
E_{max}	+127	+1023
E_{min}	–126	–1022

Floating-point number value v is determined as follows:

If $E = E_{max} + 1$ and $f \neq 0$, v is a non-number (NaN) irrespective of sign s
If $E = E_{max} + 1$ and $f = 0$, $v = (-1)^s$ (infinity) [positive or negative infinity]
If $E_{min} \leq E \leq E_{max}$, $v = (-1)^s 2^E (1.f)$ [normalized number]
If $E = E_{min} - 1$ and $f \neq 0$, $v = (-1)^s 2^{Emin} (0.f)$ [denormalized number]
If $E = E_{min} - 1$ and $f = 0$, $v = (-1)^s 0$ [positive or negative zero]

Table 6.2 shows the ranges of the various numbers in hexadecimal notation. For the signaling non-number and quiet non-number, see section 6.2.2, Non-Numbers (NaN). For the denormalized number, see section 6.2.3, Denormalized Numbers.

Table 6.2 Floating-Point Ranges

Type	Single-Precision	Double-Precision
Signaling non-number	H'7FFF FFFF to H'7FC0 0000	H'7FFF FFFF FFFF FFFF to H'7FF8 0000 0000 0000
Quiet non-number	H'7FBF FFFF to H'7F80 0001	H'7FF7 FFFF FFFF FFFF to H'7FF0 0000 0000 0001
Positive infinity	H'7F80 0000	H'7FF0 0000 0000 0000
Positive normalized number	H'7F7F FFFF to H'0080 0000	H'7FEF FFFF FFFF FFFF to H'0010 0000 0000 0000
Positive denormalized number	H'007F FFFF to H'0000 0001	H'000F FFFF FFFF FFFF to H'0000 0000 0000 0001
Positive zero	H'0000 0000	H'0000 0000 0000 0000
Negative zero	H'8000 0000	H'8000 0000 0000 0000
Negative denormalized number	H'8000 0001 to H'807F FFFF	H'8000 0000 0000 0001 to H'800F FFFF FFFF FFFF
Negative normalized number	H'8080 0000 to H'FF7F FFFF	H'8010 0000 0000 0000 to H'FFEF FFFF FFFF FFFF
Negative infinity	H'FF80 0000	H'FFF0 0000 0000 0000
Quiet non-number	H'FF80 0001 to H'FFBF FFFF	H'FFF0 0000 0000 0001 to H'FFF7 FFFF FFFF FFFF
Signaling non-number	H'FFC0 0000 to H'FFFF FFFF	H'FFF8 0000 0000 0000 to H'FFFF FFFF FFFF FFFF

6.2.2 Non-Numbers (NaN)

Figure 6.3 shows the bit pattern of a non-number (NaN). A value is NaN in the following case:

- Sign bit: Don't care
- Exponent field: All bits are 1
- Fraction field: At least one bit is 1

The NaN is a signaling NaN (sNaN) if the MSB of the fraction field is 1, and a quiet NaN (qNaN) if the MSB is 0.

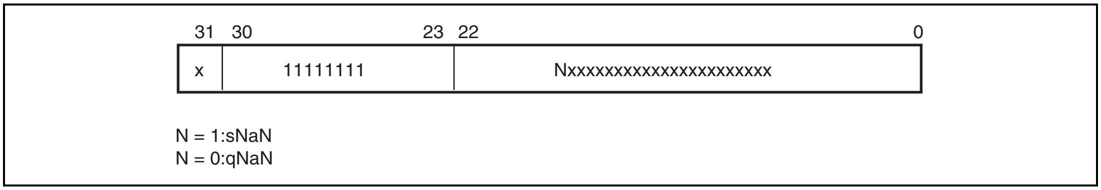


Figure 6.3 Single-Precision NaN Bit Pattern

An sNaN is assumed to be the input data in an operation, except the transfer instructions between registers, FABS, and FNEG, that generates a floating-point value.

- When the EN.V bit in FPSCR is 0, the operation result (output) is a qNaN.
- When the EN.V bit in FPSCR is 1, an invalid operation exception will be generated. In this case, the contents of the operation destination register are unchanged.

Following three instructions are used as transfer instructions between registers.

- FMOV FRm,FRn
- FLDS FRm,FPUL
- FSTS FPUL,FRn

If a qNaN is input in an operation that generates a floating-point value, and an sNaN has not been input in that operation, the output will always be a qNaN irrespective of the setting of the EN.V bit in FPSCR. An exception will not be generated in this case.

The qNAN values as operation results are as follows:

- Single-precision qNaN: H'7FBF FFFF
- Double-precision qNaN: H'7FF7 FFFF FFFF FFFF

See section 10, Instruction Descriptions for details of floating-point operations when a non-number (NaN) is input.

6.2.3 Denormalized Numbers

For a denormalized number floating-point value, the exponent field is expressed as 0, and the fraction field as a non-zero value.

When the DN bit in FPSCR of the FPU is 1, a denormalized number (source operand or operation result) is always positive or negative zero in a floating-point operation that generates a value (an operation other than transfer instructions between registers, FNEG, or FABS).

When the DN bit in FPSCR is 0, a denormalized number (source operand or operation result) is processed as it is. See section 10, Instruction Descriptions for details of floating-point operations when a denormalized number is input.

6.3 Register Descriptions

6.3.1 Floating-Point Registers

Figure 6.4 shows the floating-point register configuration. There are thirty-two 32-bit floating-point registers comprised with two banks: FPR0_BANK0 to FPR15_BANK0, and FPR0_BANK1 to FPR15_BANK1. These thirty-two registers are referenced as FR0 to FR15, DR0/2/4/6/8/10/12/14, FV0/4/8/12, XF0 to XF15, XD0/2/4/6/8/10/12/14, and XMTRX. Corresponding registers to FPR0_BANK0 to FPR15_BANK0, and FPR0_BANK1 to FPR15_BANK1 are determined according to the FR bit of FPSCR.

1. Floating-point registers, FPRi_BANKj (32 registers)

   FPR0_BANK0 to FPR15_BANK0

   FPR0_BANK1 to FPR15_BANK1

2. Single-precision floating-point registers, FRi (16 registers)

   When FPSCR.FR = 0, FR0 to FR15 are allocated to FPR0_BANK0 to FPR15_BANK0;

   when FPSCR.FR = 1, FR0 to FR15 are allocated to FPR0_BANK1 to FPR15_BANK1.

3. Double-precision floating-point registers, DRi (8 registers): A DR register comprises two FR registers.

   DR0 = {FR0, FR1}, DR2 = {FR2, FR3}, DR4 = {FR4, FR5}, DR6 = {FR6, FR7},
   DR8 = {FR8, FR9}, DR10 = {FR10, FR11}, DR12 = {FR12, FR13}, DR14 = {FR14, FR15}

4. Single-precision floating-point vector registers, FVi (4 registers): An FV register comprises four FR registers.

   FV0 = {FR0, FR1, FR2, FR3}, FV4 = {FR4, FR5, FR6, FR7},
   FV8 = {FR8, FR9, FR10, FR11}, FV12 = {FR12, FR13, FR14, FR15}

5. Single-precision floating-point extended registers, XFi (16 registers)

   When FPSCR.FR = 0, XF0 to XF15 are allocated to FPR0_BANK1 to FPR15_BANK1;

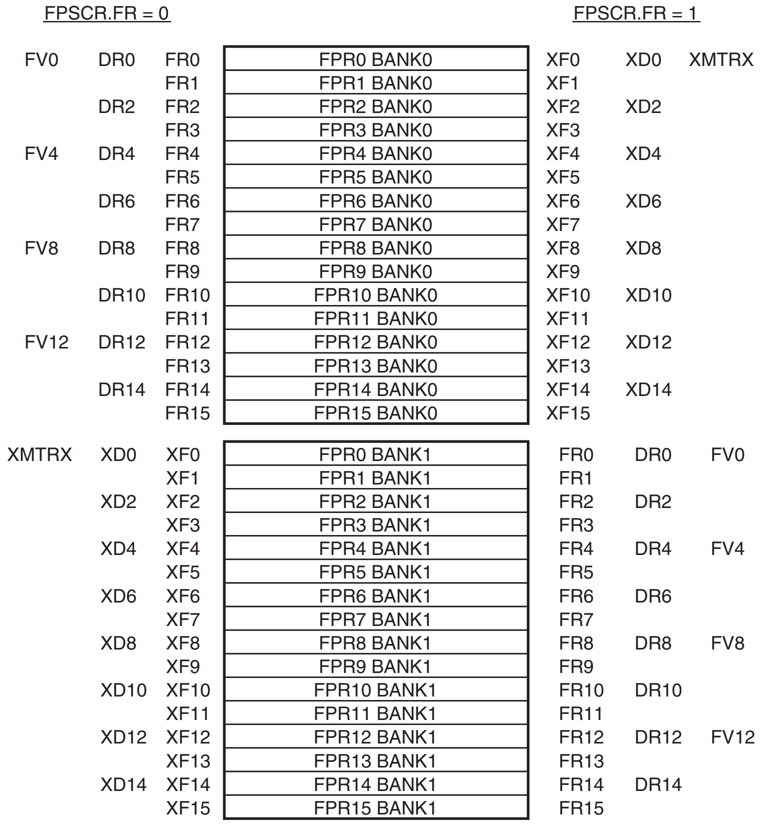
   when FPSCR.FR = 1, XF0 to XF15 are allocated to FPR0_BANK0 to FPR15_BANK0.

6. Double-precision floating-point extended registers, XDi (8 registers): An XD register comprises two XF registers.

   XD0 = {XF0, XF1}, XD2 = {XF2, XF3}, XD4 = {XF4, XF5}, XD6 = {XF6, XF7},
   XD8 = {XF8, XF9}, XD10 = {XF10, XF11}, XD12 = {XF12, XF13}, XD14 = {XF14, XF15}

7. Single-precision floating-point extended register matrix, XMTRX: XMTRX comprises all 16 XF registers.

$$\mathrm{XMTRX} = \begin{bmatrix} \mathrm{XF0} & \mathrm{XF4} & \mathrm{XF8} & \mathrm{XF12} \\ \mathrm{XF1} & \mathrm{XF5} & \mathrm{XF9} & \mathrm{XF13} \\ \mathrm{XF2} & \mathrm{XF6} & \mathrm{XF10} & \mathrm{XF14} \\ \mathrm{XF3} & \mathrm{XF7} & \mathrm{XF11} & \mathrm{XF15} \end{bmatrix}$$



FPSCR.FR = 0
FPSCR.FR = 1
FV0 DR0 FR0 FPR0 BANK0 XF0 XD0 XMTRX
FR1 FPR1 BANK0 XF1
DR2 FR2 FPR2 BANK0 XF2 XD2
FR3 FPR3 BANK0 XF3
FV4 DR4 FR4 FPR4 BANK0 XF4 XD4
FR5 FPR5 BANK0 XF5
DR6 FR6 FPR6 BANK0 XF6 XD6
FR7 FPR7 BANK0 XF7
FV8 DR8 FR8 FPR8 BANK0 XF8 XD8
FR9 FPR9 BANK0 XF9
DR10 FR10 FPR10 BANK0 XF10 XD10
FR11 FPR11 BANK0 XF11
FV12 DR12 FR12 FPR12 BANK0 XF12 XD12
FR13 FPR13 BANK0 XF13
DR14 FR14 FPR14 BANK0 XF14 XD14
FR15 FPR15 BANK0 XF15
XMTRX XD0 XF0 FPR0 BANK1 FR0 DR0 FV0
XF1 FPR1 BANK1 FR1
XD2 XF2 FPR2 BANK1 FR2 DR2
XF3 FPR3 BANK1 FR3
XD4 XF4 FPR4 BANK1 FR4 DR4 FV4
XF5 FPR5 BANK1 FR5
XD6 XF6 FPR6 BANK1 FR6 DR6
XF7 FPR7 BANK1 FR7
XD8 XF8 FPR8 BANK1 FR8 DR8 FV8
XF9 FPR9 BANK1 FR9
XD10 XF10 FPR10 BANK1 FR10 DR10
XF11 FPR11 BANK1 FR11
XD12 XF12 FPR12 BANK1 FR12 DR12 FV12
XF13 FPR13 BANK1 FR13
XD14 XF14 FPR14 BANK1 FR14 DR14
XF15 FPR15 BANK1 FR15

Figure 6.4 Floating-Point Registers

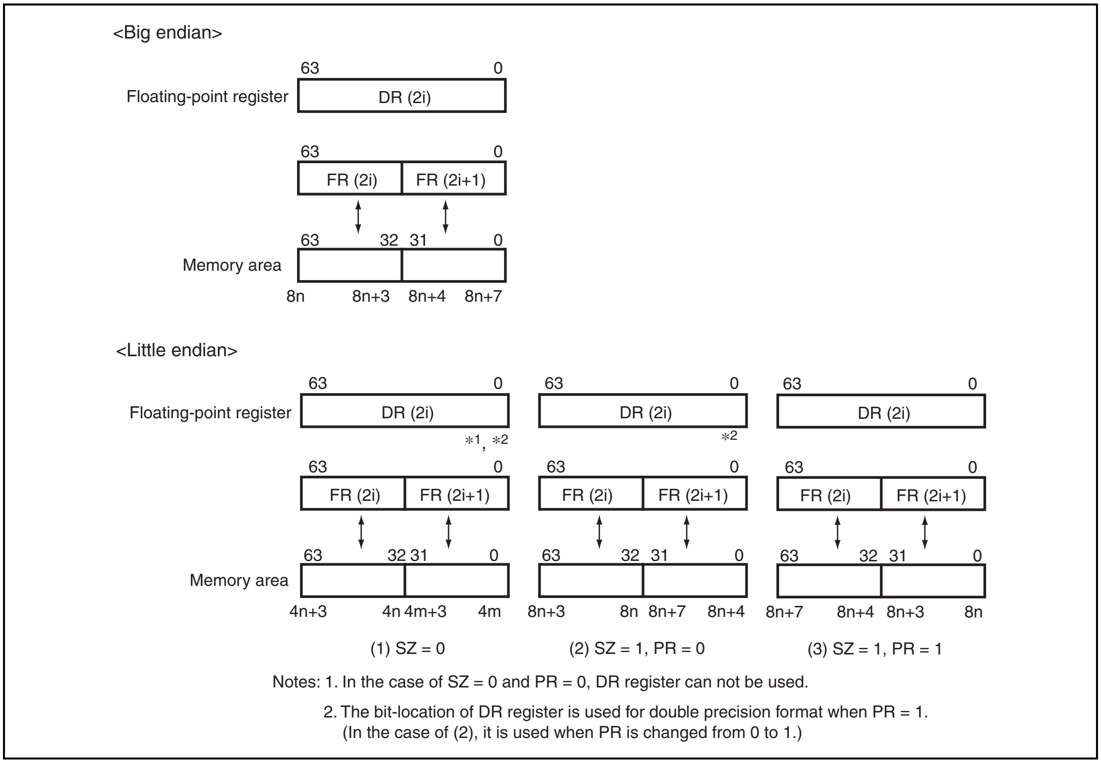
6.3.2 Floating-Point Status/Control Register (FPSCR)

bit:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	—	—	—	—	—	—	—	—	—	—	FR	SZ	PR	DN	Cause	
Initial value:	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
R/W:	R	R	R	R	R	R	R	R	R	R	R/W	R/W	R/W	R/W	R/W	R/W

bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	Cause				Enable (EN)					Flag					RM	
Initial value:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
R/W:	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

Bit	Bit Name	Initial Value	R/W	Description
31 to 22	—	All 0	R	Reserved These bits are always read as 0. The write value should always be 0.
21	FR	0	R/W	Floating-Point Register Bank 0: FPR0_BANK0 to FPR15_BANK0 are assigned to FR0 to FR15 and FPR0_BANK1 to FPR15_BANK1 are assigned to XF0 to XF15 1: FPR0_BANK0 to FPR15_BANK0 are assigned to XF0 to XF15 and FPR0_BANK1 to FPR15_BANK1 are assigned to FR0 to FR15
20	SZ	0	R/W	Transfer Size Mode 0: Data size of FMOV instruction is 32-bits 1: Data size of FMOV instruction is a 32-bit register pair (64 bits) For relations between endian and the SZ and PR bits, see figure 6.5.
19	PR	0	R/W	Precision Mode 0: Floating-point instructions are executed as single-precision operations 1: Floating-point instructions are executed as double-precision operations (graphics support instructions are undefined) For relations between endian and the SZ and PR bits, see figure 6.5.
18	DN	1	R/W	Denormalization Mode 0: Denormalized number is treated as such 1: Denormalized number is treated as zero

Bit	Bit Name	Initial Value	R/W	Description
17 to 12	Cause	All 0	R/W	FPU Exception Cause Field
11 to 7	Enable	All 0	R/W	FPU Exception Enable Field
6 to 2	Flag	All 0	R/W	FPU Exception Flag Field Each time an FPU operation instruction is executed, the FPU exception cause field is cleared to 0. When an FPU exception occurs, the bits corresponding to FPU exception cause field and flag field are set to 1. The FPU exception flag field remains set to 1 until it is cleared to 0 by software. For bit allocations of each field, see table 6.3.
1	RM1	0	R/W	Rounding Mode
0	RM0	1	R/W	These bits select the rounding mode. 00: Round to Nearest 01: Round to Zero 10: Reserved 11: Reserved



Notes: 1. In the case of SZ = 0 and PR = 0, DR register can not be used.
2. The bit-location of DR register is used for double precision format when PR = 1. (In the case of (2), it is used when PR is changed from 0 to 1.)

Figure 6.5 Relation between SZ Bit and Endian

Table 6.3 Bit Allocation for FPU Exception Handling

Field Name		FPU Error (E)	Invalid Operation (V)	Division by Zero (Z)	Overflow (O)	Underflow (U)	Inexact (I)
Cause	FPU exception cause field	Bit 17	Bit 16	Bit 15	Bit 14	Bit 13	Bit 12
Enable	FPU exception enable field	None	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7
Flag	FPU exception flag field	None	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2

6.3.3 Floating-Point Communication Register (FPUL)

Information is transferred between the FPU and CPU via FPUL. FPUL is a 32-bit system register that is accessed from the CPU side by means of LDS and STS instructions. For example, to convert the integer stored in general register R1 to a single-precision floating-point number, the processing flow is as follows:

R1 → (LDS instruction) → FPUL → (single-precision FLOAT instruction) → FR1

6.4 Rounding

In a floating-point instruction, rounding is performed when generating the final operation result from the intermediate result. Therefore, the result of combination instructions such as FMAC, FTRV, and FIPR will differ from the result when using a basic instruction such as FADD, FSUB, or FMUL. Rounding is performed once in FMAC, but twice in FADD, FSUB, and FMUL.

Which of the two rounding methods is to be used is determined by the RM bits in FPSCR.

FPSCR.RM[1:0] = 00: Round to Nearest
FPSCR.RM[1:0] = 01: Round to Zero

Round to Nearest: The operation result is rounded to the nearest expressible value. If there are two nearest expressible values, the one with an LSB of 0 is selected.

If the unrounded value is $2^{Emax} (2 - 2^{-P})$ or more, the result will be infinity with the same sign as the unrounded value. The values of Emax and P, respectively, are 127 and 24 for single-precision, and 1023 and 53 for double-precision.

Round to Zero: The digits below the round bit of the unrounded value are discarded.

If the unrounded value is larger than the maximum expressible absolute value, the value will become the maximum expressible absolute value with the same sign as unrounded value.

6.5 Floating-Point Exceptions

6.5.1 General FPU Disable Exceptions and Slot FPU Disable Exceptions

FPU-related exceptions are occurred when an FPU instruction is executed with SR.FD set to 1. When the FPU instruction is in other than delayed slot, the general FPU disable exception is occurred. When the FPU instruction is in the delay slot, the slot FPU disable exception is occurred.

6.5.2 FPU Exception Sources

The exception sources are as follows:

- FPU error (E): When FPSCR.DN = 0 and a denormalized number is input
- Invalid operation (V): In case of an invalid operation, such as NaN input
- Division by zero (Z): Division with a zero divisor
- Overflow (O): When the operation result overflows
- Underflow (U): When the operation result underflows
- Inexact exception (I): When overflow, underflow, or rounding occurs

The FPU exception cause field in FPSCR contains bits corresponding to all of above sources E, V, Z, O, U, and I, and the FPU exception flag and enable fields in FPSCR contain bits corresponding to sources V, Z, O, U, and I, but not E. Thus, FPU errors cannot be disabled.

When an FPU exception occurs, the corresponding bit in the FPU exception cause field is set to 1, and 1 is added to the corresponding bit in the FPU exception flag field. When an FPU exception does not occur, the corresponding bit in the FPU exception cause field is cleared to 0, but the corresponding bit in the FPU exception flag field remains unchanged.

6.5.3 FPU Exception Handling

FPU exception handling is initiated in the following cases:

- FPU error (E): FPSCR.DN = 0 and a denormalized number is input
- Invalid operation (V): FPSCR.Enable.V = 1 and (instruction = FTRV or invalid operation)
- Division by zero (Z): FPSCR.Enable.Z = 1 and division with a zero divisor or the input of FSRRA is zero
- Overflow (O): FPSCR.Enable.O = 1 and instruction with possibility of operation result overflow
- Underflow (U): FPSCR.Enable.U = 1 and instruction with possibility of operation result underflow
- Inexact exception (I): FPSCR.Enable.I = 1 and instruction with possibility of inexact operation result

All exception events that originate in the FPU are assigned as the same exception event. The meaning of an exception is determined by software by reading from FPSCR and interpreting the information it contains. Also, the destination register is not changed by any FPU exception handling operation.

If the FPU exception sources except for above are generated, the bit corresponding to source V, Z, O, U, or I is set to 1, and a default value is generated as the operation result.

- Invalid operation (V): qNaN is generated as the result.
- Division by zero (Z): Infinity with the same sign as the unrounded value is generated.
- Overflow (O):

  When rounding mode = RZ, the maximum normalized number, with the same sign as the unrounded value, is generated.
  When rounding mode = RN, infinity with the same sign as the unrounded value is generated.
- Underflow (U):
  When FPSCR.DN = 0, a denormalized number with the same sign as the unrounded value, or zero with the same sign as the unrounded value, is generated.
  When FPSCR.DN = 1, zero with the same sign as the unrounded value, is generated.
- Inexact exception (I): An inexact result is generated.

6.6 Graphics Support Functions

The SH-4A supports two kinds of graphics functions: new instructions for geometric operations, and pair single-precision transfer instructions that enable high-speed data transfer.

6.6.1 Geometric Operation Instructions

Geometric operation instructions perform approximate-value computations. To enable high-speed computation with a minimum of hardware, the SH-4A ignores comparatively small values in the partial computation results of four multiplications. Consequently, the error shown below is produced in the result of the computation:

$$\text{Maximum error} = \text{MAX}\,(\text{individual multiplication result} \times 2^{-\text{MIN (number of multiplier significant digits}-1\text{, number of multiplicand significant digits}-1)}) + \text{MAX}\,(\text{result value} \times 2^{-23}, 2^{-149})$$

The number of significant digits is 24 for a normalized number and 23 for a denormalized number (number of leading zeros in the fractional part).

In a future version of the SH Series, the above error is guaranteed, but the same result between different processor cores is not guaranteed.

FIPR FVm, FVn (m, n: 0, 4, 8, 12): This instruction is basically used for the following purposes:

- Inner product ($m \neq n$):
  This operation is generally used for surface/rear surface determination for polygon surfaces.
- Sum of square of elements ($m = n$):
  This operation is generally used to find the length of a vector.

Since an inexact exception is not detected by an FIPR instruction, the inexact exception (I) bit in both the FPU exception cause field and flag field are always set to 1 when an FIPR instruction is executed. Therefore, if the I bit is set in the FPU exception enable field, FPU exception handling will be executed.

FTRV XMTRX, FVn (n: 0, 4, 8, 12): This instruction is basically used for the following purposes:

- Matrix (4 × 4) · vector (4):
  This operation is generally used for viewpoint changes, angle changes, or movements called vector transformations (4-dimensional). Since affine transformation processing for angle + parallel movement basically requires a 4 × 4 matrix, the SH-4A supports 4-dimensional operations.
- Matrix (4 × 4) × matrix (4 × 4):
  This operation requires the execution of four FTRV instructions.

Since an inexact exception is not detected by an FIRV instruction, the inexact exception (I) bit in both the FPU exception cause field and flag field are always set to 1 when an FTRV instruction is executed. Therefore, if the I bit is set in the FPU exception enable field, FPU exception handling will be executed. It is not possible to check all data types in the registers beforehand when executing an FTRV instruction. If the V bit is set in the FPU exception enable field, FPU exception handling will be executed.

FRCHG: This instruction modifies banked registers. For example, when the FTRV instruction is executed, matrix elements must be set in an array in the background bank. However, to create the actual elements of a translation matrix, it is easier to use registers in the foreground bank. When the LDS instruction is used on FPSCR, this instruction takes four to five cycles in order to maintain the FPU state. With the FRCHG instruction, the FR bit in FPSCR can be changed in one cycle.

6.6.2 Pair Single-Precision Data Transfer

In addition to the powerful new geometric operation instructions, the SH-4A also supports high-speed data transfer instructions.

When the SZ bit is 1, the SH-4A can perform data transfer by means of pair single-precision data transfer instructions.

- FMOV DRm/XDm, DRn/XDRn (m, n: 0, 2, 4, 6, 8, 10, 12, 14)
- FMOV DRm/XDm, @Rn (m: 0, 2, 4, 6, 8, 10, 12, 14; n: 0 to 15)

These instructions enable two single-precision (2 × 32-bit) data items to be transferred; that is, the transfer performance of these instructions is doubled.

- FSCHG

This instruction changes the value of the SZ bit in FPSCR, enabling fast switching between use and non-use of pair single-precision data transfer.

Section 7 Memory Management Unit (MMU)

The SH-4A supports an 8-bit address space identifier, a 32-bit virtual address space, and a 29-bit physical address space. Address translation from virtual addresses to physical addresses is enabled by the memory management unit (MMU) in the SH-4A. The MMU performs high-speed address translation by caching user-created address translation table information in an address translation buffer (translation lookaside buffer: TLB).

The SH-4A has four instruction TLB (ITLB) entries and 64 unified TLB (UTLB) entries. UTLB copies are stored in the ITLB by hardware. A paging system is used for address translation, with four page sizes (1, 4, and 64 Kbytes, and 1 Mbyte) supported. It is possible to set the virtual address space access right and implement memory protection independently for privileged mode and user mode.

7.1 Overview of MMU

The MMU was conceived as a means of making efficient use of physical memory. As shown in (0) in figure 7.1, when a process is smaller in size than the physical memory, the entire process can be mapped onto physical memory, but if the process increases in size to the point where it does not fit into physical memory, it becomes necessary to divide the process into smaller parts, and map the parts requiring execution onto physical memory as occasion arises ((1) in figure 7.1). Having this mapping onto physical memory executed consciously by the process itself imposes a heavy burden on the process. The virtual memory system was devised as a means of handling all physical memory mapping to reduce this burden ((2) in figure 7.1). With a virtual memory system, the size of the available virtual memory is much larger than the actual physical memory, and processes are mapped onto this virtual memory. Thus processes only have to consider their operation in virtual memory, and mapping from virtual memory to physical memory is handled by the MMU. The MMU is normally managed by the OS, and physical memory switching is carried out so as to enable the virtual memory required by a process to be mapped smoothly onto physical memory. Physical memory switching is performed via secondary storage, etc.

The virtual memory system that came into being in this way works to best effect in a time sharing system (TSS) that allows a number of processes to run simultaneously ((3) in figure 7.1). Running a number of processes in a TSS did not increase efficiency since each process had to take account of physical memory mapping. Efficiency is improved and the load on each process reduced by the use of a virtual memory system ((4) in figure 7.1). In this virtual memory system, virtual memory is allocated to each process. The task of the MMU is to map a number of virtual memory areas onto physical memory in an efficient manner. It is also provided with memory protection functions to prevent a process from inadvertently accessing another process's physical memory.

When address translation from virtual memory to physical memory is performed using the MMU, it may happen that the translation information has not been recorded in the MMU, or the virtual memory of a different process is accessed by mistake. In such cases, the MMU will generate an exception, change the physical memory mapping, and record the new address translation information.

Although the functions of the MMU could be implemented by software alone, having address translation performed by software each time a process accessed physical memory would be very inefficient. For this reason, a buffer for address translation (the translation lookaside buffer: TLB) is provided by hardware, and frequently used address translation information is placed here. The TLB can be described as a cache for address translation information. However, unlike a cache, if address translation fails—that is, if an exception occurs—switching of the address translation information is normally performed by software. Thus memory management can be performed in a flexible manner by software.

There are two methods by which the MMU can perform mapping from virtual memory to physical memory: the paging method, using fixed-length address translation, and the segment method, using variable-length address translation. With the paging method, the unit of translation is a fixed-size address space called a page.

In the following descriptions, the address space in virtual memory in the SH-4A is referred to as virtual address space, and the address space in physical memory as physical address space.

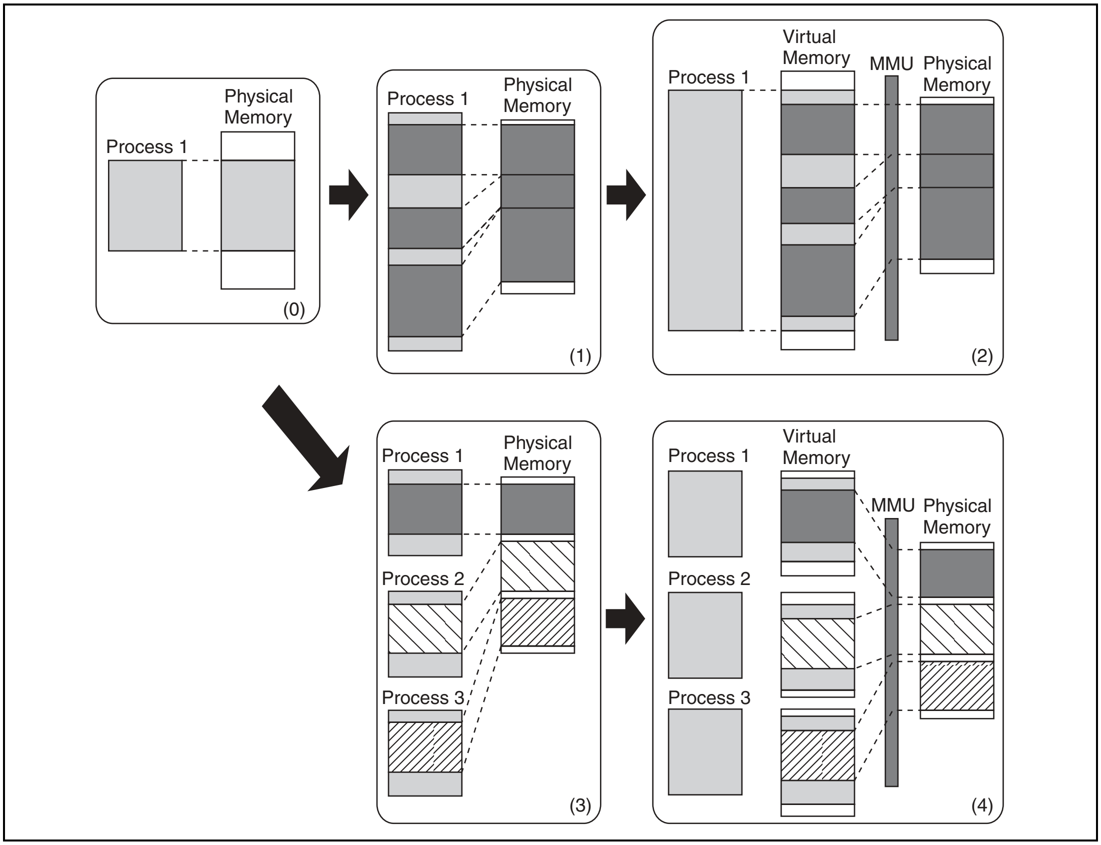


Figure 7.1 Role of MMU

7.1.1 Address Spaces

Virtual Address Space: The SH-4A supports a 32-bit virtual address space, and can access a 4-Gbyte address space. The virtual address space is divided into a number of areas, as shown in figures 7.2 and 7.3. In privileged mode, the 4-Gbyte space from the P0 area to the P4 area can be accessed. In user mode, a 2-Gbyte space in the U0 area can be accessed. When the SQMD bit in the MMU control register (MMUCR) is 0, a 64-Mbyte space in the store queue area can be accessed. When the RMD bit in the on-chip memory control register (RAMCR) is 1, a 16-Mbyte space in on-chip memory area can be accessed. Accessing areas other than the U0 area, store queue area, and on-chip memory area in user mode will cause an address error.

When the AT bit in MMUCR is set to 1 and the MMU is enabled, the P0, P3, and U0 areas can be mapped onto any physical address space in 1-, 4-, or 64-Kbyte, or 1-Mbyte page units. By using an 8-bit address space identifier, the P0, P3, and U0 areas can be increased to a maximum of 256. Mapping from the virtual address space to the 29-bit physical address space is carried out using the TLB.

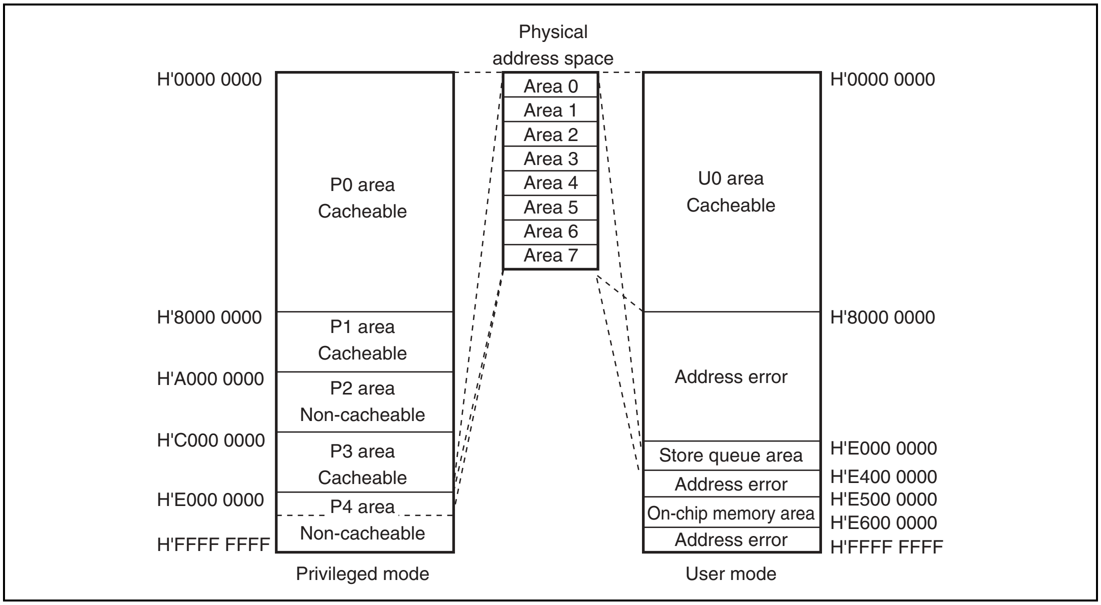


Figure 7.2 Virtual Address Space (AT in MMUCR= 0)

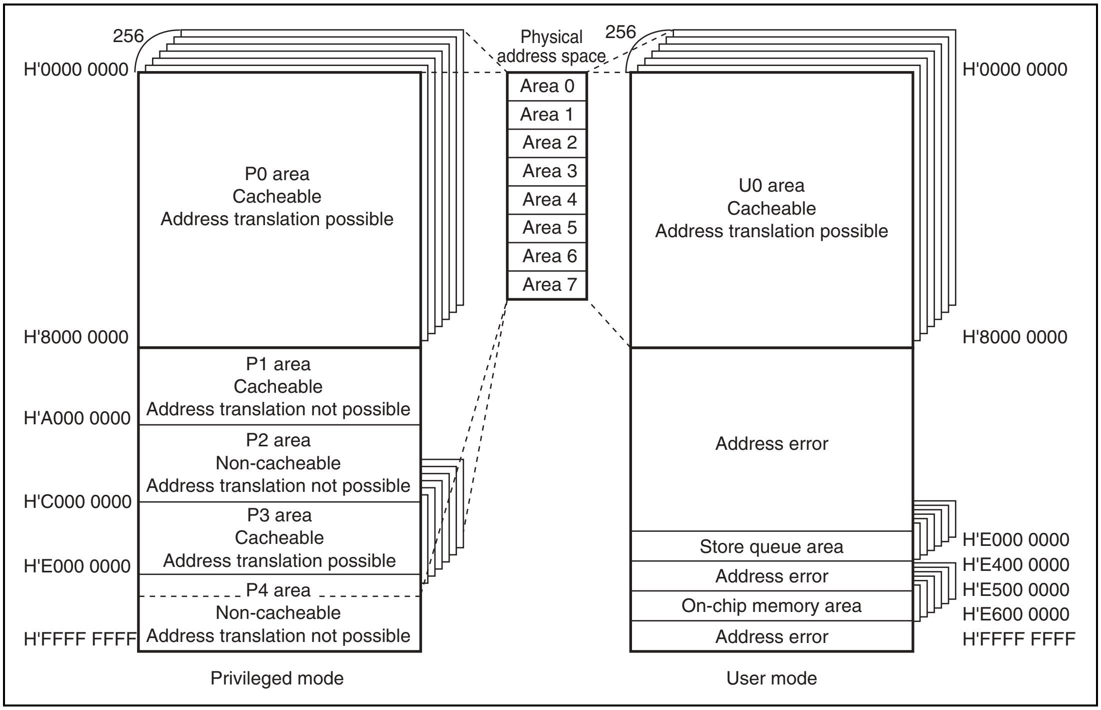


Figure 7.3 Virtual Address Space (AT in MMUCR= 1)

- P0, P3, and U0 Areas:

  The P0, P3, and U0 areas allow address translation using the TLB and access using the cache.

  When the MMU is disabled, replacing the upper 3 bits of an address with 0s gives the corresponding physical address. Whether or not the cache is used is determined by the CCR setting. When the cache is used, switching between the copy-back method and the write-through method for write accesses is specified by the WT bit in CCR.

  When the MMU is enabled, these areas can be mapped onto any physical address space in 1-, 4-, or 64-Kbyte, or 1-Mbyte page units using the TLB. When CCR is in the cache enabled state and the C bit for the corresponding page of the TLB entry is 1, accesses can be performed using the cache. When the cache is used, switching between the copy-back method and the write-through method for write accesses is specified by the WT bit of the TLB entry.

  When the P0, P3, and U0 areas are mapped onto the control register area which is allocated in the area 7 in physical address space by means of the TLB, the C bit for the corresponding page must be cleared to 0.

- P1 Area:

  The P1 area does not allow address translation using the TLB but can be accessed using the cache.

  Regardless of whether the MMU is enabled or disabled, clearing the upper 3 bits of an address to 0 gives the corresponding physical address. Whether or not the cache is used is determined by the CCR setting. When the cache is used, switching between the copy-back method and the write-through method for write accesses is specified by the CB bit in CCR.

- P2 Area:

  The P2 area does not allow address translation using the TLB and access using the cache.

  Regardless of whether the MMU is enabled or disabled, clearing the upper 3 bits of an address to 0 gives the corresponding physical address.

- P4 Area:

  The P4 area is mapped onto the internal resource of the SH-4A. This area except the store queue and on-chip memory areas does not allow address translation using the TLB. This area cannot be accessed using the cache. The P4 area is shown in detail in figure 7.4.

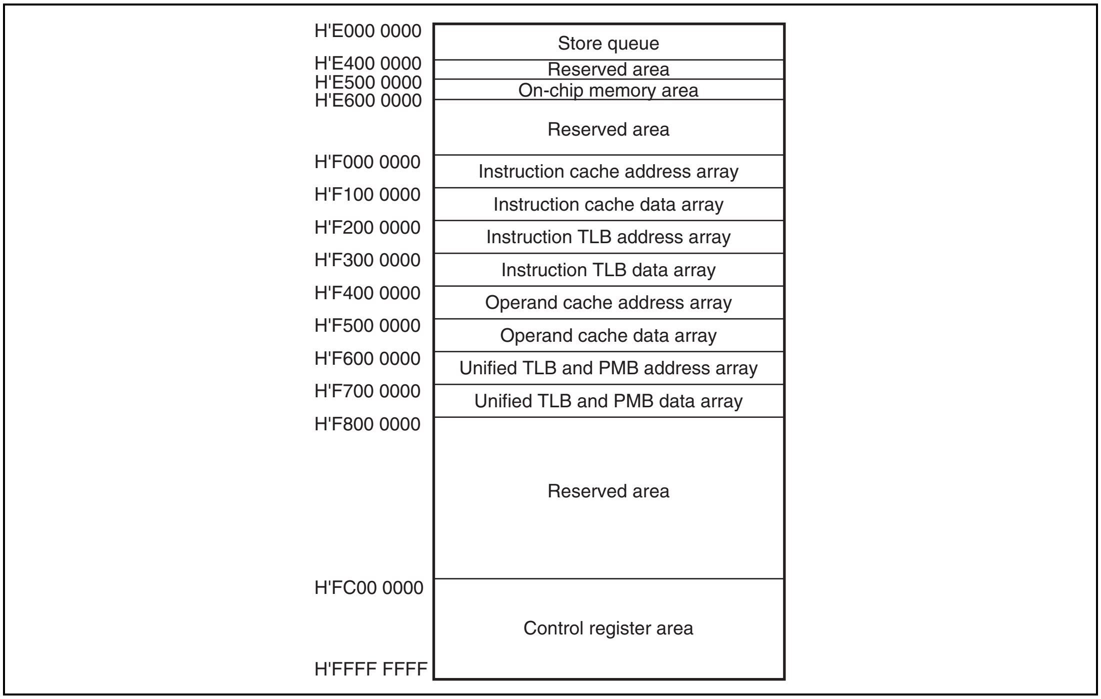


Figure 7.4   P4 Area

The area from H'E000 0000 to H'E3FF FFFF comprises addresses for accessing the store queues (SQs). In user mode, the access right is specified by the SQMD bit in MMUCR. For details, see section 8.7, Store Queues.

The area from H'E500 0000 to H'E5FF FFFF comprises addresses for accessing the on-chip memory. In user mode, the access right is specified by the RMD bit in RAMCR. For details, see section 9, L Memory.

The area from H'F000 0000 to H'F0FF FFFF is used for direct access to the instruction cache address array. For details, see section 8.6.1, IC Address Array.

The area from H'F100 0000 to H'F1FF FFFF is used for direct access to the instruction cache data array. For details, see section 8.6.2, IC Data Array.

The area from H'F200 0000 to H'F2FF FFFF is used for direct access to the instruction TLB address array. For details, see section 7.6.1, ITLB Address Array.

The area from H'F300 0000 to H'F37F FFFF is used for direct access to instruction TLB data array. For details, see section 7.6.2, ITLB Data Array.

The area from H'F400 0000 to H'F4FF FFFF is used for direct access to the operand cache address array. For details, see section 8.6.3, OC Address Array.

The area from H'F500 0000 to H'F5FF FFFF is used for direct access to the operand cache data array. For details, see section 8.6.4, OC Data Array.

The area from H'F600 0000 to H'F60F FFFF is used for direct access to the unified TLB address array. For details, see section 7.6.3, UTLB Address Array.

The area from H'F610 0000 to H'F61F FFFF is used for direct access to the PMB address array. For details, see section 7.7.5, Memory-Mapped PMB Configuration.

The area from H'F700 0000 to H'F70F FFFF is used for direct access to unified TLB data array. For details, see section 7.6.4, UTLB Data Array.

The area from H'F710 0000 to H'F71F FFFF is used for direct access to the PMB data array. For details, see section 7.7.5, Memory-Mapped PMB Configuration.

The area from H'FC00 0000 to H'FFFF FFFF is the on-chip peripheral module control register area. For details, see register descriptions in each section of the hardware manual of the target product.

Physical Address Space: The SH-4A supports a 29-bit physical address space. The physical address space is divided into eight areas as shown in figure 7.5. Area 7 is a reserved area. For details, see the Bus State Controller (BSC) section of the hardware manual of the target product.

Only when area 7 in the physical address space is accessed using the TLB, addresses H'1C00 0000 to H'1FFF FFFF of area 7 are not designated as a reserved area, but are equivalent to the control register area in the P4 area, in the virtual address space.

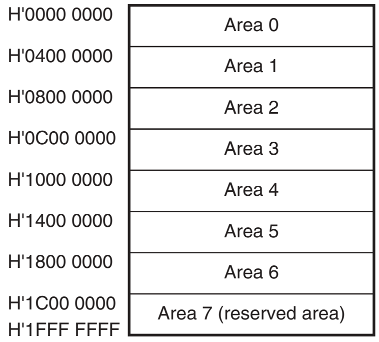


Figure 7.5 Physical Address Space

Address Translation: When the MMU is used, the virtual address space is divided into units called pages, and translation to physical addresses is carried out in these page units. The address translation table in external memory contains the physical addresses corresponding to virtual addresses and additional information such as memory protection codes. Fast address translation is achieved by caching the contents of the address translation table located in external memory into the TLB. In the SH-4A, basically, the ITLB is used for instruction accesses and the UTLB for data accesses. In the event of an access to an area other than the P4 area, the accessed virtual address is translated to a physical address. If the virtual address belongs to the P1 or P2 area, the physical address is uniquely determined without accessing the TLB. If the virtual address belongs to the P0, U0, or P3 area, the TLB is searched using the virtual address, and if the virtual address is recorded in the TLB, a TLB hit is made and the corresponding physical address is read from the TLB. If the accessed virtual address is not recorded in the TLB, a TLB miss exception is generated and processing switches to the TLB miss exception handling routine. In the TLB miss exception handling routine, the address translation table in external memory is searched, and the corresponding physical address and page management information are recorded in the TLB. After the return from the exception handling routine, the instruction which caused the TLB miss exception is re-executed.

Single Virtual Memory Mode and Multiple Virtual Memory Mode: There are two virtual memory systems, single virtual memory and multiple virtual memory, either of which can be selected with the SV bit in MMUCR. In the single virtual memory system, a number of processes run simultaneously, using virtual address space on an exclusive basis, and the physical address corresponding to a particular virtual address is uniquely determined. In the multiple virtual memory system, a number of processes run while sharing the virtual address space, and particular virtual addresses may be translated into different physical addresses depending on the process. The only difference between the single virtual memory and multiple virtual memory systems in terms of operation is in the TLB address comparison method (see section 7.3.3, Address Translation Method).

Address Space Identifier (ASID): In multiple virtual memory mode, an 8-bit address space identifier (ASID) is used to distinguish between multiple processes running simultaneously while sharing the virtual address space. Software can set the 8-bit ASID of the currently executing process in PTEH in the MMU. The TLB does not have to be purged when processes are switched by means of ASID.

In single virtual memory mode, ASID is used to provide memory protection for multiple processes running simultaneously while using the virtual address space on an exclusive basis.

Note: Two or more entries with the same virtual page number (VPN) but different ASID must not be set in the TLB simultaneously in single virtual memory mode.

7.2 Register Descriptions

The following registers are related to MMU processing.

Table 7.1 Register Configuration

Register Name	Abbreviation	R/W	P4 Address*	Area 7 Address*	Size
Page table entry high register	PTEH	R/W	H'FF00 0000	H'1F00 0000	32
Page table entry low register	PTEL	R/W	H'FF00 0004	H'1F00 0004	32
Translation table base register	TTB	R/W	H'FF00 0008	H'1F00 0008	32
TLB exception address register	TEA	R/W	H'FF00 000C	H'1F00 000C	32
MMU control register	MMUCR	R/W	H'FF00 0010	H'1F00 0010	32
Physical address space control register	PASCR	R/W	H'FF00 0070	H'1F00 0070	32
Instruction re-fetch inhibit control register	IRMCR	R/W	H'FF00 0078	H'1F00 0078	32

Note: * These P4 addresses are for the P4 area in the virtual address space. These area 7 addresses are accessed from area 7 in the physical address space by means of the TLB.

Table 7.2 Register States in Each Processing State

Register Name	Abbreviation	Power-on Reset	Manual Reset	Sleep	Standby
Page table entry high register	PTEH	Undefined	Undefined	Retained	Retained
Page table entry low register	PTEL	Undefined	Undefined	Retained	Retained
Translation table base register	TTB	Undefined	Undefined	Retained	Retained
TLB exception address register	TEA	Undefined	Retained	Retained	Retained
MMU control register	MMUCR	H'0000 0000	H'0000 0000	Retained	Retained
Physical address space control register	PASCR	H'0000 0000	H'0000 0000	Retained	Retained
Instruction re-fetch inhibit control register	IRMCR	H'0000 0000	H'0000 0000	Retained	Retained

7.2.1 Page Table Entry High Register (PTEH)

PTEH consists of the virtual page number (VPN) and address space identifier (ASID). When an MMU exception or address error exception occurs, the VPN of the virtual address at which the exception occurred is set in the VPN bit by hardware. VPN varies according to the page size, but the VPN set by hardware when an exception occurs consists of the upper 22 bits of the virtual address which caused the exception. VPN setting can also be carried out by software. The number of the currently executing process is set in the ASID bit by software. ASID is not updated by hardware. VPN and ASID are recorded in the UTLB by means of the LDTLB instruction.

After the ASID field in PTEH has been updated, execute one of the following three methods before an access (including an instruction fetch) to the P0, P3, or U0 area that uses the updated ASID value is performed.

1. Execute a branch using the RTE instruction. In this case, the branch destination may be the P0, P3, or U0 area.
2. Execute the ICBI instruction for any address (including non-cacheable area).
3. If the R2 bit in IRMCR is 0 (initial value) before updating the ASID field, the specific instruction does not need to be executed. However, note that the CPU processing performance will be lowered because the instruction fetch is performed again for the next instruction after the ASID field has been updated.

Note that the method 3 may not be guaranteed in the future SuperH Series. Therefore, it is recommended that the method 1 or 2 should be used for being compatible with the future SuperH Series.

Bit:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	VPN															
Initial value:	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
R/W:	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	VPN						—	—	ASID							
Initial value:	—	—	—	—	—	—	0	0	—	—	—	—	—	—	—	—
R/W:	R/W	R/W	R/W	R/W	R/W	R/W	R	R	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

Bit	Bit Name	Initial Value	R/W	Description
31 to 10	VPN	—	R/W	Virtual Page Number
9, 8	—	All 0	R	Reserved For details on reading from or writing to these bits, see description in General Precautions on Handling of Product.
7 to 0	ASID	—	R/W	Address Space Identifier

7.2.2 Page Table Entry Low Register (PTEL)

PTEL is used to hold the physical page number and page management information to be recorded in the UTLB by means of the LDTLB instruction. The contents of this register are not changed unless a software directive is issued.

Bit:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	—	—	—	PPN												
Initial value:	0	0	0	—	—	—	—	—	—	—	—	—	—	—	—	—
R/W:	R	R	R	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	PPN						—	V	SZ1	PR1	PR0	SZ0	C	D	SH	WT
Initial value:	—	—	—	—	—	—	0	—	—	—	—	—	—	—	—	—
R/W:	R/W	R/W	R/W	R/W	R/W	R/W	R	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

Bit	Bit Name	Initial Value	R/W	Description
31 to 29	—	All 0	R	Reserved For details on reading from or writing to these bits, see description in General Precautions on Handling of Product.
28 to 10	PPN	—	R/W	Physical Page Number
9	—	0	R	Reserved For details on reading from or writing to this bit, see description in General Precautions on Handling of Product.
8	V	—	R/W	Page Management Information The meaning of each bit is same as that of corresponding bit in Common TLB (UTLB). For details, see section 7.3, TLB Functions.
7	SZ1	—	R/W	
6	PR1	—	R/W	
5	PR0	—	R/W	
4	SZ0	—	R/W	
3	C	—	R/W	
2	D	—	R/W	
1	SH	—	R/W	
0	WT	—	R/W	

7.2.3 Translation Table Base Register (TTB)

TTB is used to store the base address of the currently used page table, and so on. The contents of TTB are not changed unless a software directive is issued. This register can be used freely by software.

Bit:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	TTB															
Initial value:	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
R/W:	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	TTB															
Initial value:	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
R/W:	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

7.2.4 TLB Exception Address Register (TEA)

After an MMU exception or address error exception occurs, the virtual address at which the exception occurred is stored. The contents of this register can be changed by software.

Bit:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
Initial value:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
R/W:	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R

Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	—	—	—	—	—	—	—	—	—	—	—	—	TC	SA2	SA1	SA0
Initial value:	0	0	0	0	0	0	0	0	0	0	0	0	—	—	—	—
R/W:	R	R	R	R	R	R	R	R	R	R	R	R	R/W	R/W	R/W	R/W

7.2.5 MMU Control Register (MMUCR)

The individual bits perform MMU settings as shown below. Therefore, MMUCR rewriting should be performed by a program in the P1 or P2 area.

After MMUCR has been updated, execute one of the following three methods before an access (including an instruction fetch) to the P0, P3, U0, or store queue area is performed.

1. Execute a branch using the RTE instruction. In this case, the branch destination may be the P0, P3, or U0 area.
2. Execute the ICBI instruction for any address (including non-cacheable area).
3. If the R2 bit in IRMCR is 0 (initial value) before updating MMUCR, the specific instruction does not need to be executed. However, note that the CPU processing performance will be lowered because the instruction fetch is performed again for the next instruction after MMUCR has been updated.

Note that the method 3 may not be guaranteed in the future SuperH Series. Therefore, it is recommended that the method 1 or 2 should be used for being compatible with the future SuperH Series.

MMUCR contents can be changed by software. However, the LRUI and URC bits may also be updated by hardware.

Bit:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	LRUI						—	—	URB						—	—
Initial value:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
R/W:	R/W	R/W	R/W	R/W	R/W	R/W	R	R	R/W	R/W	R/W	R/W	R/W	R/W	R	R

Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	URC						SQMD	SV	—	—	—	—	—	TI	—	AT
Initial value:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
R/W:	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R	R	R	R	R	R/W	R	R/W

Bit	Bit Name	Initial Value	R/W	Description
31 to 26	LRUI	All 0	R/W	Least Recently Used ITLB These bits indicate the ITLB entry to be replaced. The LRU (least recently used) method is used to decide the ITLB entry to be replaced in the event of an ITLB miss. The entry to be purged from the ITLB can be confirmed using the LRUI bits. LRUI is updated by means of the algorithm shown below. x means that updating is not performed. 000xxx: ITLB entry 0 is used 1xx00x: ITLB entry 1 is used x1x1x0: ITLB entry 2 is used xx1x11: ITLB entry 3 is used xxxxxx: Other than above When the LRUI bit settings are as shown below, the corresponding ITLB entry is updated by an ITLB miss. Ensure that values for which "Setting prohibited" is indicated below are not set at the discretion of software. After a power-on or manual reset, the LRUI bits are initialized to 0, and therefore a prohibited setting is never made by a hardware update. x means "don't care". 111xxx: ITLB entry 0 is updated 0xx11x: ITLB entry 1 is updated x0x0x1: ITLB entry 2 is updated xx0x00: ITLB entry 3 is updated Other than above: Setting prohibited
25, 24	—	All 0	R	Reserved For details on reading from or writing to these bits, see description in General Precautions on Handling of Product.
23 to 18	URB	All 0	R/W	UTLB Replace Boundary These bits indicate the UTLB entry boundary at which replacement is to be performed. Valid only when URB $\neq$ 0.
17, 16	—	All 0	R	Reserved For details on reading from or writing to these bits, see description in General Precautions on Handling of Product.

Bit	Bit Name	Initial Value	R/W	Description
15 to 10	URC	All 0	R/W	UTLB Replace Counter These bits serve as a random counter for indicating the UTLB entry for which replacement is to be performed with an LDTLB instruction. This bit is incremented each time the UTLB is accessed. If URB > 0, URC is cleared to 0 when the condition URC = URB is satisfied. Also note that if a value is written to URC by software which results in the condition of URC ≥ URB, incrementing is first performed in excess of URB until URC = H'3F. URC is not incremented by an LDTLB instruction.
9	SQMD	0	R/W	Store Queue Mode Bit Specifies the right of access to the store queues. 0: User/privileged access possible 1: Privileged access possible (address error exception in case of user access)
8	SV	0	R/W	Single Virtual Memory Mode/Multiple Virtual Memory Mode Switching Bit When this bit is changed, ensure that 1 is also written to the TI bit. 0: Multiple virtual memory mode 1: Single virtual memory mode
7 to 3	—	All 0	R	Reserved For details on reading from or writing to these bits, see description in General Precautions on Handling of Product.
2	TI	0	R/W	TLB Invalidate Bit Writing 1 to this bit invalidates (clears to 0) all valid UTLB/ITLB bits. This bit is always read as 0.
1	—	0	R	Reserved For details on reading from or writing to this bit, see description in General Precautions on Handling of Product.
0	AT	0	R/W	Address Translation Enable Bit These bits enable or disable the MMU. 0: MMU disabled 1: MMU enabled MMU exceptions are not generated when the AT bit is 0. In the case of software that does not use the MMU, the AT bit should be cleared to 0.

7.2.6 Physical Address Space Control Register (PASCR)

PASCR controls the operation in the physical address space.

Bit:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
Initial value:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
R/W:	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R

Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	—	—	—	—	—	—	—	—	UB							
Initial value:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
R/W:	R	R	R	R	R	R	R	R	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

Bit	Bit Name	Initial Value	R/W	Description
31 to 8	—	All 0	R	Reserved For details on reading from or writing to these bits, see description in General Precautions on Handling of Product.
7 to 0	UB	All 0	R/W	Buffered Write Control for Each Area (64 Mbytes) When writing is performed without using the cache or in the cache write-through mode, these bits specify whether the next bus access from the CPU waits for the end of writing for each area. 0 : The CPU does not wait for the end of writing bus access and starts the next bus access 1 : The CPU waits for the end of writing bus access and starts the next bus access UB[7]: Corresponding to the control register area UB[6]: Corresponding to area 6 UB[5]: Corresponding to area 5 UB[4]: Corresponding to area 4 UB[3]: Corresponding to area 3 UB[2]: Corresponding to area 2 UB[1]: Corresponding to area 1 UB[0]: Corresponding to area 0

7.2.7 Instruction Re-Fetch Inhibit Control Register (IRMCR)

When the specific resource is changed, IRMCR controls whether the instruction fetch is performed again for the next instruction. The specific resource means the part of control registers, TLB, and cache.

In the initial state, the instruction fetch is performed again for the next instruction after changing the resource. However, the CPU processing performance will be lowered because the instruction fetch is performed again for the next instruction every time the resource is changed. Therefore, it is recommended that each bit in IRMCR is set to 1 and the specific instruction should be executed after all necessary resources have been changed prior to execution of the program which uses changed resources.

For details on the specific sequence, see descriptions in each resource.

Bit:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
Initial value:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
R/W:	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R

Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	—	—	—	—	—	—	—	—	—	—	—	R2	R1	LT	MT	MC
Initial value:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
R/W:	R	R	R	R	R	R	R	R	R	R	R	R/W	R/W	R/W	R/W	R/W

Bit	Bit Name	Initial Value	R/W	Description
31 to 5	—	All 0	R	Reserved For details on reading from or writing to these bits, see description in General Precautions on Handling of Product.
4	R2	0	R/W	Re-Fetch Inhibit 2 after Register Change When MMUCR, PASCR, CCR, PTEH, or RAMCR is changed, this bit controls whether re-fetch is performed for the next instruction. 0: Re-fetch is performed 1: Re-fetch is not performed
3	R1	0	R/W	Re-Fetch Inhibit 1 after Register Change When a register allocated in addresses H'FF200000 to H'FF2FFFFF is changed, this bit controls whether re-fetch is performed for the next instruction. 0: Re-fetch is performed 1: Re-fetch is not performed

Bit	Bit Name	Initial Value	R/W	Description
2	LT	0	R/W	Re-Fetch Inhibit after LDTLB Execution This bit controls whether re-fetch is performed for the next instruction after the LDTLB instruction has been executed. 0: Re-fetch is performed 1: Re-fetch is not performed
1	MT	0	R/W	Re-Fetch Inhibit after Writing Memory-Mapped TLB This bit controls whether re-fetch is performed for the next instruction after writing memory-mapped ITLB/UTLB while the AT bit in MMUCR is set to 1. 0: Re-fetch is performed 1: Re-fetch is not performed
0	MC	0	R/W	Re-Fetch Inhibit after Writing Memory-Mapped IC This bit controls whether re-fetch is performed for the next instruction after writing memory-mapped IC while the ICE bit in CCR is set to 1. 0: Re-fetch is performed 1: Re-fetch is not performed

7.3 TLB Functions

7.3.1 Unified TLB (UTLB) Configuration

The UTLB is used for the following two purposes:

1. To translate a virtual address to a physical address in a data access
2. As a table of address translation information to be recorded in the ITLB in the event of an ITLB miss

The UTLB is so called because of its use for the above two purposes. Information in the address translation table located in external memory is cached into the UTLB. The address translation table contains virtual page numbers and address space identifiers, and corresponding physical page numbers and page management information. Figure 7.6 shows the UTLB configuration. The UTLB consists of 64 fully-associative type entries. Figure 7.7 shows the relationship between the page size and address format.

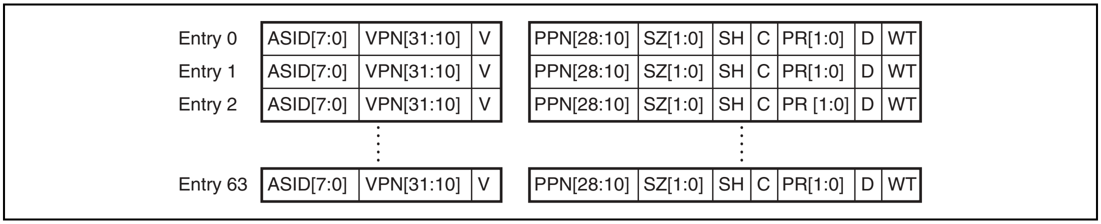


Figure 7.6 UTLB Configuration

[Legend]

- VPN: Virtual page number

  For 1-Kbyte page: Upper 22 bits of virtual address

  For 4-Kbyte page: Upper 20 bits of virtual address

  For 64-Kbyte page: Upper 16 bits of virtual address

  For 1-Mbyte page: Upper 12 bits of virtual address

- ASID: Address space identifier

  Indicates the process that can access a virtual page.

  In single virtual memory mode and user mode, or in multiple virtual memory mode, if the SH bit is 0, this identifier is compared with the ASID in PTEH when address comparison is performed.

- SH: Share status bit

  When 0, pages are not shared by processes.

  When 1, pages are shared by processes.

- SZ[1:0]: Page size bits

  Specify the page size.

  00: 1-Kbyte page

  01: 4-Kbyte page

  10: 64-Kbyte page

  11: 1-Mbyte page

- V: Validity bit

  Indicates whether the entry is valid.

  0: Invalid

  1: Valid

  Cleared to 0 by a power-on reset.

  Not affected by a manual reset.

- PPN: Physical page number

  Upper 22 bits of the physical address of the physical page number.

  With a 1-Kbyte page, PPN[28:10] are valid.

  With a 4-Kbyte page, PPN[28:12] are valid.

  With a 64-Kbyte page, PPN[28:16] are valid.

  With a 1-Mbyte page, PPN[28:20] are valid.

  The synonym problem must be taken into account when setting the PPN (see section 7.4.5, Avoiding Synonym Problems).

- PR[1:0]: Protection key data

  2-bit data expressing the page access right as a code.

  00: Can be read from only in privileged mode

  01: Can be read from and written to in privileged mode

  10: Can be read from only in privileged or user mode

  11: Can be read from and written to in privileged mode or user mode

- C: Cacheability bit

  Indicates whether a page is cacheable.

  0: Not cacheable

  1: Cacheable

  When the control register area is mapped, this bit must be cleared to 0.

- D: Dirty bit
  Indicates whether a write has been performed to a page.
  0: Write has not been performed
  1: Write has been performed

- WT: Write-through bit
  Specifies the cache write mode.
  0: Copy-back mode
  1: Write-through mode

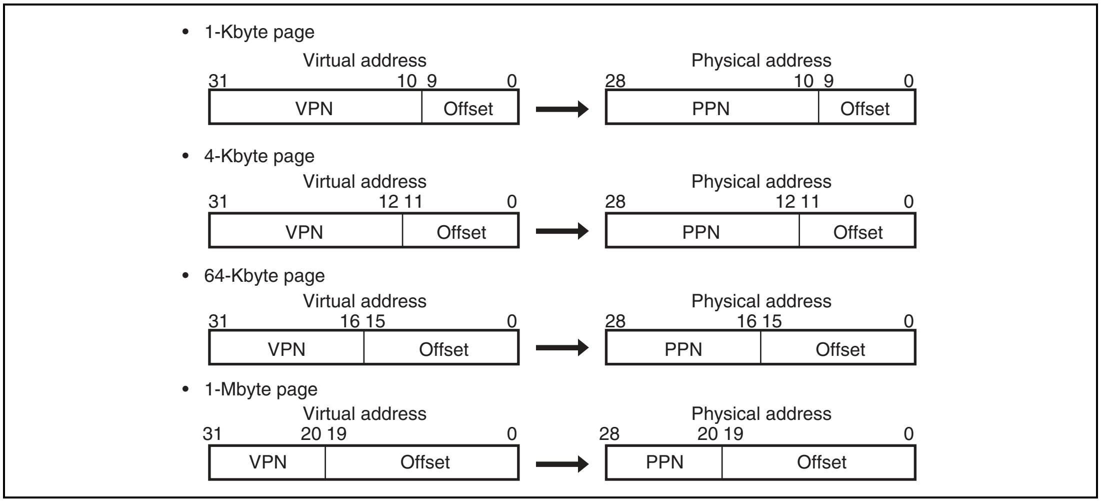


Figure 7.7 Relationship between Page Size and Address Format

7.3.2 Instruction TLB (ITLB) Configuration

The ITLB is used to translate a virtual address to a physical address in an instruction access. Information in the address translation table located in the UTLB is cached into the ITLB. Figure 7.8 shows the ITLB configuration. The ITLB consists of four fully-associative type entries.

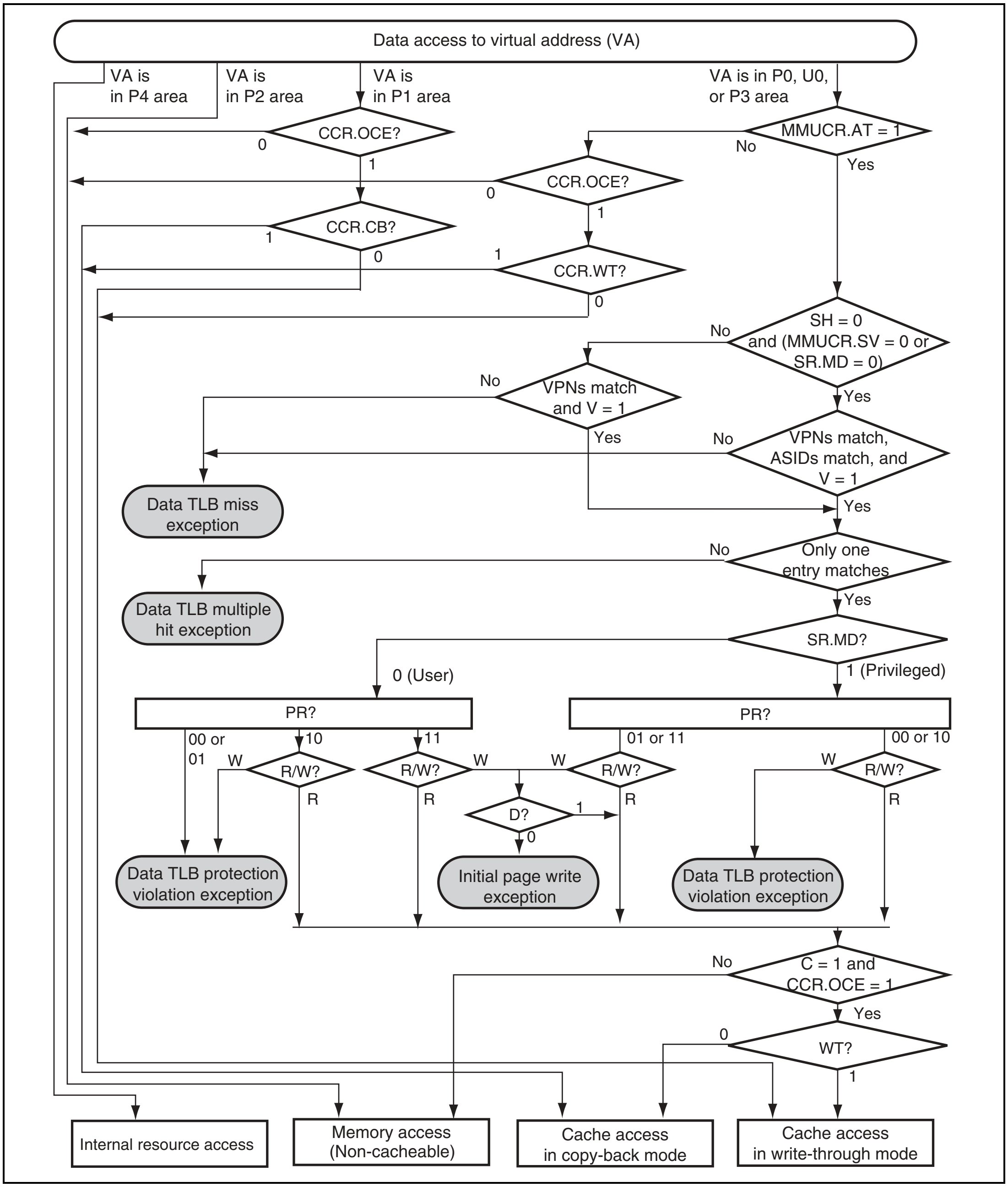
Entry 0	ASID[7:0]	VPN[31:10]	V	PPN[28:10]	SZ[1:0]	SH	C	PR
Entry 1	ASID[7:0]	VPN[31:10]	V	PPN[28:10]	SZ[1:0]	SH	C	PR
Entry 2	ASID[7:0]	VPN[31:10]	V	PPN[28:10]	SZ[1:0]	SH	C	PR
Entry 3	ASID[7:0]	VPN[31:10]	V	PPN[28:10]	SZ[1:0]	SH	C	PR

Notes: 1. The D and WT bits are not supported.
2. There is only one PR bit, corresponding to the upper bit of the PR bits in the UTLB.

Figure 7.8 ITLB Configuration

7.3.3 Address Translation Method

Figure 7.9 shows a flowchart of a memory access using the UTLB.



Data access to virtual address (VA)
VA is in P4 area
VA is in P2 area
VA is in P1 area
VA is in P0, U0, or P3 area
CCR.OCE?
0
1
MMUCR.AT = 1
No
Yes
CCR.OCE?
0
1
CCR.CB?
1
0
CCR.WT?
1
0
SH = 0 and (MMUCR.SV = 0 or SR.MD = 0)
No
Yes
VPNs match and V = 1
No
Yes
VPNs match, ASIDs match, and V = 1
No
Yes
Data TLB miss exception
Only one entry matches
No
Yes
Data TLB multiple hit exception
SR.MD?
0 (User)
1 (Privileged)
PR?
00 or 01
10
11
PR?
01 or 11
00 or 10
R/W?
W
R
D?
1
0
Data TLB protection violation exception
Initial page write exception
Data TLB protection violation exception
C = 1 and CCR.OCE = 1
No
Yes
WT?
0
1
Internal resource access
Memory access (Non-cacheable)
Cache access in copy-back mode
Cache access in write-through mode

Figure 7.9 Flowchart of Memory Access Using UTLB

Figure 7.10 shows a flowchart of a memory access using the ITLB.

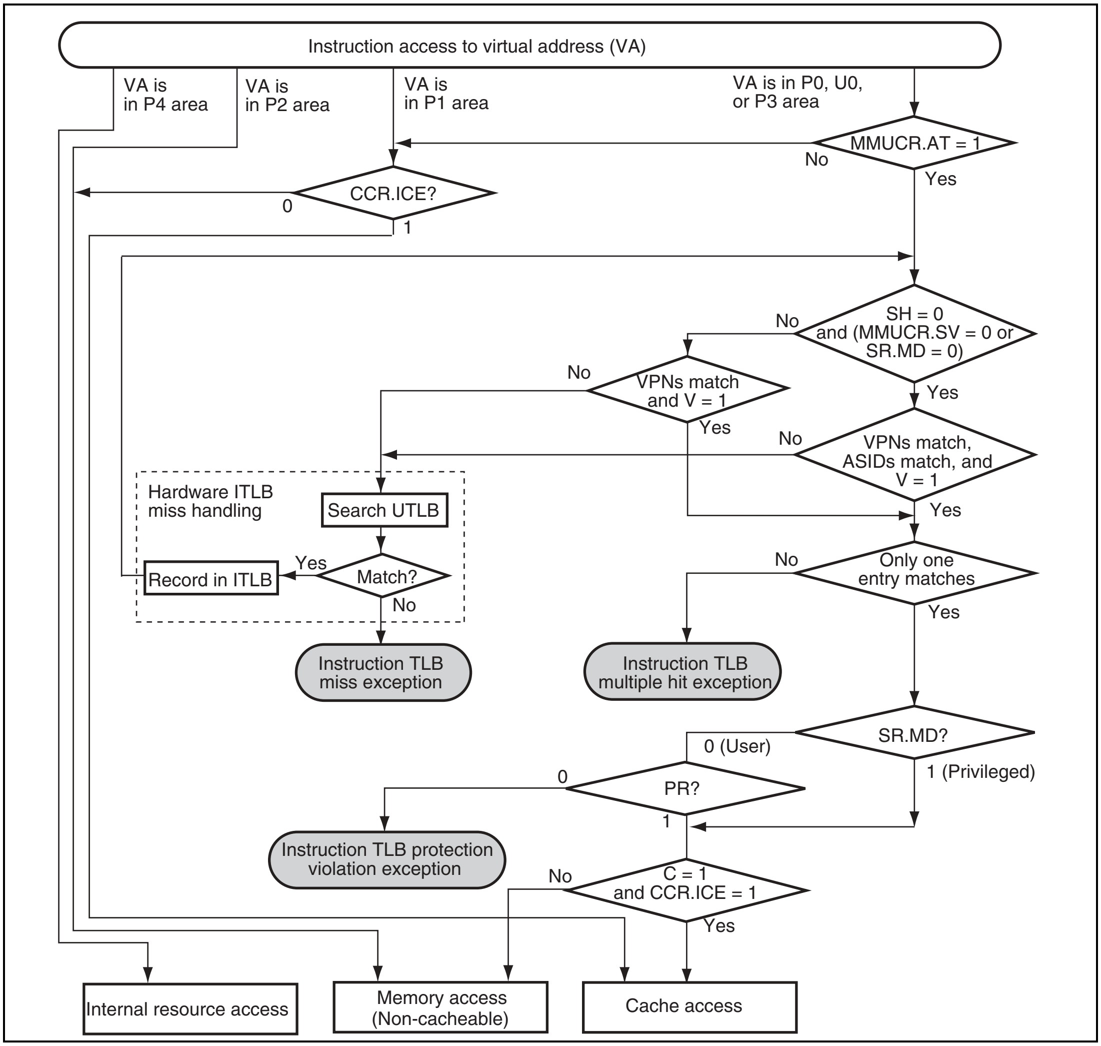

Instruction access to virtual address (VA)
VA is in P4 area
VA is in P2 area
VA is in P1 area
VA is in P0, U0, or P3 area
MMUCR.AT = 1
No
Yes
CCR.ICE?
0
1
SH = 0 and (MMUCR.SV = 0 or SR.MD = 0)
No
Yes
VPNs match and V = 1
No
Yes
VPNs match, ASIDs match, and V = 1
No
Yes
Hardware ITLB miss handling
Search UTLB
Record in ITLB
Yes
Match?
No
Only one entry matches
No
Yes
Instruction TLB miss exception
Instruction TLB multiple hit exception
SR.MD?
0 (User)
1 (Privileged)
PR?
0
1
Instruction TLB protection violation exception
C = 1 and CCR.ICE = 1
No
Yes
Internal resource access
Memory access (Non-cacheable)
Cache access

Figure 7.10 Flowchart of Memory Access Using ITLB

7.4 MMU Functions

7.4.1 MMU Hardware Management

The SH-4A supports the following MMU functions.

1. The MMU decodes the virtual address to be accessed by software, and performs address translation by controlling the UTLB/ITLB in accordance with the MMUCR settings.
2. The MMU determines the cache access status on the basis of the page management information read during address translation (C and WT bits).
3. If address translation cannot be performed normally in a data access or instruction access, the MMU notifies software by means of an MMU exception.
4. If address translation information is not recorded in the ITLB in an instruction access, the MMU searches the UTLB. If the necessary address translation information is recorded in the UTLB, the MMU copies this information into the ITLB in accordance with the LRUI bit setting in MMUCR.

7.4.2 MMU Software Management

Software processing for the MMU consists of the following:

1. Setting of MMU-related registers. Some registers are also partially updated by hardware automatically.
2. Recording, deletion, and reading of TLB entries. There are two methods of recording UTLB entries: by using the LDTLB instruction, or by writing directly to the memory-mapped UTLB. ITLB entries can only be recorded by writing directly to the memory-mapped ITLB. Deleting or reading UTLB/ITLB entries is enabled by accessing the memory-mapped UTLB/ITLB.
3. MMU exception handling. When an MMU exception occurs, processing is performed based on information set by hardware.

7.4.3 MMU Instruction (LDTLB)

A TLB load instruction (LDTLB) is provided for recording UTLB entries. When an LDTLB instruction is issued, the SH-4A copies the contents of PTEH and PTEL to the UTLB entry indicated by the URC bit in MMUCR. ITLB entries are not updated by the LDTLB instruction, and therefore address translation information purged from the UTLB entry may still remain in the ITLB entry. As the LDTLB instruction changes address translation information, ensure that it is issued by a program in the P1 or P2 area.

After the LDTLB instruction has been executed, execute one of the following three methods before an access (include an instruction fetch) the area where TLB is used to translate the address is performed.

1. Execute a branch using the RTE instruction. In this case, the branch destination may be the area where TLB is used to translate the address.
2. Execute the ICBI instruction for any address (including non-cacheable area).
3. If the LT bit in IRMCR is 0 (initial value) before executing the LDTLB instruction, the specific instruction does not need to be executed. However, note that the CPU processing performance will be lowered because the instruction fetch is performed again for the next instruction after MMUCR has been updated.

Note that the method 3 may not be guaranteed in the future SuperH Series. Therefore, it is recommended that the method 1 or 2 should be used for being compatible with the future SuperH Series.

The operation of the LDTLB instruction is shown in figure 7.11.

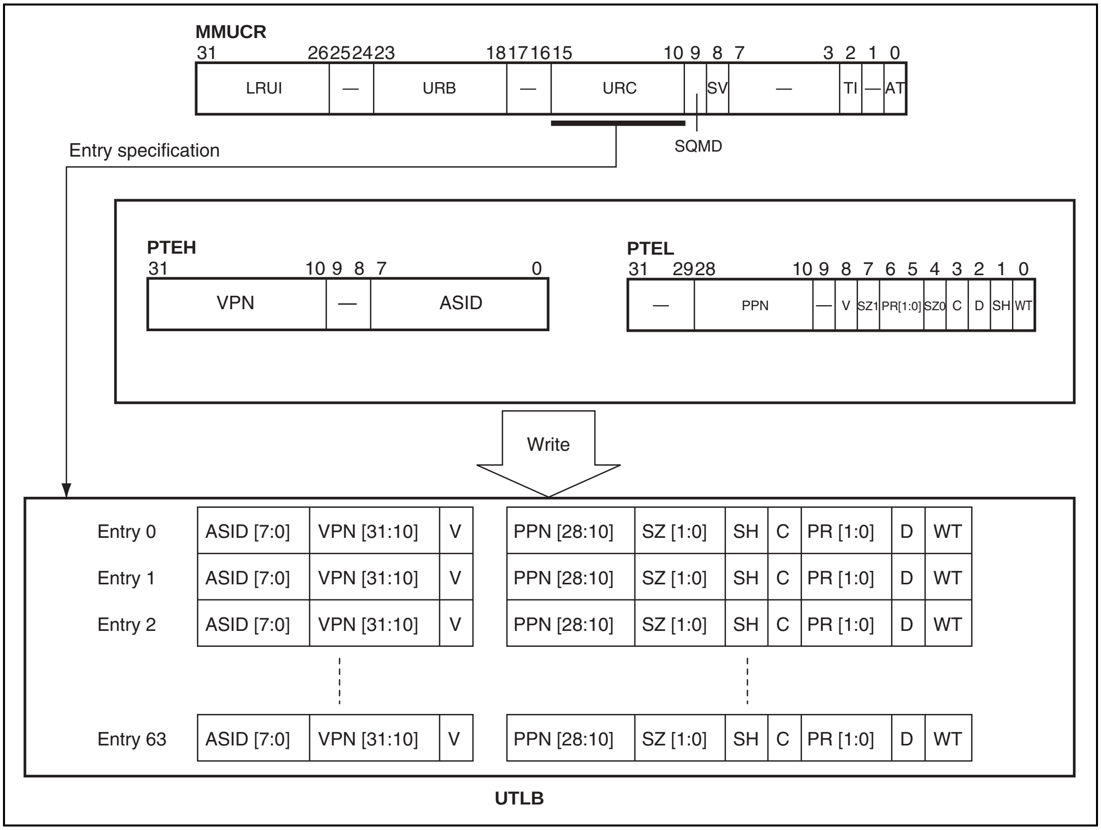


Figure 7.11 Operation of LDTLB Instruction

7.4.4 Hardware ITLB Miss Handling

In an instruction access, the SH-4A searches the ITLB. If it cannot find the necessary address translation information (ITLB miss occurred), the UTLB is searched by hardware, and if the necessary address translation information is present, it is recorded in the ITLB. This procedure is known as hardware ITLB miss handling. If the necessary address translation information is not found in the UTLB search, an instruction TLB miss exception is generated and processing passes to software.

7.4.5 Avoiding Synonym Problems

When 1- or 4-Kbyte pages are recorded in TLB entries, a synonym problem may arise. The problem is that, when a number of virtual addresses are mapped onto a single physical address, the same physical address data is recorded in a number of cache entries, and it becomes impossible to guarantee data integrity. This problem does not occur with the instruction TLB and instruction cache because data is only read in these cases. In this LSI, entry specification is performed using bits 12 to 5 of the virtual address in order to achieve fast operand cache operation. However, bits 12 to 10 of the virtual address in the case of a 1-Kbyte page, and bit 12 of the virtual address in the case of a 4-Kbyte page, are subject to address translation. As a result, bits 12 to 10 of the physical address after translation may differ from bits 12 to 10 of the virtual address.

Consequently, the following restrictions apply to the recording of address translation information in UTLB entries.

- When address translation information whereby a number of 1-Kbyte page UTLB entries are translated into the same physical address is recorded in the UTLB, ensure that the VPN[12:10] values are the same.
- When address translation information whereby a number of 4-Kbyte page UTLB entries are translated into the same physical address is recorded in the UTLB, ensure that the VPN[12] value is the same.
- Do not use 1-Kbyte page UTLB entry physical addresses with UTLB entries of a different page size.
- Do not use 4-Kbyte page UTLB entry physical addresses with UTLB entries of a different page size.

The above restrictions apply only when performing accesses using the cache.

Note: When multiple items of address translation information use the same physical memory to provide for future expansion of the SuperH RISC engine family, ensure that the VPN[20:10] values are the same. Also, do not use the same physical address for address translation information of different page sizes.

7.5 MMU Exceptions

There are seven MMU exceptions: instruction TLB multiple hit exception, instruction TLB miss exception, instruction TLB protection violation exception, data TLB multiple hit exception, data TLB miss exception, data TLB protection violation exception, and initial page write exception. Refer to figures 7.9 and 7.10 for the conditions under which each of these exceptions occurs.

7.5.1 Instruction TLB Multiple Hit Exception

An instruction TLB multiple hit exception occurs when more than one ITLB entry matches the virtual address to which an instruction access has been made. If multiple hits occur when the UTLB is searched by hardware in hardware ITLB miss handling, an instruction TLB multiple hit exception will result.

When an instruction TLB multiple hit exception occurs, a reset is executed and cache coherency is not guaranteed.

Hardware Processing: In the event of an instruction TLB multiple hit exception, hardware carries out the following processing:

1. Sets the virtual address at which the exception occurred in TEA.
2. Sets exception code H'140 in EXPEVT.
3. Branches to the reset handling routine (H'A000 0000).

Software Processing (Reset Routine): The ITLB entries which caused the multiple hit exception are checked in the reset handling routine. This exception is intended for use in program debugging, and should not normally be generated.

7.5.2 Instruction TLB Miss Exception

An instruction TLB miss exception occurs when address translation information for the virtual address to which an instruction access is made is not found in the UTLB entries by the hardware ITLB miss handling routine. The instruction TLB miss exception processing carried out by hardware and software is shown below. This is the same as the processing for a data TLB miss exception.

Hardware Processing: In the event of an instruction TLB miss exception, hardware carries out the following processing:

1. Sets the VPN of the virtual address at which the exception occurred in PTEH.
2. Sets the virtual address at which the exception occurred in TEA.
3. Sets exception code H'040 in EXPEVT.
4. Sets the PC value indicating the address of the instruction at which the exception occurred in SPC. If the exception occurred at a delay slot, sets the PC value indicating the address of the delayed branch instruction in SPC.
5. Sets the SR contents at the time of the exception in SSR. The R15 contents at this time are saved in SGR.
6. Sets the MD bit in SR to 1, and switches to privileged mode.
7. Sets the BL bit in SR to 1, and masks subsequent exception requests.
8. Sets the RB bit in SR to 1.
9. Branches to the address obtained by adding offset H'0000 0400 to the contents of VBR, and starts the instruction TLB miss exception handling routine.

Software Processing (Instruction TLB Miss Exception Handling Routine): Software is responsible for searching the external memory page table and assigning the necessary page table entry. Software should carry out the following processing in order to find and assign the necessary page table entry.

1. Write to PTEL the values of the PPN, PR, SZ, C, D, SH, V, and WT bits in the page table entry recorded in the external memory address translation table.
2. When the entry to be replaced in entry replacement is specified by software, write that value to the URC bits in MMUCR. If URC is greater than URB at this time, the value should be changed to an appropriate value after issuing an LDTLB instruction.
3. Execute the LDTLB instruction and write the contents of PTEH and PTEL to the TLB.
4. Finally, execute the exception handling return instruction (RTE), terminate the exception handling routine, and return control to the normal flow. The RTE instruction should be issued at least one instruction after the LDTLB instruction.

7.5.3 Instruction TLB Protection Violation Exception

An instruction TLB protection violation exception occurs when, even though an ITLB entry contains address translation information matching the virtual address to which an instruction access is made, the actual access type is not permitted by the access right specified by the PR bit. The instruction TLB protection violation exception processing carried out by hardware and software is shown below.

Hardware Processing: In the event of an instruction TLB protection violation exception, hardware carries out the following processing:

1. Sets the VPN of the virtual address at which the exception occurred in PTEH.
2. Sets the virtual address at which the exception occurred in TEA.
3. Sets exception code H'0A0 in EXPEVT.
4. Sets the PC value indicating the address of the instruction at which the exception occurred in SPC. If the exception occurred at a delay slot, sets the PC value indicating the address of the delayed branch instruction in SPC.
5. Sets the SR contents at the time of the exception in SSR. The R15 contents at this time are saved in SGR.
6. Sets the MD bit in SR to 1, and switches to privileged mode.
7. Sets the BL bit in SR to 1, and masks subsequent exception requests.
8. Sets the RB bit in SR to 1.
9. Branches to the address obtained by adding offset H'0000 0100 to the contents of VBR, and starts the instruction TLB protection violation exception handling routine.

Software Processing (Instruction TLB Protection Violation Exception Handling Routine): Resolve the instruction TLB protection violation, execute the exception handling return instruction (RTE), terminate the exception handling routine, and return control to the normal flow. The RTE instruction should be issued at least one instruction after the LDTLB instruction.

7.5.4 Data TLB Multiple Hit Exception

A data TLB multiple hit exception occurs when more than one UTLB entry matches the virtual address to which a data access has been made.

When a data TLB multiple hit exception occurs, a reset is executed, and cache coherency is not guaranteed. The contents of PPN in the UTLB prior to the exception may also be corrupted.

Hardware Processing: In the event of a data TLB multiple hit exception, hardware carries out the following processing:

1. Sets the virtual address at which the exception occurred in TEA.
2. Sets exception code H'140 in EXPEVT.
3. Branches to the reset handling routine (H'A000 0000).

Software Processing (Reset Routine): The UTLB entries which caused the multiple hit exception are checked in the reset handling routine. This exception is intended for use in program debugging, and should not normally be generated.

7.5.5 Data TLB Miss Exception

A data TLB miss exception occurs when address translation information for the virtual address to which a data access is made is not found in the UTLB entries. The data TLB miss exception processing carried out by hardware and software is shown below.

Hardware Processing: In the event of a data TLB miss exception, hardware carries out the following processing:

1. Sets the VPN of the virtual address at which the exception occurred in PTEH.
2. Sets the virtual address at which the exception occurred in TEA.
3. Sets exception code H'040 in the case of a read, or H'060 in the case of a write in EXPEVT (OCBP, OCBWB: read; OCBI, MOVCA.L: write).
4. Sets the PC value indicating the address of the instruction at which the exception occurred in SPC. If the exception occurred at a delay slot, sets the PC value indicating the address of the delayed branch instruction in SPC.
5. Sets the SR contents at the time of the exception in SSR. The R15 contents at this time are saved in SGR.
6. Sets the MD bit in SR to 1, and switches to privileged mode.
7. Sets the BL bit in SR to 1, and masks subsequent exception requests.
8. Sets the RB bit in SR to 1.
9. Branches to the address obtained by adding offset H'0000 0400 to the contents of VBR, and starts the data TLB miss exception handling routine.

Software Processing (Data TLB Miss Exception Handling Routine): Software is responsible for searching the external memory page table and assigning the necessary page table entry. Software should carry out the following processing in order to find and assign the necessary page table entry.

1. Write to PTEL the values of the PPN, PR, SZ, C, D, SH, V, and WT bits in the page table entry recorded in the external memory address translation table.
2. When the entry to be replaced in entry replacement is specified by software, write that value to the URC bits in MMUCR. If URC is greater than URB at this time, the value should be changed to an appropriate value after issuing an LDTLB instruction.
3. Execute the LDTLB instruction and write the contents of PTEH and PTEL to the UTLB.
4. Finally, execute the exception handling return instruction (RTE), terminate the exception handling routine, and return control to the normal flow. The RTE instruction should be issued at least one instruction after the LDTLB instruction.

7.5.6 Data TLB Protection Violation Exception

A data TLB protection violation exception occurs when, even though a UTLB entry contains address translation information matching the virtual address to which a data access is made, the actual access type is not permitted by the access right specified by the PR bit. The data TLB protection violation exception processing carried out by hardware and software is shown below.

Hardware Processing: In the event of a data TLB protection violation exception, hardware carries out the following processing:

1. Sets the VPN of the virtual address at which the exception occurred in PTEH.
2. Sets the virtual address at which the exception occurred in TEA.
3. Sets exception code H'0A0 in the case of a read, or H'0C0 in the case of a write in EXPEVT (OCBP, OCBWB: read; OCBI, MOVCA.L: write).
4. Sets the PC value indicating the address of the instruction at which the exception occurred in SPC. If the exception occurred at a delay slot, sets the PC value indicating the address of the delayed branch instruction in SPC.
5. Sets the SR contents at the time of the exception in SSR. The R15 contents at this time are saved in SGR.
6. Sets the MD bit in SR to 1, and switches to privileged mode.
7. Sets the BL bit in SR to 1, and masks subsequent exception requests.
8. Sets the RB bit in SR to 1.
9. Branches to the address obtained by adding offset H'0000 0100 to the contents of VBR, and starts the data TLB protection violation exception handling routine.

Software Processing (Data TLB Protection Violation Exception Handling Routine): Resolve the data TLB protection violation, execute the exception handling return instruction (RTE), terminate the exception handling routine, and return control to the normal flow. The RTE instruction should be issued at least one instruction after the LDTLB instruction.

7.5.7 Initial Page Write Exception

An initial page write exception occurs when the D bit is 0 even though a UTLB entry contains address translation information matching the virtual address to which a data access (write) is made, and the access is permitted. The initial page write exception processing carried out by hardware and software is shown below.

Hardware Processing: In the event of an initial page write exception, hardware carries out the following processing:

1. Sets the VPN of the virtual address at which the exception occurred in PTEH.
2. Sets the virtual address at which the exception occurred in TEA.
3. Sets exception code H'080 in EXPEVT.
4. Sets the PC value indicating the address of the instruction at which the exception occurred in SPC. If the exception occurred at a delay slot, sets the PC value indicating the address of the delayed branch instruction in SPC.
5. Sets the SR contents at the time of the exception in SSR. The R15 contents at this time are saved in SGR.
6. Sets the MD bit in SR to 1, and switches to privileged mode.
7. Sets the BL bit in SR to 1, and masks subsequent exception requests.
8. Sets the RB bit in SR to 1.
9. Branches to the address obtained by adding offset H'0000 0100 to the contents of VBR, and starts the initial page write exception handling routine.

Software Processing (Initial Page Write Exception Handling Routine): Software is responsible for the following processing:

1. Retrieve the necessary page table entry from external memory.
2. Write 1 to the D bit in the external memory page table entry.
3. Write to PTEL the values of the PPN, PR, SZ, C, D, WT, SH, and V bits in the page table entry recorded in external memory.
4. When the entry to be replaced in entry replacement is specified by software, write that value to the URC bits in MMUCR. If URC is greater than URB at this time, the value should be changed to an appropriate value after issuing an LDTLB instruction.
5. Execute the LDTLB instruction and write the contents of PTEH and PTEL to the UTLB.

6. Finally, execute the exception handling return instruction (RTE), terminate the exception handling routine, and return control to the normal flow. The RTE instruction should be issued at least one instruction after the LDTLB instruction.

7.6 Memory-Mapped TLB Configuration

To enable the ITLB and UTLB to be managed by software, their contents are allowed to be read from and written to by a program in the P2 area with a MOV instruction in privileged mode. Operation is not guaranteed if access is made from a program in another area.

After the memory-mapped TLB has been accessed, execute one of the following three methods before an access (including an instruction fetch) to an area other than the P2 area is performed.

1. Execute a branch using the RTE instruction. In this case, the branch destination may be an area other than the P2 area.
2. Execute the ICBI instruction for any address (including non-cacheable area).
3. If the MT bit in IRMCR is 0 (initial value) before accessing the memory-mapped TLB, the specific instruction does not need to be executed. However, note that the CPU processing performance will be lowered because the instruction fetch is performed again for the next instruction after MMUCR has been updated.

Note that the method 3 may not be guaranteed in the future SuperH Series. Therefore, it is recommended that the method 1 or 2 should be used for being compatible with the future SuperH Series.

The ITLB and UTLB are allocated to the P4 area in the virtual address space. VPN, V, and ASID in the ITLB can be accessed as an address array, PPN, V, SZ, PR, C, and SH as a data array. VPN, D, V, and ASID in the UTLB can be accessed as an address array, PPN, V, SZ, PR, C, D, WT, and SH as a data array. V and D can be accessed from both the address array side and the data array side. Only longword access is possible. Instruction fetches cannot be performed in these areas. For reserved bits, a write value of 0 should be specified; their read value is undefined.

7.6.1 ITLB Address Array

The ITLB address array is allocated to addresses H'F200 0000 to H'F2FF FFFF in the P4 area. An address array access requires a 32-bit address field specification (when reading or writing) and a 32-bit data field specification (when writing). Information for selecting the entry to be accessed is specified in the address field, and VPN, V, and ASID to be written to the address array are specified in the data field.

In the address field, bits [31:24] have the value H'F2 indicating the ITLB address array and the entry is specified by bits [9:8]. As only longword access is used, 0 should be specified for address field bits [1:0].

In the data field, bits [31:10] indicate VPN, bit [8] indicates V, and bits [7:0] indicate ASID.

The following two kinds of operation can be used on the ITLB address array:

1. ITLB address array read

   VPN, V, and ASID are read into the data field from the ITLB entry corresponding to the entry set in the address field.
2. ITLB address array write

   VPN, V, and ASID specified in the data field are written to the ITLB entry corresponding to the entry set in the address field.

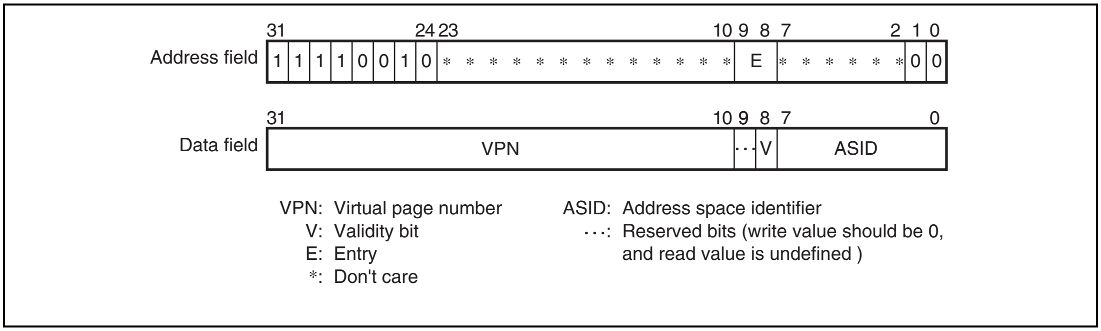

Address field
31
24 23
10 9 8 7
2 1 0
1 1 1 1 0 0 1 0 * * * * * * * * * * * * * * E * * * * * * 0 0
Data field
31
10 9 8 7
0
VPN
V
ASID
VPN: Virtual page number
V: Validity bit
E: Entry
*: Don't care
ASID: Address space identifier
· · ·: Reserved bits (write value should be 0, and read value is undefined)

Figure 7.12 Memory-Mapped ITLB Address Array

7.6.2 ITLB Data Array

The ITLB data array is allocated to addresses H'F300 0000 to H'F37F FFFF in the P4 area. A data array access requires a 32-bit address field specification (when reading or writing) and a 32-bit data field specification (when writing). Information for selecting the entry to be accessed is specified in the address field, and PPN, V, SZ, PR, C, and SH to be written to the data array are specified in the data field.

In the address field, bits [31:23] have the value H'F30 indicating ITLB data array and the entry is specified by bits [9:8].

In the data field, bits [28:10] indicate PPN, bit [8] indicates V, bits [7] and [4] indicate SZ, bit [6] indicates PR, bit [3] indicates C, and bit [1] indicates SH.

The following two kinds of operation can be used on ITLB data array:

1. ITLB data array read
   PPN, V, SZ, PR, C, and SH are read into the data field from the ITLB entry corresponding to the entry set in the address field.
2. ITLB data array write
   PPN, V, SZ, PR, C, and SH specified in the data field are written to the ITLB entry corresponding to the entry set in the address field.

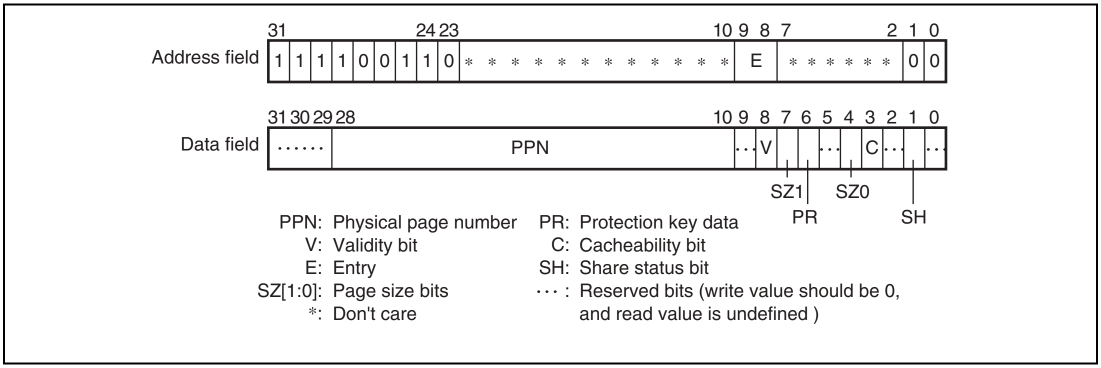


Figure 7.13 Memory-Mapped ITLB Data Array

7.6.3 UTLB Address Array

The UTLB address array is allocated to addresses H'F600 0000 to H'F60F FFFF in the P4 area. An address array access requires a 32-bit address field specification (when reading or writing) and a 32-bit data field specification (when writing). Information for selecting the entry to be accessed is specified in the address field, and VPN, D, V, and ASID to be written to the address array are specified in the data field.

In the address field, bits [31:20] have the value H'F60 indicating the UTLB address array and the entry is specified by bits [13:8]. Bit [7] that is the association bit (A bit) in the address field specifies whether address comparison is performed in a write to the UTLB address array.

In the data field, bits [31:10] indicate VPN, bit [9] indicates D, bit [8] indicates V, and bits [7:0] indicate ASID.

The following three kinds of operation can be used on the UTLB address array:

1. UTLB address array read

   VPN, D, V, and ASID are read into the data field from the UTLB entry corresponding to the entry set in the address field. In a read, associative operation is not performed regardless of whether the association bit specified in the address field is 1 or 0.
2. UTLB address array write (non-associative)

   VPN, D, V, and ASID specified in the data field are written to the UTLB entry corresponding to the entry set in the address field. The A bit in the address field should be cleared to 0.
3. UTLB address array write (associative)

   When a write is performed with the A bit in the address field set to 1, comparison of all the UTLB entries is carried out using the VPN specified in the data field and ASID in PTEH. The usual address comparison rules are followed, but if a UTLB miss occurs, the result is no operation, and an exception is not generated. If the comparison identifies a UTLB entry corresponding to the VPN specified in the data field, D and V specified in the data field are written to that entry. This associative operation is simultaneously carried out on the ITLB, and if a matching entry is found in the ITLB, V is written to that entry. Even if the UTLB comparison results in no operation, a write to the ITLB is performed as long as a matching entry is found in the ITLB. If there is a match in both the UTLB and ITLB, the UTLB information is also written to the ITLB.

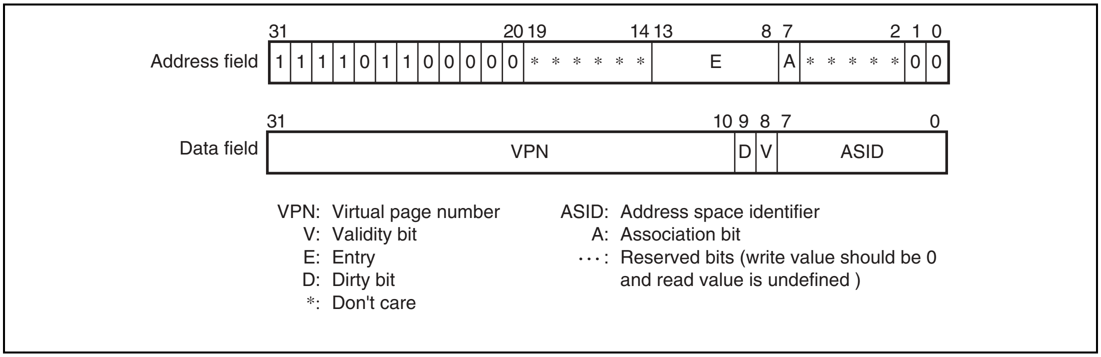


Figure 7.14 Memory-Mapped UTLB Address Array

7.6.4 UTLB Data Array

The UTLB data array is allocated to addresses H'F700 0000 to H'F70F FFFF in the P4 area. A data array access requires a 32-bit address field specification (when reading or writing) and a 32-bit data field specification (when writing). Information for selecting the entry to be accessed is specified in the address field, and PPN, V, SZ, PR, C, D, SH, and WT to be written to data array are specified in the data field.

In the address field, bits [31:20] have the value H'F70 indicating UTLB data array and the entry is specified by bits [13:8].

In the data field, bits [28:10] indicate PPN, bit [8] indicates V, bits [7] and [4] indicate SZ, bits [6:5] indicate PR, bit [3] indicates C, bit [2] indicates D, bit [1] indicates SH, and bit [0] indicates WT.

The following two kinds of operation can be used on UTLB data array:

1. UTLB data array read

   PPN, V, SZ, PR, C, D, SH, and WT are read into the data field from the UTLB entry corresponding to the entry set in the address field.
2. UTLB data array write

   PPN, V, SZ, PR, C, D, SH, and WT specified in the data field are written to the UTLB entry corresponding to the entry set in the address field.

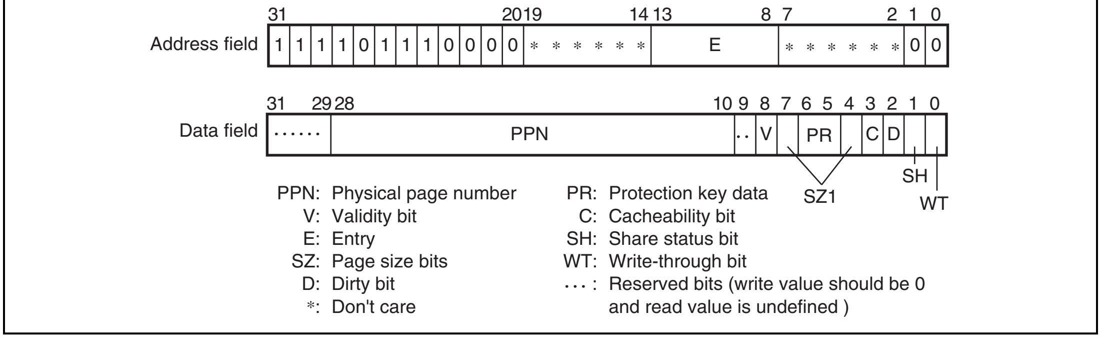


Figure 7.15 Memory-Mapped UTLB Data Array

7.7 32-Bit Address Extended Mode

Setting the SE bit in PASCR to 1 changes mode from 29-bit address mode which handles the 29-bit physical address space to 32-bit address extended mode which handles the 32-bit physical address space.

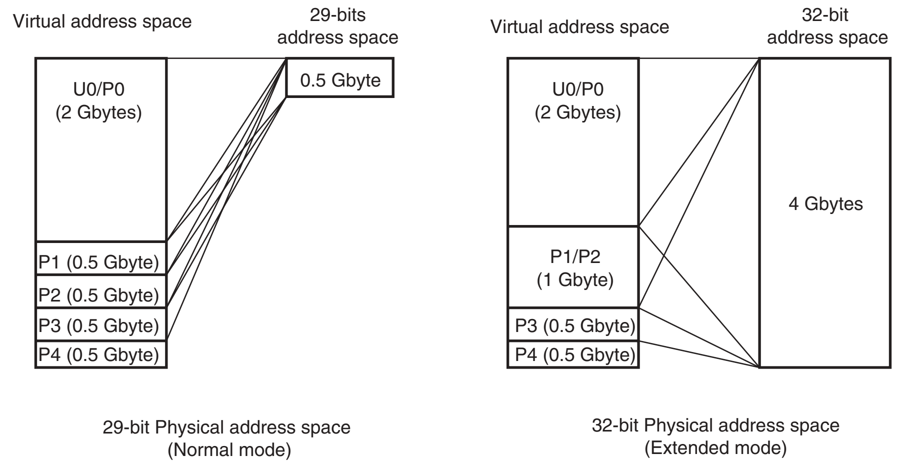


Figure 7.16 Physical Address Space (32-Bit Address Extended Mode)

7.7.1 Overview of 32-Bit Address Extended Mode

In 32-bit address extended mode, the privileged space mapping buffer (PMB) is introduced. The PMB maps virtual addresses in the P1 or P2 area which are not translated in 29-bit address mode to the 32-bit physical address space. In areas which are target for address translation of the TLB (UTLB/ITLB), upper three bits in the PPN field of the UTLB or ITLB are extended and then addresses after the TLB translation can handle the 32-bit physical addresses.

As for the cache operation, P1 area is cacheable and P2 area is non-cacheable in the case of 29-bit address mode, but the cache operation of both P1 and P2 area are determined by the C bit and WT bit in the PMB in the case of 32-bit address mode.

7.7.2 Transition to 32-Bit Address Extended Mode

The SH-4A enters 29-bit address mode after a power-on reset. Transition is made to 32-bit address extended mode by setting the SE bit in PASCR to 1. In 32-bit address extended mode, the MMU operates as follows.

1. When the AT bit in MMUCR is 0, virtual addresses in the U0, P0, or P3 area become 32-bit physical addresses. Addresses in the P1 or P2 area are translated according to the PMB mapping information.
2. When the AT bit in MMUCR is 1, virtual addresses in the U0, P0, or P3 area are translated to 32-bit physical addresses according to the TLB conversion information. Addresses in the P1 or P2 area are translated according to the PMB mapping information.
3. Regardless of the setting of the AT bit in MMUCR, bits 31 to 29 in physical addresses become B'111 in the control register area (addresses H'FC00 0000 to H'FFFF FFFF). When the control register area is recorded in the UTLB and accessed, B'111 should be set to PPN[31:29].

7.7.3 Privileged Space Mapping Buffer (PMB) Configuration

In 32-bit address extended mode, virtual addresses in the P1 or P2 area are translated according to the PMB mapping information. The PMB has 16 entries and configuration of each entry is as follows.

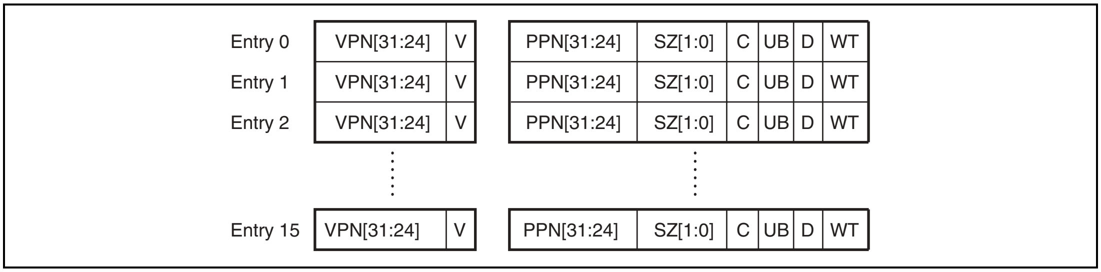


Figure 7.17 PMB Configuration

[Legend]

- VPN: Virtual page number
  For 16-Mbyte page: Upper 8 bits of virtual address
  For 64-Mbyte page: Upper 6 bits of virtual address
  For 128-Mbyte page: Upper 5 bits of virtual address
  For 512-Mbyte page: Upper 3 bits of virtual address

- SZ: Page size bits
  Specify the page size.
  00: 16-Mbyte page
  01: 64-Mbyte page
  10: 128-Mbyte page
  11: 512-Mbyte page

- V: Validity bit
  Indicates whether the entry is valid.
  0: Invalid
  1: Valid
  Cleared to 0 by a power-on reset.
  Not affected by a manual reset.

- PPN: Physical page number
  Upper 8 bits of the physical address of the physical page number.
  With a 16-Mbyte page, PPN[31:24] are valid.
  With a 64-Mbyte page, PPN[31:26] are valid.
  With a 128-Mbyte page, PPN[31:27] are valid.
  With a 512-Mbyte page, PPN[31:29] are valid.

- C: Cacheability bit
  Indicates whether a page is cacheable.
  0: Not cacheable
  1: Cacheable

- WT: Write-through bit
  Specifies the cache write mode.
  0: Copy-back mode
  1: Write-through mode

- UB: Buffered write bit

  Specifies whether a buffered write is performed.

  0: Buffered write (Data access of subsequent processing proceeds without waiting for the write to complete.)

  1: Unbuffered write (Data access of subsequent processing is stalled until the write has completed.)

7.7.4 PMB Function

The SH-4A supports the following PMB functions.

1. Only memory-mapped write can be used for writing to the PMB. The LDTLB instruction cannot be used to write to the PMB.
2. Software must ensure that every accessed P1 or P2 address has a corresponding PMB entry before the access occurs. When an access to an address in the P1 or P2 area which is not recorded in the PMB is made, the SH-4A is reset by the TLB. In this case, the accessed address in the P1 or P2 area which causes the TLB reset is stored in the TEA and code H′140 in the EXPEVT.
3. The SH-4A does not guarantee the operation when multiple hit occurs in the PMB. Special care should be taken when the PMB mapping information is recorded by software.
4. The PMB does not have an associative write function.
5. Since there is no PR field in the PMB, read/write protection cannot be preformed. The address translation target of the PMB is the P1 or P2 address. In user mode access, an address error exception occurs.
6. Both entries from the UTLB and PMB are mixed and recorded in the ITLB by means of the hardware ITLB miss handling. However, these entries can be identified by checking whether VPN[31:30] is 10 or not. When an entry from the PMB is recorded in the ITLB, H′00, 01, and 1 are recorded in the ASID, PR, and SH fields which do not exist in the PMB, respectively.

7.7.5 Memory-Mapped PMB Configuration

To enable the PMB to be managed by software, its contents are allowed to be read from and written to by a P1 or P2 area program with a MOV instruction in privileged mode. The PMB address array is allocated to addresses H'F610 0000 to H'F61F FFFF in the P4 area and the PMB data array to addresses H'F710 0000 to H'F71F FFFF in the P4 area. VPN and V in the PMB can be accessed as an address array, PPN, V, SZ, C, WT, and UB as a data array. V can be accessed from both the address array side and the data array side. A program which executes a PMB memory-mapped access should be placed in the page area at which the C bit in PMB is cleared to 0.

1. PMB address array read

   When memory reading is performed while bits 31 to 20 in the address field are specified as H'F61 which indicates the PMB address array and bits 11 to 8 in the address field as an entry, bits 31 to 24 in the data field are read as VPN and bit 8 in the data field as V.

2. PMB address array write

   When memory writing is performed while bits 31 to 20 in the address field are specified as H'F61 which indicates the PMB address array and bits 11 to 8 in the address field as an entry, and bits 31 to 24 in the data field are specified as VPN and bit 8 in the data field as V, data is written to the specified entry.

3. PMB data array read

   When memory reading is performed while bits 31 to 20 in the address field are specified as H'F71 which indicates the PMB data array and bits 11 to 8 in the address field as an entry, bits 31 to 24 in the data field are read as PPN, bit 9 in the data field as UB, bit 8 in the data field as V, bits 7 and 4 in the data field as SZ, bit 3 in the data field as C, and bit 0 in the data field as WT.

4. PMB data array write

   When memory writing is performed while bits 31 to 20 in the address field are specified as H'F71 which indicates the PMB data array and bits 11 to 8 in the address field as an entry, and bits 31 to 24 in the data field are specified as PPN, bit 9 in the data field as UB, bit 8 in the data field as V, bits 7 and 4 in the data field as SZ, bit 3 in the data field as C, and bit 0 in the data field as WT, data is written to the specified entry.

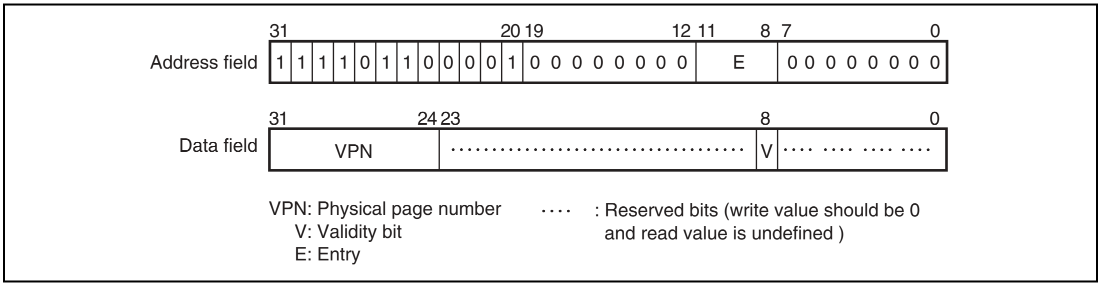


Figure 7.18 Memory-Mapped PMB Address Array

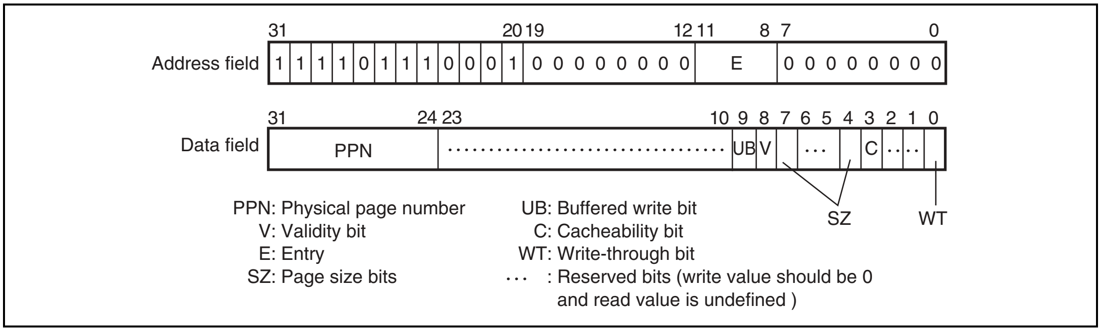


Figure 7.19 Memory-Mapped PMB Data Array

7.7.6 Notes on Using 32-Bit Address Extended Mode

When using 32-bit address extended mode, note that the items described in this section are extended or changed as follows.

PASCR: The SE bit is added in bit 31 in the control register (PASCR). The bits 6 to 0 of the UB in the PASCR are invalid (Note that the bit 7 of the UB is still valid). When writing to the P1 or P2 area, the UB bit in the PMB controls whether a buffered write is performed or not. When the MMU is enabled, the UB bit in the TLB controls writing to the P0, P3, or U0 area. When the MMU is disabled, writing to the P0, P3, or U0 area is always performed as a buffered write.

Bit	Bit Name	Initial Value	R/W	Description
31	SE	0	R/W	0: 29-bit address mode 1: 32-bit address extended mode
30 to 8	—	All 0	R	Reserved For details on reading from or writing to these bits, see description in General Precautions on Handling of Product.
7 to 0	UB	All 0	R/W	Buffered Write Control for Each Area (64 Mbytes) When writing is performed without using the cache or in the cache write-through mode, these bits specify whether the CPU waits for the end of writing for each area. 0: The CPU does not wait for the end of writing 1: The CPU stalls and waits for the end of writing UB[7]: Corresponding to the control register area UB[6:0]: These bits are invalid in 32-bit address extended mode.

ITLB: The PPN field in the ITLB is extended to bits 31 to 10.

UTLB: The PPN field in the UTLB is extended to bits 31 to 10. The same UB bit as that in the PMB is added in each entry of the UTLB.

- UB: Buffered write bit

  Specifies whether a buffered write is performed.

  0: Buffered write (Subsequent processing proceeds without waiting for the write to complete.)

  1: Unbuffered write (Subsequent processing is stalled until the write has completed.)

In a memory-mapped TLB access, the UB bit can be read from or written to by bit 9 in the data array.

PTEL: The same UB bit as that in the PMB is added in bit 9 in PTEL. This UB bit is written to the UB bit in the UTLB by the LDTLB instruction. The PPN field is extended to bits 31 to 10.

CCR.CB: The CB bit in CCR is invalid. Whether a cacheable write for the P1 area is performed in copy-back mode or write-though mode is determined by the WT bit in the PMB.

IRMCR.MT: The MT bit in IRMCR is valid for a memory-mapped PMB write.

QACR0, QACR1: AREA0[4:2]/AREA1[4:2] fields of QACR0/QACR1 are extended to AREA0[7:2]/AREA1[7:2] corresponding to physical address [31:26]. See section 8.2.2, Queue Address Control Register 0 (QACR0) and 8.2.3, Queue Address Control Register 1 (QACR1).

LSA0, LSA1, LDA0, LDA1: L0SADR, L1SADR, L0DADR, and L1DADR fields are extended to bits 31 to 10. See section 9.2.2, L Memory Transfer Source Address Register 0 (LSA0), section 9.2.3, L Memory Transfer Source Address Register 1 (LSA1), section 9.2.4, L Memory Transfer Destination Address Register 0 (LDA0), and section 9.2.5, L Memory Transfer Destination Address Register 1 (LDA1).

When using 32-bit address mode, the following notes should be applied to software.

1. For the SE bit switching, only switching from 0 to 1 is supported in boot routine after a power-on reset or manual reset.
2. After switching the SE bit, an area in which the program is allocated becomes the target of the PMB address translation. Therefore, the area should be recorded in the PMB before switching the SE bit. An address which may be accessed in the P1 or P2 area such as the exception handler should also be recorded in the PMB.
3. When an external memory access occurs by an operand memory access located before the MOV.L instruction which switches the SE bit, external memory space addresses accessed in both address modes should be the same.
4. Note that the V bit is mapped to both address array and data array in PMB registration. That is, first write 0 to the V bit in one of arrays and then write 1 to the V bit in another array.

Section 8 Caches

The SH-4A has an on-chip 32-Kbyte instruction cache (IC) for instructions and an on-chip 32-Kbyte operand cache (OC) for data.

Note: For the size of instruction cache and operand cache, see the hardware manual of the target product. This manual describes the 32-Kbyte case for each cache memory.

8.1 Features

The features of the cache are given in table 8.1.

The SH-4A supports two 32-byte store queues (SQs) to perform high-speed writes to external memory. The features of the store queues are given in table 8.2.

Table 8.1 Cache Features

Item	Instruction Cache	Operand Cache
Capacity	32-Kbyte cache	32-Kbyte cache
Type	4-way set-associative, virtual address index/physical address tag	4-way set-associative, virtual address index/physical address tag
Line size	32 bytes	32 bytes
Entries	256 entries/way	256 entries/way
Write method	—	Copy-back/write-through selectable
Replacement method	LRU (least-recently-used) algorithm	LRU (least-recently-used) algorithm

Table 8.2 Store Queue Features

Item	Store Queues
Capacity	32 bytes × 2
Addresses	H'E000 0000 to H'E3FF FFFF
Write	Store instruction (1-cycle write)
Write-back	Prefetch instruction (PREF instruction)
Access right	When MMU is disabled: Determined by SQMD bit in MMUCR When MMU is enabled: Determined by PR for each page

The operand cache of the SH-4A is 4-way set associative, each may comprising 256 cache lines. Figure 8.1 shows the configuration of the operand cache.

The instruction cache is 4-way set-associative, each way comprising 256 cache lines. Figure 8.2 shows the configuration of the instruction cache.

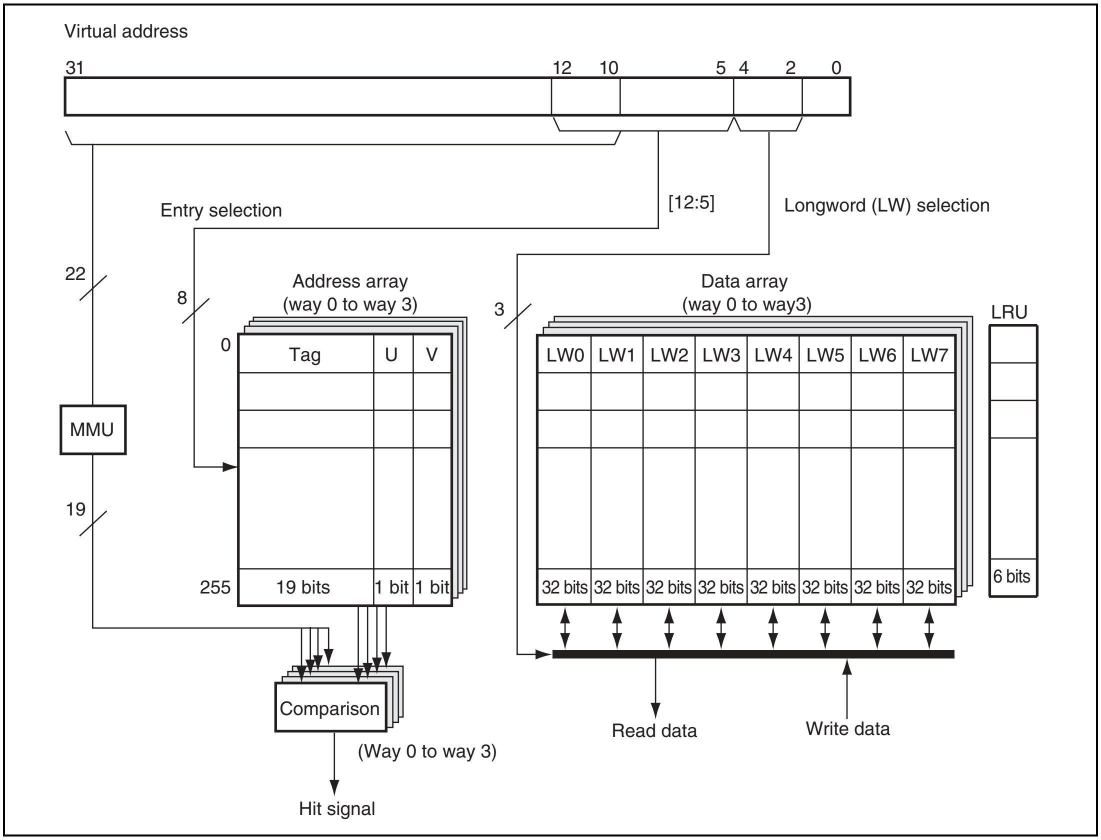


Figure 8.1 Configuration of Operand Cache (OC)

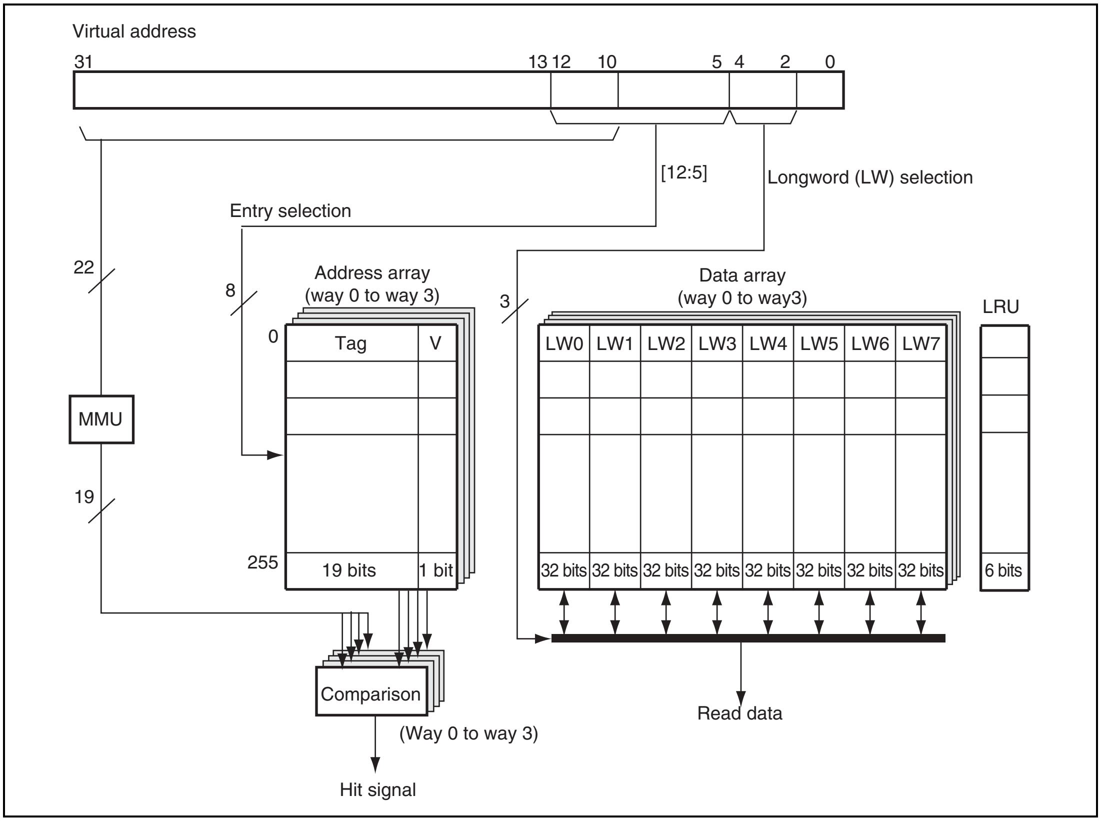


Figure 8.2 Configuration of Instruction Cache (IC)

- Tag

  Stores the upper 19 bits of the 29-bit physical address of the data line to be cached. The tag is not initialized by a power-on or manual reset.

- V bit (validity bit)

  Indicates that valid data is stored in the cache line. When this bit is 1, the cache line data is valid. The V bit is initialized to 0 by a power-on reset, but retains its value in a manual reset.

- U bit (dirty bit)

  The U bit is set to 1 if data is written to the cache line while the cache is being used in copy-back mode. That is, the U bit indicates a mismatch between the data in the cache line and the data in external memory. The U bit is never set to 1 while the cache is being used in write-through mode, unless it is modified by accessing the memory-mapped cache (see section 8.6, Memory-Mapped Cache Configuration). The U bit is initialized to 0 by a power-on reset, but retains its value in a manual reset.

- Data array

  The data field holds 32 bytes (256 bits) of data per cache line. The data array is not initialized by a power-on or manual reset.

- LRU

  In a 4-way set-associative method, up to 4 items of data can be registered in the cache at each entry address. When an entry is registered, the LRU bit indicates which of the 4 ways it is to be registered in. The LRU mechanism uses 6 bits of each entry, and its usage is controlled by hardware. The LRU (least-recently-used) algorithm is used for way selection, and selects the less recently accessed way. The LRU bits are initialized to 0 by a power-on reset but not by a manual reset. The LRU bits cannot be read from or written to by software.

8.2 Register Descriptions

The following registers are related to cache.

Table 8.3 Register Configuration

Register Name	Abbreviation	R/W	P4 Address*	Area 7 Address*	Size
Cache control register	CCR	R/W	H'FF00 001C	H'1F00 001C	32
Queue address control register 0	QACR0	R/W	H'FF00 0038	H'1F00 0038	32
Queue address control register 1	QACR1	R/W	H'FF00 003C	H'1F00 003C	32
On-chip memory control register	RAMCR	R/W	H'FF00 0074	H'1F00 0074	32

Note: * These P4 addresses are for the P4 area in the virtual address space. These area 7 addresses are accessed from area 7 in the physical address space by means of the TLB.

Table 8.4 Register States in Each Processing State

Register Name	Abbreviation	Power-on Reset	Manual Reset	Sleep	Standby
Cache control register	CCR	H'0000 0000	H'0000 0000	Retained	Retained
Queue address control register 0	QACR0	Undefined	Undefined	Retained	Retained
Queue address control register 1	QACR1	Undefined	Undefined	Retained	Retained
On-chip memory control register	RAMCR	H'0000 0000	H'0000 0000	Retained	Retained

8.2.1 Cache Control Register (CCR)

CCR controls the cache operating mode, the cache write mode, and invalidation of all cache entries.

CCR modifications must only be made by a program in the non-cacheable P2 area. After CCR has been updated, execute one of the following three methods before an access (including an instruction fetch) to the cacheable area is performed.

1. Execute a branch using the RTE instruction. In this case, the branch destination may be the cacheable area.
2. Execute the ICBI instruction for any address (including non-cacheable area).
3. If the R2 bit in IRMCR is 0 (initial value) before updating CCR, the specific instruction does not need to be executed. However, note that the CPU processing performance will be lowered because the instruction fetch is performed again for the next instruction after CCR has been updated.

Note that the method 3 may not be guaranteed in the future SuperH Series. Therefore, it is recommended that the method 1 or 2 should be used for being compatible with the future SuperH Series.

Bit:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
Initial value:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
R/W:	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R

Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	—	—	—	—	ICI	—	—	ICE	—	—	—	—	OCI	CB	WT	OCE
Initial value:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
R/W:	R	R	R	R	R/W	R	R	R/W	R	R	R	R	R/W	R/W	R/W	R/W

Bit	Bit Name	Initial Value	R/W	Description
31 to 12	—	All 0	R	Reserved For details on reading from or writing to these bits, see description in General Precautions on Handling of Product.
11	ICI	0	R/W	IC Invalidation Bit When 1 is written to this bit, the V bits of all IC entries are cleared to 0. This bit is always read as 0.

Bit	Bit Name	Initial Value	R/W	Description
10, 9	—	All 0	R	Reserved For details on reading from or writing to these bits, see description in General Precautions on Handling of Product.
8	ICE	0	R/W	IC Enable Bit Selects whether the IC is used. Note however when address translation is performed, the IC cannot be used unless the C bit in the page management information is also 1. 0: IC not used 1: IC used
7 to 4	—	All 0	R	Reserved For details on reading from or writing to these bits, see description in General Precautions on Handling of Product.
3	OCI	0	R/W	OC Invalidation Bit When 1 is written to this bit, the V and U bits of all OC entries are cleared to 0. This bit is always read as 0.
2	CB	0	R/W	Copy-Back Bit Indicates the P1 area cache write mode. 0: Write-through mode 1: Copy-back mode
1	WT	0	R/W	Write-Through Mode Indicates the P0, U0, and P3 area cache write mode. When address translation is performed, the value of the WT bit in the page management information has priority. 0: Copy-back mode 1: Write-through mode
0	OCE	0	R/W	OC Enable Bit Selects whether the OC is used. Note however when address translation is performed, the OC cannot be used unless the C bit in the page management information is also 1. 0: OC not used 1: OC used

8.2.2 Queue Address Control Register 0 (QACR0)

QACR0 specifies the area onto which store queue 0 (SQ0) is mapped when the MMU is disabled.

Bit:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
Initial value:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
R/W:	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R

Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	—	—	—	—	—	—	—	—	—	—	—	AREA0			—	—
Initial value:	0	0	0	0	0	0	0	0	0	0	0	—	—	—	0	0
R/W:	R	R	R	R	R	R	R	R	R	R	R	R/W	R/W	R/W	R	R

Bit	Bit Name	Initial Value	R/W	Description
31 to 5	—	All 0	R	Reserved For details on reading from or writing to these bits, see description in General Precautions on Handling of Product.
4 to 2	AREA0	Undefined	R/W	When the MMU is disabled, these bits generate physical address bits [28:26] for SQ0.
1, 0	—	All 0	R	Reserved For details on reading from or writing to these bits, see description in General Precautions on Handling of Product.

8.2.3 Queue Address Control Register 1 (QACR1)

QACR1 specifies the area onto which store queue 1 (SQ1) is mapped when the MMU is disabled.

Bit:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
Initial value:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
R/W:	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R

Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	—	—	—	—	—	—	—	—	—	—	—	AREA1			—	—
Initial value:	0	0	0	0	0	0	0	0	0	0	0	—	—	—	0	0
R/W:	R	R	R	R	R	R	R	R	R	R	R	R/W	R/W	R/W	R	R

Bit	Bit Name	Initial Value	R/W	Description
31 to 5	—	All 0	R	Reserved For details on reading from or writing to these bits, see description in General Precautions on Handling of Product.
4 to 2	AREA1	Undefined	R/W	When the MMU is disabled, these bits generate physical address bits [28:26] for SQ1.
1, 0	—	All 0	R	Reserved For details on reading from or writing to these bits, see description in General Precautions on Handling of Product.

8.2.4 On-Chip Memory Control Register (RAMCR)

RAMCR controls the number of ways in the IC and OC.

RAMCR modifications must only be made by a program in the non-cacheable P2 area. After RAMCR has been updated, execute one of the following three methods before an access (including an instruction fetch) to the cacheable area or the L memory area is performed.

1. Execute a branch using the RTE instruction. In this case, the branch destination may be the non-cacheable area or the L memory area.
2. Execute the ICBI instruction for any address (including non-cacheable area).
3. If the R2 bit in IRMCR is 0 (initial value) before updating RAMCR, the specific instruction does not need to be executed. However, note that the CPU processing performance will be lowered because the instruction fetch is performed again for the next instruction after RAMCR has been updated.

Note that the method 3 may not be guaranteed in the future SuperH Series. Therefore, it is recommended that the method 1 or 2 should be used for being compatible with the future SuperH Series.

Bit:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
Initial value:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
R/W:	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R

Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	—	—	—	—	—	—	RMD	RP	IC2W	OC2W	—	—	—	—	—	—
Initial value:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
R/W:	R	R	R	R	R	R	R/W	R/W	R/W	R/W	R	R	R	R	R	R

Bit	Bit Name	Initial Value	R/W	Description
31 to 10	—	All 0	R	Reserved For details on reading from or writing to these bits, see description in General Precautions on Handling of Product.
9	RMD	0	R/W	On-Chip Memory Access Mode Bit For details, see section 9.4, L Memory Protective Functions.
8	RP	0	R/W	On-Chip Memory Protection Enable Bit For details, see section 9.4, L Memory Protective Functions.

Bit	Bit Name	Initial Value	R/W	Description
7	IC2W	0	R/W	IC Two-Way Mode bit 0: IC is a four-way operation 1: IC is a two-way operation For details, see section 8.4.3, IC Two-Way Mode.
6	OC2W	0	R/W	OC Two-Way Mode bit 0: OC is a four-way operation 1: OC is a two-way operation For details, see section 8.3.6, OC Two-Way Mode.
5 to 0	—	All 0	R	Reserved For details on reading from or writing to these bits, see description in General Precautions on Handling of Product.

8.3 Operand Cache Operation

8.3.1 Read Operation

When the Operand Cache (OC) is enabled (OCE = 1 in CCR) and data is read from a cacheable area, the cache operates as follows:

1. The tag, V bit, U bit, and LRU bits on each way are read from the cache line indexed by virtual address bits [12:5].
2. The tags read from the each way is compared with bits [28:10] of the physical address resulting from virtual address translation by the MMU:

- If there is a way whose tag matches and its V bit is 1, see No. 3.
- If there is no way whose tag matches and its V bit is 1 and the U bit of the way which is selected to replace using the LRU bits is 0, see No. 4.
- If there is no way whose tag matches and its V bit is 1 and the U bit of the way which is selected to replace using the LRU bits is 1, see No. 5.

3. Cache hit

   The data indexed by virtual address bits [4:0] is read from the data field of the cache line on the hitted way in accordance with the access size. Then the LRU bits are updated to indicate the hitted way is the latest one.
4. Cache miss (no write-back)

   Data is read into the cache line on the way, which is selected to replace, from the physical address space corresponding to the virtual address. Data reading is performed, using the wraparound method, in order from the quad-word data(8 bytes) including the cache-missed data. When the corresponding data arrives in the cache, the read data is returned to the CPU. While the remaining data on the cache line is being read, the CPU can execute the next processing. When reading of one line of data is completed, the tag corresponding to the physical address is recorded in the cache, 1 is written to the V bit and 0 is written to the U bit on the way. Then the LRU bit is updated to indicate the way is latest one.
5. Cache miss (with write-back)

   The tag and data field of the cache line on the way which is selected to replace are saved in the write-back buffer. Then data is read into the cache line on the way which is selected to replace from the physical address space corresponding to the virtual address. Data reading is performed, using the wraparound method, in order from the quad-word data (8 bytes) including the cache-missed data, and when the corresponding data arrives in the cache, the read data is returned to the CPU. While the remaining one cache line of data is being read, the CPU can execute the next processing. When reading of one line of data is completed, the tag corresponding to the physical address is recorded in the cache, 1 is written to the V bit, and 0 to the U bit. And the LRU bits are updated to indicate the way is latest one. The data in the write-back buffer is then written back to external memory.

8.3.2 Prefetch Operation

When the Operand Cache (OC) is enabled (OCE = 1 in CCR) and data is prefetched from a cacheable area, the cache operates as follows:

1. The tag, V bit, U bit, and LRU bits on each way are read from the cache line indexed by virtual address bits [12:5].
2. The tag, read from each way, is compared with bits [28:10] of the physical address resulting from virtual address translation by the MMU:

- If there is a way whose tag matches and its V bit is 1, see No. 3.
- If there is no way whose tag matches and the V bit is 1, and the U bit of the way which is selected to replace using the LRU bits is 0, see No. 4.
- If there is no way whose tag matches and the V bit is 1, and the U bit of the way which is selected to replace using the LRU bits is 1, see No. 5.

3. Cache hit

   Then the LRU bits are updated to indicate the hitted way is the latest one.
4. Cache miss (no write-back)

   Data is read into the cache line on the way, which is selected to replace, from the physical address space corresponding to the virtual address. Data reading is performed, using the wraparound method, in order from the quad-word data (8 bytes) including the cache-missed data. In the prefetch operation the CPU doesn't wait the data arrives. While the one cache line of data is being read, the CPU can execute the next processing. When reading of one line of data is completed, the tag corresponding to the physical address is recorded in the cache, 1 is written to the V bit and 0 is written to the U bit on the way. And the LRU bit is updated to indicate the way is latest one.
5. Cache miss (with write-back)

   The tag and data field of the cache line on the way which is selected to replace are saved in the write-back buffer. Then data is read into the cache line on the way which is selected to replace from the physical address space corresponding to the virtual address. Data reading is performed, using the wraparound method, in order from the quad-word data (8 bytes) including the cache-missed data. In the prefetch operation the CPU doesn't wait the data arrives. While the one cache line of data is being read, the CPU can execute the next processing. And the LRU bits are updated to indicate the way is latest one. The data in the write-back buffer is then written back to external memory.

8.3.3 Write Operation

When the Operand Cache (OC) is enabled (OCE = 1 in CCR) and data is written to a cacheable area, the cache operates as follows:

1. The tag, V bit, U bit, and LRU bits on each way are read from the cache line indexed by virtual address bits [12:5].
2. The tag, read from each way, is compared with bits [28:10] of the physical address resulting from virtual address translation by the MMU:

- If there is a way whose tag matches and its V bit is 1, see No. 3 for copy-back and No. 4 for write-through.
- I If there is no way whose tag matches and its V bit is 1 and the U bit of the way which is selected to replace using the LRU bits is 0, see No. 5 for copy-back and No. 7 for write-through.
- If there is no way whose tag matches and its V bit is 1 and the U bit of the way which is selected to replace using the LRU bits is 1, see No. 6 for copy-back and No. 7 for write-through.

3. Cache hit (copy-back)

   A data write in accordance with the access size is performed for the data field on the hit way which is indexed by virtual address bits [4:0]. Then 1 is written to the U bit. The LRU bits are updated to indicate the way is the latest one.
4. Cache hit (write-through)

   A data write in accordance with the access size is performed for the data field on the hit way which is indexed by virtual address bits [4:0]. A write is also performed to external memory corresponding to the virtual address. Then the LRU bits are updated to indicate the way is the latest one. In this case, the U bit isn't updated.
5. Cache miss (copy-back, no write-back)

   A data write in accordance with the access size is performed for the data field on the hit way which is indexed by virtual address bits [4:0]. Then, the data, excluding the cache-missed data which is written already, is read into the cache line on the way which is selected to replace from the physical address space corresponding to the virtual address.

   Data reading is performed, using the wraparound method, in order from the quad-word data (8 bytes) including the cache-missed data. While the remaining data on the cache line is being read, the CPU can execute the next processing. When reading of one line of data is completed, the tag corresponding to the physical address is recorded in the cache, 1 is written to the V bit and the U bit on the way. Then the LRU bit is updated to indicate the way is latest one.

6. Cache miss (copy-back, with write-back)

   The tag and data field of the cache line on the way which is selected to replace are saved in the write-back buffer. Then a data write in accordance with the access size is performed for the data field on the hit way which is indexed by virtual address bits [4:0]. Then, the data, excluding the cache-missed data which is written already, is read into the cache line on the way which is selected to replace from the physical address space corresponding to the virtual address. Data reading is performed, using the wraparound method, in order from the quad-word data (8 bytes) including the cache-missed data. While the remaining data on the cache line is being read, the CPU can execute the next processing. When reading of one line of data is completed, the tag corresponding to the physical address is recorded in the cache, 1 is written to the V bit and the U bit on the way. Then the LRU bit is updated to indicate the way is latest one. Then the data in the write-back buffer is then written back to external memory.

7. Cache miss (write-through)

   A write of the specified access size is performed to the external memory corresponding to the virtual address. In this case, a write to cache is not performed.

8.3.4 Write-Back Buffer

In order to give priority to data reads to the cache and improve performance, the SH-4A has a write-back buffer which holds the relevant cache entry when it becomes necessary to purge a dirty cache entry into external memory as the result of a cache miss. The write-back buffer contains one cache line of data and the physical address of the purge destination.

Physical address bits [28:5]	LW0	LW1	LW2	LW3	LW4	LW5	LW6	LW7

Figure 8.3 Configuration of Write-Back Buffer

8.3.5 Write-Through Buffer

The SH-4A has a 64-bit buffer for holding write data when writing data in write-through mode or writing to a non-cacheable area. This allows the CPU to proceed to the next operation as soon as the write to the write-through buffer is completed, without waiting for completion of the write to external memory.

Physical address bits[28:0]	LW0	LW1

Figure 8.4 Configuration of Write-Through Buffer

8.3.6 OC Two-Way Mode

When the OC2W bit in RAMCR is set to 1, OC two-way mode which only uses way 0 and way 1 in the OC is entered. Thus, power consumption can be reduced. In this mode, only way 0 and way 1 are used even if a memory-mapped OC access is made.

The OC2W bit should be modified by a program in the P2 area. At that time, if the valid line has already been recorded in the OC, data should be written back by software, if necessary, 1 should be written to the OCI bit in CCR, and all entries in the OC should be invalid before modifying the OC2W bit.

8.4 Instruction Cache Operation

8.4.1 Read Operation

When the IC is enabled (ICE = 1 in CCR) and instruction fetches are performed from a cacheable area, the instruction cache operates as follows:

1. The tag, V bit, U bit and LRU bits on each way are read from the cache line indexed by virtual address bits [12:5].
2. The tag, read from each way, is compared with bits [28:10] of the physical address resulting from virtual address translation by the MMU:

- If there is a way whose tag matches and the V bit is 1, see No. 3.
- If there is no way whose tag matches and the V bit is 1, see No. 4.

3. Cache hit

   The data indexed by virtual address bits [4:2] is read as an instruction from the data field on the hit way. The LRU bits are updated to indicate the way is the latest one.
4. Cache miss

   Data is read into the cache line on the way which selected using LRU bits to replace from the physical address space corresponding to the virtual address. Data reading is performed, using the wraparound method, in order from the quad-word data (8 bytes) including the cache-missed data, and when the corresponding data arrives in the cache, the read data is returned to the CPU as an instruction. While the remaining one cache line of data is being read, the CPU can execute the next processing. When reading of one line of data is completed, the tag corresponding to the physical address is recorded in the cache, and 1 is written to the V bit, the LRU bits are updated to indicate the way is the latest one.

8.4.2 Prefetch Operation

When the IC is enabled (ICE = 1 in CCR) and instruction prefetches are performed from a cacheable area, the instruction cache operates as follows:

1. The tag, V bit, Ubit and LRU bits on each way are read from the cache line indexed by virtual address bits [12:5].
2. The tag, read from each way, is compared with bits [28:10] of the physical address resulting from virtual address translation by the MMU:
- If there is a way whose tag matches and the V bit is 1, see No. 3.
- If there is no way whose tag matches and the V bit is 1, see No. 4.
3. Cache hit

   The LRU bits is updated to indicate the way is the latest one.
4. Cache miss

   Data is read into the cache line on a way which selected using the LRU bits to replace from the physical address space corresponding to the virtual address. Data reading is performed, using the wraparound method, in order from the quad-word data (8 bytes) including the cache-missed data. In the prefetch operation, the CPU doesn't wait the data arrived. While the one cache line of data is being read, the CPU can execute the next processing. When reading of one line of data is completed, the tag corresponding to the physical address is recorded in the cache, and 1 is written to the V bit, the LRU bits is updated to indicate the way is the latest one.

8.4.3 IC Two-Way Mode

When the IC2W bit in RAMCR is set to 1, IC two-way mode which only uses way 0 and way 1 in the IC is entered. Thus, power consumption can be reduced. In this mode, only way 0 and way 1 are used even if a memory-mapped IC access is made.

The IC2W bit should be modified by a program in the P2 area. At that time, if the valid line has already been recorded in the IC, 1 should be written to the ICI bit in CCR and all entries in the IC should be invalid before modifying the IC2W bit.

8.5 Cache Operation Instruction

8.5.1 Coherency between Cache and External Memory

Coherency between cache and external memory should be assured by software. In the SH-4A, the following six instructions are supported for cache operations. Details of these instructions are given in section 10, Instruction Descriptions.

- Operand cache invalidate instruction: OCBI @Rn
  Operand cache invalidation (no write-back)

- Operand cache purge instruction: OCBP @Rn
  Operand cache invalidation (with write-back)

- Operand cache write-back instruction: OCBWB @Rn
  Operand cache write-back

- Operand cache allocate instruction: MOVCA.L R0,@Rn
  Operand cache allocation

- Instruction cache invalidate instruction: ICBI @Rn
  Instruction cache invalidation

- Operand access synchronization instruction: SYNCO
  Wait for data transfer completion

The operand cache can receive "PURGE" and "FLUSH" transaction from SuperHyway bus to control the cache coherency. Since the address used by the PURGE and FLUSH transaction is a physical address, the following restrictions occur to avoid cache synonym problem in MMU enable mode.

- 1Kbyte page size cannot be used.

PURGE transaction: When the operand cache is enabled, the PURGE transaction checks the operand cache and invalidates the hit entry. If the invalidated entry is dirty, the data is written back to the external memory. If the transaction is not hit to the cache, it is no-operation.

FLUSH transaction: When the operand cache is enabled, the FLUSH transaction checks the operand cache and if the hit line is dirty, then the data is written back to the external memory. If the transaction is not hit to the cache or the hit entry is not dirty, it is no-operation.

8.5.2 Prefetch Operation

The SH-4A supports a prefetch instruction to reduce the cache fill penalty incurred as the result of a cache miss. If it is known that a cache miss will result from a read or write operation, it is possible to fill the cache with data beforehand by means of the prefetch instruction to prevent a cache miss due to the read or write operation, and so improve software performance. If a prefetch instruction is executed for data already held in the cache, or if the prefetch address results in a UTLB miss or a protection violation, the result is no operation, and an exception is not generated. Details of the prefetch instruction are given in section 10, Instruction Descriptions.

• Prefetch instruction (OC)	: PREF @Rn
• Prefetch instruction (IC)	: PREFI @Rn

8.6 Memory-Mapped Cache Configuration

To enable the IC and OC to be managed by software, the IC contents can be read from or written to by a program in the P2 area by means of a MOV instruction in privileged mode. Operation is not guaranteed if access is made from a program in another area. In this case, execute one of the following three methods for executing a branch to the P0, U0, P1, or P3 area.

1. Execute a branch using the RTE instruction.
2. Execute a branch to the P0, U0, P1, or P3 area after executing the ICBI instruction for any address (including non-cacheable area).
3. If the MC bit in IRMCR is 0 (initial value) before making an access to the memory-mapped IC, the specific instruction does not need to be executed. However, note that the CPU processing performance will be lowered because the instruction fetch is performed again for the next instruction after making an access to the memory-mapped IC.

Note that the method 3 may not be guaranteed in the future SuperH Series. Therefore, it is recommended that the method 1 or 2 should be used for being compatible with the future SuperH Series.

In privileged mode, the OC contents can be read from or written to by a program in the P1 or P2 area by means of a MOV instruction. Operation is not guaranteed if access is made from a program in another area. The IC and OC are allocated to the P4 area in the virtual address space. Only data accesses can be used on both the IC address array and data array and the OC address array and data array, and accesses are always longword-size. Instruction fetches cannot be performed in these areas. For reserved bits, a write value of 0 should be specified and the read value is undefined.

8.6.1 IC Address Array

The IC address array is allocated to addresses H'F000 0000 to H'F0FF FFFF in the P4 area. An address array access requires a 32-bit address field specification (when reading or writing) and a 32-bit data field specification. The way and entry to be accessed are specified in the address field, and the write tag and V bit are specified in the data field.

In the address field, bits [31:24] have the value H'F0 indicating the IC address array, and the way is specified by bits [14:13] and the entry by bits [12:5]. The association bit (A bit) [3] in the address field specifies whether or not association is performed when writing to the IC address array. As only longword access is used, 0 should be specified for address field bits [1:0].

In the data field, the tag is indicated by bits [31:10], and the V bit by bit [0]. As the IC address array tag is 19 bits in length, data field bits [31:29] are not used in the case of a write in which association is not performed. Data field bits [31:29] are used for the virtual address specification only in the case of a write in which association is performed.

The following three kinds of operation can be used on the IC address array:

1. IC address array read

   The tag and V bit are read into the data field from the IC entry corresponding to the way and entry set in the address field. In a read, associative operation is not performed regardless of whether the association bit specified in the address field is 1 or 0.
2. IC address array write (non-associative)

   The tag and V bit specified in the data field are written to the IC entry corresponding to the way and entry set in the address field. The A bit in the address field should be cleared to 0.
3. IC address array write (associative)

   When a write is performed with the A bit in the address field set to 1, the tag in each way stored in the entry specified in the address field is compared with the tag specified in the data field. The way numbers of bits [14:13] in the address field are not used. If the MMU is enabled at this time, comparison is performed after the virtual address specified by data field bits [31:10] has been translated to a physical address using the ITLB. If the addresses match and the V bit in the way is 1, the V bit specified in the data field is written into the IC entry. In other cases, no operation is performed. This operation is used to invalidate a specific IC entry. If an ITLB miss occurs during address translation, or the comparison shows a mismatch, an exception is not generated, no operation is performed, and the write is not executed.

Note: This function may not be supported in the future SuperH Series. Therefore, it is recommended that the ICBI instruction should be used to operate the IC definitely by handling ITLB miss and reporting ITLB miss exception.

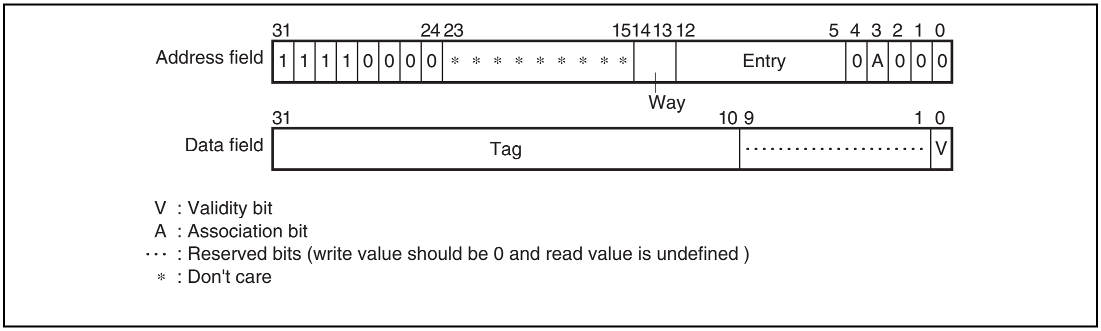


Figure 8.5 Memory-Mapped IC Address Array

8.6.2 IC Data Array

The IC data array is allocated to addresses H'F100 0000 to H'F1FF FFFF in the P4 area. A data array access requires a 32-bit address field specification (when reading or writing) and a 32-bit data field specification. The way and entry to be accessed are specified in the address field, and the longword data to be written is specified in the data field.

In the address field, bits [31:24] have the value H'F1 indicating the IC data array, and the way is specified by bits [14:13] and the entry by bits [12:5]. Address field bits [4:2] are used for the longword data specification in the entry. As only longword access is used, 0 should be specified for address field bits [1:0].

The data field is used for the longword data specification.

The following two kinds of operation can be used on the IC data array:

1. IC data array read

   Longword data is read into the data field from the data specified by the longword specification bits in the address field in the IC entry corresponding to the way and entry set in the address field.

2. IC data array write

   The longword data specified in the data field is written for the data specified by the longword specification bits in the address field in the IC entry corresponding to the way and entry set in the address field.

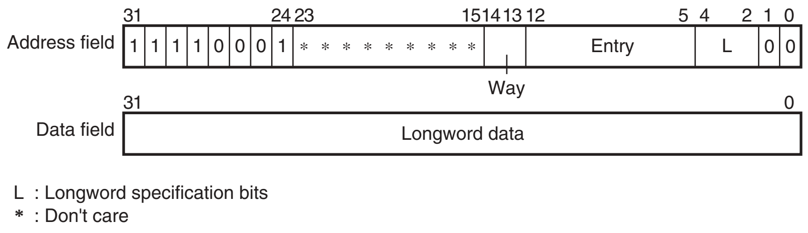


Figure 8.6 Memory-Mapped IC Data Array

8.6.3 OC Address Array

The OC address array is allocated to addresses H'F400 0000 to H'F4FF FFFF in the P4 area. An address array access requires a 32-bit address field specification (when reading or writing) and a 32-bit data field specification. The way and entry to be accessed are specified in the address field, and the write tag, U bit, and V bit are specified in the data field.

In the address field, bits [31:24] have the value H'F4 indicating the OC address array, and the way is specified by bits [14:13] and the entry by bits [12:5]. The association bit (A bit) [3] in the address field specifies whether or not association is performed when writing to the OC address array. As only longword access is used, 0 should be specified for address field bits [1:0].

In the data field, the tag is indicated by bits [31:10], the U bit by bit [1], and the V bit by bit [0]. As the OC address array tag is 19 bits in length, data field bits [31:29] are not used in the case of a write in which association is not performed. Data field bits [31:29] are used for the virtual address specification only in the case of a write in which association is performed.

The following three kinds of operation can be used on the OC address array:

1. OC address array read

   The tag, U bit, and V bit are read into the data field from the OC entry corresponding to the way and entry set in the address field. In a read, associative operation is not performed regardless of whether the association bit specified in the address field is 1 or 0.

2. OC address array write (non-associative)

   The tag, U bit, and V bit specified in the data field are written to the OC entry corresponding to the way and entry set in the address field. The A bit in the address field should be cleared to 0.

   When a write is performed to a cache line for which the U bit and V bit are both 1, after write-back of that cache line, the tag, U bit, and V bit specified in the data field are written.

3. OC address array write (associative)

   When a write is performed with the A bit in the address field set to 1, the tag in each way stored in the entry specified in the address field is compared with the tag specified in the data field. The way numbers of bits [14:13] in the address field are not used. If the MMU is enabled at this time, comparison is performed after the virtual address specified by data field bits [31:10] has been translated to a physical address using the UTLB. If the addresses match and the V bit in the way is 1, the U bit and V bit specified in the data field are written into the OC entry. In other cases, no operation is performed. This operation is used to invalidate a specific OC entry. If the OC entry U bit is 1, and 0 is written to the V bit or to the U bit, write-back is performed. If a UTLB miss occurs during address translation, or the comparison shows a mismatch, an exception is not generated, no operation is performed, and the write is not executed.

Note: This function may not be supported in the future SuperH Series. Therefore, it is recommended that the OCBI, OCBP, or OCBWB instruction should be used to operate the OC definitely by reporting data TLB miss exception.

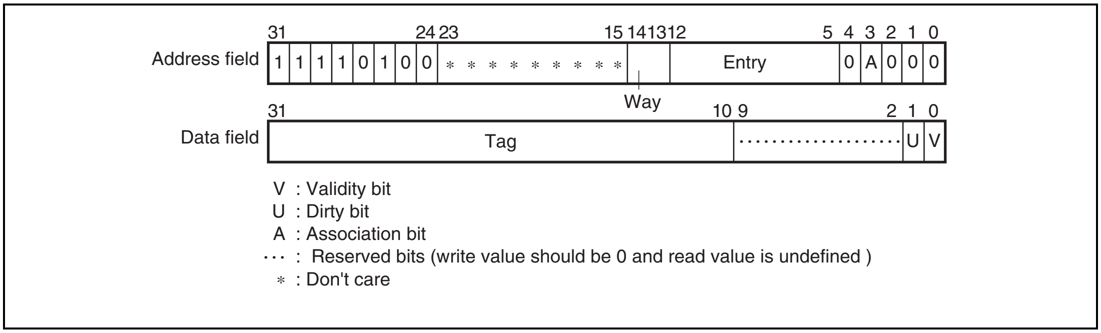


Figure 8.7 Memory-Mapped OC Address Array

8.6.4 OC Data Array

The OC data array is allocated to addresses H'F500 0000 to H'F5FF FFFF in the P4 area. A data array access requires a 32-bit address field specification (when reading or writing) and a 32-bit data field specification. The way and entry to be accessed are specified in the address field, and the longword data to be written is specified in the data field.

In the address field, bits [31:24] have the value H'F5 indicating the OC data array, and the way is specified by bits [14:13] and the entry by bits [12:5]. Address field bits [4:2] are used for the longword data specification in the entry. As only longword access is used, 0 should be specified for address field bits [1:0].

The data field is used for the longword data specification.

The following two kinds of operation can be used on the OC data array:

1. OC data array read

   Longword data is read into the data field from the data specified by the longword specification bits in the address field in the OC entry corresponding to the way and entry set in the address field.

2. OC data array write

   The longword data specified in the data field is written for the data specified by the longword specification bits in the address field in the OC entry corresponding to the way and entry set in the address field. This write does not set the U bit to 1 on the address array side.

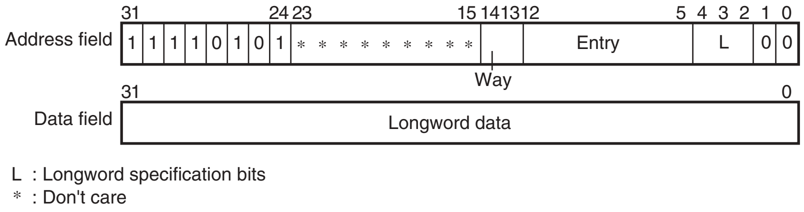


Figure 8.8 Memory-Mapped OC Data Array

8.7 Store Queues

The SH-4A supports two 32-byte store queues (SQs) to perform high-speed writes to external memory.

8.7.1 SQ Configuration

There are two 32-byte store queues, SQ0 and SQ1, as shown in figure 8.9. These two store queues can be set independently.

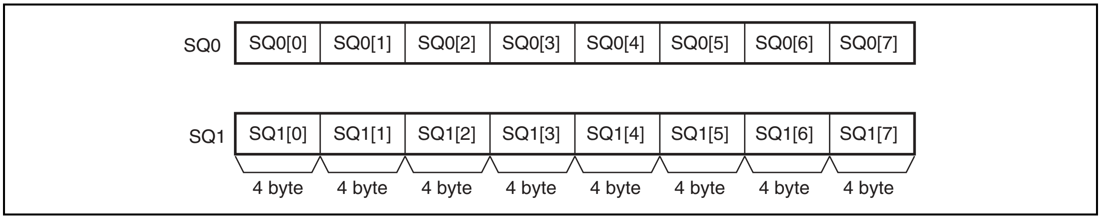


Figure 8.9 Store Queue Configuration

8.7.2 Writing to SQ

A write to the SQs can be performed using a store instruction for addresses H'E000 0000 to H'E3FF FFFC in the P4 area. A longword or quadword access size can be used. The meanings of the address bits are as follows:

[31:26]	: 111000	Store queue specification
[25:6]	: Don't care	Used for external memory transfer/access right
[5]	: 0/1	0: SQ0 specification 1: SQ1 specification
[4:2]	: LW specification	Specifies longword position in SQ0/SQ1
[1:0]	: 00	Fixed at 0

8.7.3 Transfer to External Memory

Transfer from the SQs to external memory can be performed with a prefetch instruction (PREF). Issuing a PREF instruction for addresses H'E000 0000 to H'E3FF FFFC in the P4 area starts a transfer from the SQs to external memory. The transfer length is fixed at 32 bytes, and the start address is always at a 32-byte boundary. While the contents of one SQ are being transferred to external memory, the other SQ can be written to without a penalty cycle. However, writing to the SQ involved in the transfer to external memory is kept waiting until the transfer is completed.

The physical address bits [28:0] of the SQ transfer destination are specified as shown below, according to whether the MMU is enabled or disabled.

- When MMU is enabled (AT = 1 in MMUCR)

  The SQ area (H'E000 0000 to H'E3FF FFFF) is set in VPN of the UTLB, and the transfer destination physical address in PPN. The ASID, V, SZ, SH, PR, and D bits have the same meaning as for normal address translation, but the C and WT bits have no meaning with regard to this page. When a prefetch instruction is issued for the SQ area, address translation is performed and physical address bits [28:10] are generated in accordance with the SZ bit specification. For physical address bits [9:5], the address prior to address translation is generated in the same way as when the MMU is disabled. Physical address bits [4:0] are fixed at 0. Transfer from the SQs to external memory is performed to this address.
- When MMU is disabled (AT = 0 in MMUCR)

  The SQ area (H'E000 0000 to H'E3FF FFFF) is specified as the address at which a PREF instruction is issued. The meanings of address bits [31:0] are as follows:

[31:26]	: 111000	Store queue specification
[25:6]	: Address	Transfer destination physical address bits [25:6]
[5]	: 0/1	0: SQ0 specification 1: SQ1 specification and transfer destination physical address bit [5]
[4:2]	: Don't care	No meaning in a prefetch
[1:0]	: 00	Fixed at 0

  Physical address bits [28:26], which cannot be generated from the above address, are generated from QACR0 and QACR1.

QACR0[4:2]	: Physical address bits [28:26] corresponding to SQ0
QACR1[4:2]	: Physical address bits [28:26] corresponding to SQ1

  Physical address bits [4:0] are always fixed at 0 since burst transfer starts at a 32-byte boundary.

8.7.4 Determination of SQ Access Exception

Determination of an exception in a write to an SQ or transfer to external memory (PREF instruction) is performed as follows according to whether the MMU is enabled or disabled. If an exception occurs during a write to an SQ, the SQ contents before the write are retained. If an exception occurs in a data transfer from an SQ to external memory, the transfer to external memory will be aborted.

- When MMU is enabled (AT = 1 in MMUCR)

  Operation is in accordance with the address translation information recorded in the UTLB, and the SQMD bit in MMUCR. Write type exception judgment is performed for writes to the SQs, and read type exception judgment for transfer from the SQs to external memory (using a PREF instruction). As a result, a TLB miss exception or protection violation exception is generated as required. However, if SQ access is enabled in privileged mode only by the SQMD bit in MMUCR, an address error will occur even if address translation is successful in user mode.

- When MMU is disabled (AT = 0 in MMUCR)

  Operation is in accordance with the SQMD bit in MMUCR.

  0: Privileged/user mode access possible

  1: Privileged mode access possible

  If the SQ area is accessed in user mode when the SQMD bit in MMUCR is set to 1, an address error will occur.

8.7.5 Reading from SQ

In privileged mode in the SH-4A, reading the contents of the SQs may be performed by means of a load instruction for addresses H'FF00 1000 to H'FF00 103C in the P4 area. Only longword access is possible.

[31:6]	: H'FF00 1000	Store queue specification
[5]	: 0/1	0: SQ0 specification 1: SQ1 specification
[4:2]	: LW specification	Specifies longword position in SQ0/SQ1
[1:0]	: 00	Fixed at 0

8.8 Notes on Using 32-Bit Address Extended Mode

In 32-bit address extended mode, the items described in this section are extended as follows.

1. The tag bits [28:10] (19 bits) in the IC and OC are extended to bits [31:10] (22 bits).
2. An instruction which operates the IC (a memory-mapped IC access and writing to the ICI bit in CCR) should be located in the P1 or P2 area. The cacheable bit (C bit) in the corresponding entry in the PMB should be 0.
3. Bits [4:2] (3 bits) for the AREA0 bit in QACR0 and the AREA1 bit in QACR1 are extended to bits [7:2] (6 bits).

Section 9 L Memory

The SH-4A includes on-chip L-memory which stores instructions or data.

Note: For the size of L-memory, see the hardware manual of the target product.

9.1 Features

- Capacity

  Total L memory can be selected from 16 Kbytes, 32 Kbytes, 64 Kbytes, or 128 Kbytes.
- Page

  The L memory is divided into two pages (pages 0 and 1).
- Memory map

  The L memory is allocated in the addresses shown in table 9.1 in both the virtual address space and the physical address space.

Table 9.1 L Memory Addresses

	Memory Size (Two Pages Total)			
Page	**16 Kbytes**	**32 Kbytes**	**64 Kbytes**	**128 Kbytes**
Page 0 of L memory	H'E500E000 to H'E500FFFF	H'E500C000 to H'E500FFFF	H'E5008000 to H'E500FFFF	H'E5000000 to H'E500FFFF
Page 1 of L memory	H'E5010000 to H'E5011FFF	H'E5010000 to H'E5013FFF	H'E5010000 to H'E5017FFF	H'E5010000 to H'E501FFFF

- Ports

  Each page has three independent read/write ports and is connected to each bus. The instruction bus is used when L memory is accessed through instruction fetch. The operand bus is used when L memory is accessed through operand access. The SuperHyway bus is used for L memory access from the SuperHyway bus master module.
- Priority

  In the event of simultaneous accesses to the same page from different buses, the access requests are processed according to priority. The priority order is: SuperHyway bus > operand bus > instruction bus.

9.2 Register Descriptions

The following registers are related to L memory.

Table 9.2 Register Configuration

Name	Abbreviation	R/W	P4 Address*	Area 7 Address*	Access Size
On-chip memory control register	RAMCR	R/W	H'FF000074	H'1F000074	32
L memory transfer source address register 0	LSA0	R/W	H'FF000050	H'1F000050	32
L memory transfer source address register 1	LSA1	R/W	H'FF000054	H'1F000054	32
L memory transfer destination address register 0	LDA0	R/W	H'FF000058	H'1F000058	32
L memory transfer destination address register 1	LDA1	R/W	H'FF00005C	H'1F00005C	32

Note: * The P4 address is the address used when using P4 area in the virtual address space. The area 7 address is the address used when accessing from area 7 in the physical address space using the TLB.

Table 9.3 Register Status in Each Processing State

Name	Abbreviation	Power-On Reset	Manual Reset	Sleep	Standby
On-chip memory control register	RAMCR	H'00000000	H'00000000	Retained	Retained
L memory transfer source address register 0	LSA0	Undefined	Undefined	Retained	Retained
L memory transfer source address register 1	LSA1	Undefined	Undefined	Retained	Retained
L memory transfer destination address register 0	LDA0	Undefined	Undefined	Retained	Retained
L memory transfer destination address register 1	LDA1	Undefined	Undefined	Retained	Retained

9.2.1 On-Chip Memory Control Register (RAMCR)

RAMCR controls the protective functions in the L memory.

Bit :	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
Initial value :	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
R/W:	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R

Bit :	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	—	—	—	—	—	—	RMD	RP	IC2W	OC2W	—	—	—	—	—	—
Initial value :	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
R/W:	R	R	R	R	R	R	R/W	R/W	R/W	R/W	R	R	R	R	R	R

Bit	Bit Name	Initial Value	R/W	Description
31to10	—	All 0	R	Reserved For read/write in these bits, refer to General Precautions on Handling of Product.
9	RMD	0	R/W	On-Chip Memory Access Mode Specifies the right of access to the L memory from the virtual address space. 0: An access in privileged mode is allowed. (An address error exception occurs in user mode.) 1: An access in user/ privileged mode is allowed.
8	RP	0	R/W	On-Chip Memory Protection Enable Selects whether or not to use the protective functions using ITLB and UTLB for accessing the L memory from the virtual address space. 0: Protective functions are not used. 1: Protective functions are used. For further details, refer to section 9.4, L Memory Protective Functions.
7	IC2W	0	R/W	IC Two-Way Mode For further details, refer to section 8.4.3, IC Two-Way Mode.
6	OC2W	0	R/W	OC Two-Way Mode For further details, refer to section 8.3.6, OC Two-Way Mode.
5 to 0	—	All 0	R	Reserved For read/write in these bits, refer to General Precautions on Handling of Product.

9.2.2 L Memory Transfer Source Address Register 0 (LSA0)

When MMUCR.AT = 0 or RAMCR.RP = 0, the LSA0 specifies the transfer source physical address for block transfer to page 0 of the L memory.

Bit :	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	—	—	—	L0SADR												
Initial value :	0	0	0	—	—	—	—	—	—	—	—	—	—	—	—	—
R/W:	R	R	R	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

Bit :	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	L0SADR						—	—	—	—	L0SSZ					
Initial value :	—	—	—	—	—	—	0	0	0	0	—	—	—	—	—	—
R/W:	R/W	R/W	R/W	R/W	R/W	R/W	R	R	R	R	R/W	R/W	R/W	R/W	R/W	R/W

Bit	Bit Name	Initial Value	R/W	Description
31 to 29	—	All 0	R	Reserved For read/write in these bits, refer General Precautions on Handling of Product.
28 to 10	L0SADR	Undefined	R/W	L Memory Page 0 Block Transfer Source Address When MMUCR.AT = 0 or RAMCR.RP = 0, these bits specify the transfer source physical address for block transfer to page 0 in the L memory.
9 to 6	—	All 0	R	Reserved For read/write in these bits, refer to General Precautions on Handling of Product.

Bit	Bit Name	Initial Value	R/W	Description
5 to 0	L0SSZ	Undefined	R/W	L Memory Page 0 Block Transfer Source Address Select When MMUCR.AT = 0 or RAMCR.RP = 0, these bits select whether the operand addresses or L0SADR values are used as bits 15 to 10 of the transfer source physical address for block transfer to the L memory. L0SSZ[5:0] correspond to the transfer source physical addresses [15:10]. 0: The operand address is used as the transfer source physical address. 1: The L0SADR value is used as the transfer source physical address. Settable values: 111111: Transfer source physical address is specified in 1-Kbyte units. 111110: Transfer source physical address is specified in 2-Kbyte units. 111100: Transfer source physical address is specified in 4-Kbyte units. 111000: Transfer source physical address is specified in 8-Kbyte units. 110000: Transfer source physical address is specified in 16-Kbyte units. 100000: Transfer source physical address is specified in 32-Kbyte units. 000000: Transfer source physical address is specified in 64-Kbyte units. Settings other than the ones given above are prohibited.

9.2.3 L Memory Transfer Source Address Register 1 (LSA1)

When MMUCR.AT = 0 or RAMCR.RP = 0, the LSA1 specifies the transfer source physical address for block transfer to page 1 in the L memory.

Bit :	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	—	—	—	L1SADR												
Initial value :	0	0	0	—	—	—	—	—	—	—	—	—	—	—	—	—
R/W:	R	R	R	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

Bit :	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	L1SADR						—	—	—	—	L1SSZ					
Initial value :	—	—	—	—	—	—	0	0	0	0	—	—	—	—	—	—
R/W:	R/W	R/W	R/W	R/W	R/W	R/W	R	R	R	R	R/W	R/W	R/W	R/W	R/W	R/W

Bit	Bit Name	Initial Value	R/W	Description
31 to 29	—	All 0	R	Reserved For read/write in these bits, refer to General Precautions on Handling of Product.
28 to 10	L1SADR	Undefined	R/W	L Memory Page 1 Block Transfer Source Address When MMUCR.AT = 0 or RAMCR.RP = 0, these bits specify transfer source physical address for block transfer to page 1 in the L memory.
9 to 6	—	All 0	R	Reserved For read/write in these bits, refer to General Precautions on Handling of Product.
5 to 0	L1SSZ	Undefined	R/W	L Memory Page 1 Block Transfer Source Address Select When MMUCR.AT = 0 or RAMCR.RP = 0, these bits select whether the operand addresses or L1SADR values are used as bits 15 to 10 of the transfer source physical address for block transfer to page 1 in the L memory. L1SSZ bits [5:0] correspond to the transfer source physical addresses [15:10]. 0: The operand address is used as the transfer source physical address. 1: The L1SADR value is used as the transfer source physical address. Settable values: 111111: Transfer source physical address is specified in 1-Kbyte units. 111110: Transfer source physical address is specified in 2-Kbyte units. 111100: Transfer source physical address is specified in 4-Kbyte units. 111000: Transfer source physical address is specified in 8-Kbyte units. 110000: Transfer source physical address is specified in 16-Kbyte units. 100000: Transfer source physical address is specified in 32-Kbyte units. 000000: Transfer source physical address is specified in 64-Kbyte units. Settings other than the ones given above are prohibited.

9.2.4 L Memory Transfer Destination Address Register 0 (LDA0)

When MMUCR.AT = 0 or RAMCR.RP = 0, LDA0 specifies the transfer destination physical address for block transfer to page 0 of the L memory.

Bit :	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	—	—	—	L0DADR												
Initial value :	0	0	0	—	—	—	—	—	—	—	—	—	—	—	—	—
R/W:	R	R	R	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

Bit :	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	L0DADR						—	—	—	—	L0DSZ					
Initial value :	—	—	—	—	—	—	0	0	0	0	—	—	—	—	—	—
R/W:	R/W	R/W	R/W	R/W	R/W	R/W	R	R	R	R	R/W	R/W	R/W	R/W	R/W	R/W

Bit	Bit Name	Initial Value	R/W	Description
31 to 29	—	All 0	R	Reserved For read/write in these bits, refer to General Precautions on Handling of Product.
28 to 10	L0DADR	Undefined	R/W	L Memory Page 0 Block Transfer Destination Address When MMUCR.AT = 0 or RAMCR.RP = 0, these bits specify transfer destination physical address for block transfer to page 0 in the L memory.
9 to 6	—	All 0	R	Reserved For read/write in these bits, refer to General Precautions on Handling of Product.

Bit	Bit Name	Initial Value	R/W	Description
5 to 0	L0DSZ	Undefined	R/W	L Memory Page 0 Block Transfer Destination Address Select When MMUCR.AT = 0 or RAMCR.RP = 0, these bits select whether the operand addresses or L0DADR values are used as bits 15 to 10 of the transfer destination physical address for block transfer to page 0 in the L memory. L0DSZ bits [5:0] correspond to the transfer destination physical address bits [15:10]. 0: The operand address is used as the transfer destination physical address. 1: The L0DADR value is used as the transfer destination physical address. Settable values: 111111: Transfer destination physical address is specified in 1-Kbyte units. 111110: Transfer destination physical address is specified in 2-Kbyte units. 111100: Transfer destination physical address is specified in 4-Kbyte units. 111000: Transfer destination physical address is specified in 8-Kbyte units. 110000: Transfer destination physical address is specified in 16-Kbyte units. 100000: Transfer destination physical address is specified in 32-Kbyte units. 000000: Transfer destination physical address is specified in 64-Kbyte units. Settings other than the ones given above are prohibited.

9.2.5 L Memory Transfer Destination Address Register 1 (LDA1)

When MMUCR.AT = 0 or RAMCR.RP = 0, LDA1 specifies the transfer destination physical address for block transfer to page 1 in the L memory.

Bit :	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	—	—	—	L1DADR												
Initial value :	0	0	0	—	—	—	—	—	—	—	—	—	—	—	—	—
R/W:	R	R	R	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

Bit :	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	L1DADR						—	—	—	—	L1DSZ					
Initial value :	—	—	—	—	—	—	0	0	0	0	—	—	—	—	—	—
R/W:	R/W	R/W	R/W	R/W	R/W	R/W	R	R	R	R	R/W	R/W	R/W	R/W	R/W	R/W

Bit	Bit Name	Initial Value	R/W	Description
31 to 29	—	All 0	R	Reserved For read/write in these bits, refer to General Precautions on Handling of Product.
28 to 10	L1DADR	Undefined	R/W	L Memory Page 1 Block Transfer Destination Address When MMUCR.AT = 0 or RAMCR.RP = 0, these bits specify transfer destination physical address for block transfer to page 1 in the L memory.
9 to 6	—	All 0	R	Reserved For read/write in these bits, refer to General Precautions on Handling of Product.

Bit	Bit Name	Initial Value	R/W	Description
5 to 0	L1DSZ	Undefined	R/W	L Memory Page 1 Block Transfer Destination Address Select When MMUCR.AT = 0 or RAMCR.RP = 0, these bits select whether the operand addresses or L1DADR values are used as bits 15 to 10 of the transfer destination physical address for block transfer to page 1 in the L memory. L1DSZ bits [5:0] correspond to the transfer destination physical addresses [15:10]. 0: The operand address is used as the transfer destination physical address. 1: The L1DADR value is used as the transfer destination physical address. Settable values: 111111: Transfer destination physical address is specified in 1-Kbyte units. 111110: Transfer destination physical address is specified in 2-Kbyte units. 111100: Transfer destination physical address is specified in 4-Kbyte units. 111000: Transfer destination physical address is specified in 8-Kbyte units. 110000: Transfer destination physical address is specified in 16-Kbyte units. 100000: Transfer destination physical address is specified in 32-Kbyte units. 000000: Transfer destination physical address is specified in 64-Kbyte units. Settings other than the ones given above are prohibited.

9.3 Operation

9.3.1 Access from the CPU and FPU

L memory access from the CPU and FPU is direct via the instruction bus and operand bus by means of the virtual address. As long as there is no conflict on the page, the L memory is accessed in one cycle.

9.3.2 Access from the SuperHyway Bus Master Module

L memory is always accessed by the SuperHyway bus master module, such as DMAC, via the SuperHyway bus which is a physical address bus. The same addresses as for the virtual addresses must be used.

9.3.3 Block Transfer

High-speed data transfer can be performed through block transfer between the L memory and external memory without cache utilization.

Data can be transferred from the external memory to the L memory through a prefetch instruction (PREF). Block transfer from the external memory to the L memory begins when the PREF instruction is issued to the address in the L memory area in the virtual address space.

Data can be transferred from the L memory to the external memory through a write-back instruction (OCBWB). Block transfer from the L memory to the external memory begins when the OCBWB instruction is issued to the address in the L memory area in the virtual address space.

In either case, transfer rate is fixed to 32 bytes. Since the start address is always limited to a 32-byte boundary, the lower five bits of the address indicated by Rn are ignored, and are always dealt with as all 0s. In either case, other pages and cache can be accessed during block transfer, but the CPU will stall if the page which is being transferred is accessed before data transfer ends.

The physical addresses [28:0] of the external memory performing data transfers with the L memory are specified as follows according to whether the MMU is enabled or disabled.

When MMU is Enabled (MMUCR.AT = 1) and RAMCR.RP = 1: An address of the L memory area is specified to the UTLB VPN field, and to the physical address of the transfer source (in the case of the PREF instruction) or the transfer destination (in the case of the OCBWB instruction) to the PPN field. The ASID, V, SZ, SH, PR, and D bits have the same meaning as normal address conversion; however, the C and WT bits have no meaning in this page.

When the PREF instruction is issued to the L memory area, address conversion is performed in order to generate the physical address bits [28:10] in accordance with the SZ bit specification. The physical address bits [9:5] are generated from the virtual address prior to address conversion. The physical address bits [4:0] are fixed to 0. Block transfer is performed to the L memory from the external memory which is specified by these physical addresses.

When the OCBWB instruction is issued to the L memory area, address conversion is performed in order to generate the physical address bits [28:10] in accordance with the SZ bit specification. The physical address bits [9:5] are generated from the virtual address prior to address conversion. The physical address bits [4:0] are fixed to 0. Block transfer is performed from the L memory to the external memory specified by these physical addresses.

In PREF or OCBWB instruction execution, an MMU exception is checked as read type. After the MMU execution check, a TLB miss exception or protection error exception occurs if necessary. If an exception occurs, the block transfer is inhibited.

When MMU is Disabled (MMUCR.AT = 0) or RAMCR.RP = 0: The transfer source physical address in block transfer to page 0 in the L memory is set in the L0SADR bits of the LSA0 register. And the L0SSZ bits in the LSA0 register choose either the virtual addresses specified through the PRFF instruction or the L0SADR values as bits 15 to 10 of the transfer source physical address. In other words, the transfer source area can be specified in units of 1 Kbyte to 64 Kbytes.

The transfer destination physical address in block transfer from page 0 in the L memory is set in the L0DADR bits of the LDA0 register. And the L0DSZ bits in the LDA0 register choose either the virtual addresses specified through the OCBWB instruction or the L0DADR values as bits 15 to 10 of the transfer destination physical address. In other words, the transfer source area can be specified in units of 1 Kbyte to 64 Kbytes.

Block transfer to page 1 in the L memory is set to LSA1 and LDA1 as with page 0 in the L memory.

When the PREF instruction is issued to the L memory area, the physical address bits [28:10] are generated in accordance with the LSA0 or LSA1 specification. The physical address bits [9:5] are generated from the virtual address. The physical address bits [4:0] are fixed to 0. Block transfer is performed from the external memory specified by these physical addresses to the L memory.

When the OCBWB instruction is issued to the L memory area, the physical address bits [28:10] are generated in accordance with the LDA0 or LDA1 specification. The physical address bits [9:5] are generated from the virtual address. The physical address bits [4:0] are fixed to 0. Block transfer is performed from the L memory to the external memory specified by these physical addresses.

9.4 L Memory Protective Functions

The SH-4A implements the following protective functions to the L memory by using the on-chip memory access mode bit (RMD) and the on-chip memory protection enable bit (RP) in the on-chip memory control register (RAMCR).

- Protective functions for access from the CPU and FPU

  When RAMCR.RMD = 0, and the L memory is accessed in user mode, it is determined to be an address error exception.

  When MMUCR.AT = 1 and RAMCR.RP = 1, MMU exception and address error exception are checked in the L memory area which is a part of area P4 as with the area P0/P3/U0.

The above descriptions are summarized in table 9.4.

Table 9.4 Protective Function Exceptions to Access L Memory

MMUCR.AT	RAMCR.RP	SR.MD	RAMCR. RMD	Always Occurring Exceptions	Possibly Occurring Exceptions
0	*	0	0	Address error exception	—
			1	—	—
		1	*	—	—
1	0	0	0	Address error exception	—
			1	—	—
		1	*	—	—
	1	0	0	Address error exception	—
			1	—	MMU exception
		1	*	—	MMU exception

[Legend] *: Don't care

9.5 Usage Notes

9.5.1 Page Conflict

In the event of simultaneous access to the same page from different buses, page conflict occurs. Although each access is completed correctly, this kind of conflict tends to lower L memory accessibility. Therefore it is advisable to provide all possible preventative software measures. For example, conflicts will not occur if each bus accesses different pages.

9.5.2 L Memory Coherency

In order to allocate instructions in the L memory, write an instruction to the L memory, execute the following sequence, then branch to the rewritten instruction.

- SYNCO
- ICBI @Rn

In this case, the target for the ICBI instruction can be any address (L memory address may be possible) within the range where no address error exception occurs, and cache hit/miss is possible.

9.5.3 Sleep Mode

The SuperHyway bus master module, such as DMAC, cannot access L memory in sleep mode.

9.6 Note on Using 32-Bit Address Extended Mode

In 32-bit address extended mode, L0SADR fields in LSA0, L1SADR fields in LSA1, L0DADR fields in LDA0, and L1DADR fields in LDA1 are extended from 19-bit [28:10] to 22-bit [31:10].

Section 10 Instruction Descriptions

This section describes instructions in alphabetical order using the format shown below.

Instruction Name (Full Name): Instruction Type (Indication of delayed branch instruction or interrupt-disabling instruction)

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

Description:

Description of operation

Notes:

Notes on using the instruction

Operation:

Operation written in C language

Examples:

An example is shown using assembler mnemonics, indicating the states before and after execution of the instruction.
Italics (e.g., .align) indicate an assembler control instruction. The meaning of the assembler control instructions is given below. For details, refer to the C/C++ Compiler, Assembler, Optimizing linkage editor User's Manual.

.org	Location counter setting
.data.w	Word integer data allocation
.data.1	Longword integer data allocation
.sdata	String data allocation
.align 2	2-byte boundary alignment
.align 4	4-byte boundary alignment
.align 32	32-byte boundary alignment
.arepeat 16	16-times repeat expansion
.arepeat 32	32-times repeat expansion
.aendr	Count-specification repeat expansion end

Note: SH Series cross-assembler version 1.0 does not support conditional assembler function

Possible Exceptions:

A list of exceptions that may occur when an instruction is executed is shown bellow.
But "Instruction TLB multiple-hit exception", "Instruction TLB miss exception", "Instruction TLB protection exception", and "Instruction address error" are omitted because these exceptions may occur in all instructions. As for the overflow/underflow exceptions, the detailed occurrence conditions are also described.

10.1 CPU instruction

Note: Of the SH-4A's section, CPU instructions, those which support the FPU or differ functionally from those of the SH4AL-DSP are described in section 10.2, CPU instructions (FPU Related). The other instructions are described in section 10.1, CPU instructions.

The following resources and functions are used in C-language descriptions of the operation of CPU instructions.

```
char     8-bit integer
short    16-bit integer
int      32-bit integer
long     64-bit integer
float    single-precision floating point number(32 bits)
double   double-precision floating point number(64 bits)
```

These are data types.

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

These reflect the respective sizes of address Addr. A word read from other than a 2n address, or a longword read from other than a 4n address, will be detected as an address error.

```
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);
```

These write data Data to address Addr, using the respective sizes. A word write to other than a 2n address, or a longword write to other than a 4n address, will be detected as an address error.

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

Shifts to execution of the slot instruction at address (Addr).

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

Registers

```
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;
};
```

SR structure definitions

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

Definitions of bits in SR

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

Error display function

10.1.1 ADD (Add binary): Arithmetic Instruction

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

Description: This instruction adds together the contents of general registers Rn and Rm and stores the result in Rn.

8-bit immediate data can also be added to the contents of general register Rn.

8-bit immediate data is sign-extended to 32 bits, allowing use in decrement operations.

Notes: None

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;
}
```

Example:

```
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
```

10.1.2 ADDC (Add with Carry): Arithmetic Instruction

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

Description: This instruction adds together the contents of general registers Rn and Rm and the T bit, and stores the result in Rn. A carry resulting from the operation is reflected in the T bit. This instruction is used for additions exceeding 32 bits.

Notes: None

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;
}
```

Example:

```
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'00000000
ADDC   R2,R0        ; Before execution  T = 1, R0 = H'00000000, R2 = H'00000000
                    ; After execution   T = 0, R0 = H'00000001
```

10.1.3 ADDV (Add with (V flag) Overflow Check): Arithmetic Instruction

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

Description: This instruction adds together the contents of general registers Rn and Rm and stores the result in Rn. If overflow occurs, the T bit is set.

Notes: None

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;
}
```

Example:

```
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
```

10.1.4 AND (AND Logical): Logical Instruction

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

Description: This instruction ANDs the contents of general registers Rn and Rm and stores the result in Rn.

This instruction can be used to AND general register R0 contents with zero-extended 8-bit immediate data, or, in indexed GBR indirect addressing mode, to AND 8-bit memory with 8-bit immediate data.

Notes: With AND #imm,R0, the upper 24 bits of R0 are always cleared as a result of the operation.

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;
  }
```

Example:

```
      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
```

Possible Exceptions: Exceptions may occur when AND.B instruction is executed.

- Data TLB multiple-hit exception
- Data TLB miss exception
- Data TLB protection violation exception
- Initial page write exception
- Data address error

Exceptions are checked taking a data access by this instruction as a byte load and a byte store.

10.1.5 BF (Branch if False): Branch Instruction

Format	Operation	Instruction Code	Cycle	T Bit
`BF label`	If T = 0 PC + 4 + disp × 2 → PC If T = 1, nop	`10001011dddddddd`	1	—

Description: This is a conditional branch instruction that references the T bit. The branch is taken if T = 0, and not taken if T = 1. The branch destination is address (PC + 4 + displacement × 2). The PC source value is the BF instruction address. As the 8-bit displacement is multiplied by two after sign-extension, the branch destination can be located in the range from –256 to +254 bytes from the BF instruction.

Notes: If the branch destination cannot be reached, the branch must be handled by using BF in combination with a BRA or JMP instruction, for example.

Operation:

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

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

Example:

```
    CLRT                   ;Normally T = 0
    BT    TRGET_T          ;T = 0, so branch is not taken.
    BF    TRGET_F          ;T = 0, so branch to TRGET_F.
    NOP                    ;
    NOP                    ;
TRGET_F:                   ;← BF instruction branch destination
```

Possible Exceptions:

- Slot illegal instruction exception

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

Format	Operation	Instruction Code	Cycle	T Bit
`BF/S label`	If T = 0, PC + 4 + disp × 2 → PC If T = 1, nop	`10001111dddddddd`	1	—

Description: This is a delayed conditional branch instruction that references the T bit. If T = 1, the next instruction is executed and the branch is not taken. If T = 0, the branch is taken after execution of the next instruction.

The branch destination is address (PC + 4 + displacement × 2). The PC source value is the BF/S instruction address. As the 8-bit displacement is multiplied by two after sign-extension, the branch destination can be located in the range from –256 to +254 bytes from the BF/S instruction.

Notes: As this is a delayed branch instruction, when the branch condition is satisfied, the instruction following this instruction is executed before the branch destination instruction.

Interrupts are not accepted between this instruction and the following instruction.

If the following instruction is a branch instruction, it is identified as a slot illegal instruction.

If this instruction is located in the delay slot immediately following a delayed branch instruction, it is identified as a slot illegal instruction.

If the branch destination cannot be reached, the branch must be handled by using BF/S in combination with a BRA or JMP instruction, for example.

Operation:

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

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

Example:

```
    CLRT                    ;Normally T = 0
    BT/S   TRGET_T          ;T = 0, so branch is not taken.
    NOP                     ;
    BF/S   TRGET_F          ;T = 0, so branch to TRGET.
    ADD    R0,R1            ;Executed before branch.
    NOP                     ;
TRGET_F:                    ;← BF/S instruction branch destination
```

Possible Exceptions:

- Slot illegal instruction exception

10.1.7 BRA (Branch): Branch Instruction

Format	Operation	Instruction Code	Cycle	T Bit
`BRA label`	PC + 4 + disp × 2 → PC	`1010dddddddddddd`	1	—

Description: This is an unconditional branch instruction. The branch destination is address (PC + 4 + displacement × 2). The PC source value is the BRA instruction address. As the 12-bit displacement is multiplied by two after sign-extension, the branch destination can be located in the range from –4096 to +4094 bytes from the BRA instruction. If the branch destination cannot be reached, this branch can be performed with a JMP instruction.

Notes: As this is a delayed branch instruction, the instruction following this instruction is executed before the branch destination instruction.

Interrupts are not accepted between this instruction and the following instruction. If the following instruction is a branch instruction, it is identified as a slot illegal instruction.

Operation:

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

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

Example:

```
    BRA   TRGET              ;Branch to TRGET.
    ADD   R0,R1              ;ADD executed before branch.
    NOP                      ;
TRGET:                       ;← BRA instruction branch destination
```

Possible Exceptions:

- Slot illegal instruction exception

10.1.8 BRAF (Branch Far): Branch Instruction (Delayed Branch Instruction)

Format	Operation	Instruction Code	Cycle	T Bit
`BRAF Rn`	PC + 4 + Rn → PC	`0000nnnn00100011`	1	—

Description: This is an unconditional branch instruction. The branch destination is address (PC + 4 + Rn). The branch destination address is the result of adding 4 plus the 32-bit contents of general register Rn to PC.

Notes: As this is a delayed branch instruction, the instruction following this instruction is executed before the branch destination instruction.

Interrupts are not accepted between this instruction and the following instruction. If the following instruction is a branch instruction, it is identified as a slot illegal instruction.

Operation:

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

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

Example:

```
   MOV.L #(TRGET-BRAF_PC),R0    ;Set displacement.
   BRAF  R0                     ;Branch to TRGET.
   ADD   R0,R1                  ;ADD executed before branch.
 BRAF_PC:                       ;
   NOP
 TRGET:                         ;← BRAF instruction branch destination
```

Possible Exceptions:

- Slot illegal instruction exception

10.1.9 BT (Branch if True): Branch Instruction

Format	Operation	Instruction Code	Cycle	T Bit
`BT    label`	If T = 1 PC + 4 + disp × 2 → PC If T = 0, nop	`10001001dddddddd`	1	—

Description: This is a conditional branch instruction that references the T bit. The branch is taken if T = 1, and not taken if T = 0.

The branch destination is address (PC + 4 + displacement × 2). The PC source value is the BT instruction address. As the 8-bit displacement is multiplied by two after sign-extension, the branch destination can be located in the range from –256 to +254 bytes from the BT instruction.

Notes: If the branch destination cannot be reached, the branch must be handled by using BT in combination with a BRA or JMP instruction, for example.

Operation:

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

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

Example:

```
     SETT                ;Normally T = 1
     BF    TRGET_F       ;T = 1, so branch is not taken.
     BT    TRGET_T       ;T = 1, so branch to TRGET_T.
     NOP                 ;
     NOP                 ;
TRGET_T:                 ;← BT instruction branch destination
```

Possible Exceptions:

- Slot illegal instruction exception

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

Format	Operation	Instruction Code	Cycle	T Bit
`BT/S label`	If T = 1, PC + 4 + disp × 2 → PC If T = 0, nop	`10001101dddddddd`	1	—

Description: This is a conditional branch instruction that references the T bit. The branch is taken if T = 1, and not taken if T = 0.

The PC source value is the BT/S instruction address. As the 8-bit displacement is multiplied by two after sign-extension, the branch destination can be located in the range from –256 to +254 bytes from the BT/S instruction.

Notes: As this is a delayed branch instruction, when the branch condition is satisfied, the instruction following this instruction is executed before the branch destination instruction.

Interrupts are not accepted between this instruction and the following instruction.

If the following instruction is a branch instruction, it is identified as a slot illegal instruction.

If the branch destination cannot be reached, the branch must be handled by using BT/S in combination with a BRA or JMP instruction, for example.

Operation:

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

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

Example:

```
    SETT                    ; Normally T = 1
    BF/S    TRGET_F         ; T = 1, so branch is not taken.
    NOP                     ;
    BT/S    TRGET_T         ; T = 1, so branch to TRGET_T.
    ADD     R0,R1           ; Executed before branch.
    NOP                     ;
 TRGET_T:                   ; ← BT/S instruction branch destination
```

Possible Exceptions:

- Slot illegal instruction exception

10.1.11 CLRMAC (Clear MAC Register): System Control Instruction

Format	Operation	Instruction Code	Cycle	T Bit
`CLRMAC`	0 → MACH, MACL	`0000000000101000`	1	—

Description: This instruction clears the MACH and MACL registers.

Notes: None

Operation:

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

Example:

```
CLRMAC                  ;Clear MAC register to initialize.
MAC.W   @R0+,@R1+       ;Multiply-and-accumulate operation
MAC.W   @R0+,@R1+       ;
```

10.1.12 CLRS (Clear S Bit): System Control Instruction

Format	Operation	Instruction Code	Cycle	T Bit
`CLRS`	0 → S	`0000000001001000`	1	—

Description: This instruction clears the S bit to 0.

Notes: None

Operation:

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

Example:

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

10.1.13 CLRT (Clear T Bit): System Control Instruction

Format	Operation	Instruction Code	Cycle	T Bit
CLRT	$0 \rightarrow T$	0000000000001000	1	0

Description: This instruction clears the T bit.

Notes: None

Operation:

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

Example:

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

10.1.14 CMP/cond (Compare Conditionally): Arithmetic Instruction

Format		Operation	Instruction Code	Cycle	T Bit
`CMP/EQ`	`Rm,Rn`	If Rn = Rm, 1 → T Otherwise, 0 → T	`0011nnnnmmmm0000`	1	Result of comparison
`CMP/GE`	`Rm,Rn`	If Rn ≥ Rm, signed, 1 → T Otherwise, 0 → T	`0011nnnnmmmm0011`	1	Result of comparison
`CMP/GT`	`Rm,Rn`	If Rn > Rm, signed, 1 → T Otherwise, 0 → T	`0011nnnnmmmm0111`	1	Result of comparison
`CMP/HI`	`Rm,Rn`	If Rn > Rm, unsigned, 1 → T Otherwise, 0 → T	`0011nnnnmmmm0110`	1	Result of comparison
`CMP/HS`	`Rm,Rn`	If Rn ≥ Rm, unsigned, 1 → T Otherwise, 0 → T	`0011nnnnmmmm0010`	1	Result of comparison
`CMP/PL`	`Rn`	If Rn > 0, 1 → T Otherwise, 0 → T	`0100nnnn00010101`	1	Result of comparison
`CMP/PZ`	`Rn`	If Rn ≥ 0, 1 → T Otherwise, 0 → T	`0100nnnn00010001`	1	Result of comparison
`CMP/STR`	`Rm,Rn`	If any bytes are equal, 1 → T Otherwise, 0 → T	`0010nnnnmmmm1100`	1	Result of comparison
`CMP/EQ`	`#imm,R0`	If R0 = imm, 1 → T Otherwise, 0 → T	`10001000iiiiiiii`	1	Result of comparison

Description: This instruction compares general registers Rn and Rm, and sets the T bit if the specified condition (cond) is true. If the condition is false, the T bit is cleared. The contents of Rn are not changed. Nine conditions can be specified. For the two conditions PZ and PL, Rn is compared with 0.

With the EQ condition, sign-extended 8-bit immediate data can be compared with R0. The contents of R0 are not changed.

Mnemonic		Description
CMP/EQ	Rm,Rn	If Rn = Rm, T = 1
CMP/GE	Rm,Rn	If Rn ≥ Rm as signed values, T = 1
CMP/GT	Rm,Rn	If Rn > Rm as signed values, T = 1
CMP/HI	Rm,Rn	If Rn > Rm as unsigned values, T = 1
CMP/HS	Rm,Rn	If Rn ≥ Rm as unsigned values, T = 1
CMP/PL	Rn	If Rn > 0, T = 1
CMP/PZ	Rn	If Rn ≥ 0, T = 1
CMP/STR	Rm,Rn	If any bytes are equal, T = 1
CMP/EQ	#imm,R0	If R0 = imm, T = 1

Notes: None

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) >> 24;
    HL = (temp & 0x00FF0000) >> 16;
    LH = (temp & 0x0000FF00) >> 8;
    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       ; T = 0, so branch is not taken.
CMP/HS   R0,R1         ; R0 = H'7FFFFFFF, R1 = H'80000000
BT       TRGET_T       ; T = 1, so branch is taken.
CMP/STR  R2,R3         ; R2 = "ABCD", R3 = "XYCZ"
BT       TRGET_T       ; T = 1, so branch is taken.
```

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

Format	Operation	Instruction Code	Cycle	T Bit
`DIV0S   Rm,Rn`	MSB of Rn → Q, MSB of Rm → M, M^Q → T	`0010nnnnmmmm0111`	1	Result of calculation

Description: This instruction performs initial settings for signed division. This instruction is followed by a DIV1 instruction that executes 1-digit division, for example, and repeated divisions are executed to find the quotient. See the description of the DIV1 instruction for details.

Notes: None

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 the examples for the DIV1 instruction.

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

Format	Operation	Instruction Code	Cycle	T Bit
`DIV0U`	0 → M/Q/T	`0000000000011001`	1	0

Description: This instruction performs initial settings for unsigned division. This instruction is followed by a DIV1 instruction that executes 1-digit division, for example, and repeated divisions are executed to find the quotient. See the description of the DIV1 instruction for details.

Notes: None

Operation:

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

Example:

See the examples for the DIV1 instruction.

10.1.17 DIV1 (Divide 1 Step): Arithmetic Instruction

Format	Operation	Instruction Code	Cycle	T Bit
`DIV1 Rm,Rn`	1-step division (Rn ÷ Rm)	`0011nnnnmmmm0100`	1	Result of calculation

Description: This instruction performs 1-digit division (1-step division) of the 32-bit contents of general register Rn (dividend) by the contents of Rm (divisor). The quotient is obtained by repeated execution of this instruction alone or in combination with other instructions. The specified registers and the M, Q, and T bits must not be modified during these repeated executions.

In 1-step division, the dividend is shifted 1 bit to the left, the divisor is subtracted from this, and the quotient bit is reflected in the Q bit according to whether the result is positive or negative.

The remainder can be found as follows after first finding the quotient using the DIV1 instruction:

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

Detection of division by zero or overflow is not provided. Check for division by zero and overflow division before executing the division. A remainder operation is not provided. Find the remainder by finding the product of the divisor and the obtained quotient, and subtracting this value from the dividend.

Initial settings should first be made with the DIV0S or DIV0U instruction. DIV1 is executed once for each bit of the divisor. If a quotient of more than 17 bits is required, place an ROTCL instruction before the DIV1 instruction. See the examples for details of the division sequence.

Notes: None

Operation:

```
DIV1(long m, long n)    /* DIV1 Rm,Rn */
{
   unsigned long tmp0, tmp2;
   unsigned char old_q, tmp1;

   old_q = Q;
   Q = (unsigned char)((0x80000000 & R[n])!=0);
   tmp2 = R[m];
   R[n] <<= 1;
   R[n] |= (unsigned long)T;
```

```
switch(old_q){
case 0:switch(M){
        case 0:tmp0 = R[n];
                R[n] -= tmp2;
                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] += tmp2;
                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] += tmp2;
                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] -= tmp2;
                        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                     ; Set divisor in upper 16 bits, clear lower 16 bits to 0
 TST        R0,R0                  ; Check for division by zero
 BT         ZERO_DIV               ;
 CMP/HS     R0,R1                  ; Check for overflow
 BT         OVER_DIV               ;
 DIV0U                             ; Flag initialization
 .arepeat   16                     ;
 DIV1       R0,R1                  ; Repeat 16 times
 .aendr                            ;
 ROTCL      R1                     ;
 EXTU.W     R1,R1                  ; R1 = quotient
```

Example 2:

```
                                 ;  R1:R2 (64 bits) ÷ R0 (32 bits) = R2 (32 bits); unsigned
    TST         R0,R0            ; Check for division by zero
    BT          ZERO_DIV         ;
    CMP/HS      R0,R1            ; Check for overflow
    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               ; Set divisor in upper 16 bits, clear lower 16 bits to 0
    EXTS.W      R1,R1            ; Dividend sign-extended to 32 bits
    XOR         R2,R2            ; R2 = 0
    MOV         R1,R3            ;
    ROTCL       R3               ;
    SUBC        R2,R1            ; If dividend is negative, subtract 1
    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 notation)
    ADDC        R2,R1            ; If MSB of quotient is 1, add 1 to convert to two's complement notation
    EXTS.W      R1,R1            ; R1 = quotient (two's complement notation)
```

Example 4:

```
                                    ; R2 (32 bits) ÷ R0 (32 bits) = R2 (32 bits); signed
  MOV         R2,R3                 ;
  ROTCL       R3                    ;
  SUBC        R1,R1                 ; Dividend sign-extended to 64 bits (R1:R2)
  XOR         R3,R3                 ; R3 = 0
  SUBC        R3,R2                 ; If dividend is negative, subtract 1 to convert to one's complement notation
  DIV0S       R0,R1                 ; Flag initialization
  .arepeat    32                    ;
  ROTCL       R2                    ; Repeat 32 times
  DIV1        R0,R1                 ;
  .aendr                            ;
  ROTCL       R2                    ; R2 = quotient (one's complement notation)
  ADDC        R3,R2                 ; If MSB of quotient is 1, add 1 to convert to two's complement notation
                                    ; R2 = quotient (two's complement notation)
```

10.1.18 DMULS.L (Double-length Multiply as Signed): Arithmetic Instruction

Format	Operation	Instruction Code	Cycle	T Bit
`DMULS.L Rm,Rn`	Signed, Rn × Rm →MAC	`0011nnnnmmmm1101`	2	—

Description: This instruction performs 32-bit multiplication of the contents of general register Rn by the contents of Rm, and stores the 64-bit result in the MACH and MACL registers. The multiplication is performed as a signed arithmetic operation.

Notes: None

Operation:

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

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

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

    RnL = temp1&0x0000FFFF;
    RnH = (temp1>>16)&0x0000FFFF;
    RmL = temp2&0x0000FFFF;
    RmH = (temp2>>16)&0x0000FFFF;
    temp0 = RmL*RnL;
    temp1 = RmH*RnL;
    temp2 = RmL*RnH;
    temp3 = RmH*RnH;

    Res2 = 0;
    Res1 = temp1+temp2;
    if (Res1<temp1) Res2 += 0x00010000;
    temp1 = (Res1<<16)&0xFFFF0000;
    Res0 = temp0 + temp1;
    if (Res0<temp0) Res2++;

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

        if (fnLmL<0) {
            Res2 = ~Res2;
            if (Res0==0)
               Res2++;
            else
               Res0 = (~Res0) + 1;
        }

        MACH = Res2;
        MACL = Res0;
        PC +=2;
}
```

Example:

```
DMULS.L   R0,R1        ;Before execution   R0 = H'FFFFFFFE, R1 = H'00005555
                       ;After execution    MACH = H'FFFFFFFF, MACL = H'FFFF5556
STS       MACH,R0      ;Get operation result (upper)
STS       MACL,R1      ;Get operation result (lower)
```

10.1.19 DMULU.L (Double-length Multiply as Unsigned): Arithmetic Instruction

Format	Operation	Instruction Code	Cycle	T Bit
`DMULU.L Rm,Rn`	Unsigned, Rn × Rm →MAC	`0011nnnnmmmm0101`	2	—

Description: This instruction performs 32-bit multiplication of the contents of general register Rn by the contents of Rm, and stores the 64-bit result in the MACH and MACL registers. The multiplication is performed as an unsigned arithmetic operation.

Notes: None

Operation:

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

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

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

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

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

    temp1 = (Res1<<16) & 0xFFFF0000;
    Res0 = temp0 + temp1;
    if (Res0<temp0) Res2++;
    Res2 = Res2 + ((Res1>>16)&0x0000FFFF) + temp3;

    MACH = Res2;
    MACL = Res0;
    PC += 2;
}
```

Example:

```
DMULU.L    R0,R1          ;Before execution   R0 = H'FFFFFFFE, R1 = H'00005555
                          ;After execution    MACH = H'00005554, MACL = H'FFFF5556
STS        MACH,R0        ;Get operation result (upper)
STS        MACL,R1        ;Get operation result (lower)
```

10.1.20 DT (Decrement and Test): Arithmetic Instruction

Format	Operation	Instruction Code	Cycle	T Bit

`DT Rn`	Rn – 1 → Rn;
	if Rn = 0, 1 → T
	if Rn ≠ 0, 0 → T
		`0100nnnn00010000`	1	Result of comparison

Description: This instruction decrements the contents of general register Rn by 1 and compares the result with zero. If the result is zero, the T bit is set to 1. If the result is nonzero, the T bit is cleared to 0.

Notes: None

Operation:

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

Example:

```
    MOV    #4,R5       ;Set loop count
LOOP:
    ADD    R0,R1       ;
    DT     R5          ;Decrement R5 value and check for 0.
    BF     LOOP        ;If T = 0, branch to LOOP (in this example, 4 loops are executed).
```

10.1.21 EXTS (Extend as Signed): Arithmetic Instruction

Format	Operation	Instruction Code	Cycle	T Bit
`EXTS.B Rm,Rn`	Rm sign-extended from byte → Rn	`0110nnnnmmmm1110`	1	—
`EXTS.W Rm,Rn`	Rm sign-extended from word → Rn	`0110nnnnmmmm1111`	1	—

Description:

This instruction sign-extends the contents of general register Rm and stores the result in Rn.

For a byte specification, the value of Rm bit 7 is transferred to Rn bits 8 to 31. For a word specification, the value of Rm bit 15 is transferred to Rn bits 16 to 31.

Notes: None

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;
}
```

Example:

```
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
```

10.1.22 EXTU (Extend as Unsigned): Arithmetic Instruction

Format	Operation	Instruction Code	Cycle	T Bit
`EXTU.B  Rm,Rn`	Rm zero-extended from byte → Rn	`0110nnnnmmmm1100`	1	—
`EXTU.W  Rm,Rn`	Rm zero-extended from word → Rn	`0110nnnnmmmm1101`	1	—

Description: This instruction zero-extends the contents of general register Rm and stores the result in Rn.

For a byte specification, 0 is transferred to Rn bits 8 to 31. For a word specification, 0 is transferred to Rn bits 16 to 31.

Notes: None

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;
}
```

Example:

```
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
```

10.1.23 ICBI (Instruction Cache Block Invalidate): Data Transfer Instruction

Format	Operation	Instruction Code	Cycle	T Bit
`ICBI @Rn`	Invalidates the instruction cache block indicated by logical address Rn	`0000nnnn11100011`	13	—

Description: This instruction accesses the instruction cache at the effective address indicated by the contents of Rn. When the cache is hit, the corresponding cache block is invalidated (the V bit is cleared to 0). At this time, write-back is not performed. No operation is performed in the case of a cache miss or access to a non-cache area.

Notes: None

Operation:

```
ICBI(int n)   /* ICBI @Rn */
{
    invalidate_instruction_cache_block(R[n]);
    PC += 2;
}
```

Example: When a program is overwriting RAM to modify its own execution, the corresponding block of the instruction cache should be invalidated by the ICBI instruction. This prevents execution of the program from the instruction cache, where the non-overwritten instructions are stored.

Possible Exceptions: Exceptions may occur when invalidation is not performed.

- Instruction TLB multiple-hit exception
- Instruction TLB miss exception
- Instruction TLB protection violation exception
- Instruction address error
- Slot illegal instruction exception

10.1.24 JMP (Jump): Branch Instruction

Format	Operation	Instruction Code	Cycle	T Bit
`JMP @Rn`	Rn → PC	`0100nnnn00101011`	1	—

Description: Unconditionally makes a delayed branch to the address specified by Rn.

Notes: As this is a delayed branch instruction, the instruction following this instruction is executed before the branch destination instruction.

Interrupts are not accepted between this instruction and the following instruction. If the following instruction is a branch instruction, it is identified as a slot illegal instruction.

Operation:

```
JMP(int n)/* JMP @Rn */
{
    unsigned int temp;

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

Example:

```
             MOV.L     JMP_TABLE,R0    ; R0 = TRGET address
             JMP       @R0             ; Branch to TRGET.
             MOV       R0,R1           ; MOV executed before branch.
             .align    4
JMP_TABLE:   .data.l   TRGET           ; Jump table
        ...........
TRGET:       ADD       #1,R1           ; ← Branch destination
```

Possible Exceptions:

- Slot illegal instruction exception

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

Format	Operation	Instruction Code	Cycle	T Bit
`LDC Rm, GBR`	Rm → GBR	0100mmmm00011110	1	—
`LDC Rm, VBR`	Rm → VBR	0100mmmm00101110	1	—
`LDC Rm, SGR`	Rm → SGR	0100mmmm00111010	4	—
`LDC Rm, SSR`	Rm → SSR	0100mmmm00111110	1	—
`LDC Rm, SPC`	Rm → SPC	0100mmmm01001110	1	
`LDC Rm, DBR`	Rm → DBR	0100mmmm11111010	4	—
`LDC Rm, R0_BANK`	Rm → R0_BANK	0100mmmm10001110	1	—
`LDC Rm, R1_BANK`	Rm → R1_BANK	0100mmmm10011110	1	—
`LDC Rm, R2_BANK`	Rm → R2_BANK	0100mmmm10101110	1	—
`LDC Rm, R3_BANK`	Rm → R3_BANK	0100mmmm10111110	1	—
`LDC Rm, R4_BANK`	Rm → R4_BANK	0100mmmm11001110	1	—
`LDC Rm, R5_BANK`	Rm → R5_BANK	0100mmmm11011110	1	—
`LDC Rm, R6_BANK`	Rm → R6_BANK	0100mmmm11101110	1	—
`LDC Rm, R7_BANK`	Rm → R7_BANK	0100mmmm11111110	1	—
`LDC.L @Rm+, GBR`	(Rm) → GBR, Rm+4 → Rm	0100mmmm00010111	1	—
`LDC.L @Rm+, VBR`	(Rm) → VBR, Rm+4 → Rm	0100mmmm00100111	1	—
`LDC.L @Rm+, SGR`	(Rm) → SGR, Rm+4 → Rm	0100mmmm00110110	4	—
`LDC.L @Rm+, SSR`	(Rm) → SSR, Rm+4 → Rm	0100mmmm00110111	1	—
`LDC.L @Rm+, SPC`	(Rm) → SPC, Rm+4 → Rm	0100mmmm01000111	1	—
`LDC.L @Rm+, DBR`	(Rm) → DBR, Rm+4 → Rm	0100mmmm11110110	4	—
`LDC.L @Rm+, R0_BANK`	(Rm) → R0_BANK, Rm+4 → Rm	0100mmmm10000111	1	—
`LDC.L @Rm+, R1_BANK`	(Rm) → R1_BANK, Rm+4 → Rm	0100mmmm10010111	1	—
`LDC.L @Rm+, R2_BANK`	(Rm) → R2_BANK, Rm+4 → Rm	0100mmmm10100111	1	—
`LDC.L @Rm+, R3_BANK`	(Rm) → R3_BANK, Rm+4 → Rm	0100mmmm10110111	1	—
`LDC.L @Rm+, R4_BANK`	(Rm) → R4_BANK, Rm+4 → Rm	0100mmmm11000111	1	—
`LDC.L @Rm+, R5_BANK`	(Rm) → R5_BANK, Rm+4 → Rm	0100mmmm11010111	1	—
`LDC.L @Rm+, R6_BANK`	(Rm) → R6_BANK, Rm+4 → Rm	0100mmmm11100111	1	—
`LDC.L @Rm+, R7_BANK`	(Rm) → R7_BANK, Rm+4 → Rm	0100mmmm11110111	1	—

Description: These instructions store the source operand in the control register GBR, VBR, SSR, SPC, DBR, SGR, or R0_BANK to R7_BANK.

Notes: With the exception of LDC Rm,GBR and LDC.L @Rm+,GBR, the LDC/LDC.L instructions are privileged instructions and can only be used in privileged mode. Use in user mode will cause an illegal instruction exception. However, LDC Rm,GBR and LDC.L @Rm+,GBR can also be used in user mode.

With the LDC Rm, Rn_BANK and LDC.L @Rm, Rn_BANK instructions, Rn_BANK0 is accessed when the RB bit in the SR register is 1, and Rn_BANK1 is accessed when this bit is 0.

Operation:

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

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

LDCSGR(int m)      /* LDC Rm,SGR : Privileged */
{
  SGR = R[m];
  PC += 2;
}

LDCSSR(int m)      /* LDC Rm,SSR : Privileged */
{
  SSR = R[m],
  PC += 2;
}

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

```
LDCDBR(int m)       /* LDC Rm,DBR : Privileged */
{
  DBR = R[m];
  PC += 2;
}

LDCRn_BANK(int m)   /* LDC Rm,Rn_BANK : Privileged */
                 /* n=0-7 */
{
  Rn_BANK = R[m];
  PC += 2;
}

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

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

LDCMSGR(int m)      /* LDC.L @Rm+,SGR : Privileged */
{
  SGR = Read_Long(R[m]);
  R[m] += 4;
  PC += 2;
}
```

```
LDCMSSR(int m)      /* LDC.L @Rm+,SSR : Privileged */
{
  SSR=Read_Long(R[m]);
  R[m] += 4;
  PC += 2;
}

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

LDCMDBR(int m)      /* LDC.L @Rm+,DBR : Privileged */
{
  DBR = Read_Long(R[m]);
  R[m] += 4;
  PC += 2;
}

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

Possible Exceptions:

- Data TLB multiple-hit exception
- General illegal instruction exception
- Slot illegal instruction exception
- Data TLB miss exception
- Data TLB protection violation exception
- Data address error

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

Format	Operation	Instruction Code	Cycle	T Bit
`LDS    Rm,MACH`	Rm → MACH	0100mmmm00001010	1	—
`LDS    Rm,MACL`	Rm → MACL	0100mmmm00011010	1	—
`LDS    Rm,PR`	Rm→ PR	0100mmmm00101010	1	—
`LDS.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.

Notes: None

Operation:

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

LDSMACL(long m)      /* LDS Rm,MACL */
{
    MACL = R[m];
    PC += 2;
}

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

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

LDSMMACL(lomg 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;
}
```

Example:

```
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)
```

Possible Exceptions: Exception may occur when LDS.L instruction is executed.

- Data TLB multiple-hit exception
- Data TLB miss exception
- Data TLB protection violation exception
- Data address error

10.1.27 LDTLB (Load PTEH/PTEL to TLB): System Control Instruction (Privileged Instruction)

Format	Operation	Instruction Code	Cycle	T Bit
`LDTLB`	PTEH/PTEL → TLB	`0000000000111000`	1	—

Description: This instruction loads the contents of the PTEH/PTEL registers into the TLB (translation lookaside buffer) specified by MMUCR.URC (random counter field in the MMC control register).

LDTLB is a privileged instruction, and can only be used in privileged mode. Use of this instruction in user mode will cause an illegal instruction exception.

Notes: As this instruction loads the contents of the PTEH/PTEL registers into a TLB, it should be used either with the MMU disabled, or in the P1 or P2 virtual space with the MMU enabled (see section 7, Memory Management Unit (MMU), for details). After this instruction is issued, there must be at least one instruction between the LDTLB instruction and issuance of an instruction relating to address to the P0, U0, and P3 areas (i.e. BRAF, BSRF, JMP, JSR, RTS, or RTE).

Operation:

```
LDTLB( )   /*LDTLB */
{
    TLB[MMUCR.URC].ASID = PTEH & 0x000000FF;
    TLB[MMUCR.URC].VPN = (PTEH & 0xFFFFFC00) >> 10;
    TLB[MMUCR.URC].PPN = (PTEH & 0x1FFFFC00) >> 10;
    TLB[MMUCR.URC].SZ = (PTEL & 0x00000080) >> 6 |
        (PTEL & 0x00000010) >> 4;
    TLB[MMUCR.URC].SH = (PTEH & 0x00000002) >> 1;
    TLB[MMUCR.URC].PR = (PTEH & 0x00000060) >> 5;
    TLB[MMUCR.URC].WT = (PTEH & 0x00000001);
    TLB[MMUCR.URC].C = (PTEH & 0x00000008) >> 3;
    TLB[MMUCR.URC].D = (PTEH & 0x00000004) >> 2;
    TLB[MMUCR.URC].V = (PTEH & 0x00000100) >> 8;

    PC += 2;
}
```

Example:

```
MOV    @R0,R1         ; Load page table entry (upper) into R1
MOV    R1,@R2         ; Load R1 into PTEH; R2 is PTEH address (H'FF000000)
LDTLB                 ; Load PTEH, PTEL registers into TLB
```

Possible Exceptions:

- General illegal instruction exception
- Slot illegal instruction exception

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

Format	Operation	Instruction Code	Cycle	T Bit
`MAC.L @Rm+,@Rn+`	Signed, (Rn) × (Rm) + MAC → MAC Rn + 4 → Rn, Rm + 4 → Rm	`0000nnnnmmmm1111`	5	—

Description: This instruction performs signed multiplication of the 32-bit operands whose addresses are the contents of general registers Rm and Rn, adds the 64-bit result to the MAC register contents, and stores the result in the MAC register. Operands Rm and Rn are each incremented by 4 each time they are read.

If the S bit is 0, the 64-bit result is stored in the linked MACH and MACL registers.

If the S bit is 1, the addition to the MAC register contents is a saturation operation at the 48th bit from the LSB. In a saturation operation, only the lower 48 bits of the MAC register are valid, and the result range is limited to H'FFFF800000000000 (minimum value) to H'00007FFFFFFFFFFF (maximum value).

Notes: None

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(fnLmL<0){
        Res2 = ˜Res2;
        if (Res0==0) Res2++;
        else Res0 = (˜Res0)+1;
    }
    if(S==1){
        Res0 = MACL + Res0;
        if (MACL>Res0) Res2++;
        if (MACH & 0x00008000);
        else Res2 += MACH|0xFFFF0000;
              Res2 += MACH&0x00007FFF;
        if(((long)Res2<0)&&(Res2 < 0xFFFF8000)){
            Res2 = 0xFFFF8000;
            Res0 = 0x00000000;
        }
        if(((long)Res2>0)&&(Res2 > 0x00007FFF)){
            Res2 = 0x00007FFF;
            Res0 = 0xFFFFFFFF;
        };

        MACH = (Res2 & 0x0000FFFF)|(MACH & 0xFFFF0000);
        MACL = Res0;
    }
        else {
            Res0 = MACL + Res0;
            if (MACL>Res0) Res2++;
            Res2 += MACH;

            MACH = Res2;
            MACL = Res0;
        }
        PC += 2;
    }
```

Example:

```
          MOVA       TBLM,R0         ;Get table address
          MOV        R0,R1           ;
          MOVA       TBLN,R0         ;Get table address
          CLRMAC                     ;MAC register initialization
          MAC.L      @R0+,@R1+       ;
          MAC.L      @R0+,@R1+       ;
          STS        MACL,R0         ;Get result in R0
          .........
          .align     2               ;
TBLM      .data.l    H'1234ABCD      ;
          .data.l    H'5678EF01      ;
TBLN      .data.l    H'0123ABCD      ;
          .data.l    H'4567DEF0      ;
```

Possible Exceptions:

- Data TLB multiple-hit exception
- Data TLB miss exception
- Data TLB protection violation exception
- Data address error

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

Format		Operation	Instruction Code	Cycle	T Bit
`MAC.W`	`@Rm+,@Rn+`	Signed, (Rn) × (Rm) + MAC →MAC	`0100nnnnmmmm1111`	4	—
`MAC`	`@Rm+,@Rn+`	Rn + 2 → Rn, Rm + 2 → Rm			

Description: This instruction performs signed multiplication of the 16-bit operands whose addresses are the contents of general registers Rm and Rn, adds the 32-bit result to the MAC register contents, and stores the result in the MAC register. Operands Rm and Rn are each incremented by 2 each time they are read.

If the S bit is 0, a 16 × 16 + 64 → 64-bit multiply-and-accumulate operation is performed, and the 64-bit result is stored in the linked MACH and MACL registers.

If the S bit is 1, a 16 × 16 + 32 → 32-bit multiply-and-accumulate operation is performed, and the addition to the MAC register contents is a saturation operation. In a saturation operation, only the MACL register is valid, and the result range is limited to H'80000000 (minimum value) to H'7FFFFFFF (maximum value). If overflow occurs, the LSB of the MACH register is set to 1. H'80000000 (minimum value) is stored in the MACL register if the result overflows in the negative direction, and H'7FFFFFFF (maximum value) is stored if the result overflows in the positive direction

Notes: If the S bit is 0, a 16 × 16 + 64 → 64-bit multiply-and-accumulate operation is performed.

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) MACL = 0x7FFFFFFF;
            if (src==2) MACL = 0x80000000;
        }
    }
    else {
        MACH += tempn;
        if (templ>MACL) MACH += 1;

    }
    PC += 2;
}
```

Example:

```
             MOVA         TBLM,R0       ;Get table address
             MOV          R0,R1         ;
             MOVA         TBLN,R0       ;Get table address
             CLRMAC                     ;MAC register initialization
             MAC.W        @R0+,@R1+     ;
             MAC.W        @R0+,@R1+     ;
             STS          MACL,R0       ;Get result in R0
             ...........
             .align 2                   ;
TBLM         .data.w      H'1234        ;
             .data.w      H'5678        ;
TBLN         .data.w      H'0123        ;
             .data.w      H'4567        ;
```

Possible Exceptions:

- Data TLB multiple-hit exception
- Data TLB miss exception
- Data TLB protection violation exception
- Data address error

10.1.30 MOV (Move data): Data Transfer Instruction

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

Description: This instruction transfers the source operand to the destination. When an operand is memory, the data size can be specified as byte, word, or longword. When the source operand is memory, the loaded data is sign-extended to longword before being stored in the register.

Notes: None

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)
```

Possible Exceptions: Exceptions may occur when MOV instructions without MOV Rm,Rn are executed.

- Data TLB multiple-hit exception
- Data TLB miss exception
- Data TLB protection violation exception
- Data address error
- Initial page write exception (only write operation)

10.1.31 MOV (Move Constant Value): Data Transfer Instruction

Format	Operation	Instruction Code	Cycle	T Bit
`MOV #imm,Rn`	imm → sign extension → Rn	`1110nnnniiiiiiii`	1	—
`MOV.W @(disp*,PC),Rn`	(disp × 2 + PC + 4) → sign extension → Rn	`1001nnnndddddddd`	1	—
`MOV.L @(disp*,PC),Rn`	(disp × 4 + PC & H'FFFFFFFC + 4) → Rn	`1101nnnndddddddd`	1	—

Note: * The assembler of Renesas Technology uses the value after scaling (×1, ×2, or ×4) as the displacement (disp).

Description: This instruction stores immediate data, sign-extended to longword, in general register Rn. In the case of word or longword data, the data is stored from memory address (PC + 4 + displacement × 2) or (PC + 4 + displacement × 4).

With word data, the 8-bit displacement is multiplied by two after zero-extension, and so the relative distance from the table is in the range up to PC + 4 + 510 bytes. The PC value is the address of this instruction.

With longword data, the 8-bit displacement is multiplied by four after zero-extension, and so the relative distance from the operand is in the range up to PC + 4 + 1020 bytes. The PC value is the address of this instruction. A value with the lower 2 bits adjusted to B'00 is used in address calculation.

Notes: If a PC-relative load instruction is executed in a delay slot, a slot illegal instruction exception will be generated.

Operation:

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

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

      disp = (unsigned int)(0x000000FF & d);
      R[n] = (int)Read_Word(PC+4+(disp<<1));
      if ((R[n]&0x8000)==0) R[n] &= 0x0000FFFF;
      else R[n] |= 0xFFFF0000;
      PC += 2;
}

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

      disp = (unsigned int)(0x000000FF & (int)d);
      R[n] = Read_Long((PC & 0xFFFFFFFC)+4+(disp<<2));
      PC += 2;
}
```

Example:

```
Address
1000        MOV      #H'80,R1       ;R1 = H'FFFFFF80
1002        MOV.W    IMM,R2         ;R2 = H'FFFF9ABC     IMM means (PC + 4 + H'08)
1004        ADD      #-1,R0         ;
1006        TST      R0,R0          ;
1008        MOV.L    @(3*,PC),R3    ;R3 = H'12345678
100A        BRA      NEXT           ;Delayed branch instruction
100C        NOP
100E IMM    .data.w  H'9ABC         ;
1010        .data.w  H'1234         ;
1012 NEXT   JMP      @R3            ;Distination of BRA branch instruction
1014        CMP/EQ   #0,R0          ;
            .align   4              ;
1018        .data.l  H'12345678     ;
101C        .data.l  H'9ABCDEF0     ;
```

Note: * The assembler of Renesas Technology uses the value after scaling (×1, ×2, or ×4) as the displacement (disp).

Possible Exceptions: Exceptions may occur when PC-relative load instruction is executed.

- Data TLB multiple-hit exception
- Slot illegal instruction exception
- Data TLB miss exception
- Data TLB protection violation exception
- Data address error

10.1.32 MOV (Move Global Data): Data Transfer Instruction

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

Note: * The assembler of Renesas Technology uses the value after scaling (×1, ×2, or ×4) as the displacement (disp).

Description: This instruction transfers the source operand to the destination. Byte, word, or longword can be specified as the data size, but the register is always R0. If the transfer data is byte-size, the 8-bit displacement is only zero-extended, so a range up to +255 bytes can be specified. If the transfer data is word-size, the 8-bit displacement is multiplied by two after zero-extension, enabling a range up to +510 bytes to be specified. With longword transfer data, the 8-bit displacement is multiplied by four after zero-extension, enabling a range up to +1020 bytes to be specified.

When the source operand is memory, the loaded data is sign-extended to longword before being stored in the register.

Notes: When loading, the destination register is always R0.

Operation:

```
MOVBLG(int d) /* MOV.B @(disp,GBR),R0 */
{
        unsigned int disp;

        disp = (unsigned int)(0x000000FF & d);
        R[0] = (int)Read_Byte(GBR+disp);
        if ((R[0]&0x80)==0) R[0] &= 0x000000FF;
        else R[0] |= 0xFFFFFF00;
        PC += 2;
}
```

```
MOVWLG(int d) /* MOV.W @(disp,GBR),R0 */
{
        unsigned int disp;

        disp = (unsigned int)(0x000000FF & d);

        R[0] = (int)Read_Word(GBR+(disp<<1));
        if ((R[0]&0x8000)==0) R[0] &= 0x0000FFFF;
        else R[0] |= 0xFFFF0000;
        PC += 2;
}

MOVLLG(int d) /* MOV.L @(disp,GBR),R0 */
{


        unsigned int disp;

        disp = (unsigned int)(0x000000FF & d);
        R[0] = Read_Long(GBR+(disp<<2));
        PC += 2;
}
MOVBSG(int d) /* MOV.B R0,@(disp,GBR) */
{
        unsigned int disp;

        disp = (unsigned int)(0x000000FF & d);
        Write_Byte(GBR+disp,R[0]);
        PC += 2;
}
```

```
MOVWSG(int d) /* MOV.W R0,@(disp,GBR) */
{
        unsigned int disp;

        disp = (unsigned int)(0x000000FF & d);
        Write_Word(GBR+(disp<<1),R[0]);
        PC += 2;
}

MOVLSG(int d) /* MOV.L R0,@(disp,GBR) */
{
        unsigned int disp;

        disp = (unsigned int)(0x000000FF & (long)d);
        Write_Long(GBR+(disp<<2),R[0]);
        PC += 2;
}
```

Example:

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

Note: * The assembler of Renesas Technology uses the value after scaling (×1, ×2, or ×4) as the displacement (disp).

Possible Exceptions:

- Data TLB multiple-hit exception
- Slot illegal instruction exception
- Data TLB miss exception
- Data TLB protection violation exception
- Data address error
- Initial page write exception (only write operation)

10.1.33 MOV (Move Structure Data): Data Transfer Instruction

Format	Operation	Instruction Code	Cycle	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	—

Note: * The assembler of Renesas Technology uses the value after scaling (×1, ×2, or ×4) as the displacement (disp).

Description: This instruction transfers the source operand to the destination. It is ideal for accessing data inside a structure or stack. Byte, word, or longword can be specified as the data size, but with byte or word data the register is always R0.

If the data is byte-size, the 4-bit displacement is only zero-extended, so a range up to +15 bytes can be specified. If the data is word-size, the 4-bit displacement is multiplied by two after zero-extension, enabling a range up to +30 bytes to be specified. With longword data, the 4-bit displacement is multiplied by four after zero-extension, enabling a range up to +60 bytes to be specified. If a memory operand cannot be reached, the previously described @(R0,Rn) mode must be used.

When the source operand is memory, the loaded data is sign-extended to longword before being stored in the register.

Notes: When loading byte or word data, the destination register is always R0. Therefore, if the following instruction attempts to reference R0, it is kept waiting until completion of the load instruction. This allows optimization by changing the order of instructions.

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

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;
}
```

Example:

```
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
```

Note: * The assembler of Renesas Technology uses the value after scaling (×1, ×2, or ×4) as the displacement (disp).

Possible Exceptions:

- Data TLB multiple-hit exception
- Slot illegal instruction exception
- Data TLB miss exception
- Data TLB protection violation exception
- Data address error
- Initial page write exception (only write operation)

10.1.34 MOVA (Move Effective Address): Data Transfer Instruction

Format	Operation	Instruction Code	Cycle	T Bit
`MOVA @(disp*,PC),R0`	disp × 4 + PC & H'FFFFFFFC + 4 → R0	`11000111dddddddd`	1	—

Note: * The assembler of Renesas Technology uses the value after scaling (×1, ×2, or ×4) as the displacement (disp).

Description: This instruction stores the source operand effective address in general register R0. The 8-bit displacement is multiplied by four after zero-extension. The PC value is the address of this instruction, but a value with the lower 2 bits adjusted to B'00 is used in address calculation.

Notes: If this instruction is executed in a delay slot, a slot illegal instruction exception will be generated.

Operation:

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

         disp = (unsigned int)(0x000000FF & d);
         R[0] = (PC&0xFFFFFFFC) + 4 + (disp<<2);
         PC += 2;
}
```

Example:

```
Address .org     H'1006
1006    MOVA     STR*,R0     ;STR address → R0
1008    MOV.B    @R0,R1      ;R1 = "X" ← Position after adjustment of lower 2 bits of PC
100A    ADD      R4,R5
        .align   4
100C    STR:.sdata  "XYZP12"
```

Note: * The assembler of Renesas Technology uses the value after scaling (×1, ×2, or ×4) as the displacement (disp).

Possible Exceptions:

- Slot illegal instruction exception

10.1.35 MOVCA.L (Move with Cache Block Allocation): Data Transfer Instruction

Format	Operation	Instruction Code	Cycle	T Bit
`MOVCA.L R0,@Rn`	R0 → (Rn) (without fetching cache block)	`0000nnnn11000011`	1	—

Description: This instruction stores the contents of general register R0 in the memory location indicated by effective address Rn. This instruction differs from other store instructions as follows.

If write-back is selected for the accessed memory, and a cache miss occurs, the cache block will be allocated but an R0 data write will be performed to that cache block without performing a block read. Other cache block contents are undefined.

Notes: None

Operation:

```
MOVCAL(int n) /*MOVCA.L  R0,@Rn */
     {
       if ((is_write_back_memory(R[n]))
            && (look_up_in_operand_cache(R[n]) == MISS))
                     allocate_operand_cache_block(R[n]);
       Write_Long(R[n], R[0]);
       PC += 2;
     }
```

Possible Exceptions:

- Data TLB multiple-hit exception
- Data TLB miss exception
- Data TLB protection violation exception
- Initial page write exception
- Data address error

10.1.36 MOVCO (Move Conditional): Data Transfer Instruction

Format	Operation	Instruction Code	Cycle	T Bit
`MOVCO.L R0,@Rn`	LDST → T if (T==1) R0 → (Rn) 0 → LDST	`0000nnnn01110011`	1	LDST

Description: MOVCO is used in combination with MOVLI to realize an atomic read-modify-write operation in a single processor.

This instruction copies the value of the LDST flag to the T bit. When the T bit is set to 1, the value of R0 is stored at the address in Rm. If the T bit is cleared to 0, the value is not stored at the address in Rm. Finally, the LDST flag is cleared to 0. Since the LDST flag is cleared by an instruction or exception, storage by the MOVCO instruction only proceeds when no interrupt or exception has occurred between the execution of the MOVLI and MOVCO instructions.

Notes: None

Operation:

```
MOVCO(long n) /* MOVCO Rn,@Rn */
{
     T = LDST;
     if(T==1)
          Write_Long(R[n],R[0]);
     LDST = 0;
     PC += 2
}
```

Example:

```
Retry:        MOVLI.L   @Rn,R0     ; Atomic incrementation
              ADD       #1,R0
              MOVCO.L   R0,@Rn
              BF        Retry      ; Reexecute if an interrupt or other
                                   exception occurs between the MOVLI and
                                   MOVCO instructions
              NOP
```

Possible Exceptions:

- Data TLB multiple-hit exception
- Data TLB miss exception
- Data TLB protection violation exception
- Initial page write exception
- Data address error

10.1.37 MOVLI (Move Linked): Data Transfer Instruction

Format	Operation	Instruction Code	Cycle	T Bit
`MOVLI.L @Rm,R0`	1 → LDST (Rm) → R0 If an interrupt or exception has occurred 0 → LDST	`0000nnnn01100011`	1	—

Description: MOVLI is used in combination with MOVCO to realize an atomic read-modify-write operation in a single processor.

This instruction sets the LDST flag to 1 and reads the four bytes of data indicated by Rm into R0. If, however, an interrupt or exception occurs, LDST is cleared to 0.

Storage by the MOVCO instruction only proceeds when the instruction is executed after the LDST bit has been set by the MOVLI instruction and not cleared by an interrupt or other exception. When LDST has been cleared to 0, the MOVCO instruction clears the T bit and does not proceed with storage.

Notes: None

Operation:

```
MOVLINK(long m) /* MOVLI Rm,@Rn */
{
     LDST = 1;
     R[0] = Read_Long(R[m]);
     PC += 2
}
```

Example:
See the examples for the MOVCO instruction.

Possible Exceptions:

- Data TLB multiple-hit exception
- Data TLB miss exception
- Data TLB protection violation exception
- Data address error

10.1.38 MOVT (Move T Bit): Data Transfer Instruction

Format	Operation	Instruction Code	Cycle	T Bit
`MOVT Rn`	T → Rn	`0000nnnn00101001`	1	—

Description: This instruction stores the T bit in general register Rn. When T = 1, Rn = 1; when T = 0, Rn = 0.

Notes: None

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
```

10.1.39 MOVUA (Move Unaligned): Data Transfer Instruction

Format	Operation	Instruction Code	Cycle	T Bit
`MOVUA.L @Rm,R0`	(Rm) → R0 Load non-boundary-aligned data	`0100nnnn10101001`	2	—
`MOVUA.L @Rm+,R0`	(Rm) → R0, Rm + 4 → Rm Load non-boundary-aligned data	`0100nnnn11101001`	2	—

Description: This instruction loads the longword of data from the effective address indicated by the contents of Rm in memory to R0. The address is not restricted to longword boundaries address (4n); this instruction allows loading from non-longword-boundary addresses (4n + 1, 4n + 2, and 4n + 3). Data address error exceptions do not occur when access is to non-longword-boundary addresses (4n + 1, 4n + 2, and 4n + 3).

Notes: None

Operation:

```
MOVUAL(int m) /* MOVUA.L Rm,R0*/
{
    Read_Unaligned_Long(R0,R[m]);
    PC += 2;
}
MOVUALP(int m) /* MOVUA.L Rm+,R0*/
{
    Read_Unaligned_Long(R0,R[m]);
    if(m != 0) R[m] += 4;
    PC += 2;
}
```

Example:

```
MOVUA.L   @R1,R0    ;Before execution  R1=H'00001001, R0=H'00000000
                    ;After execution   R0=(H'00001001)
MOVUA.L   @R1+,R0   ;Before execution  R1=H'00001007, R0=H'00000000
                    ;After execution   R1=H'0000100B, R0=(H'00001007)
; Special case in which the source operand is @R0
MOVUA.L   @R0,R0    ;Before execution  R0=H'00001001
                    ;After execution   R0=(H'00001001)
MOVUA.L   @R0+,R0   ;Before execution  R0=H'00001001
                    ;After execution   R0=(H'00001001)
```

Possible Exceptions:

- Data TLB multiple-hit exception
- Data TLB miss exception
- Data TLB protection violation exception
- Data address error (when the privileged area is accessed from user)

10.1.40 MUL.L (Multiply Long): Arithmetic Instruction

Format	Operation	Instruction Code	Cycle	T Bit
`MUL.L  Rm,Rn`	Rn × Rm → MACL	`0000nnnnmmmm0111`	2	—

Description: This instruction 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 contents of MACH are not changed.

Notes: None

Operation:

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

Example:

```
MUL.L     R0,R1        ; Before execution   R0 = H'FFFFFFFE, R1 = H'00005555
                       ; After execution    MACL = H'FFFF5556
STS       MACL,R0      ; Get operation result
```

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

Format	Operation	Instruction Code	Cycle	T Bit
`MULS.W  Rm,Rn`	Signed, Rn × Rm → MACL	`0010nnnnmmmm1111`	1	—

Description: This instruction performs 16-bit multiplication of the contents of general registers Rn and Rm, and stores the 32-bit result in the MACL register. The multiplication is performed as a signed arithmetic operation. The contents of MACH are not changed.

Notes: None

Operation:

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

Example:

```
MULS.W       R0,R1          ; Before execution  R0 = H'FFFFFFFE, R1 = H'00005555
                            ; After execution   MACL = H'FFFF5556
STS          MACL,R0        ; Get operation result
```

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

Format	Operation	Instruction Code	Cycle	T Bit
`MULU.W Rm,Rn`	Unsigned, Rn × Rm → MACL	`0010nnnnmmmm1110`	1	—

Description: This instruction performs 16-bit multiplication of the contents of general registers Rn and Rm, and stores the 32-bit result in the MACL register. The multiplication is performed as an unsigned arithmetic operation. The contents of MACH are not changed.

Notes: None

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.W     R0,R1          ;Before execution   R0 = H'00000002, R1 = H'FFFFAAAA
                          ;After execution    MACL = H'00015554
STS        MACL,R0        ;Get operation result
```

10.1.43 NEG (Negate): Arithmetic Instruction

Format	Operation	Instruction Code	Cycle	T Bit
`NEG Rm,Rn`	0 - Rm → Rn	`0110nnnnmmmm1011`	1	—

Description: This instruction finds the two's complement of the contents of general register Rm and stores the result in Rn. That is, it subtracts Rm from 0 and stores the result in Rn.

Notes: None

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
```

10.1.44 NEGC (Negate with Carry): Arithmetic Instruction

Format	Operation	Instruction Code	Cycle	T Bit
`NEGC Rm,Rn`	0 – Rm – T → Rn, borrow → T	`0110nnnnmmmm1010`	1	Borrow

Description: This instruction subtracts the contents of general register Rm and the T bit from 0 and stores the result in Rn. A borrow resulting from the operation is reflected in the T bit. The NEGC instruction is used for sign inversion of a value exceeding 32 bits.

Notes: None

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;
}
```

Example:

```
CLRT                      ; Sign inversion of R0:R1 (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
```

10.1.45 NOP (No Operation): System Control Instruction

Format	Operation	Instruction Code	Cycle	T Bit
`NOP`	No operation	`0000000000001001`	1	—

Description: This instruction simply increments the program counter (PC), advancing the processing flow to execution of the next instruction.

Notes: None

Operation:

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

Example:

```
NOP                    ;Time equivalent to one execution state elapses.
```

10.1.46 NOT (Not-logical Complement): Logical Instruction

Format	Operation	Instruction Code	Cycle	T Bit
`NOT Rm,Rn`	~Rm → Rn	`0110nnnnmmmm0111`	1	—

Description: This instruction finds the one's complement of the contents of general register Rm and stores the result in Rn. That is, it inverts the Rm bits and stores the result in Rn.

Notes: None

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
```

10.1.47 OCBI (Operand Cache Block Invalidate): Data Transfer Instruction

Format	Operation	Instruction Code	Cycle	T Bit
`OCBI @Rn`	Operand cache block invalidation	`0000nnnn10010011`	1	—

Description: This instruction accesses data using the contents indicated by effective address Rn. In the case of a hit in the cache, the corresponding cache block is invalidated (the V bit is cleared to 0). If there is unwritten information (U bit = 1), write-back is not performed even if write-back mode is selected. No operation is performed in the case of a cache miss or an access to a non-cache area.

Notes: None

Operation:

```
OCBI(int n) /* OCBI @Rn */
{
  invalidate_operand_cache_block(R[n]);
  PC += 2;
}
```

Possible Exceptions:

- Data TLB multiple-hit exception
- Data TLB miss exception
- Data TLB protection violation exception
- Initial page write exception
- Data address error

Note that the above exceptions are generated even if OCBI does not operate.

10.1.48 OCBP (Operand Cache Block Purge): Data Transfer Instruction

Format	Operation	Instruction Code	Cycle	T Bit
`OCBP @Rn`	Writes back and invalidates operand cache block	`0000nnnn10100011`	1	—

Description: This instruction accesses data using the contents indicated by effective address Rn. If the cache is hit and there is unwritten information (U bit = 1), the corresponding cache block is written back to external memory and that block is invalidated (the V bit is cleared to 0). If there is no unwritten information (U bit = 0), the block is simply invalidated. No operation is performed in the case of a cache miss or an access to a non-cache area.

Notes: None

Operation:

```
OCBP(int n)         /* OCBP @Rn */
{
  if(is_dirty_block(R[n]))  write_back(R[n])
  invalidate_operand_cache_block(R[n]);
  PC += 2;
}
```

Possible Exceptions:

- Data TLB multiple-hit exception
- Data TLB miss exception
- Data TLB protection violation exception
- Data address error

Note that the above exceptions are generated even if OCBP does not operate.

10.1.49 OCBWB (Operand Cache Block Write Back): Data Transfer Instruction

Format	Operation	Instruction Code	Cycle	T Bit
`OCBWB @Rn`	Writes back operand cache block	`0000nnnn10110011`	1	—

Description: This instruction accesses data using the contents indicated by effective address Rn. If the cache is hit and there is unwritten information (U bit = 1), the corresponding cache block is written back to external memory and that block is cleaned (the U bit is cleared to 0). In other cases (i.e. in the case of a cache miss or an access to a non-cache area, or if the block is already clean), no operation is performed.

Notes: None

Operation:

```
OCBWB(int n)        /* OCBWB @Rn */
      {
        if(is_dirty_block(R[n]))  write_back(R[n]);
        PC += 2;
      }
```

Possible Exceptions:

- Data TLB multiple-hit exception
- Data TLB miss exception
- Data TLB protection violation exception
- Data address error

Note that the above exceptions are generated even if OCBWB does not operate.

10.1.50 OR (OR Logical): Logical Instruction

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

Description: This instruction ORs the contents of general registers Rn and Rm and stores the result in Rn.

This instruction can be used to OR general register R0 contents with zero-extended 8-bit immediate data, or, in indexed GBR indirect addressing mode, to OR 8-bit memory with 8-bit immediate data.

Notes: None

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;
}
```

Example:

```
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
```

Possible Exceptions: Exceptions may occur when OR.B instruction is executed.

- Data TLB multiple-hit exception
- Data TLB miss exception
- Data TLB protection violation exception
- Initial page write exception
- Data address error

Exceptions are checked taking a data access by this instruction as a byte load and a byte store.

10.1.51 PREF (Prefetch Data to Cache): Data Transfer Instruction

Format	Operation	Instruction Code	Cycle	T Bit
`PREF @Rn`	(Rn) → operand cache	`0000nnnn10000011`	1	—

Description: This instruction reads a 32-byte data block starting at a 32-byte boundary into the operand cache. The lower 5 bits of the address specified by Rn are masked to zero.

This instruction does not generate data address error and MMU exceptions except data TLB multiple-hit exception. In the event of an error, the PREF instruction is treated as an NOP (no operation) instruction.

Notes: None

Operation:

```
PREF(int n) /* PREF @Rn */
{
         PC += 2;
}
```

Example:

```
          MOV.L     #SOFT_PF,R1     ;R1 address is SOFT_PF
          PREF      @R1             ;Load SOFT_PF data into on-chip cache

          .align 32
SOFT_PF:  .data.l   H'12345678
          .data.l   H'9ABCDEF0
          .data.l   H'AAAA5555
          .data.l   H'5555AAAA
          .data.l   H'11111111
          .data.l   H'22222222
          .data.l   H'33333333
          .data.l   H'44444444
```

Possible Exceptions:

- Data TLB multiple-hit exception

10.1.52 PREFI (Prefetch Instruction Cache Block): Data Transfer Instruction

Format	Operation	Instruction Code	Cycle	T Bit
`PREFI @Rn`	Invalidation of instruction cache indicated by logical address Rn	`0000nnnn11010011`	10	—

Description: This instruction reads a 32-byte block of data starting at a 32-byte boundary within the instruction cache. The lower 5 bits of the address specified by Rn are masked by zeroes.

This instruction does not generate data address error and MMU exceptions. In the event of an error, the PREFI instruction is treated as an NOP (no operation) instruction.

When the address to be prefetched is missing from UTLB or is protected, the PREFI instruction is treated as an NOP instruction and a TLB exception does not occur.

Notes: None

Operation:

```
PREFI(int n) /* PREFI @Rn*/
{
   prefetch_instruction_cache_block(R[n]);
   PC += 2;
}
```

Example:

```
       MOVA     WakeUp,R0 ; Wakeup address
       PREFI    @R0       ; Prefetching of instructions to be
                            executed after release from the SLEEP state
       SLEEP
WakeUp:
       NOP
This instruction is used, before the SLEEP command is issued, to
prefetch instructions for execution on return from the SLEEP state.
```

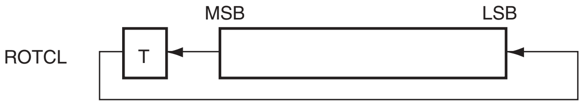
Possible Exceptions:

- Slot illegal instruction exception

10.1.53 ROTCL (Rotate with Carry Left): Shift Instruction

Format	Operation	Instruction Code	Cycle	T Bit
`ROTCL Rn`	T ← Rn ← T	`0100nnnn00100100`	1	MSB

Description: This instruction rotates the contents of general register Rn one bit to the left through the T bit, and stores the result in Rn. The bit rotated out of the operand is transferred to the T bit.



Notes: None

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;
}
```

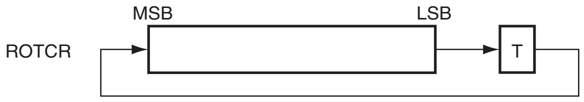
Example:

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

10.1.54 ROTCR (Rotate with Carry Right): Shift Instruction

Format	Operation	Instruction Code	Cycle	T Bit
`ROTCR Rn`	T → Rn → T	`0100nnnn00100101`	1	LSB

Description: This instruction rotates the contents of general register Rn one bit to the right through the T bit, and stores the result in Rn. The bit rotated out of the operand is transferred to the T bit.



Notes: None

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;
}
```

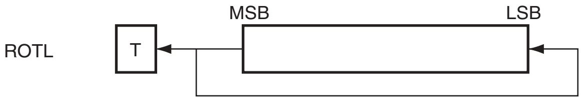
Example:

```
ROTCR  R0          ; Before execution  R0 = H'00000001, T = 1
                   ; After execution   R0 = H'80000000, T = 1
```

10.1.55 ROTL (Rotate Left): Shift Instruction

Format	Operation	Instruction Code	Cycle	T Bit
`ROTL Rn`	T ← Rn ← MSB	`0100nnnn00000100`	1	MSB

Description: This instruction rotates the contents of general register Rn one bit to the left, and stores the result in Rn. The bit rotated out of the operand is transferred to the T bit.



Notes: None

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;
}
```

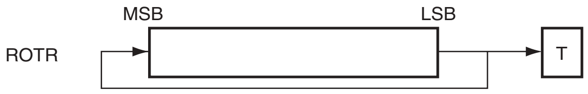
Example:

```
ROTL   R0              ; Before execution   R0 = H'80000000, T = 0
                       ; After execution    R0 = H'00000001, T = 1
```

10.1.56 ROTR (Rotate Right): Shift Instruction

Format		Operation	Instruction Code	Cycle	T Bit
ROTR	Rn	LSB → Rn → T	0100nnnn00000101	1	LSB

Description: This instruction rotates the contents of general register Rn one bit to the right, and stores the result in Rn. The bit rotated out of the operand is transferred to the T bit.



Notes: None

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;
}
```

Example:

ROTR R0	; Before execution	R0 = H'00000001, T = 0
	; After execution	R0 = H'80000000, T = 1

10.1.57 RTE (Return from Exception): System Control Instruction

Format	Operation	Instruction Code	Cycle	T Bit
RTE	SSR → SR, SPC→ PC	0000000000101011	4	—

Description: This instruction returns from an exception or interrupt handling routine by restoring the PC and SR values from SPC and SSR. Program execution continues from the address specified by the restored PC value.

RTE is a privileged instruction, and can only be used in privileged mode. Use of this instruction in user mode will cause an illegal instruction exception.

Notes: As this is a delayed branch instruction, the instruction following the RTE instruction is executed before the branch destination instruction.

Interrupts are not accepted between this instruction and the following instruction. An exception must not be generated by the instruction in this instruction's delay slot. If the following instruction is a branch instruction, it is identified as a slot illegal instruction.

If this instruction is located in the delay slot immediately following a delayed branch instruction, it is identified as a slot illegal instruction.

The SR value accessed by the instruction in the RTE delay slot is the value restored from SSR by the RTE instruction. The SR and MD values defined prior to RTE execution are used to fetch the instruction in the RTE delay slot.

Operation:

```
RTE( ) /* RTE */
{
        unsigned int temp;
        temp = PC;
        SR = SSR;
        PC = SPC;
        Delay_Slot(temp+2);
}
```

Example:

```
RTE                  ;Return to original routine.
ADD    #8,R14        ;Executed before branch.
```

Note: In a delayed branch, the actual branch operation occurs after execution of the slot instruction, but instruction execution (register updating, etc.) is in fact performed in delayed branch instruction → delay slot instruction order. For example, even if the register holding the branch destination address is modified in the delay slot, the branch destination address will still be the register contents prior to the modification.

Possible Exceptions:

- General illegal instruction exception
- Slot illegal instruction exception

10.1.58 RTS (Return from Subroutine): Branch Instruction

Format	Operation	Instruction Code	Cycle	T Bit
RTS	PR → PC	0000000000001011	1	—

Description: This instruction returns from a subroutine procedure by restoring the PC from PR. Processing continues from the address indicated by the restored PC value. This instruction can be used to return from a subroutine procedure called by a BSR or JSR instruction to the source of the call.

Notes: As this is a delayed branch instruction, the instruction following this instruction is executed before the branch destination instruction.

Interrupts are not accepted between this instruction and the following instruction. If the following instruction is a branch instruction, it is identified as a slot illegal instruction.

The instruction that restores PR must be executed before the RTS instruction. This restore instruction cannot be in the RTS delay slot.

Operation:

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

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

Example:

```
            MOV.L      TABLE,R3      ; R3 = TRGET address
            JSR        @R3           ;  Branch to TRGET.
            NOP                      ; NOP executed before branch.
            ADD        R0,R1         ; ← Subroutine procedure return destination (PR contents)
            ........
TABLE:      .data.l    TRGET         ; Jump table
            ........
TRGET:      MOV        R1,R0         ; ← Entry to procedure
            RTS                      ; PR contents → PC
            MOV        #12,R0        ; MOV executed before branch.
```

Possible Exceptions:

- Slot illegal instruction exception

10.1.59 SETS (Set S Bit): System Control Instruction

Format	Operation	Instruction Code	Cycle	T Bit
`SETS`	1 → S	`0000000001011000`	1	—

Description: This instruction sets the S bit to 1.

Notes: None

Operation:

```
SETS( ) /* SETS */
{
        S = 1;
        PC += 2;
}
```

Example:

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

10.1.60 SETT (Set T Bit): System Control Instruction

Format	Operation	Instruction Code	Cycle	T Bit
SETT	$1 \rightarrow T$	0000000000011000	1	1

Description: This instruction sets the T bit to 1.

Notes: None

Operation:

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

Example:

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

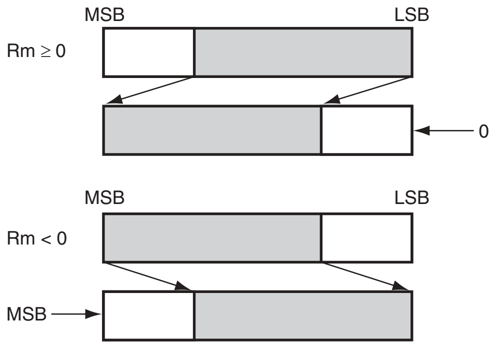
10.1.61 SHAD (Shift Arithmetic Dynamically): Shift Instruction

Format	Operation	Instruction Code	Cycle	T Bit
`SHAD Rm, Rn`	When Rm ≥ 0, Rn << Rm → Rn When Rm < 0, Rn >> Rm → [MSB → Rn]	`0100nnnnmmmm1100`	1	—

Description: This instruction arithmetically shifts the contents of general register Rn. General register Rm specifies the shift direction and the number of bits to be shifted.

Rn register contents are shifted to the left if the Rm register value is positive, and to the right if negative. In a shift to the right, the MSB is added at the upper end.

The number of bits to be shifted is specified by the lower 5 bits (bits 4 to 0) of the Rm register. If the value is negative (MSB = 1), the Rm register is represented as a two's complement. The left shift range is 0 to 31, and the right shift range, 1 to 32.



Notes: None

Operation:

```
SHAD(int m,n) /*SHAD Rm,Rn */
{
        int sgn = R[m] & 0x80000000;
        if (sgn==0)
               R[n] <<= (R[m] & 0x1F);
        else if ((R[m] & 0x1F) == 0) {
               if ((R[n] & 0x80000000) == 0)
                        R[n] = 0;
               else
                        R[n] = 0xFFFFFFFF;
       }
       else
               R[n] = (long)R[n] >> ((~R[m] & 0x1F)+1);
       PC += 2;
}
```

Example:

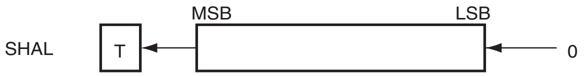
```
SHAD   R1,R2        ; Before execution   R1 = H'FFFFFFEC, R2 = H'80180000
                    ; After execution    R1 = H'FFFFFFEC, R2 = H'FFFFF801
SHAD   R3,R4        ; Before execution   R3 = H'00000014, R4 = H'FFFFF801
                    ; After execution    R3 = H'00000014, R4 = H'80100000
```

10.1.62 SHAL (Shift Arithmetic Left): Shift Instruction

Format	Operation	Instruction Code	Cycle	T Bit
`SHAL Rn`	T ← Rn ← 0	`0100nnnn00100000`	1	MSB

Description:

This instruction arithmetically shifts the contents of general register Rn one bit to the left, and stores the result in Rn. The bit shifted out of the operand is transferred to the T bit.



Notes: None

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:

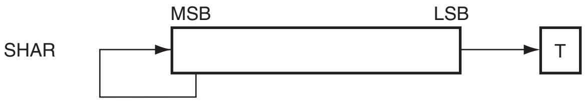
```
SHAL   R0           ; Before execution   R0 = H'80000001, T = 0
                    ; After execution    R0 = H'00000002, T = 1
```

10.1.63 SHAR (Shift Arithmetic Right): Shift Instruction

Format		Operation	Instruction Code	Cycle	T Bit
`SHAR`	`Rn`	MSB → Rn → T	`0100nnnn00100001`	1	LSB

Description:

This instruction arithmetically shifts the contents of general register Rn one bit to the right, and stores the result in Rn. The bit shifted out of the operand is transferred to the T bit.



Notes: None

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
```

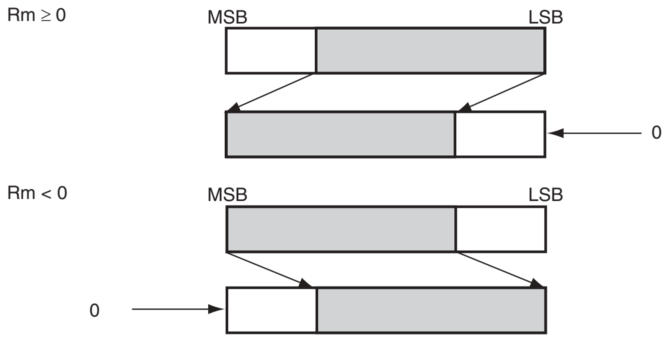
10.1.64 SHLD (Shift Logical Dynamically): Shift Instruction

Format	Operation	Instruction Code	Cycle	T Bit
`SHLD Rm, Rn`	When Rm ≥ 0, Rn << Rm → Rn When Rm < 0, Rn >> Rm → [0 → Rn]	`0100nnnnmmmm1101`	1	—

Description: This instruction logically shifts the contents of general register Rn. General register Rm specifies the shift direction and the number of bits to be shifted.

Rn register contents are shifted to the left if the Rm register value is positive, and to the right if negative. In a shift to the right, 0s are added at the upper end.

The number of bits to be shifted is specified by the lower 5 bits (bits 4 to 0) of the Rm register. If the value is negative (MSB = 1), the Rm register is represented as a two's complement. The left shift range is 0 to 31, and the right shift range, 1 to 32.



Notes: None

Operation:

```
SHLD(int m,n)/*SHLD Rm,Rn */
{
     int sgn = R[m] & 0x80000000;
     if (sgn == 0)
               R[n] <<= (R[m] & 0x1F);
     else if ((R[m] & 0x1F) == 0)
               R[n] = 0;
     else
               R[n] = (unsigned)R[n] >> ((~R[m] & 0x1F)+1);
     PC += 2;
}
```

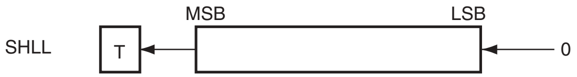
Example:

```
SHLD   R1, R2          ; Before execution   R1 = H'FFFFFFEC,  R2 = H'80180000
                       ; After execution    R1 = H'FFFFFFEC,  R2 = H'00000801
SHLD   R3, R4          ; Before execution   R3 = H'00000014,  R4 = H'FFFFF801
                       ; After execution    R3 = H'00000014,  R4 = H'80100000
```

10.1.65 SHLL (Shift Logical Left): Shift Instruction

Format	Operation	Instruction Code	Cycle	T Bit
`SHLL Rn`	T ← Rn ← 0	`0100nnnn00000000`	1	MSB

Description: This instruction logically shifts the contents of general register Rn one bit to the left, and stores the result in Rn. The bit shifted out of the operand is transferred to the T bit.



Notes: None

Operation:

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

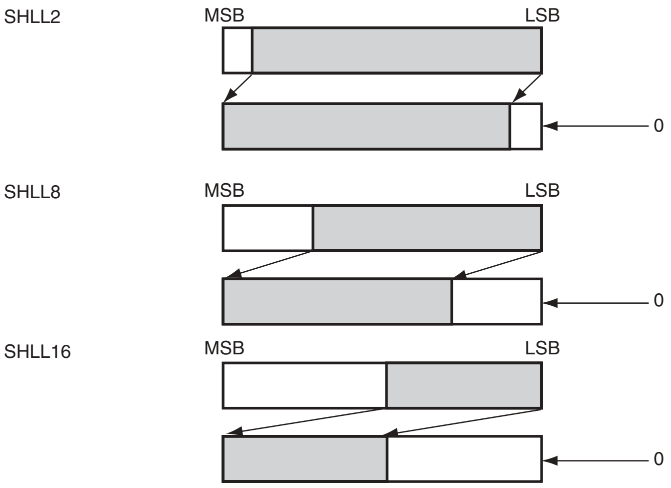
Example:

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

10.1.66 SHLLn (n bits Shift Logical Left): Shift Instruction

Format		Operation	Instruction Code	Cycle	T Bit
`SHLL2`	`Rn`	Rn<<2 → Rn	`0100nnnn00001000`	1	—
`SHLL8`	`Rn`	Rn<<8 → Rn	`0100nnnn00011000`	1	—
`SHLL16`	`Rn`	Rn<<16 → Rn	`0100nnnn00101000`	1	—

Description: This instruction logically shifts the contents of general register Rn 2, 8, or 16 bits to the left, and stores the result in Rn. The bits shifted out of the operand are discarded.



Notes: None

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;
}
```

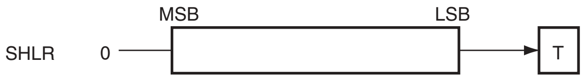
Example:

```
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
```

10.1.67 SHLR (Shift Logical Right): Shift Instruction

Format	Operation	Instruction Code	Cycle	T Bit
`SHLR Rn`	$0 \rightarrow Rn \rightarrow T$	`0100nnnn00000001`	1	LSB

Description: This instruction logically shifts the contents of general register Rn one bit to the right, and stores the result in Rn. The bit shifted out of the operand is transferred to the T bit.



Notes: None

Operation:

```
SHLR(long n)    /* SHLR Rn */
{
    if ((R[n] & 0x00000001)==0) T = 0;
    else T = 1;
    R[n] >>= 1;
    R[n] &= 0x7FFFFFFF;
    PC += 2;
}
```

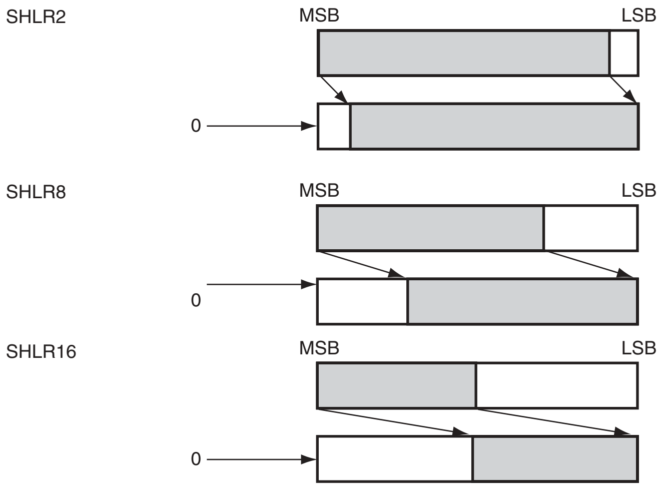
Example:

```
SHLR   R0            ; Before execution   R0 = H'80000001, T = 0
                     ; After execution    R0 = H'40000000, T = 1
```

10.1.68 SHLRn (n bits Shift Logical Right): Shift Instruction

Format		Operation	Instruction Code	Cycle	T Bit
`SHLR2`	`Rn`	Rn>>2 → Rn	`0100nnnn00001001`	1	—
`SHLR8`	`Rn`	Rn>>8 → Rn	`0100nnnn00011001`	1	—
`SHLR16`	`Rn`	Rn>>16 → Rn	`0100nnnn00101001`	1	—

Description: This instruction logically shifts the contents of general register Rn 2, 8, or 16 bits to the right, and stores the result in Rn. The bits shifted out of the operand are discarded.



Notes: None

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;
}
```

Example:

```
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
```

10.1.69 SLEEP (Sleep): System Control Instruction (Privileged Instruction)

Format	Operation	Instruction Code	Cycle	T Bit
`SLEEP`	Sleep or standby	`0000000000011011`	Undefined	—

Description: This instruction places the CPU in the power-down state.

In power-down mode, the CPU retains its internal state, but immediately stops executing instructions and waits for an interrupt request. When it receives an interrupt request, the CPU exits the power-down state.

SLEEP is a privileged instruction, and can only be used in privileged mode. Use of this instruction in user mode will cause an illegal instruction exception.

Notes: SLEEP performance depends on the standby control register (STBCR). See Power-Down Modes in the target product's hardware manual, for details.

Operation:

```
SLEEP( )     /* SLEEP */
{
    Sleep_standby();
}
```

Example:

```
SLEEP                          ;Transition to power-down mode
```

Possible Exceptions:

- General illegal instruction exception
- Slot illegal instruction exception

10.1.70 STC (Store Control Register): System Control Instruction (Privileged Instruction)

Format	Operation	Instruction Code	Cycle	T Bit
`STC GBR, Rn`	GBR → Rn	0000nnnn00010010	1	—
`STC VBR, Rn`	VBR → Rn	0000nnnn00100010	1	—
`STC SSR, Rn`	SSR → Rn	0000nnnn00110010	1	—
`STC SPC, Rn`	SPC → Rn	0000nnnn01000010	1	—
`STC SGR, Rn`	SGR → Rn	0000nnnn00111010	1	—
`STC DBR, Rn`	DBR → Rn	0000nnnn11111010	1	—
`STC R0_BANK, Rn`	R0_BANK → Rn	0000nnnn10000010	1	—
`STC R1_BANK, Rn`	R1_BANK → Rn	0000nnnn10010010	1	—
`STC R2_BANK, Rn`	R2_BANK → Rn	0000nnnn10100010	1	—
`STC R3_BANK, Rn`	R3_BANK → Rn	0000nnnn10110010	1	—
`STC R4_BANK, Rn`	R4_BANK → Rn	0000nnnn11000010	1	—
`STC R5_BANK, Rn`	R5_BANK → Rn	0000nnnn11010010	1	—
`STC R6_BANK, Rn`	R6_BANK → Rn	0000nnnn11100010	1	—
`STC R7_BANK, Rn`	R7_BANK → Rn	0000nnnn11110010	1	—
`STC.L GBR, @-Rn`	Rn-4 → Rn, GBR → (Rn)	0100nnnn00010011	1	—
`STC.L VBR, @-Rn`	Rn-4 → Rn, VBR → (Rn)	0100nnnn00100011	1	—
`STC.L SSR, @-Rn`	Rn-4 → Rn, SSR → (Rn)	0100nnnn00110011	1	—
`STC.L SPC, @-Rn`	Rn-4 → Rn, SPC → (Rn)	0100nnnn01000011	1	—
`STC.L SGR, @-Rn`	Rn-4 → Rn, SGR → (Rn)	0100nnnn00110010	1	—
`STC.L DBR, @-Rn`	Rn-4 → Rn, DBR → (Rn)	0100nnnn11110010	1	—
`STC.L R0_BANK, @-Rn`	Rn-4 → Rn, R0_BANK → (Rn)	0100nnnn10000011	1	—
`STC.L R1_BANK, @-Rn`	Rn-4 → Rn, R1_BANK → (Rn)	0100nnnn10010011	1	—
`STC.L R2_BANK, @-Rn`	Rn-4 → Rn, R2_BANK → (Rn)	0100nnnn10100011	1	—
`STC.L R3_BANK, @-Rn`	Rn-4 → Rn, R3_BANK → (Rn)	0100nnnn10110011	1	—
`STC.L R4_BANK, @-Rn`	Rn-4 → Rn, R4_BANK → (Rn)	0100nnnn11000011	1	—
`STC.L R5_BANK, @-Rn`	Rn-4 → Rn, R5_BANK → (Rn)	0100nnnn11010011	1	—
`STC.L R6_BANK, @-Rn`	Rn-4 → Rn, R6_BANK → (Rn)	0100nnnn11100011	1	—
`STC.L R7_BANK, @-Rn`	Rn-4 → Rn, R7_BANK → (Rn)	0100nnnn11110011	1	—

Description: This instruction stores control register GBR, VBR, SSR, SPC, SGR, DBR or Rm_BANK (m = 0–7) in the destination.

Rm_BANK operands are specified by the RB bit of the SR register:
when the RB bit is 1 Rm_BANK0 is accessed,
when the RB bit is 0 Rm_BANK1 is accessed.

Notes: STC/STC.L can only be used in privileged mode excepting STC GBR, Rn/STC.L GBR, @-Rn. Use of these instructions in user mode will cause illegal instruction exceptions.

Operation:

```
STCGBR(int n)     /* STC GBR,Rn */
     {
         R[n] = GBR;
         PC += 2;
     }

STCVBR(int n)     /* STC VBR,Rn : Privileged */
     {
         R[n] = VBR;
         PC += 2;
     }

STCSSR(int n)     /* STC SSR,Rn : Privileged */
     {
         R[n] = SSR;
         PC += 2;
     }

STCSPC(int n)     /* STC SPC,Rn : Privileged */
     {
         R[n] = SPC;
         PC += 2;
     }

STCSGR(int n)     /* STC SGR,Rn : Privileged */
     {
         R[n] = SGR;
         PC += 2;
     }

STCDBR(int n)     /* STC DBR,Rn : Privileged */
     {
         R[n] = DBR;
         PC += 2;
     }
```

```
STCRm_BANK(int n)      /* STC Rm_BANK,Rn : Privileged */
                       /* m=0-7 */
     {
         R[n] = Rm_BANK;
         PC += 2;
     }

STCMGBR(int n)      /* STC.L GBR,@-Rn */
     {
         R[n] -= 4;
         Write_Long(R[n],GBR);
         PC += 2;
     }

STCMVBR(int n)      /* STC.L VBR,@-Rn : Privileged */
     {
         R[n] -= 4;
         Write_Long(R[n],VBR);
         PC += 2;
     }

STCMSSR(int n)      /* STC.L SSR,@-Rn : Privileged */
     {
         R[n] -= 4;
         Write_Long(R[n],SSR);
         PC += 2;
     }

STCMSPC(int n)      /* STC.L SPC,@-Rn : Privileged */
     {
         R[n] -= 4;
         Write_Long(R[n],SPC);
         PC += 2;
     }
```

```
STCMSGR(int n)      /* STC.L SGR,@-Rn : Privileged */
     {
         R[n] -= 4;
         Write_Long(R[n],SGR);
         PC += 2;
     }

STCMDBR(int n)      /* STC.L DBR,@-Rn : Privileged */
     {
         R[n] -= 4;
         Write_Long(R[n],DBR);
         PC += 2;
     }

STCMRm_BANK(int n)      /* STC.L Rm_BANK,@-Rn : Privileged */
                        /* m=0-7 */
     {
         R[n] -= 4;
         Write_Long(R[n],Rm_BANK);
         PC += 2;
     }
```

Possible Exceptions:

- Data TLB multiple-hit exception
- General illegal instruction exception
- Slot illegal instruction exception
- Data TLB miss exception
- Data TLB protection violation exception
- Initial page write exception
- Data address error

10.1.71 STS (Store System Register): System Control Instruction

Format	Operation	Instruction Code	Cycle	T Bit
`STS MACH,Rn`	MACH → Rn	`0000nnnn00001010`	1	—
`STS MACL,Rn`	MACL → Rn	`0000nnnn00011010`	1	—
`STS PR,Rn`	PR → Rn	`0000nnnn00101010`	1	—
`STS.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: This instruction stores system register MACH, MACL, or PR in the destination.

Notes: None

Operation:

```
STSMACH(int n)     /* STS MACH,Rn */
{
   R[n] = MACH;
   PC += 2;
}

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

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

```
STSMMACH(int n)      /* STS.L MACH,@-Rn */
{
   R[n] -= 4;
   Write_Long(R[n],MACH);
   PC += 2;
}

STSMMACL(int n)      /* STS.L MACL,@-Rn */
{
   R[n] -= 4;
   Write_Long(R[n],MACL);
   PC += 2;
}

STSMPR(int 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
```

Possible Exceptions:

- Data TLB multiple-hit exception
- Data TLB miss exception
- Data TLB protection violation exception
- Initial page write exception
- Data address error

10.1.72 SUB (Subtract Binary): Arithmetic Instruction

Format	Operation	Instruction Code	Cycle	T Bit
`SUB Rm,Rn`	Rn - Rm → Rn	`0011nnnnmmmm1000`	1	—

Description: This instruction subtracts the contents of general register Rm from the contents of general register Rn and stores the result in Rn. For immediate data subtraction, ADD #imm,Rn should be used.

Notes: None

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
```

10.1.73 SUBC (Subtract with Carry): Arithmetic Instruction

Format	Operation	Instruction Code	Cycle	T Bit
`SUBC Rm,Rn`	Rn - Rm-T → Rn, borrow → T	`0011nnnnmmmm1010`	1	Borrow

Description: This instruction subtracts the contents of general register Rm and the T bit from the contents of general register Rn, and stores the result in Rn. A borrow resulting from the operation is reflected in the T bit. This instruction is used for subtractions exceeding 32 bits.

Notes: None

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;
}
```

Example:

```
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
```

10.1.74 SUBV (Subtract with (V flag) Underflow Check): Arithmetic Instruction

Format	Operation	Instruction Code	Cycle	T Bit
`SUBV Rm,Rn`	Rn - Rm → Rn, underflow → T	`0011nnnnmmmm1011`	1	Underflow

Description: This instruction subtracts the contents of general register Rm from the contents of general register Rn, and stores the result in Rn. If underflow occurs, the T bit is set.

Notes: None

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;
}
```

Example:

```
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
```

10.1.75 SWAP (Swap Register Halves): Data Transfer Instruction

Format	Operation	Instruction Code	Cycle	T Bit
`SWAP.B Rm,Rn`	Rm → lower-2-byte upper/ lower-byte swap → Rn	`0110nnnnmmmm1000`	1	—
`SWAP.W Rm,Rn`	Rm → upper-/lower-word swap → Rn	`0110nnnnmmmm1001`	1	

Description: This instruction swaps the upper and lower parts of the contents of general register Rm, and stores the result in Rn.

In the case of a byte specification, the 8 bits from bit 15 to bit 8 of Rm are swapped with the 8 bits from bit 7 to bit 0. The upper 16 bits of Rm are transferred directly to the upper 16 bits of Rn.

In the case of a word specification, the 16 bits from bit 31 to bit 16 of Rm are swapped with the 16 bits from bit 15 to bit 0.

Notes: None

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;
}
```

Example:

```
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
```

10.1.76 SYNCO (Synchronize Data Operation): Data Transfer Instruction

Format	Operation	Instruction Code	Cycle	T Bit
`SYNCO`	Data accesses invoked by the following instruction are not executed until execution of data accesses which precede this instruction has been completed.	`0000000010101011`	Undefined	—

Description: This instruction is used to synchronize data operations. When this instruction is executed, the subsequent bus accesses are not executed until the execution of all preceding bus accesses has been completed.

Notes: The SYNCO instruction can not guarantee the ordering of receipt timing which is notified by the memory-mapped peripheral resources through the method except bus when the register is changed by bus accesses. Refer to the description of each registers to guarantee this ordering.

Operation:

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

Example:

1. Ordering access to memory areas which are shared with other memory users
2. Flushing all write buffers
3. Stopping memory-access operations from merging and becoming ineffective
4. Waiting for the completion of cache-control instructions

10.1.77 TAS (Test And Set): Logical Instruction

Format		Operation	Instruction Code	Cycle	T Bit
`TAS.B`	`@Rn`	If (Rn) = 0, 1 → T, else 0 → T 1 → MSB of (Rn)	`0100nnnn00011011`	4	Test result

Description: This instruction purges the cache block corresponding to the memory area specified by the contents of general register Rn, reads the byte data indicated by that address, and sets the T bit to 1 if that data is zero, or clears the T bit to 0 if the data is nonzero. The instruction then sets bit 7 to 1 and writes to the same address. The bus is not released during this period.

The purge operation is executed as follows.

In a purge operation, data is accessed using the contents of general register Rn as the effective address. If there is a cache hit and the corresponding cache block is dirty (U bit = 1), the contents of that cache block are written back to external memory, and the cache block is then invalidated (by clearing the V bit to 0). If there is a cache hit and the corresponding cache block is clean (U bit = 0), the cache block is simply invalidated (by clearing the V bit to 0). A purge is not executed in the event of a cache miss, or if the accessed memory location is non-cacheable.

The two TAS.B memory accesses are executed automatically. Another memory access is not executed between the two TAS.B accesses.

Notes: None

Operation:

```
TAS(int n) /* TAS.B @Rn */
{
     int temp;

     temp = (int)Read_Byte(R[n]); /* Bus Lock */
     if (temp==0) T = 1;
     else T = 0;
     temp |= 0x00000080;
     Write_Byte(R[n],temp);     /* Bus unlock */
     PC += 2;
}
```

Possible Exceptions:

- Data TLB multiple-hit exception
- Data TLB miss exception
- Data TLB protection violation exception
- Initial page write exception
- Data address error

Exceptions are checked taking a data access by this instruction as a byte load and a byte store.

10.1.78 TRAPA (Trap Always): System Control Instruction

Format	Operation	Instruction Code	Cycle	T Bit
`TRAPA #imm`	Imm<<2 → TRA, PC + 2 → SPC, SR → SSR, R15 → SGR, 1 → SR.MD/BL/RB, H'160 → EXPEVT, VBR + H'00000100 → PC	`11000011iiiiiiii`	13	—

Description: This instruction starts trap exception handling. The values of (PC + 2), SR, and R15 are saved to SPC, SSR and SGR, and 8-bit immediate data is stored in the TRA register (bits 9 to 2). The processor mode is switched to privileged mode (the MD bit in SR is set to 1), and the BL bit and RB bit in SR are set to 1. As a result, exception and interrupt requests are masked (not accepted), and the BANK1 registers (R0_BANK1 to R7_BANK1) are selected. Exception code H'160 is written to the EXPEVT register (bits 11 to 0). The program branches to address (VBR + H'00000100), indicated by the sum of the VBR register contents and offset H'00000100.

Notes: None

Operation:

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

     imm = (0x000000FF & i);
     TRA = imm<<2;
     SSR = SR;
     SPC = PC+2;
     SGR = R15;
     SR.MD = 1;
     SR.BL = 1;
     SR.RB=1;
     EXPEVT = 0x00000160;
     PC = VBR + 0x00000100;
}
```

Possible Exceptions:

- Unconditional trap
- Slot illegal instruction exception

10.1.79 TST (Test Logical): Logical Instruction

Format	Operation	Instruction Code	Cycle	T Bit
`TST Rm,Rn`	Rn & Rm; if result is 0, 1 → T, else 0 → T	`0010nnnnmmmm1000`	1	Test result
`TST #imm,R0`	R0 & imm; if result is 0, 1 → T, else 0 → T	`11001000iiiiiiii`	1	Test result
`TST.B #imm,@(R0,GBR)`	(R0 + GBR) & imm; if result is 0, 1 → T, else 0 → T	`11001100iiiiiiii`	3	Test result

Description: This instruction ANDs the contents of general registers Rn and Rm, and sets the T bit if the result is zero. If the result is nonzero, the T bit is cleared. The contents of Rn are not changed.

This instruction can be used to AND general register R0 contents with zero-extended 8-bit immediate data, or, in indexed GBR indirect addressing mode, to AND 8-bit memory with 8-bit immediate data. The contents of R0 or the memory are not changed.

Notes: None

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)  /* TST #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;
}
```

Example:

```
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
```

Possible Exceptions: Exceptions may occur when TST.B instruction is executed.

- Data TLB multiple-hit exception
- Data TLB miss exception
- Data TLB protection violation exception
- Initial page write exception
- Data address error

Exceptions are checked taking a data access by this instruction as a byte load and a byte store.

10.1.80 XOR (Exclusive OR Logical): Logical Instruction

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

Description: This instruction exclusively ORs the contents of general registers Rn and Rm, and stores the result in Rn.

This instruction can be used to exclusively OR register R0 contents with zero-extended 8-bit immediate data, or, in indexed GBR indirect addressing mode, to exclusively OR 8-bit memory with 8-bit immediate data.

Notes: None

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) */
{
     int temp;

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

Example:

```
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
```

Possible Exceptions: Exceptions may occur when XOR.B instruction is executed.

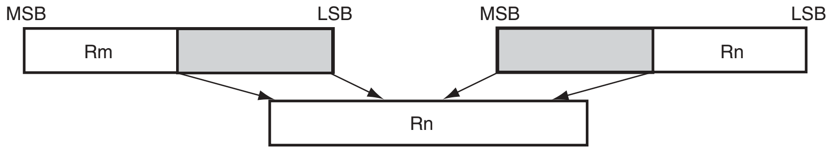
- Data TLB multiple-hit exception
- Data TLB miss exception
- Data TLB protection violation exception
- Initial page write exception
- Data address error

Exceptions are checked taking a data access by this instruction as a byte load and a byte store.

10.1.81 XTRCT (Extract): Data Transfer Instruction

Format	Operation	Instruction Code	Cycle	T Bit
`XTRCT Rm,Rn`	Middle 32 bits of Rm:Rn → Rn	`0010nnnnmmmm1101`	1	—

Description: This instruction extracts the middle 32 bits from the 64-bit contents of linked general registers Rm and Rn, and stores the result in Rn.



Notes: None

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
```

10.2 CPU Instructions (FPU related)

Of the SH-4A CPU's instructions, those which support the FPU and those which differ in function from instructions of the SH3A-DSP are described in this section.

10.2.1 BSR (Branch to Subroutine): Branch Instruction (Delayed Branch Instruction)

Format	Operation	Instruction Code	Cycle	T Bit
`BSR label`	PC+4 $\rightarrow$ PR, PC+4+disp$\times$2 $\rightarrow$ PC	`1011dddddddddddd`	1	—

Description: This instruction branches to address (PC + 4 + displacement × 2), and stores address (PC + 4) in PR. The PC source value is the BSR instruction address. As the 12-bit displacement is multiplied by two after sign-extension, the branch destination can be located in the range from –4096 to +4094 bytes from the BSR instruction. If the branch destination cannot be reached, this branch can be performed with a JSR instruction.

Notes: As this is a delayed branch instruction, the instruction following this instruction is executed before the branch destination instruction.

Interrupts are not accepted between this instruction and the following instruction. If the following instruction is a branch instruction, it is identified as a slot illegal instruction.

Operation:

```
BSR(int d)   /* BSR disp */
{
    int disp;
    unsigned int temp;

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

Example

```
        BSR   TRGET          ;Branch to TRGET.
        MOV   R3,R4          ;MOV executed before branch.
        ADD   R0,R1          ;Subroutine procedure return destination (contents of PR)
        .....
        .....
  TRGET:                     ;← Entry to procedure
        MOV   R2,R3          ;
        RTS                  ;Return to above ADD instruction.
        MOV   #1,R0          ;MOV executed before branch.
```

Possible Exceptions:

- Slot illegal instruction exception

10.2.2 BSRF (Branch to Subroutine Far): Branch Instruction (Delayed Branch Instruction)

Format	Operation	Instruction Code	Cycle	T Bit
`BSRF Rn`	PC+4 → PR, PC+4+Rn → PC	`0000nnnn00000011`	1	—

Description: This instruction branches to address (PC + 4 + Rn), and stores address (PC + 4) in PR. The PC source value is the BSRF instruction address. The branch destination address is the result of adding the 32-bit contents of general register Rn to PC + 4.

Notes: As this is a delayed branch instruction, the instruction following this instruction is executed before the branch destination instruction.

Interrupts are not accepted between this instruction and the following instruction. If the following instruction is a branch instruction, it is identified as a slot illegal instruction.

Operation:

```
BSRF(int n)     /* BSRF Rn */
{
    unsigned int temp;

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

Example:

```
    MOV.L   #(TRGET-BSRF_PC),R0   ;Set displacement.
    BSRF    R0                    ;Branch to TRGET.
    MOV     R3,R4                 ;MOV executed before branch.
  BSRF_PC:                        ;
    ADD     R0,R1                 ;
     .....
  TRGET:                          ;← Entry to procedure
    MOV     R2,R3                 ;
    RTS                           ;Return to above ADD instruction.
    MOV     #1,R0                 ;MOV executed before branch.
```

Possible Exceptions:

- Slot illegal instruction exception

10.2.3 JSR (Jump to Subroutine): Branch Instruction (Delayed Branch Instruction)

Format	Operation	Instruction Code	Cycle	T Bit
`JSR @Rn`	PC+4 → PR, Rn → PC	`0100nnnn00001011`	1	—

Description: This instruction makes a delayed branch to the subroutine procedure at the specified address after execution of the following instruction. Return address (PC + 4) is saved in PR, and a branch is made to the address indicated by general register Rn. JSR is used in combination with RTS for subroutine procedure calls.

Notes: As this is a delayed branch instruction, the instruction following this instruction is executed before the branch destination instruction.

Interrupts are not accepted between this instruction and the following instruction. If the following instruction is a branch instruction, it is identified as a slot illegal instruction.

Operation:

```
JSR(int n)/* JSR @Rn */
{
    unsigned int temp;

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

Example:

```
                MOV.L       JSR_TABLE,R0    ; R0 = TRGET address
                JSR         @R0             ; Branch to TRGET.
                XOR         R1,R1           ; XOR executed before branch.
                ADD         R0,R1           ; ← Procedure return destination (PR contents)
                .......
                .align      4
JSR_TABLE:      .data.l     TRGET           ; Jump table
TRGET:          NOP                         ; ← Entry to procedure
                MOV         R2,R3           ;
                RTS                         ; Return to above ADD instruction.
                MOV         #70,R1          ; MOV executed before RTS.
```

Possible Exceptions:

- Slot illegal instruction exception

10.2.4 LDC (Load to Control Register): System Control Instruction (Privileged Instruction)

Format	Operation	Instruction Code	Cycle	T Bit
`LDC Rm,SR`	Rm → SR	`0100mmmm00001110`	4	LSB
`LDC.L @Rm+,SR`	(Rm) → SR, Rm+4 → Rm	`0100mmmm00000111`	4	LSB

Description: This instruction stores the source operand in the control register SR.

Notes: This instruction is only usable in privileged mode. Issuing this instruction in user mode will cause an illegal instruction exception.

Operation:

```
LDCSR(int m)      /* LDC Rm,SR : Privileged */
{
    SR = R[m] & 0x700083F3;
    PC += 2;
}
LDCMSR(int m)      /* LDC.L @Rm+,SR: Privileged */
{
    SR = Read_Long(R[m]) & 0x700083F3;
    R[m] += 4;
    PC += 2;
}
```

Possible exception:

- Data TLB multiple-hit exception
- General illegal instruction exception
- Slot illegal instruction exception
- Data TLB miss exception
- Data TLB protection violation exception
- Data address error

10.2.5 LDS (Load to FPU System register): System Control Instruction

Format	Operation	Instruction Code	Cycle	T Bit
`LDS Rm,FPUL`	Rm → FPUL	0100mmmm01011010	1	—
`LDS.L @Rm+,FPUL`	(Rm) → FPUL, Rm + 4 → Rm	0100mmmm01010110	1	—
`LDS Rm,FPSCR`	Rm → FPSCR	0100mmmm01101010	1	—
`LDS.L @Rm+,FPSCR`	(Rm) → FPSCR, Rm + 4 → Rm	0100mmmm01100110	1	—

Description: This instruction loads the source operand into FPU system registers FPUL and FPSCR.

Notes: None

Operation:

```
#define FPSCR_MASK 0x003FFFFF

LDSFPUL(int m, int *FPUL)        /* LDS Rm,FPUL  */
{
     *FPUL = R[m];
     PC += 2;
}
LDSMFPUL(int m, int *FPUL)        /* LDS.L @Rm+,FPUL  */
{
     *FPUL = Read_Long(R[m]);
     R[m] += 4;
     PC += 2;
}
LDSFPSCR(int  m)        /* LDS Rm,FPSCR  */
{
     FPSCR = R[m] & FPSCR_MASK;
     PC += 2;
}
```

```
LDSMFPSCR(int  m)        /* LDS.L @Rm+,FPSCR  */
{
     FPSCR = Read_Long(R[m]) & FPSCR_MASK;
     R[m] += 4;
     PC += 2;
}
```

Possible Exceptions:

- Data TLB multiple-hit exception
- Data TLB miss exception
- Data TLB protection violation exception
- Data address error

10.2.6 STC (Store Control Register): System Control Instruction (Privileged Instruction)

Format	Operation	Instruction Code	Cycle	T Bit
`STC SR,Rn`	SR → Rn	`0000nnnn00000010`	1	—
`STC.L SR,@-Rn`	Rn - 4 →Rn, SR → (Rn)	`0100nnnn00000011`	1	—

Description: This instruction stores the control register SR in the destination.

Notes: STC can only be used in privileged mode. Use of this instruction in user mode will cause illegal instruction exception.

Operation:

```
STCSR(int n)      /* STC SR,Rn : Privileged */
{
      R[n] = SR;
      PC += 2;
}
STCMSR(int n)     /* STC.L SR,@-Rn : Privileged */
{
      R[n] -= 4;
      Write_Long(R[n],SR);
      PC += 2;
}
```

Possible exceptions:

- Data TLB multiple-hit exception
- General illegal instruction exception
- Slot illegal instruction exception
- Data TLB miss exception
- Data TLB protection violation exception
- Initial page write exception
- Data address error

10.2.7 STS (Store from FPU System Register): System Control Instruction

Format	Operation	Instruction Code	Cycle	T Bit
`STS FPUL,Rn`	FPUL → Rn	0000nnnn01011010	1	—
`STS FPSCR,Rn`	FPSCR → Rn	0000nnnn01101010	1	—
`STS.L FPUL,@-Rn`	Rn-4 → Rn, FPUL → (Rn)	0100nnnn01010010	1	—
`STS.L FPSCR,@-Rn`	Rn-4 → Rn, FPSCR → (Rn)	0100nnnn01100010	1	—

Description: This instruction stores FPU system register FPUL or FPSCR in the destination.

Notes: None

Operation:

```
STSFPUL(int n, int *FPUL)           /* STS FPUL,Rn  */
{
     R[n] = *FPUL;
     PC += 2;
}
STSMFPUL(int n, int *FPUL)     /* STS.L FPUL,@-Rn  */
{
     R[n] -= 4;
     Write_Long(R[n],*FPUL) ;
     PC += 2;
}
STSFPSCR(int  n)          /* STS FPSCR,Rn  */
{
     R[n] = FPSCR & 0x003FFFFF;
     PC += 2;
}
STSMFPSCR(int  n)  /* STS.L FPSCR,@-Rn  */
{
     R[n] -= 4;
     Write_Long(R[n],FPSCR & 0x003FFFFF)
     PC += 2;
}
```

Examples:

- STS

  Example 1:

```
MOV.L   #H'12ABCDEF, R12
LDS     R12, FPUL
STS     FPUL, R13
                      ; After executing the STS instruction:
                      ; R13 = 12ABCDEF
```

  Example 2:

```
STS     FPSCR, R2
                      ; After executing the STS instruction:
                      ; The current content of FPSCR is stored in register R2
```

- STS.L

  Example 1:

```
MOV.L   #H'0C700148, R7
STS.L   FPUL, @-R7
                      ; Before executing the STS.L instruction:
                      ; R7 = 0C700148
                      ; After executing the STS.L instruction:
                      ; R7 = 0C700144, and the content of FPUL is saved at memory
                      ; location 0C700144.
```

  Example 2:

```
MOV.L   #H'0C700154, R8
STS.L   FPSCR, @-R8
                      ; After executing the STS.L instruction:
                      ; The content of FPSCR is saved at memory location 0C700150.
```

Possible Exceptions:

- Data TLB multiple-hit exception
- Data TLB miss exception
- Data TLB protection violation exception
- Initial page write exception
- Data address error

10.3 FPU Instruction

The following resources and functions are for use in C-language descriptions of the operation of FPU instructions and supplement the resources and functions used in describing the operation of CPU instructions.

These are floating-point number definition statements.

```
#define PZERO        0
#define NZERO        1
#define DENORM       2
#define NORM         3
#define PINF         4
#define NINF         5
#define qNaN         6
#define sNaN         7
#define EQ           0
#define GT           1
#define LT           2
#define UO           3
#define INVALID      4
#define FADD         0
#define FSUB         1

#define CAUSE        0x0003f000  /* FPSCR(bit17-12) */
#define SET_E        0x00020000  /* FPSCR(bit17) */
#define SET_V        0x00010040  /* FPSCR(bit16,6) */
#define SET_Z        0x00008020  /* FPSCR(bit15,5) */
#define SET_O        0x00004010  /* FPSCR(bit14,4) */
#define SET_U        0x00002008  /* FPSCR(bit13,3) */
#define SET_I        0x00001004  /* FPSCR(bit12,2) */
#define ENABLE_VOUI  0x00000b80  /* FPSCR(bit11,9-7) */
#define ENABLE_V     0x00000800  /* FPSCR(bit11) */
#define ENABLE_Z     0x00000400  /* FPSCR(bit10) */
#define ENABLE_OUI   0x00000380  /* FPSCR(bit9-7) */
#define ENABLE_I     0x00000080  /* FPSCR(bit7) */
#define FLAG         0x0000007C  /* FPSCR(bit6-2) */
```

```
#define FPSCR_FR    FPSCR>>21&1
#define FPSCR_PR    FPSCR>>19&1
#define FPSCR_DN    FPSCR>>18&1
#define FPSCR_I     FPSCR>>12&1
#define FPSCR_RM    FPSCR&1
#define FR_HEX      frf.l[ FPSCR_FR]
#define FR          frf.f[ FPSCR_FR]
#define DR_HEX      frf.l[ FPSCR_FR]
#define DR          frf.d[ FPSCR_FR]
#define XF_HEX      frf.l[~FPSCR_FR]
#define XF          frf.f[~FPSCR_FR]
#define XD          frf.d[~FPSCR_FR]

union {
      int  l[2][16];
      float  f[2][16];
      double d[2][8];
} frf;
int FPSCR;

int sign_of(int n)
{
      return(FR_HEX[n]>>31);
}
int data_type_of(int n) {
int abs;
     abs = FR_HEX[n] & 0x7fffffff;
     if(FPSCR_PR == 0) { /* Single-precision */
         if(abs < 0x00800000){
             if((FPSCR_DN == 1) || (abs == 0x00000000)){
                 if(sign_of(n) == 0)  {zero(n, 0); return(PZERO);}
                 else                 {zero(n, 1); return(NZERO);}
             }
             else                          return(DENORM);
         }
```

```
          else if(abs < 0x7f800000)    return(NORM);
          else if(abs == 0x7f800000) {
              if(sign_of(n) == 0)      return(PINF);
              else                     return(NINF);
          }
          else if(abs < 0x7fc00000)    return(qNaN);
          else                         return(sNaN);
      }
      else {   /* Double-precision */
          if(abs < 0x00100000){
              if((FPSCR_DN == 1) ||
                ((abs == 0x00000000) && (FR_HEX[n+1] == 0x00000000)){
                  if(sign_of(n) == 0)  {zero(n, 0); return(PZERO);}
                  else                 {zero(n, 1); return(NZERO);}
             }
             else                 return(DENORM);
          }
      }
          else if(abs < 0x7ff00000) return(NORM);
          else if((abs == 0x7ff00000) &&
                  (FR_HEX[n+1] == 0x00000000)) {
              if(sign_of(n) == 0) return(PINF);
              else                return(NINF);
          }
          else if(abs < 0x7ff80000) return(qNaN);
          else                      return(sNaN);
      }
}
void register_copy(int m,n)
{
                        FR[n]   = FR[m];
     if(FPSCR_PR == 1)   FR[n+1] = FR[m+1];
}
void normal_faddsub(int m,n,type)
{
```

```
union {
     float f;
     int l;
}    dstf,srcf;
union {
     long d;
     int l[2];
}    dstd,srcd;
union {                /* "long double" format: */
     long double x;    /*    1-bit sign           */
     int l[4];         /*   15-bit exponent       */
}    dstx;             /*  112-bit mantissa       */
     if(FPSCR_PR == 0) {
         if(type == FADD)      srcf.f =  FR[m];
         else                     srcf.f = -FR[m];
         dstd.d = FR[n]; /* Conversion from single-precision to double-precision */
         dstd.d += srcf.f;
         if(((dstd.d == FR[n]) && (srcf.f != 0.0)) ||
             ((dstd.d == srcf.f) && (FR[n] != 0.0))) {
             set_I();
             if(sign_of(m)^ sign_of(n)) {
                 dstd.l[1] -= 1;
                 if(dstd.l[1] == 0xffffffff) dstd.l[0] -= 1;
             }
         }
         if(dstd.l[1] & 0x1fffffff) set_I();
         dstf.f += srcf.f; /* Round to nearest */
         if(FPSCR_RM == 1) {
             dstd.l[1] &= 0xe0000000; /* Round to zero */
             dstf.f = dstd.d;
         }
         check_single_exception(&FR[n],dstf.f);
     } else {
         if(type == FADD)  srcd.d =  DR[m>>1];
         else              srcd.d = -DR[m>>1];
         dstx.x = DR[n>>1];
                      /* Conversion from double-precision to extended double-precision */
         dstx.x += srcd.d;
         if(((dstx.x == DR[n>>1]) && (srcd.d != 0.0)) ||
             ((dstx.x == srcd.d) && (DR[n>>1] != 0.0)) ) {
             set_I();
             if(sign_of(m)^ sign_of(n)) {
                 dstx.l[3] -= 1;
                 if(dstx.l[3] == 0xffffffff) {dstx.l[2] -= 1;
                 if(dstx.l[2] == 0xffffffff) {dstx.l[1] -= 1;
                 if(dstx.l[1] == 0xffffffff) {dstx.l[0] -= 1;}}}
             }
         }
         if((dstx.l[2] & 0x0fffffff) || dstx.l[3]) set_I();
         dst.d += srcd.d; /* Round to nearest */
         if(FPSCR_RM == 1) {
             dstx.l[2] &= 0xf0000000; /* Round to zero */
             dstx.l[3]  = 0x00000000;
             dst.d = dstx.x;
         }
         check_double_exception(&DR[n>>1] ,dst.d);
     }
}
void normal_fmul(int m,n)
{
union {
     float f;
     int l;
}    tmpf;
union {
     double d;
     int l[2];
}    tmpd;
```

```
union {
     long double x;
     int l[4];
}    tmpx;
     if(FPSCR_PR == 0) {
         tmpd.d = FR[n]; /* Single-precision to double-precision */
         tmpd.d *= FR[m]; /* Precise creation */
         tmpf.f *= FR[m]; /* Round to nearest */
         if(tmpf.f != tmpd.d) set_I();
         if((tmpf.f > tmpd.d) && (FPSCR_RM == 1)) {
             tmpf.l -= 1; /* Round to zero */
         }
         check_single_exception(&FR[n],tmpf.f);
     } else {
         tmpx.x = DR[n>>1]; /* Single-precision to double-precision */
         tmpx.x *= DR[m>>1]; /* Precise creation */
         tmpd.d *= DR[m>>1]; /* Round to nearest */
         if(tmpd.d != tmpx.x) set_I();
         if(tmpd.d > tmpx.x) && (FPSCR_RM == 1)) {
             tmpd.l[1] -= 1; /* Round to zero */
             if(tmpd.l[1] == 0xffffffff) tmpd.l[0] -= 1;
         }
         check_double_exception(&DR[n>>1], tmpd.d);
     }
}
void fipr(int m,n)
{
union {
     double d;
     int l[2];
}    mlt[4];
```

```
 float dstf;
      if((data_type_of(m) == sNaN) || (data_type_of(n) == sNaN) ||
           (data_type_of(m+1) == sNaN) || (data_type_of(n+1) == sNaN) ||
           (data_type_of(m+2) == sNaN) || (data_type_of(n+2) == sNaN) ||
           (data_type_of(m+3) == sNaN) || (data_type_of(n+3) == sNaN) ||
           (check_product_invalid(m,n)) ||
           (check_product_invalid(m+1,n+1)) ||
           (check_product_invalid(m+2,n+2)) ||
           (check_product_invalid(m+3,n+3)) )   invalid(n+3);
      else if((data_type_of(m) == qNaN)|| (data_type_of(n) == qNaN)||
           (data_type_of(m+1) == qNaN) || (data_type_of(n+1) == qNaN) ||
           (data_type_of(m+2) == qNaN) || (data_type_of(n+2) == qNaN) ||
           (data_type_of(m+3) == qNaN) || (data_type_of(n+3) == qNaN))
 qnan(n+3);
      else if (check_ positive_infinity() &&
               (check_ negative_infinity())       invalid(n+3);
      else if (check_ positive_infinity())        inf(n+3,0);
      else if (check_ negative_infinity())        inf(n+3,1);
      else {
           for(i=0;i<4;i++) {
               /* If FPSCR_DN == 1, zeroize  */
               if      (data_type_of(m+i) == PZERO)  FR[m+i] = +0.0;
               else if(data_type_of(m+i) == NZERO)  FR[m+i] = -0.0;
               if      (data_type_of(n+i) == PZERO)  FR[n+i] = +0.0;
               else if(data_type_of(n+i) == NZERO)  FR[n+i] = -0.0;
               mlt[i].d = FR[m+i];
               mlt[i].d *= FR[n+i];

      /* To be precise, with FIPR, the lower 18 bits are discarded; therefore, this description
         is simplified, and differs from the hardware.  */
           mlt[i].l[1] &= 0xff000000;
           mlt[i].l[1] |= 0x00800000;
           }
           mlt[0].d += mlt[1].d + mlt[2].d + mlt[3].d;
           mlt[0].l[1] &= 0xff800000;
           dstf = mlt[0].d;
           set_I();
```

```
            check_single_exception(&FR[n+3],dstf);
        }
}
void check_single_exception(float *dst,result)
{
union {
        float f;
        int l;
}       tmp;
float abs;
        if(result < 0.0)   tmp.l = 0xff800000; /* −infinity */
        else               tmp.l = 0x7f800000; /* +infinity */
        if(result == tmp.f) {
            set_O(); set_I();
            if(FPSCR_RM == 1)      {
                tmp.l -= 1; /* Maximum value of normalized number */
                result = tmp.f;
            }
        }
        if(result < 0.0)   abs = -result;
        else               abs =  result;
        tmp.l = 0x00800000; /* Minimum value of normalized number */
        if(abs < tmp.f) {
            if((FPSCR_DN == 1) && (abs != 0.0)) {
                set_I();
                if(result < 0.0) result = -0.0; /* Zeroize denormalized number */
                else             result =  0.0;
            }
            if(FPSCR_I == 1) set_U();
        }
        if(FPSCR & ENABLE_OUI) fpu_exception_trap();
        else                    *dst = result;
}
```

```
void check_double_exception(double *dst,result)
{
union {
     double d;
     int l[2];
}    tmp;
double abs;
     if(result < 0.0)  tmp.l[0] = 0xfff00000; /* –infinity */
     else              tmp.l[0] = 0x7ff00000; /* +infinity */
                       tmp.l[1] = 0x00000000;
     if(result == tmp.d)
          set_O(); set_I();
          if(FPSCR_RM == 1) {
               tmp.l[0] -= 1;
               tmp.l[1] = 0xffffffff;
               result = tmp.d; /* Maximum value of normalized number */
          }
     }
     if(result < 0.0)  abs = -result;
     else              abs =  result;
     tmp.l[0] = 0x00100000; /* Minimum value of normalized number */
     tmp.l[1] = 0x00000000;
     if(abs < tmp.d) {
          if((FPSCR_DN == 1) && (abs != 0.0)) {
               set_I();
               if(result < 0.0) result = -0.0;
                             /* Zeroize denormalized number */
               else             result =  0.0;
          }
          if(FPSCR_I == 1) set_U();
     }
     if(FPSCR & ENABLE_OUI) fpu_exception_trap();
     else                    *dst = result;
}
```

```
int check_product_invalid(int m,n)
{
     return(check_product_infinity(m,n)  &&
           ((data_type_of(m) == PZERO) || (data_type_of(n) == PZERO) ||
            (data_type_of(m) == NZERO) || (data_type_of(n) == NZERO)));
}
int check_ product_infinity(int m,n)
{
     return((data_type_of(m) == PINF) || (data_type_of(n) == PINF) ||
            (data_type_of(m) == NINF) || (data_type_of(n) == NINF));
}
int check_ positive_infinity(int m,n)
{
     return(((check_ product_infinity(m,n) && (~sign_of(m)^
sign_of(n))) ||
     ((check_ product_infinity(m+1,n+1) && (~sign_of(m+1)^
sign_of(n+1))) ||
     ((check_ product_infinity(m+2,n+2) && (~sign_of(m+2)^
sign_of(n+2))) ||
     ((check_ product_infinity(m+3,n+3) && (~sign_of(m+3)^
sign_of(n+3))));
}
int check_ negative_infinity(int m,n)
{
  return(((check_ product_infinity(m,n) && (sign_of(m)^ sign_of(n))) ||
     ((check_ product_infinity(m+1,n+1) && (sign_of(m+1)^
sign_of(n+1))) ||
     ((check_ product_infinity(m+2,n+2) && (sign_of(m+2)^
sign_of(n+2))) ||
     ((check_ product_infinity(m+3,n+3) && (sign_of(m+3)^
sign_of(n+3))));
}
void clear_cause () {FPSCR &= ~CAUSE;}
void set_E() {FPSCR |= SET_E; fpu_exception_trap();}
void set_V() {FPSCR |= SET_V;}
void set_Z() {FPSCR |= SET_Z;}
void set_O() {FPSCR |= SET_O;}
void set_U() {FPSCR |= SET_U;}
void set_I() {FPSCR |= SET_I;}
void invalid(int n)
{
     set_V();
     if((FPSCR & ENABLE_V) == 0 qnan(n);
     else    fpu_exception_trap();
}

void dz(int n,sign)
{
     set_Z();
     if((FPSCR & ENABLE_Z) == 0 inf(n,sign);
     else    fpu_exception_trap();
}
void zero(int n,sign)
{
     if(sign == 0)    FR_HEX [n]   = 0x00000000;
     else             FR_HEX [n]   = 0x80000000;
     if (FPSCR_PR==1) FR_HEX [n+1] = 0x00000000;
}
void inf(int n,sign) {
     if (FPSCR_PR==0) {
         if(sign == 0)  FR_HEX [n]   = 0x7f800000;
         else           FR_HEX [n]   = 0xff800000;
     } else {
         if(sign == 0)  FR_HEX [n]   = 0x7ff00000;
         else           FR_HEX [n]   = 0xfff00000;
                        FR_HEX [n+1] = 0x00000000;
     }
}
void qnan(int n)
{
     if (FPSCR_PR==0)  FR[n]   = 0x7fbfffff;
     else {            FR[n]   = 0x7ff7ffff;
                       FR[n+1] = 0xffffffff;
     }
}
```

10.3.1 FABS (Floating-point Absolute Value): Floating-Point Instruction

PR	Format	Operation	Instruction Code	Cycle	T Bit
0	`FABS FRn`	FRn & H'7FFFFFFF → FRn	`1111nnnn01011101`	1	—
1	`FABS DRn`	DRn & H'7FFFFFFFFFFFFFFF → DRn	`1111nnn001011101`	1	—

Description: This instruction clears the most significant bit of the contents of floating-point register FRn/DRn to 0, and stores the result in FRn/DRn.

The cause and flag fields in FPSCR are not updated.

Notes: None

Operation:

```
void FABS (int n){
     FR[n] = FR[n] & 0x7fffffff;
     pc += 2;
}
/* Same operation is performed regardless of precision. */
```

Possible Exceptions: None

10.3.2 FADD (Floating-point ADD): Floating-Point Instruction

PR	Format	Operation	Instruction Code	Cycle	T Bit
0	`FADD FRm,FRn`	FRn+FRm → FRn	`1111nnnnmmmm0000`	1	—
1	`FADD DRm,DRn`	DRn+DRm → DRn	`1111nnn0mmm00000`	1	—

Description: When FPSCR.PR = 0: Arithmetically adds the two single-precision floating-point numbers in FRn and FRm, and stores the result in FRn.

When FPSCR.PR = 1: Arithmetically adds the two double-precision floating-point numbers in DRn and DRm, and stores the result in DRn.

When FPSCR.enable.I is set, an FPU exception trap is generated regardless of whether or not an exception has occurred. When FPSCR.enable.O/U is set, FPU exception traps are generated on actual generation by the FPU exception source and on the satisfaction of certain special conditions that apply to this the instruction. These special conditions are described in the remaining parts of this section. When an exception occurs, correct exception information is reflected in FPSCR.cause and FPSCR.flag, and FRn or DRn is not updated. Appropriate processing should therefore be performed by software.

Notes: None

Operation:

```
void FADD (int m,n)
{
     pc += 2;
     clear_cause();
     if((data_type_of(m) == sNaN) ||
         (data_type_of(n) == sNaN)) invalid(n);
     else if((data_type_of(m) == qNaN) ||
             (data_type_of(n) == qNaN)) qnan(n);
     else if((data_type_of(m) == DENORM) ||
             (data_type_of(n) == DENORM)) set_E();
     else switch (data_type_of(m)){
         case NORM: switch (data_type_of(n)){
             case NORM:    normal_faddsub(m,n,ADD); break;
             case PZERO:
             case NZERO:register_copy(m,n); break;
             default:      break;
```

```
            }           break;
            case PZERO: switch (data_type_of(n)){
                case NZERO:   zero(n,0); break;
                default:      break;
            }           break;
            case NZERO:    break;
            case PINF: switch (data_type_of(n)){
                case NINF:     invalid(n);     break;
                default:       inf(n,0);       break;
            }           break;
            case NINF: switch (data_type_of(n)){
                case PINF:     invalid(n);     break;
                default:       inf(n,1);       break;
            }           break;
        }
    }
```

FADD Special Cases

FADD	FRn,DRm							
FRm,DRm	+NORM	-NORM	+0	–0	+inf	–inf	qNaN	sNaN
+NORM	FADD							
-NORM								
+0			+0					
–0				–0		–inf		
+inf					+inf	invalid		
–inf				–inf	invalid	–inf		
qNaN							qNaN	
sNaN								invalid

Note: When DN = 1, the value of a denormalized number is treated as 0.
When DN = 0, calculation for denormalized numbers is the same as for normalized numbers.

Possible Exceptions and Overflow/Underflow Exception Trap Generating Conditions:

- FPU error
- Invalid operation
- Overflow

  Generation of overflow-exception traps

  FPSCR.PR = 0: FRn and FRm have the same sign and the exponent of at least one value is H'FE

  FPSCR.PR = 1: DRn and DRm have the same sign and the exponent of at least one value is H'7FE
- Underflow

  Generation of underflow-exception traps
  FPSCR.PR = 0: FRn and FRm have different signs and neither has an exponent greater than H'18

  FPSCR.PR = 1: DRn and DRm have different signs and neither has an exponent greater than H'035
- Inexact

10.3.3 FCMP (Floating-point Compare): Floating-Point Instruction

No.	PR	Format	Operation	Instruction Code	Cycle	T Bit
1.	0	`FCMP/EQ FRm,FRn`	When FRn = FRm,1 $\rightarrow$ T Otherwise, 0 $\rightarrow$ T	`1111nnnnmmmm0100`	1	1/0
2.	1	`FCMP/EQ DRm,DRn`	When DRn = DRm,1 $\rightarrow$ T Otherwise, 0 $\rightarrow$ T	`1111nnn0mmm00100`	1	1/0
3.	0	`FCMP/GT FRm,FRn`	When FRn > FRm,1 $\rightarrow$ T Otherwise, 0 $\rightarrow$ T	`1111nnnnmmmm0101`	1	1/0
4.	1	`FCMP/GT DRm,DRn`	When DRn > DRm,1 $\rightarrow$ T Otherwise, 0 $\rightarrow$ T	`1111nnn0mmm00101`	1	1/0

Description:

1. When FPSCR.PR = 0: Arithmetically compares the two single-precision floating-point numbers in FRn and FRm, and stores 1 in the T bit if they are equal, or 0 otherwise.
2. When FPSCR.PR = 1: Arithmetically compares the two double-precision floating-point numbers in DRn and DRm, and stores 1 in the T bit if they are equal, or 0 otherwise.
3. When FPSCR.PR = 0: Arithmetically compares the two single-precision floating-point numbers in FRn and FRm, and stores 1 in the T bit if FRn > FRm, or 0 otherwise.
4. When FPSCR.PR = 1: Arithmetically compares the two double-precision floating-point numbers in DRn and DRm, and stores 1 in the T bit if DRn > DRm, or 0 otherwise.

Notes: None

Operation:

```
void FCMP_EQ(int m,n) /* FCMP/EQ  FRm,FRn */
{
     pc += 2;
     clear_cause();
     if(fcmp_chk(m,n) == INVALID) fcmp_invalid();
     else if(fcmp_chk(m,n) == EQ)  T = 1;
     else                           T = 0;
}
void FCMP_GT(int m,n) /* FCMP/GT  FRm,FRn */
{
     pc += 2;
     clear_cause();
     if ((fcmp_chk(m,n) == INVALID) ||
          (fcmp_chk(m,n) == UO)) fcmp_invalid();
     else if(fcmp_chk(m,n) == GT)  T = 1;
     else                           T = 0;
}
int fcmp_chk (int m,n)
{
     if((data_type_of(m) == sNaN) ||
        (data_type_of(n) == sNaN))        return(INVALID);
     else if((data_type_of(m) == qNaN) ||
               (data_type_of(n) == qNaN))      return(UO);
     else switch(data_type_of(m)){
           case NORM:     switch(data_type_of(n)){
                 case PINF   :return(GT);  break;
                 case NINF   :return(LT);  break;
                 default:                  break;
                 }      break;
           case PZERO:
           case NZERO:    switch(data_type_of(n)){
                 case PZERO  :
                 case NZERO  :return(EQ);  break;
                 default:                  break;
                 }      break;
          case PINF :    switch(data_type_of(n)){
                case PINF   :return(EQ);  break;
                default:return(LT);       break;
                }      break;
          case NINF :    switch(data_type_of(n)){
                case NINF   :return(EQ);  break;
                default:return(GT);       break;
                }      break;
     }
     if(FPSCR_PR == 0) {
         if(FR[n] == FR[m])           return(EQ);
         else if(FR[n] > FR[m])       return(GT);
         else                         return(LT);
     }else {
         if(DR[n>>1] == DR[m>>1])     return(EQ);
         else if(DR[n>>1] > DR[m>>1]) return(GT);
         else                         return(LT);
     }
}
void fcmp_invalid()
{
     set_V();     if((FPSCR & ENABLE_V) == 0)  T = 0;
                  else fpu_exception_trap();
}
```

FCMP Special Cases

FCMP/EQ	FRn,DRn							
FRm,DRm	NORM	DNORM	+0	–0	+INF	–INF	qNaN	sNaN
NORM	CMP							
DNORM								
+0			EQ					
–0								
+INF					EQ			
–INF						EQ		
qNaN							!EQ	
sNaN								Invalid

Note: When DN = 1, the value of a denormalized number is treated as 0.

FCMP/GT	FRn,DRn							
FRm,DRm	NORM	DENORM	+0	–0	+INF	–INF	qNaN	sNaN
NORM	CMP				GT	!GT		
DENORM								
+0			!GT					
–0								
+INF	!GT				!GT			
–INF	GT					!GT		
qNaN							UO	
sNaN								Invalid

Note: When DN = 1, the value of a denormalized number is treated as 0.
UO means unordered. Unordered is treated as false (!GT).

Possible Exceptions:

- Invalid operation

10.3.4 FCNVDS (Floating-point Convert Double to Single Precision): Floating-Point Instruction

PR	Format	Operation	Instruction Code	Cycle	T Bit
0	—	—	—	—	—
1	`FCNVDS DRm,FPUL`	(float)DRm → FPUL	`1111mmm010111101`	1	—

Description: When FPSCR.PR = 1: This instruction converts the double-precision floating-point number in DRm to a single-precision floating-point number, and stores the result in FPUL.

When FPSCR.enable. I is set, an FPU exception trap is generated regardless of whether or not an exception has occurred. When FPSCR.enable.O/U is set, FPU exception traps are generated on actual generation by the FPU exception source and on the satisfaction of certain special conditions that apply to this the instruction. These special conditions are described in the remaining parts of this section. When an exception occurs, correct exception information is reflected in FPSCR.cause and FPSCR.flag, and FPUL is not updated. Appropriate processing should therefore be performed by software.

Notes: None

Operation:

```
void FCNVDS(int m, float *FPUL){
     case((FPSCR.PR){
          0:  undefined_operation();    /* reserved */
          1:  fcnvds(m, *FPUL);  break;  /* FCNVDS */
     }
}
void fcnvds(int m, float *FPUL)
{
     pc += 2;
     clear_cause();
     case(data_type_of(m)){
          NORM   :
          PZERO  :
          NZERO  :    normal_ fcnvds(m, *FPUL);  break;
          DENORM : set_E();
          PINF   :    *FPUL = 0x7f800000; break;
          NINF   :    *FPUL = 0xff800000; break;
          qNaN  :    *FPUL = 0x7fbfffff; break;
          sNaN  :     set_V();
                        if((FPSCR & ENABLE_V) == 0) *FPUL = 0x7fbfffff;
                        else fpu_exception_trap();   break;
     }
}
void normal_fcnvds(int m, float *FPUL)
{
int sign;
float abs;
union {
      float f;
      int l;
}     dstf,tmpf;
union {
      double d;
      int l[2];
}     dstd;
      dstd.d = DR[m>>1];
      if(dstd.l[1] & 0x1fffffff)) set_I();
      if(FPSCR_RM == 1) dstd.l[1] &= 0xe0000000; /* round toward zero*/
      dstf.f = dstd.d;
      check_single_exception(FPUL, dstf.f);
}
```

FCNVDS Special Cases

DRn	+NORM	–NORM	+0	–0	+INF	–INF	qNaN	sNaN
FCNVDS(DRn FPUL)	FCNVDS	FCNVDS	+0	–0	+INF	–INF	qNaN	Invalid

Note: When DN = 1, the value of a denormalized number is treated as 0.

Possible Exceptions and Overflow/Underflow Exception Trap Generating Conditions:

- FPU error
- Invalid operation
- Overflow

  Generation of overflow-exception traps

  The exponent of DRn is not less than H'47E
- Underflow

  Generation of underflow-exception traps

  The exponent of DRn is not more than H'380
- Inexact

10.3.5 FCNVSD (Floating-point Convert Single to Double Precision): Floating-Point Instruction

PR	Format	Operation	Instruction Code	Cycle	T Bit
0	—	—	—	—	—
1	`FCNVSD FPUL,DRn`	(double) FPUL → DRn	1111nnn010101101	1	—

Description: When FPSCR.PR = 1: This instruction converts the single-precision floating-point number in FPUL to a double-precision floating-point number, and stores the result in DRn.

Notes: None

Operation:

```
void FCNVSD(int n, float *FPUL){
     pc += 2;
     clear_cause();
     case((FPSCR_PR){
          0:  undefined_operation();    /* reserved */
          1:  fcnvsd (n, *FPUL);  break;  /* FCNVSD */
     }
}
void fcnvsd(int n, float *FPUL)
{
     case(fpul_type(*FPUL)){
          PZERO :
          NZERO :
          PINF  :
          NINF  :      DR[n>>1] = *FPUL;     break;
          DENORM :  set_E();          break;
          qNaN  :      qnan(n);       break;
          sNaN  :      invalid(n);          break;
     }
}
int fpul_type(int *FPUL)
{
int abs;
```

```
        abs = *FPUL & 0x7fffffff;
        if(abs < 0x00800000){
            if((FPSCR_DN == 1) || (abs == 0x00000000)){
                if(sign_of(src) == 0) return(PZERO);
                else                  return(NZERO);
            }
            else                      return(DENORM);
        }
        else if(abs < 0x7f800000)  return(NORM);
        else if(abs == 0x7f800000) {
                if(sign_of(src) == 0) return(PINF);
                else               return(NINF);
        }
        else if(abs < 0x7fc00000)  return(qNaN);
        else                       return(sNaN);
   }
```

FCNVSD Special Cases

FRn	+NORM	–NORM	+0	–0	+INF	–INF	qNaN	sNaN
FCNVSD(FPUL FRn)	+NORM	–NORM	+0	–0	+INF	–INF	qNaN	Invalid

Note: When DN = 1, the value of a denormalized number is treated as 0.

Possible Exceptions:

- FPU error
- Invalid operation

10.3.6 FDIV (Floating-point Divide): Floating-Point Instruction

PR	Format		Operation	Instruction Code	Cycle	T Bit
0	`FDIV`	`FRm,FRn`	FRn/FRm → FRn	`1111nnnnmmmm0011`	14	—
1	`FDIV`	`DRm,DRn`	DRn/DRm → DRn	`1111nnn0mmm00011`	30	—

Description: When FPSCR.PR = 0: Arithmetically divides the single-precision floating-point number in FRn by the single-precision floating-point number in FRm, and stores the result in FRn.

When FPSCR.PR = 1: Arithmetically divides the double-precision floating-point number in DRn by the double-precision floating-point number in DRm, and stores the result in DRn.

When FPSCR.enable.I is set, an FPU exception trap is generated regardless of whether or not an exception has occurred. When FPSCR.enable.O/U is set, FPU exception traps are generated on actual generation by the FPU exception source and on the satisfaction of certain special conditions that apply to this the instruction. These special conditions are described in the remaining parts of this section. When an exception occurs, correct exception information is reflected in FPSCR.cause and FPSCR.flag, and FRn or DRn is not updated. Appropriate processing should therefore be performed by software.

Notes: None

Operation:

```
void FDIV(int m,n)      /* FDIV FRm,FRn */
{
     pc += 2;
     clear_cause();
     if((data_type_of(m) == sNaN) ||
        (data_type_of(n) == sNaN)) invalid(n);
     else if((data_type_of(m) == qNaN) ||
             (data_type_of(n) == qNaN)) qnan(n);
     else switch (data_type_of(m)){
         case NORM: switch (data_type_of(n)){
             case PINF:
             case NINF:    inf(n,sign_of(m)^sign_of(n));break;
             case PZERO:
             case NZERO:   zero(n,sign_of(m)^sign_of(n));break;
             case DENORM:set_E();            break;
             default:    normal_fdiv(m,n);  break;
```

```
            }    break;
        case PZERO: switch (data_type_of(n)){
            case PZERO:
            case NZERO: invalid(n);break;
            case PINF:
            case NINF:  break;
            default:      dz(n,sign_of(m)^sign_of(n));break;
               }     break;
        case NZERO: switch (data_type_of(n)){
            case PZERO:
            case NZERO:  invalid(n);  break;
            case PINF:   inf(n,1);    break;
            case NINF:   inf(n,0);    break;
            default:     dz(FR[n],sign_of(m)^sign_of(n)); break;
            }     break;
        case DENORM:     set_E();     break;
        case PINF :
        case NINF : switch (data_type_of(n)){
            case DENORM: set_E(); break;
            case PINF:
            case NINF: invalid(n);     break;
            default:   zero(n,sign_of(m)^sign_of(n));break
            }    break;
        }
}
void normal_fdiv(int m,n)
{
union {
      float f;
      int l;
}     dstf,tmpf;
union {
      double d;
      int l[2];
}     dstd,tmpd;
union {
      int double x;
```

```
        int l[4];
}       tmpx;
      if(FPSCR_PR == 0) {
          tmpf.f = FR[n]; /* save destination value */
          dstf.f /= FR[m]; /* round toward nearest or even */
          tmpd.d = dstf.f; /* convert single to double */
          tmpd.d *= FR[m];
          if(tmpf.f != tmpd.d) set_I();
          if((tmpf.f < tmpd.d) && (FPSCR_RM == 1))
              dstf.l -= 1; /* round toward zero */
           check_single_exception(&FR[n], dstf.f);
      } else {
          tmpd.d = DR[n>>1]; /* save destination value */
          dstd.d /= DR[m>>1]; /* round toward nearest or even */
          tmpx.x = dstd.d; /* convert double to int double */
          tmpx.x *= DR[m>>1];
          if(tmpd.d != tmpx.x) set_I();
          if((tmpd.d < tmpx.x) && (FPSCR_RM == 1)) {
              dstd.l[1] -= 1; /* round toward zero */
              if(dstd.l[1] == 0xffffffff) dstd.l[0] -= 1;
          }
          check_double_exception(&DR[n>>1], dstd.d);
      }
}
```

FDIV Special Cases

FDIV	FRn,DRn									
FRm,DRm	+NORM	-NORM	+DENORM	–DENORM	+0	-0	+inf	–inf	qNaN	sNaN
+NORM	FDIV				+0	-0	+inf	-inf		
-NORM					-0	+0	-inf	+inf		
+DENORM					+0	-0	+inf	-inf		
–DENORM				Error	-0	+0	-inf	+inf		
+0							+inf	-inf		
-0				DZ		invalid	-inf	DZ+inf		
+inf	+0	-0	+0	-0	+0	-0				
–inf	-0	+0	-0	+0	-0	+0		invalid		
qNaN									qNaN	
sNaN										invalid

Note: When DN = 1, the value of a denormalized number is treated as 0.

Possible Exceptions and Overflow/Underflow Exception Trap Generating Conditions:

- FPU error
- Invalid operation
- Divide by zero
- Overflow

  Generation of overflow-exception traps

  FPSCR.PR = 0: (exponent of FRn) – (exponent of FRm) + H'7F is not less than H'FF

  FPSCR.PR = 1: (exponent of DRn) – (exponent of DRm) + H'3FF is not less than H'7FF
- Underflow

  Generation of underflow-exception traps

  FPSCR.PR = 0: (exponent of FRn) – (exponent of FRm) + H'7F is not more than H'01

  FPSCR.PR = 1: (exponent of DRn) – (exponent of DRm) + H'3FF is not more than H'001
- Inexact

10.3.7 FIPR (Floating-point Inner Product): Floating-Point Instruction

PR	Format	Operation	Instruction Code	Cycle	T Bit
0	`FIPR FVm,FVn`	Inner_product(FVm, FVn) $\rightarrow$ FR[n+3]	`1111nnmm11101101`	1	—
—	—	—	—	—	—

Notes: FV0 = {FR0, FR1, FR2, FR3}
FV4 = {FR4, FR5, FR6, FR7}
FV8 = {FR8, FR9, FR10, FR11}
FV12 = {FR12, FR13, FR14, FR15}

Description: When FPSCR.PR = 0: This instruction calculates the inner products of the 4-dimensional single-precision floating-point vector indicated by FVn and FVm, and stores the results in FR[n + 3].

The FIPR instruction is intended for speed rather than accuracy, and therefore the results will differ from those obtained by using a combination of FADD and FMUL instructions. The FIPR execution sequence is as follows:

1. Multiplies all terms. The results are 28 bits long.
2. Aligns these results, rounding them to fit within 30 bits.
3. Adds the aligned values.
4. Performs normalization and rounding.

Special processing is performed in the following cases:

1. If an input value is an sNaN, an invalid exception is generated.
2. If the input values to be multiplied include a combination of 0 and infinity, an invalid exception is generated.
3. In cases other than the above, if the input values include a qNaN, the result will be a qNaN.
4. In cases other than the above, if the input values include infinity:
   a. If multiplication results in two or more infinities and the signs are different, an invalid exception will be generated.
   b. Otherwise, correct infinities will be stored.
5. If the input values do not include an sNaN, qNaN, or infinity, processing is performed in the normal way.

When FPSCR.enable.U/I is set, an FPU exception trap is generated regardless of whether or not an exception has occurred. When FPSCR.enable.O is set, FPU exception traps are generated on actual generation by the FPU exception source and on the satisfaction of certain special conditions that apply to this the instruction. These special conditions are described in the remaining parts of this section. When an exception occurs, correct exception information is reflected in FPSCR.cause and FPSCR.flag, and FR[n+3] is not updated. Appropriate processing should therefore be performed by software.

Notes: None

Operation:

```
void FIPR(int m,n)    /* FIPR FVm,FVn */
{
     if(FPSCR_PR == 0)  {
         pc += 2;
         clear_cause();
         fipr(m,n);
     }
     else    undefined_operation();
}
```

Possible Exceptions and Overflow Exception Trap Generating Conditions:

- Invalid operation
- Overflow
  Generation of overflow-exception traps
  At least one of following results is not less than H'FC
  (exponent of FRn) + (exponent of FRm)
  (exponent of FR(n + 1)) + (exponent of FR(m + 1))
  (exponent of FR(n + 2)) + (exponent of FR(m + 2))
  (exponent of FR(n + 3)) + (exponent of FR(m + 3))
- Underflow
- Inexact

10.3.8 FLDI0 (Floating-point Load Immediate 0.0): Floating-Point Instruction

PR	Format	Operation	Instruction Code	Cycle	T Bit
0	`FLDI0 FRn`	0x00000000 → FRn	`1111nnnn10001101`	1	—
1	—	—	—	—	—

Description: When FPSCR.PR = 0, this instruction loads floating-point 0.0 (0x00000000) into FRn.

Notes: None

Operation:

```
void FLDI0(int n)
{
     FR[n] = 0x00000000;
     pc += 2;
}
```

Possible Exceptions: None

10.3.9 FLDI1 (Floating-point Load Immediate 1.0): Floating-Point Instruction

Format	Operation	Instruction Code	Cycle	T Bit
`FLDI1 FRn`	0x3F800000 → FRn	`1111nnnn10011101`	1	—
—	—	—	—	—

Description: When FPSCR.PR = 0, this instruction loads floating-point 1.0 (0x3F800000) into FRn.

Notes: None

Operation:

```
void FLDI1(int n)
{
     FR[n] = 0x3F800000;
     pc += 2;
}
```

Possible Exceptions: None

10.3.10 FLDS (Floating-point Load to System register): Floating-Point Instruction

Format	Operation	Instruction Code	Cycle	T Bit
`FLDS FRm,FPUL`	FRm → FPUL	`1111mmmm00011101`	1	—

Description: This instruction loads the contents of floating-point register FRm into system register FPUL.

Notes: None

Operation:

```
void FLDS(int m, float *FPUL)
{
     *FPUL = FR[m];
     pc += 2;
}
```

Possible Exceptions: None

10.3.11 FLOAT (Floating-point Convert from Integer): Floating-Point Instruction

PR	Format	Operation	Instruction Code	Cycle	T Bit
0	`FLOAT  FPUL,FRn`	(float)FPUL → FRn	`1111nnnn00101101`	1	—
1	`FLOAT  FPUL,DRn`	(double)FPUL → DRn	`1111nnn000101101`	1	—

Description:

When FPSCR.PR = 0: Taking the contents of FPUL as a 32-bit integer, converts this integer to a single-precision floating-point number and stores the result in FRn.

When FPSCR.PR = 1: Taking the contents of FPUL as a 32-bit integer, converts this integer to a double-precision floating-point number and stores the result in DRn.

When FPSCR.enable.I = 1 and FPSCR.PR = 0, an FPU exception trap is generated regardless of whether or not an exception has occurred. When an exception occurs, correct exception information is reflected in FPSCR.cause and FPSCR.flag, and FRn is not updated. Appropriate processing should therefore be performed by software.

Notes: None

Operation:

```
void FLOAT(int n, float *FPUL)
{
union {
     double d;
     int l[2];
}    tmp;
     pc += 2;
     clear_cause();
     if(FPSCR.PR==0){
         FR[n] = *FPUL; /* convert from integer to float */
         tmp.d = *FPUL;
         if(tmp.l[1] & 0x1fffffff) inexact();
     } else {
         DR[n>>1] = *FPUL; /* convert from integer to double */
     }
}
```

Possible Exceptions:

- Inexact: Not generated when FPSCR.PR = 1.

10.3.12 FMAC (Floating-point Multiply and Accumulate): Floating-Point Instruction

PR	Format	Operation	Instruction Code	Cycle	T Bit
0	`FMAC FR0,FRm,FRn`	FR0 × FRm+FRn → FRn	`1111nnnnmmmm1110`	1	—
1	—	—	—	—	—

Description: When FPSCR.PR = 0: This instruction arithmetically multiplies the two single-precision floating-point numbers in FR0 and FRm, arithmetically adds the contents of FRn, and stores the result in FRn.

When FPSCR.enable.I is set, an FPU exception trap is generated regardless of whether or not an exception has occurred. When FPSCR.enable.O/U is set, FPU exception traps are generated on actual generation by the FPU exception source and on the satisfaction of certain special conditions that apply to this the instruction. These special conditions are described in the remaining parts of this section. When an exception occurs, correct exception information is reflected in FPSCR.cause and FPSCR.flag, and FRn is not updated. Appropriate processing should therefore be performed by software.

Notes: None

Operation:

```
void FMAC(int m,n)
{
     pc += 2;
     clear_cause();
     if(FPSCR_PR == 1) undefined_operation();
     else if((data_type_of(0) == sNaN) ||
             (data_type_of(m) == sNaN) ||
             (data_type_of(n) == sNaN)) invalid(n);
     else if((data_type_of(0) == qNaN) ||
             (data_type_of(m) == qNaN)) qnan(n);
     else if((data_type_of(0) == DENORM) ||
             (data_type_of(m) == DENORM)) set_E();
     else switch (data_type_of(0){
         case NORM: switch (data_type_of(m)){
         case PZERO:
         case NZERO: switch (data_type_of(n)){
             case DENORM: set_E();  break;
             case qNaN:   qnan(n);  break;
```

```
            case PZERO:
            case NZERO: zero(n,sign_of(0)^ sign_of(m)^sign_of(n));
break;
            default:     break;
            }
    case PINF:
    case NINF: switch (data_type_of(n)){
        case DENORM:  set_E(); break;
        case qNaN:    qnan(n); break;
        case PINF:
        case NINF: if(sign_of(0)^ sign_of(m)^sign_of(n))  invalid(n);
                   else   inf(n,sign_of(0)^ sign_of(m)); break;
        default:          inf(n,sign_of(0)^ sign_of(m)); break;
        }
    case NORM: switch (data_type_of(n)){
        case DENORM: set_E();  break;
        case qNaN:   qnan(n);  break;
        case PINF:
        case NINF:   inf(n,sign_of(n)); break;
        case PZERO:
        case NZERO:
        case NORM:   normal_fmac(m,n);  break;
    }      break;
    case PZERO:
    case NZERO: switch (data_type_of(m)){
        case PINF:
        case NINF:  invalid(n); break;
        case PZERO:
        case NZERO:
        case NORM: switch (data_type_of(n)){
        case DENORM: set_E();    break;
        case qNaN:   qnan(n);    break;
        case PZERO:
        case NZERO: zero(n,sign_of(0)^ sign_of(m)^sign_of(n));  break;
        default:    break;
        }        break;
    }      break;
    case PINF :
    case NINF : switch (data_type_of(m)){
        case PZERO:
        case NZERO:invalid(n);  break;
        default: switch (data_type_of(n)){
        case DENORM: set_E();  break;
        case qNaN:    qnan(n);  break;
            default:   inf(n,sign_of(0)^sign_of(m)^sign_of(n));break
            }      break;
        }      break;
    }
}
void normal_fmac(int m,n)
{
union {
     int double x;
     int l[4];
}     dstx,tmpx;
float dstf,srcf;
     if((data_type_of(n) == PZERO)|| (data_type_of(n) == NZERO))
            srcf = 0.0; /* flush denormalized value */
     else    srcf = FR[n];
     tmpx.x = FR[0]; /* convert single to int double */
     tmpx.x *= FR[m]; /* exact product */
     dstx.x = tmpx.x + srcf;
     if(((dstx.x == srcf) && (tmpx.x != 0.0)) ||
        ((dstx.x == tmpx.x) && (srcf != 0.0))) {
        set_I();
        if(sign_of(0)^ sign_of(m)^ sign_of(n))  {
            dstx.l[3] -= 1; /* correct result */
            if(dstx.l[3] == 0xffffffff) dstx.l[2] -= 1;
            if(dstx.l[2] == 0xffffffff) dstx.l[1] -= 1;
            if(dstx.l[1] == 0xffffffff) dstx.l[0] -= 1;
        }
        else   dstx.l[3] |= 1;
     }
     if((dstx.l[1] & 0x01ffffff) || dstx.l[2] || dstx.l[3]) set_I();
     if(FPSCR_RM == 1) {
            dstx.l[1] &= 0xfe000000; /* round toward zero */
            dstx.l[2]  = 0x00000000;
            dstx.l[3]  = 0x00000000;
       }
       dstf = dstx.x;
       check_single_exception(&FR[n],dstf);
}
```

FMAC Special Cases

FMAC		FRm							
FRn	FR0	+NORM	-NORM	+0	–0	+inf	–inf	qNaN	sNaN
NORM	+NORM	FMAC				+inf	-inf		
	-NORM					-inf	+inf		
	+0								
	-0						invalid		
	+inf	+inf	-inf			+inf	-inf		
	-inf	-inf	+inf		invalid	-inf	+inf		
+0	+NORM	FMAC				+inf	-inf		
	-NORM					-inf	+inf		
	+0								
	-0				+0		invalid		
	+inf	+inf	-inf			+inf	-inf		
	-inf	-inf	+inf		invalid	-inf	+inf		
-0	+NORM	FMAC		+0	-0	+inf	-inf		
	-NORM			-0	+0	-inf	+inf		
	+0	+0	-0	+0	-0				
	-0	-0	+0	-0	+0		invalid		
	+inf	+inf	-inf			+inf	-inf		
	-inf	-inf	+inf		invalid	-inf	+inf		
+inf	+NORM					+inf	invalid		
	-NORM					invalid	+inf		
	+0								
	-0				+inf		invalid		
	+inf	+inf	invalid			+inf	invalid		
	-inf	invalid	+inf		invalid	invalid	+inf		
-inf	+NORM					invalid	-inf		
	-NORM					-inf	invalid		
	+0								
	-0				-inf		invalid		
	+inf	invalid	-inf			invalid	-inf		
	-inf	-inf	invalid		invalid	-inf	invalid	qNaN	invalid

FMAC		FRm							
FRn	FR0	+NORM	-NORM	+0	–0	+inf	–inf	qNaN	sNaN
qNaN	+NORM	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	invalid
	-NORM	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	invalid
	+0	qNaN	qNaN	qNaN	qNaN	invalid	invalid	qNaN	invalid
	-0	qNaN	qNaN	qNaN	qNaN	invalid	invalid	qNaN	invalid
	+inf	qNaN	qNaN	invalid	invalid	qNaN	qNaN	qNaN	invalid
	-inf	qNaN	qNaN	invalid	invalid	qNaN	qNaN	qNaN	invalid
!sNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	invalid
all types	sNaN	invalid	invalid	invalid	invalid	invalid	invalid	invalid	invalid
sNaN	all types	invalid	invalid	invalid	invalid	invalid	invalid	invalid	invalid

Notes: When DN = 1, the value of a denormalized numbers is treated as 0.
When DN = 0, calculation for denormalized numbers is the same as for normalized numbers.

Possible Exceptions and Overflow/Underflow Exception Trap Generating Conditions:

- FPU error
- Invalid operation
- Overflow

  Generation of overflow-exception traps

  At least one of following results is not less than H′FD

  (exponent of FR0) + (exponent of FRm)

  exponent of FRn
- Underflow

  Generation of underflow-exception traps

  At least one of following results is not more than H′2E

  (exponent of FR0) + (exponent of FRm)

  exponent of FRn
- Inexact

10.3.13 FMOV (Floating-point Move): Floating-Point Instruction

No.	SZ	Format		Operation	Instruction Code	Cycle	T Bit
1.	0	`FMOV`	`FRm,FRn`	FRm → FRn	1111nnnnmmmm1100	1	—
2.	1	`FMOV`	`DRm,DRn`	DRm → DRn	1111nnn0mmm01100	1	—
3.	0	`FMOV.S`	`FRm,@Rn`	FRm → (Rn)	1111nnnnmmmm1010	1	—
4.	1	`FMOV`	`DRm,@Rn`	DRm → (Rn)	1111nnnnmmm01010	1	—
5.	0	`FMOV.S`	`@Rm,FRn`	(Rm) → FRn	1111nnnnmmmm1000	1	—
6.	1	`FMOV`	`@Rm,DRn`	(Rm) → DRn	1111nnn0mmmm1000	1	—
7.	0	`FMOV.S`	`@Rm+,FRn`	(Rm) → FRn,Rm+4 → Rm	1111nnnnmmmm1001	1	—
8.	1	`FMOV`	`@Rm+,DRn`	(Rm) → DRn, Rm+8 → Rm	1111nnn0mmmm1001	1	—
9.	0	`FMOV.S`	`FRm,@-Rn`	Rn-4 → Rn,FRm → (Rn)	1111nnnnmmmm1011	1	—
10.	1	`FMOV`	`DRm,@-Rn`	Rn-8 → Rn,DRm → (Rn)	1111nnnnmmm01011	1	—
11.	0	`FMOV.S`	`@(R0,Rm),FRn`	(R0+Rm) → FRn	1111nnnnmmmm0110	1	—
12.	1	`FMOV`	`@(R0,Rm),DRn`	(R0+Rm) → DRn	1111nnn0mmmm0110	1	—
13.	0	`FMOV.S`	`FRm,@(R0,Rn)`	FRm → (R0+Rn)	1111nnnnmmmm0111	1	—
14.	1	`FMOV`	`DRm,@(R0,Rn)`	DRm → (R0+Rn)	1111nnnnmmm00111	1	—

Description:

1. This instruction transfers FRm contents to FRn.
2. This instruction transfers DRm contents to DRn.
3. This instruction transfers FRm contents to memory at address indicated by Rn.
4. This instruction transfers DRm contents to memory at address indicated by Rn.
5. This instruction transfers contents of memory at address indicated by Rm to FRn.
6. This instruction transfers contents of memory at address indicated by Rm to DRn.
7. This instruction transfers contents of memory at address indicated by Rm to FRn, and adds 4 to Rm.
8. This instruction transfers contents of memory at address indicated by Rm to DRn, and adds 8 to Rm.
9. This instruction subtracts 4 from Rn, and transfers FRm contents to memory at address indicated by resulting Rn value.
10. This instruction subtracts 8 from Rn, and transfers DRm contents to memory at address indicated by resulting Rn value.
11. This instruction transfers contents of memory at address indicated by (R0 + Rm) to FRn.

12. This instruction transfers contents of memory at address indicated by (R0 + Rm) to DRn.
13. This instruction transfers FRm contents to memory at address indicated by (R0 + Rn).
14. This instruction transfers DRm contents to memory at address indicated by (R0 + Rn).

Notes: None

Operation:

```
void FMOV(int m,n)                  /* FMOV FRm,FRn */
{
     FR[n] = FR[m];
     pc += 2;
}
void FMOV_DR(int m,n)            /* FMOV DRm,DRn */
{
     DR[n>>1] = DR[m>>1];
     pc += 2;
}
void FMOV_STORE(int m,n)       /* FMOV.S FRm,@Rn */
{
     store_int(FR[m],R[n]);
     pc += 2;
}
void FMOV_STORE_DR(int m,n)   /* FMOV DRm,@Rn */
{
     store_quad(DR[m>>1],R[n]);
     pc += 2;
}
 void FMOV_LOAD(int m,n)       /* FMOV.S @Rm,FRn */
{
     load_int(R[m],FR[n]);
     pc += 2;
}
void FMOV_LOAD_DR(int m,n)    /* FMOV @Rm,DRn */
{
     load_quad(R[m],DR[n>>1]);
     pc += 2;
}
void FMOV_RESTORE(int m,n)     /* FMOV.S @Rm+,FRn */
```

```
{
     load_int(R[m],FR[n]);
     R[m] += 4;
     pc += 2;
}
void FMOV_RESTORE_DR(int m,n) /* FMOV @Rm+,DRn */
{
     load_quad(R[m],DR[n>>1]) ;
     R[m] += 8;
     pc += 2;
}
void FMOV_SAVE(int m,n)        /* FMOV.S FRm,@-Rn */
{
     store_int(FR[m],R[n]-4);
     R[n] -= 4;
     pc += 2;
}
void FMOV_SAVE_DR(int m,n)    /* FMOV DRm,@-Rn */
{
     store_quad(DR[m>>1],R[n]-8);
     R[n] -= 8;
     pc += 2;
}
void FMOV_INDEX_LOAD(int m,n)  /* FMOV.S @(R0,Rm),FRn */
{
     load_int(R[0] + R[m],FR[n]);
     pc += 2;
}
void FMOV_INDEX_LOAD_DR(int m,n) /*FMOV @(R0,Rm),DRn */
{
     load_quad(R[0] + R[m],DR[n>>1]);
     pc += 2;
}
void FMOV_INDEX_STORE(int m,n)  /*FMOV.S FRm,@(R0,Rn)*/
{
     store_int(FR[m], R[0] + R[n]);
     pc += 2;
}
void FMOV_INDEX_STORE_DR(int m,n)/*FMOV DRm,@(R0,Rn)*/
{
     store_quad(DR[m>>1], R[0] + R[n]);
     pc += 2;
}
```

Possible Exceptions:

- Data TLB miss exception
- Data protection violation exception
- Initial page write exception
- Data address error

10.3.14 FMOV (Floating-point Move Extension): Floating-Point Instruction

No.	SZ	Format	Operation	Instruction Code	Cycle	T Bit
1.	1	`FMOV XDm,@Rn`	XRm → (Rn)	1111nnnnmmm11010	1	—
2.	1	`FMOV @Rm,XDn`	(Rm) → XDn	1111nnn1mmmm1000	1	—
3.	1	`FMOV @Rm+,XDn`	(Rm) → XDn, Rm+8 → Rm	1111nnn1mmmm1001	1	—
4.	1	`FMOV XDm,@-Rn`	Rn-8 → Rn,XDm → (Rn)	1111nnnnmmm11011	1	—
5.	1	`FMOV @(R0,Rm),XDn`	(R0+Rm) → XDn	1111nnn1mmmm0110	1	—
6.	1	`FMOV XDm,@(R0,Rn)`	XDm → (R0+Rn)	1111nnnnmmm10111	1	—
7.	1	`FMOV XDm,XDn`	XDm → XDn	1111nnn1mmm11100	1	—
8.	1	`FMOV XDm,DRn`	XDm → DRn	1111nnn0mmm11100	1	—
9.	1	`FMOV DRm,XDn`	DRm → XDn	1111nnn1mmm01100	1	—

Description:

1. This instruction transfers XDm contents to memory at address indicated by Rn.
2. This instruction transfers contents of memory at address indicated by Rm to XDn.
3. This instruction transfers contents of memory at address indicated by Rm to XDn, and adds 8 to Rm.
4. This instruction subtracts 8 from Rn, and transfers XDm contents to memory at address indicated by resulting Rn value.
5. This instruction transfers contents of memory at address indicated by (R0 + Rm) to XDn.
6. This instruction transfers XDm contents to memory at address indicated by (R0 + Rn).
7. This instruction transfers XDm contents to XDn.
8. This instruction transfers XDm contents to DRn.
9. This instruction transfers DRm contents to XDn.

Operation:

```
void FMOV_STORE_XD(int m,n)       /* FMOV XDm,@Rn */
{
     store_quad(XD[m>>1],R[n]);
     pc += 2;
}
void FMOV_LOAD_XD(int m,n)       /* FMOV @Rm,XDn */
{
   load_quad(R[m],XD[n>>1]);
   pc += 2;
}
void FMOV_RESTORE_XD(int m,n)    /* FMOV @Rm+,XDn */
{
   load_quad(R[m],XD[n>>1]);
   R[m] += 8;
   pc += 2;
}
void FMOV_SAVE_XD(int m,n)       /* FMOV XDm,@-Rn */
{
   store_quad(XD[m>>1],R[n]-8);
   R[n] -= 8;
   pc += 2;
}
void FMOV_INDEX_LOAD_XD(int m,n)       /* FMOV @(R0,Rm),XDn */
{
   load_quad(R[0] + R[m],XD[n>>1]);
   pc += 2;
}
void FMOV_INDEX_STORE_XD(int m,n)      /* FMOV XDm,@(R0,Rn) */
{
     store_quad(XD[m>>1], R[0] + R[n]);
     pc += 2;
}
 void FMOV_XDXD(int m,n)        /* FMOV XDm,XDn */
{
     XD[n>>1] = XD[m>>1];
     pc += 2;
```

```
  }
  void FMOV_XDDR(int m,n)  /* FMOV XDm,DRn */
  {
       DR[n>>1] = XD[m>>1];
       pc += 2;
  }
  void FMOV_DRXD(int m,n)  /* FMOV DRm,XDn */
  {
       XD[n>>1] = DR[m>>1];
       pc += 2;
  }
```

Possible Exceptions:

- Data TLB miss exception
- Data protection violation exception
- Initial page write exception
- Data address error

10.3.15 FMUL (Floating-point Multiply): Floating-Point Instruction

PR	Format	Operation	Instruction Code	Cycle	T Bit
0	`FMUL FRm,FRn`	FRn × FRm → FRn	`1111nnnnmmmm0010`	1	—
1	`FMUL DRm,DRn`	DRn × DRm → DRn	`1111nnn0mmm00010`	3	—

Description: When FPSCR.PR = 0: Arithmetically multiplies the two single-precision floating-point numbers in FRn and FRm, and stores the result in FRn.

When FPSCR.PR = 1: Arithmetically multiplies the two double-precision floating-point numbers in DRn and DRm, and stores the result in DRn.

When FPSCR.enable.I is set, an FPU exception trap is generated regardless of whether or not an exception has occurred. When FPSCR.enable.O/U is set, FPU exception traps are generated on actual generation by the FPU exception source and on the satisfaction of certain special conditions that apply to this the instruction. These special conditions are described in the remaining parts of this section. When an exception occurs, correct exception information is reflected in FPSCR.cause and FPSCR.flag, and FRn or DRn is not updated. Appropriate processing should therefore be performed by software.

Notes: None

Operation:

```
void FMUL(int m,n)
{
     pc += 2;
         clear_cause();
         if((data_type_of(m) == sNaN) ||
             (data_type_of(n) == sNaN)) invalid(n);
         else if((data_type_of(m) == qNaN) ||
                 (data_type_of(n) == qNaN)) qnan(n);
         else if((data_type_of(m) == DENORM) ||
                 (data_type_of(n) == DENORM)) set_E();
         else switch (data_type_of(m){
             case NORM: switch (data_type_of(n)){
                 case PZERO:
                 case NZERO: zero(n,sign_of(m)^sign_of(n));  break;
                 case PINF:
                 case NINF:      inf(n,sign_of(m)^sign_of(n));  break;
                 default:       normal_fmul(m,n);  break;
      }      break;
         case PZERO:
         case NZERO: switch (data_type_of(n)){
             case PINF:
             case NINF:  invalid(n); break;
             default:    zero(n,sign_of(m)^sign_of(n));break;
      }      break;
          case PINF :
          case NINF : switch (data_type_of(n)){
              case PZERO:
              case NZERO: invalid(n);   break;
              default:       inf(n,sign_of(m)^sign_of(n));break
      }       break;
      }
}
```

FMUL Special Cases (FPSCR.PR = 0)

FMUL	FRn							
FRm	+NORM	-NORM	+0	–0	+inf	–inf	qNaN	sNaN
+NORM	FMUL		+0	–0	+inf	-inf		
-NORM			–0	+0	-inf	+inf		
+0	+0	–0	+0	–0				
–0	–0	+0	–0	+0		invalid		
+inf	+inf	-inf			+inf	-inf		
–inf	-inf	+inf		invalid	-inf	+inf		
qNaN							qNaN	
sNaN								invalid

Note: When DN = 0, calculation for denormalized numbers is the same as for normalized numbers.

FMUL Special Cases (FPSCR.PR = 1)

FMUL	DRn									
DRm	+NORM	-NORM	+DENORM	–DENORM	+0	-0	+inf	–inf	qNaN	sNaN
+NORM	FMUL				+0	-0	+inf	-inf		
-NORM					-0	+0	-inf	+inf		
+DENORM					+0	-0	+inf	-inf		
–DENORM				Error	-0	+0	-inf	+inf		
+0	+0	-0	+0	-0	+0	-0				
-0	-0	+0	-0	+0	-0	+0		invalid		
+inf	+inf	-inf	+inf	-inf			+inf	-inf		
–inf	-inf	+inf	-inf	+inf		invalid	-inf	+inf		
qNaN									qNaN	
sNaN										invalid

Note: When DN = 1, the value of a denormalized number is treated as 0.

Possible Exceptions and Overflow/Underflow Exception Trap Generating Conditions:

- FPU error
- Invalid operation
- Overflow

  Generation of overflow-exception traps

  FPSCR.PR = 0: (exponent of FRn) + (exponent of FRm) – H'7F is not less than H'FE

  FPSCR.PR = 1: (exponent of DRn) + (exponent of DRm) – H'3FF is not less than H'7FE
- Underflow

  Generation of underflow-exception traps

  FPSCR.PR = 0:

  When both FRn and FRm are normalized numbers: (exponent of FRn) + (exponent of FRm) – H'7F is not more than H'00

  When at least FRn or FRm is not a normalized number: (exponent of FRn) + (exponent of FRm) – H'7F is not more than H'18

  FPSCR.PR = 1: (exponent of DRn) + (exponent of DRm) – H'3FF is not more than H'000
- Inexact

10.3.16 FNEG (Floating-point Negate Value): Floating-Point Instruction

PR	Format	Operation	Instruction Code	Cycle	T Bit
0	`FNEG FRn`	FRn ^ H'80000000 → FRn	`1111nnnn01001101`	1	—
1	`FNEG DRn`	DRn ^ H'8000000000000000 → DRn	`1111nnn001001101`	1	—

Description: This instruction inverts the most significant bit (sign bit) of the contents of floating-point register FRn/DRn, and stores the result in FRn/DRn.

The cause and flag fields in FPSCR are not updated.

Notes: None

Operation:

```
void FNEG (int n){
     FR[n] = -FR[n];
     pc += 2;
}

/* Same operation is performed regardless of precision. */
```

Possible Exceptions: None

10.3.17 FPCHG (Pr-bit Change): Floating-Point Instruction

Format	Operation	Instruction Code	Cycle	T Bit
FPCHG	~FPSCR.PR → FPSCR.PR	1111011111111101	1	—

Description: This instruction inverts the PR bit of the floating-point status register FPSCR. The value of this bit selects single-precision or double-precision operation.

Notes: None

Operation:

```
void FPCHG(){/* FPCHG */}
{
FPSCR ^= 0x00080000; /* bit 19 */
PC += 2;
}
```

Possible Exceptions: None

10.3.18 FRCHG (FR-bit Change): Floating-Point Instruction

PR	Format	Operation	Instruction Code	Cycle	T Bit
0	`FRCHG`	~FPSCR.FR → FPSCR.FR	`1111101111111101`	1	—
1	—	—	—	—	—

Description: This instruction inverts the FR bit in floating-point register FPSCR. When the FR bit in FPSCR is changed, FR0 to FR15 in FPR0_BANK0 to FPR15_BANK0 and FPR0_BANK1 to FPR15_BANK1 become XR0 to XR15, and XR0 to XR15 become FR0 to FR15. When FPSCR.FR = 0, FPR0_BANK0 to FPR15_BANK0 correspond to FR0 to FR15, and FPR0_BANK1 to FPR15_BANK1 correspond to XR0 to XR15. When FPSCR.FR = 1, FPR0_BANK1 to FPR15_BANK1 correspond to FR0 to FR15, and FPR0_BANK0 to FPR15_BANK0 correspond to XR0 to XR15.

Notes: None

Operation:

```
void FRCHG()    /* FRCHG */
{
     if(FPSCR_PR == 0){
         FPSCR ^= 0x00200000; /* bit 21 */
         PC += 2;
     }
     else undefined_operation();
}
```

Possible Exceptions: None

10.3.19 FSCA (Floating Point Sine And Cosine Approximate): Floating-Point Instruction

PR	Format	Operation	Instruction Code	Cycle	T Bit
0	`FSCA FPUL,DRn`	sin(FPUL) → FRn cos(FPUL) → FR[n+1]	`1111nnn011111101`	3	—
1	—	reserved	`1111nnnn11111101`	—	—

Description: This instruction calculates the sine and cosine approximations of FPUL (absolute error is within ±2^–21) as single-precision floating point values, and places the values of the sine and cosine in FRn and FR[n + 1], respectively. Since this instruction is an approximate operation instruction, an imprecision exception is always required (even if the input is a 0, the result is imprecise).

When FPSCR.enable.I is set, an FPU exception trap is generated. When an exception occurs, correct exception information is reflected in FPSCR.cause and FPSCR.flag, and FRn and FR[n + 1] is not updated. Appropriate processing should therefore be performed by software.

Notes: None

Operation:

```
void FSCA(int n){
    float angle;
    long offset = 0x00010000;
    long fraction = 0x0000ffff;

    case((FPSCR.PR){

      0: clear_cause();
              set_I();
      /* extract sub-rotation (fraction) part */
              fraction &= FPUL;
      /* convert to float */
              angle = fraction;
      /* convert to radian */
              angle = 2*M_PI*angle / offset;
              FR[n] ~= sin(angle)
              FR[n+1] ~= cos(angle)
              pc += 2; break;
```

```
        1: undefined_operation(); /* reserved */

      }
  }
```

Data Format of Source Operand:

Angle is specified as shown below, i.e., as a signed fraction in twos complement. The result of sin/cos is a single-precision floating-point number.

0x7FFFFFFF to 0x00000001	: $360 \times 2^{15} - 360/2^{16}$ to $360/2^{16}$ degrees
0x00000000	: 0 degree
0xffffffff to 0x80000000	: $-360/2^{16}$ to -360×2^{15} degrees

Possible Exceptions:

- Inexact

10.3.20 FSCHG (Sz-bit Change): Floating-Point Instruction

PR	Format	Operation	Instruction Code	Cycle	T Bit
0	FSCHG	~FPSCR.SZ → FPSCR.SZ	1111001111111101	1	—

Description: This instruction inverts the SZ bit of the floating-point status register FPSCR. Changing the value of the SZ bit in FPSCR switches the amount of data for transfer by the FMOV instruction between one single-precision data and a pair of single-precision data. When FPSCR.SZ = 0, an FMOV instruction transfers a single-precision number. When FPSCR.SZ = 1, the FMOV instruction transfers a pair of single-precision numbers.

Notes: None

Operation:

```
void FSCHG()    /* FSCHG */
{
    if(FPSCR_PR == 0){
        FPSCR ^= 0x00100000; /* bit 20 */
        PC += 2;
    }
    else undefined_operation();
}
```

Possible Exceptions: None

10.3.21 FSQRT (Floating-point Square Root): Floating-Point Instruction

PR	Format	Operation	Instruction Code	Cycle	T Bit
0	`FSQRT FRn`	sqrt (FRn)* → FRn	`1111nnnn01101101`	14	—
1	`FSQRT DRn`	sqrt (DRn)* → DRn	`1111nnn001101101`	30	—

Note: * sqrt(FRn) and sqrt(DRn) are the square roots of FRn and DRn, respectively.

Description: When FPSCR.PR = 0: Finds the arithmetical square root of the single-precision floating-point number in FRn, and stores the result in FRn.

When FPSCR.PR = 1: Finds the arithmetical square root of the double-precision floating-point number in DRn, and stores the result in DRn.

When FPSCR.enable.I is set, an FPU exception trap is generated regardless of whether or not an exception has occurred. When an exception occurs, correct exception information is reflected in FPSCR.cause and FPSCR.flag, and FRn or DRn is not updated. Appropriate processing should therefore be performed by software.

Notes: None

Operation:

```
void FSQRT(int n){
     pc += 2;
     clear_cause();
     switch(data_type_of(n)){
         case NORM   :   if(sign_of(n) == 0) normal_ fsqrt(n);
                   else      invalid(n); break;
         case DENORM:   if(sign_of(n) == 0) set_E();
                   else      invalid(n); break;
         case PZERO :
         case NZERO :
         case PINF  :     break;
         case NINF  :     invalid(n); break;
         case qNaN  :     qnan(n);    break;
         case sNaN  :     invalid(n); break;
     }
}
```

```
void normal_fsqrt(int n)
{
union {
      float f;
      int l;
}     dstf,tmpf;
union {
      double d;
      int l[2];
}     dstd,tmpd;
union {
      int double x;
      int l[4];
}     tmpx;
      if(FPSCR_PR == 0) {
         tmpf.f = FR[n]; /* save destination value */
         dstf.f = sqrt(FR[n]); /* round toward nearest or even */
         tmpd.d = dstf.f; /* convert single to double */
         tmpd.d *= dstf.f;
         if(tmpf.f != tmpd.d) set_I();
         if((tmpf.f < tmpd.d) && (FPSCR_RM == 1))
             dstf.l -= 1; /* round toward zero */
         if(FPSCR & ENABLE_I) fpu_exception_trap();
         else                    FR[n] = dstf.f;
      } else {
         tmpd.d = DR[n>>1]; /* save destination value */
         dstd.d = sqrt(DR[n>>1]); /* round toward nearest or even */
         tmpx.x = dstd.d; /* convert double to int double */
         tmpx.x *= dstd.d;
         if(tmpd.d != tmpx.x) set_I();
         if((tmpd.d < tmpx.x) && (FPSCR_RM == 1)) {
             dstd.l[1] -= 1; /* round toward zero */
             if(dstd.l[1] == 0xffffffff) dstd.l[0] -= 1;
         }
         if(FPSCR & ENABLE_I) fpu_exception_trap();
         else                    DR[n>>1] = dstd.d;
      }
}
```

FSQRT Special Cases:

FRn	+NORM	–NORM	+DENORM	–DENORM	+0	–0	+INF	–INF	qNaN	sNaN
FSQRT (FRn)	SQRT	Invalid	Error	Error	+0	–0	+INF	Invalid	qNaN	Invalid

Note: When DN = 1, the value of a denormalized number is treated as 0.

Possible Exceptions:

- FPU error
- Invalid operation
- Inexact

10.3.22 FSRRA (Floating Point Square Reciprocal Approximate): Floating-Point Instruction

PR	Format	Operation	Instruction Code	Cycle	T Bit
0	`FSRRA FRn`	1/ sqrt(FRn)* $\rightarrow$ FRn	`1111nnnn01111101`	1	—
1	—	reserved	`1111nnnn01111101`		

Note: * sqrt(FRn) is the square root of FRn.

Description: This instruction takes the approximate inverse of the arithmetic square root (absolute error is within ±2^–21) of the single-precision floating-point in FRn and writes the result to FRn. Since the this instruction operates by approximation, an imprecision exception is required when the input is a normalized value. In other cases, the instruction does not require an imprecision exception.

When FPSCR.enable.I is set, an FPU exception trap is generated. When an exception occurs, correct exception information is reflected in FPSCR.cause and FPSCR.flag, and FRn is not updated. Appropriate processing should therefore be performed by software.

Notes: None

Operation:

```
Void FSRRA(int n){
    case(FPSCR.PR){
        0:  fsrra_single(n);      break;
        1:  undefined_operation();      break;
    }
    PC += 2;
}
    fsrra_single(int n)
{   clear_cause();
    case(data_type_of(n)){
        NORM:  if(sign_of(n)==0)
            set_I();
            FR[n] = 1/sqrt(FR[n]);
            else invalid(n); break;
        DENORM:   if(sign_of(n)==0)
            fpu_error(); break;
            else invalid(n); break
        PZERO:
        NZERO: dz(n,sign_of(n)); break;
        PINF:  FR[n]=0;break;
        NINF:  invalid(n); break;
        qNAN:  qnan(n);  break;
        sNAN   invalid(n); break;
        }
 }
```

FSRRA Special Cases

FRn	+NORM	–NORM	+DENORM	–DENORM	+0	–0	+INF	–INF	qNaN	sNaN
FSRRA(FRn)	1/SQRT	Invalid	Error	Invalid	DZ	DZ	+0	Invalid	qNaN	Invalid

Note: When DN = 1, the value of denormalized number is treated as 0.

Possible Exceptions:

- FPU error
- Invalid operation
- Divided by Zero
- Inexact

10.3.23 FSTS (Floating-point Store System Register): Floating-Point Instruction

Format	Operation	Instruction Code	Cycle	T Bit
`FSTS FPUL,FRn`	FPUL → FRn	`1111nnnn00001101`	1	—

Description: This instruction transfers the contents of system register FPUL to floating-point register FRn.

Notes: None

Operation:

```
void FSTS(int n, float *FPUL)
{
     FR[n] = *FPUL;
     pc += 2;
}
```

Possible Exceptions: None

10.3.24 FSUB (Floating-point Subtract): Floating-Point Instruction

PR	Format		Operation	Instruction Code	Cycle	T Bit
0	`FSUB`	`FRm,FRn`	FRn-FRm → FRn	`1111nnnnmmmm0001`	1	—
1	`FSUB`	`DRm,DRn`	DRn-DRm → DRn	`1111nnn0mmm00001`	1	—

Description: When FPSCR.PR = 0: Arithmetically subtracts the single-precision floating-point number in FRm from the single-precision floating-point number in FRn, and stores the result in FRn.

When FPSCR.PR = 1: Arithmetically subtracts the double-precision floating-point number in DRm from the double-precision floating-point number in DRn, and stores the result in DRn.

When FPSCR.enable.I is set, an FPU exception trap is generated regardless of whether or not an exception has occurred. When FPSCR.enable.O/U is set, FPU exception traps are generated on actual generation by the FPU exception source and on the satisfaction of certain special conditions that apply to this the instruction. These special conditions are described in the remaining parts of this section. When an exception occurs, correct exception information is reflected in FPSCR.cause and FPSCR.flag, and FRn or DRn is not updated. Appropriate processing should therefore be performed by software.

Operation

```
void FSUB (int m,n)
{
     pc += 2;
     clear_cause();
     if((data_type_of(m) == sNaN) ||
         (data_type_of(n) == sNaN)) invalid(n);
     else if((data_type_of(m) == qNaN) ||
             (data_type_of(n) == qNaN)) qnan(n);
     else if((data_type_of(m) == DENORM) ||
             (data_type_of(n) == DENORM)) set_E();
     else switch (data_type_of(m)){
         case NORM: switch (Pe-of(n)){
             case NORM:  normal_faddsub(m,n,SUB); break;
             case PZERO:
             case NZERO: register_copy(m,n); FR[n] = -FR[n];break;
             default:       break;
         }           break;
```

```
          case PZERO: break;
          case NZERO: switch (data_type_of(n)){
              case NZERO: zero(n,0); break;
              default:       break;
          }           break;
          case PINF: switch (data_type_of(n)){
              case PINF:    invalid(n);     break;
              default:      inf(n,1);       break;
          }    break;
          case NINF: switch (data_type_of(n)){
              case NINF:    invalid(n);     break;
              default:      inf(n,0);       break;
          }           break;
      }
 }
```

FSUB Special Cases

FSUB	FRn,DRn							
FRm,DRm	+NORM	-NORM	+0	–0	+inf	–inf	qNaN	sNaN
+NORM	FSUB	FSUB	FSUB	FSUB	+inf	–inf	qNaN	invalid
-NORM	FSUB	FSUB	FSUB	FSUB	+inf	–inf	qNaN	invalid
+0	FSUB	FSUB	+0	–0	+inf	–inf	qNaN	invalid
–0	FSUB	FSUB	+0	+0	+inf	–inf	qNaN	invalid
+inf	–inf	–inf	–inf	–inf	invalid	–inf	qNaN	invalid
–inf	+inf	+inf	+inf	+inf	+inf	invalid	qNaN	invalid
qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	invalid
sNaN	invalid	invalid	invalid	invalid	invalid	invalid	invalid	invalid

Notes: When DN = 1, the value of a denormalized number is treated as 0.

When DN = 0, calculation for denormalized numbers is the same as for normalized numbers.

Possible Exceptions and Overflow/Underflow Exception Trap Generating Conditions:

- FPU error
- Invalid operation
- Overflow

  Generation of overflow-exception traps

  FPSCR.PR = 0: FRn and FRm have the different signs and the exponent of at least one value is H'FE

  FPSCR.PR = 1: DRn and DRm have the different signs and the exponent of at least one value is H'7FE
- Underflow

  Generation of underflow-exception traps

  FPSCR.PR = 0: FRn and FRm have same sign and neither has an exponent greater than H'18

  FPSCR.PR = 1: DRn and DRm have same sign and neither has an exponent greater than H'035
- Inexact

10.3.25 FTRC (Floating-point Truncate and Convert to integer): Floating-Point Instruction

PR	Format	Operation	Instruction Code	Cycle	T Bit
0	`FTRC FRm,FPUL`	(long)FRm → FPUL	1111mmmm00111101	1	—
1	`FTRC DRm,FPUL`	(long)DRm → FPUL	1111mmm000111101	1	—

Description: When FPSCR.PR = 0: Converts the single-precision floating-point number in FRm to a 32-bit integer, and stores the result in FPUL.

When FPSCR.PR = 1: Converts the double-precision floating-point number in FRm to a 32-bit integer, and stores the result in FPUL.

The rounding mode is always truncation.

Notes: None

Operation:

```
#define N_INT_SINGLE_RANGE 0xcf000000 & 0x7fffffff /* -1.000000 * 2^31 */
#define P_INT_SINGLE_RANGE 0x4effffff  /* 1.fffffe * 2^30 */
#define N_INT_DOUBLE_RANGE 0xc1e0000000200000 & 0x7fffffffffffffff
#define P_INT_DOUBLE_RANGE 0x41e0000000000000

void FTRC(int m, int *FPUL)
{
     pc += 2;
     clear_cause();
     if(FPSCR.PR==0){
          case(ftrc_single_ type_of(m)){
          NORM:     *FPUL = FR[m];     break;
          PINF:     ftrc_invalid(0,*FPUL);   break;
          NINF:     ftrc_invalid(1,*FPUL);   break;
          }
     }
     else{                  /* case FPSCR.PR=1 */
          case(ftrc_double_type_of(m)){
          NORM:     *FPUL = DR[m>>1]; break;
          PINF:     ftrc_invalid(0,*FPUL);   break;
          NINF:     ftrc_invalid(1, *FPUL);  break;
          }
     }
}
int ftrc_signle_type_of(int m)
{
     if(sign_of(m) == 0){
         if(FR_HEX[m] > 0x7f800000)    return(NINF);    /* NaN */
         else if(FR_HEX[m] > P_INT_SINGLE_RANGE)
                    return(PINF);    /* out of range,+INF */
         else       return(NORM);    /* +0,+NORM          */
     } else {
         if((FR_HEX[m] & 0x7fffffff) > N_INT_SINGLE_RANGE)
                    return(NINF);  /* out of range ,+INF,NaN*/
         else       return(NORM);   /* -0,-NORM              */
     }
}
int ftrc_double_type_of(int m)
{
     if(sign_of(m) == 0){
         if((FR_HEX[m] > 0x7ff00000) ||
           ((FR_HEX[m] == 0x7ff00000) &&
            (FR_HEX[m+1] != 0x00000000)))   return(NINF);    /* NaN */
         else if(DR_HEX[m>>1] >= P_INT_DOUBLE_RANGE)
                    return(PINF);    /* out of range,+INF */
         else       return(NORM);    /* +0,+NORM          */
     } else {
         if((DR_HEX[m>>1] & 0x7fffffffffffffff) >= N_INT_DOUBLE_RANGE)
                    return(NINF);    /* out of range ,+INF,NaN*/
         else       return(NORM);    /* -0,-NORM              */
     }
}
```

```
void ftrc_invalid(int sign, int *FPUL)
{
     set_V();
      if((FPSCR & ENABLE_V) == 0){
            if(sign == 0)      *FPUL = 0x7fffffff;
            else               *FPUL = 0x80000000;
     }
     else fpu_exception_trap();
}
```

FTRC Special Cases

FRn,DRn	NORM	+0	–0	Positive Out of Range	Negative Out of Range	+INF	–INF	qNaN	sNaN
FTRC (FRn,DRn)	TRC	0	0	Invalid +MAX	Invalid –MAX	Invalid +MAX	Invalid –MAX	Invalid –MAX	Invalid –MAX

Note: When DN = 1, the value of a denormalized number is treated as 0.

Possible Exceptions:

- Invalid operation

10.3.26 FTRV (Floating-point Transform Vector): Floating-Point Instruction

PR	Format	Operation	Instruction Code	Cycle	T Bit
0	`FTRV XMTRX,FVn`	transform_vector (XMTRX, FVn) → FVn	`1111nn0111111101`	4	—
1	—	—	—	—	—

Description: When FPSCR.PR = 0: This instruction takes the contents of floating-point registers XF0 to XF15 indicated by XMTRX as a 4-row × 4-column matrix, takes the contents of floating-point registers FR[n] to FR[n + 3] indicated by FVn as a 4-dimensional vector, multiplies the array by the vector, and stores the results in FV[n].

$$\overset{\text{XMTRX}}{\begin{bmatrix} XF[0] & XF[4] & XF[8] & XF[12] \\ XF[1] & XF[5] & XF[9] & XF[13] \\ XF[2] & XF[6] & XF[10] & XF[14] \\ XF[3] & XF[7] & XF[11] & XF[15] \end{bmatrix}} \times \overset{\text{FVn}}{\begin{bmatrix} FR[n] \\ FR[n+1] \\ FR[n+2] \\ FR[n+3] \end{bmatrix}} \rightarrow \overset{\text{FVn}}{\begin{bmatrix} FR[n] \\ FR[n+1] \\ FR[n+2] \\ FR[n+3] \end{bmatrix}}$$

The FTRV instruction is intended for speed rather than accuracy, and therefore the results will differ from those obtained by using a combination of FADD and FMUL instructions. The FTRV execution sequence is as follows:

1. Multiplies all terms. The results are 28 bits long.
2. Aligns these results, rounding them to fit within 30 bits.
3. Adds the aligned values.
4. Performs normalization and rounding.

Special processing is performed in the following cases:

1. If an input value is an sNaN, an invalid exception is generated.
2. If the input values to be multiplied include a combination of 0 and infinity, an invalid operation exception is generated.
3. In cases other than the above, if the input values include a qNaN, the result will be a qNaN.
4. In cases other than the above, if the input values include infinity:
   a. If multiplication results in two or more infinities and the signs are different, an invalid exception will be generated.
   b. Otherwise, correct infinities will be stored.
5. If the input values do not include an sNaN, qNaN, or infinity, processing is performed in the normal way.

When FPSCR.enable.V/O/U/I is set, an FPU exception trap is generated regardless of whether or not an exception has occurred. When an exception occurs, correct exception information is reflected in FPSCR.cause and FPSCR.flag, and FVn is not updated. Appropriate processing should therefore be performed by software.

Notes: None

Operation:

```
void FTRV (int n)     /* FTRV FVn */
{
float saved_vec[4],result_vec[4];
int saved_fpscr;
int dst,i;
     if(FPSCR_PR == 0) {
         PC += 2;
         clear_cause();
         saved_fpscr  = FPSCR;
         FPSCR &= ~ENABLE_VOUI; /* mask VOUI enable */
         dst = 12 - n;          /* select other vector than FVn */
         for(i=0;i<4;i++)saved_vec [i] = FR[dst+i];
         for(i=0;i<4;i++){
             for(j=0;j<4;j++) FR[dst+j] = XF[i+4j];
             fipr(n,dst);
             saved_fpscr |= FPSCR & (CAUSE|FLAG) ;
             result_vec [i] = FR[dst+3];
         }
         for(i=0;i<4;i++)FR[dst+i] = saved_vec [i];
         FPSCR = saved_fpscr;
         if(FPSCR & ENABLE_VOUI) fpu_exception_trap();
         else    for(i=0;i<4;i++)   FR[n+i] = result_vec [i];
     }
     else undefined_operation();
}
```

Possible Exceptions:

- Invalid operation
- Overflow
- Underflow
- Inexact

Section 11 List of Registers

The address map gives information on the on-chip I/O registers and is configured as described below.

Register Addresses (by functional module, in order of the corresponding section numbers):

- Descriptions by functional module, in order of the corresponding section numbers
- Access to reserved addresses which are not described in this list is disabled.

Register States in Each Operating Mode:

- Register states are described in the same order as the Register Addresses (by functional module, in order of the corresponding section numbers).
- For the initial state of each bit, refer to the description of the register in the corresponding section.
- The register states described are for the basic operating modes. If there is a specific reset for an on-chip module, refer to the section on that on-chip module.

11.1 Register Addresses (by functional module, in order of the corresponding section numbers)

Entries under Access size indicates numbers of bits.

Note: Access to undefined or reserved addresses is prohibited. Since operation or continued operation is not guaranteed when these registers are accessed, do not attempt such access.

Module	Name	Abbreviation	R/W	P4 Address*	Area 7 Address*	Access Size
Exception handling	TRAPA exception register	TRA	R/W	H'FF00 0020	H'1F00 0020	32
	Exception event register	EXPEVT	R/W	H'FF00 0024	H'1F00 0024	32
	Interrupt event register	INTEVT	R/W	H'FF00 0028	H'1F00 0028	32
MMU	Page table entry high register	PTEH	R/W	H'FF00 0000	H'1F00 0000	32
	Page table entry low register	PTEL	R/W	H'FF00 0004	H'1F00 0004	32
	Translation table base register	TTB	R/W	H'FF00 0008	H'1F00 0008	32
	TLB exception address register	TEA	R/W	H'FF00 000C	H'1F00 000C	32
	MMU control register	MMUCR	R/W	H'FF00 0010	H'1F00 0010	32
	Physical address space control register	PASCR	R/W	H'FF00 0070	H'1F00 0070	32
	Instruction re-fetch inhibit control register	IRMCR	R/W	H'FF00 0078	H'1F00 0078	32
Cache	Cache control register	CCR	R/W	H'FF00 001C	H'1F00 001C	32
	Queue address control register 0	QACR0	R/W	H'FF00 0038	H'1F00 0038	32
	Queue address control register 1	QACR1	R/W	H'FF00 003C	H'1F00 003C	32
	On-chip memory control register	RAMCR	R/W	H'FF00 0074	H'1F00 0074	32

RENESAS

Module	Name	Abbreviation	R/W	P4 Address*	Area 7 Address*	Access Size
L memory	L memory transfer source address register 0	LSA0	R/W	H'FF00 0050	H'1F00 0050	32
	L memory transfer source address register 1	LSA1	R/W	H'FF00 0054	H'1F00 0054	32
	L memory transfer destination address register 0	LDA0	R/W	H'FF00 0058	H'1F00 0058	32
	L memory transfer destination address register 1	LDA1	R/W	H'FF00 005C	H'FF00 005C	32

Note: * The P4 address is the address used when using P4 area in the virtual address space. The area 7 address is the address used when accessing from area 7 in the physical address space using the TLB.

11.2 Register States in Each Operating Mode

Module	Name	Abbreviation	Power-on Reset	Manual Reset	Sleep	Standby
Exception handling	TRAPA exception register	TRA	Undefined	Undefined	Retained	Retained
	Exception event register	EXPEVT	H'0000 0000	H'0000 0020	Retained	Retained
	Interrupt event register	INTEVT	Undefined	Undefined	Retained	Retained
MMU	Page table entry high register	PTEH	Undefined	Undefined	Retained	Retained
	Page table entry low register	PTEL	Undefined	Undefined	Retained	Retained
	Translation table base register	TTB	Undefined	Undefined	Retained	Retained
	TLB exception address register	TEA	Undefined	Retained	Retained	Retained
	MMU control register	MMUCR	H'0000 0000	H'0000 0000	Retained	Retained
	Physical address space control register	PASCR	H'0000 0000	H'0000 0000	Retained	Retained
	Instruction re-fetch inhibit control register	IRMCR	H'0000 0000	H'0000 0000	Retained	Retained
Cache	Cache control register	CCR	H'0000 0000	H'0000 0000	Retained	Retained
	Queue address control register 0	QACR0	Undefined	Undefined	Retained	Retained
	Queue address control register 1	QACR1	Undefined	Undefined	Retained	Retained
	On-chip memory control register	RAMCR	H'0000 0000	H'0000 0000	Retained	Retained
L memory	L memory transfer source address register 0	LSA0	Undefined	Undefined	Retained	Retained
	L memory transfer source address register 1	LSA1	Undefined	Undefined	Retained	Retained
	L memory transfer destination address register 0	LDA0	Undefined	Undefined	Retained	Retained
	L memory transfer destination address register 1	LDA1	Undefined	Undefined	Retained	Retained

Appendix

A. CPU Operation Mode Register (CPUOPM)

The CPUOPM is used to control the CPU operation mode. This register can be read from or written to the address H'FF2F0000 in P4 area or H'1F2F0000 in area 7 as 32-bit size.

The write value to the reserved bits should be the initial value.

The operation is not guaranteed if the write value is not the initial value.

The CPUOPM register should be updated by the CPU store instruction not the access from SuperHyway bus master except CPU.

After the CPUOPM is updated, read CPUOPM once, and execute one of the following two methods.

1. Execute a branch using the RTE instruction.
2. Execute the ICBI instruction for any address (including non-cacheable area).

After one of these methods are executed, it is guaranteed that the CPU runs under the updated CPUOPM value.

Bit:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
Initial value:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
R/W:	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R

Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	—	—	—	—	—	—	—	—	—	—	RABD	—	INTMU	—	—	—
Initial value:	0	0	0	0	0	0	1	1	1	1	1	0	0	0	0	0
R/W:	R	R	R	R	R	R	R	R	R	R	R/W	R	R/W	R	R	R

Bit	Bit Name	Initial Value	R/W	Description
31 to 6	—	H'000000F	R	Reserved The write value must be the initial value.
5	RABD	1	R/W	Speculative execution bit for subroutine return 0: Instruction fetch for subroutine return is issued speculatively. When this bit is set to 0, refer to Appendix C, Speculative Execution for Subroutine Return. 1: Instruction fetch for subroutine return is not issued speculatively.
4	—	0	R	Reserved The write value must be the initial value.
3	INTMU	0	R/W	Interrupt mode switch bit 0: SR.IMASK is not changed when an interrupt is accepted. 1: SR.IMASK is changed to the accepted interrupt level.
2 to 0	—	All 0	R	Reserved The write value must be the initial value.

B. Instruction Prefetching and Its Side Effects

This LSI is provided with an internal buffer for holding pre-read instructions, and always performs pre-reading. Therefore, program code must not be located in the last 64-byte area of any memory space. If program code is located in these areas, a bus access for instruction prefetch may occur exceeding the memory areas boundary. A case in which this is a problem is shown below.

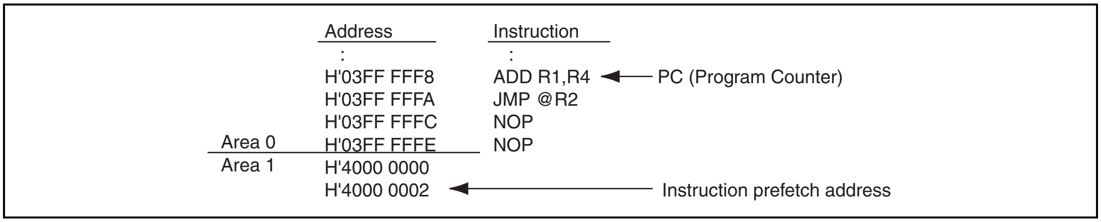


Figure B.1 Instruction Prefetch

Figure B.1 presupposes a case in which the instruction (ADD) indicated by the program counter (PC) and the address H'04000002 instruction prefetch are executed simultaneously. It is also assumed that the program branches to an area other than area 1 after executing the following JMP instruction and delay slot instruction.

In this case, a bus access (instruction prefetch) to area 1 may unintentionally occur from the programming flow.

Instruction Prefetch Side Effects

1. It is possible that an external bus access caused by an instruction prefetch may result in misoperation of an external device, such as a FIFO, connected to the area concerned.
2. If there is no device to reply to an external bus request caused by an instruction prefetch, hang-up will occur.

Remedies

1. These illegal instruction fetches can be avoided by using the MMU.
2. The problem can be avoided by not locating program code in the last 64 bytes of any area.

C. Speculative Execution for Subroutine Return

The SH-4A has the mechanism to issue an instruction fetch speculatively when returning from subroutine. By issuing an instruction fetch speculatively, the execution cycles to return from subroutine may be shortened.

This function is enabled by setting 0 to the bit 5 (RABD) of CPU Operation Mode register (CPUOPM). But this speculative instruction fetch may issue the access to the address that should not be accessed from the program. Therefore, a bus access to an unexpected area or an internal instruction address error may cause a problem. As for the effect of this bus access to unexpected memory area, refer to Appendix B, Instruction Prefetching and Its Side Effects.

Usage Condition:	When the speculative execution for subroutine return is enabled, the RTS instruction should be used to return to the address set in PR by the JSR, BSR, or BSRF instructions. It can prevent the access to unexpected address and avoid the problem.

D. Version Registers (PVR, PRR)

The SH-4A has the read-only registers which show the version of a processor core, and the version of a product. By using the value of these registers, it becomes possible to be able to distinguish the version and product of a processor from software, and to realize the scalability of the high system. Since the values of the version registers differ for every product, please refer to the hardware manual or contact Renesas Technology Corp..

Note: The bit 7 to bit 0 of PVR register and the bit 3 to bit 0 of PRR register should be masked by the software.

Table D.1 Register Configuration

Register Name	Abbr.	R/W	P4 Address	Area 7 Address	Size
Processor version register	PVR	R	H'FF000030	H'1F000030	32
Product register	PRR	R	H'FF000044	H'1F000044	32

Processor Version Register (PVR):

Bit:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	CHIP								VER							
Initial value:	0	0	0	1	0	0	0	0	*	*	*	*	*	*	*	*
R/W:	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R

Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	CUT								—	—	—	—	—	—	—	—
Initial value:	*	*	*	*	*	*	*	*	—	—	—	—	—	—	—	—
R/W:	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R

Bit	Bit Name	Initial Value	R/W	Description
31 to 24	CHIP	H'10	R	Processor Family The read value is always H'10 in the SH-4A.
23 to 16	VER	*	R	Major Version This value is changed when performing major enhancement of the architecture. The version of this manual is H'20.
15 to 8	CUT	*	R	Minor Version This value is changed when performing minor enhancement of the architecture. It differs from one product to another.
7 to 0	—	Undefined	R	This value is undefined. It should be masked by software when using it.

Note: * This value depends on a product.

Product Register (PRR):

Bit:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
Initial value:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
R/W:	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R

Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	Product								CUT				—	—	—	—
Initial value:	*	*	*	*	*	*	*	*	*	*	*	*	—	—	—	—
R/W:	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R

Bit	Bit Name	Initial Value	R/W	Description
31 to 16	—	All 0	R	Reserved For details on reading from or writing to these bits, see description in General Precautions on Handling of Product.
15 to 8	Product	*	R	Major Version This value is changed when performing major enhancement of the product. It differs from one product to another.
7 to 4	CUT	*	R	Minor Version This value is changed when performing minor enhancement of the product. It differs from one product to another.
3 to 0	—	Undefined	R	This value is undefined. It should be masked by software when using it.

Note: * This value depends on a product.

Main Revisions and Additions in this Edition

Item	Page	Revision (See Manual for Details)
Preface	—	Deleted. The SH-4A is a RISC (Reduced Instruction Set Computer) microcomputer which includes a Renesas Technology-original RISC CPU as its core.~~ and the peripheral functions required to configure a system.~~
1.1 Features	1	Amended. The SH-4A is a 32-bit RISC (reduced instruction set computer) microprocessor that is upward compatible with the SH-1, SH-2, SH-3, ~~SH-3E,~~ and SH-4 microcomputers at instruction set code level. Its 16-bit fixed-length instruction set enables program code size to be reduced by almost 50% compared with 32-bit instructions.
Table 1.1 Features CPU	1	Amended. • RISC-type instruction set (upward compatible with the SH-1, SH-2, SH-3, and SH-4 microcomputers)
Table 1.1 Features L memory	3	Amended. • Two independent read/write ports — 8-/16-/32-/64-bit access from the CPU — 8-/16-/32-/64-bit and 16-/32-byte access from the external devices Note: For the size of L memory, see the hardware manual of the target product.
Table 1.2 Changes from SH-4 to SH-4A	4	Amended. (see table below)
	5	Added. (see table below)

Page 4, Amended.

Section No. and Name	Sub-section	Sub-section Name	Changes
3. Instruction Set	3.3	Instruction Set	9 instructions are added as CPU instructions.
			3 instructions are added as FPU instructions.
4. Pipelining	4.2	Parallel-Executability	9 instructions are added as CPU instructions.
			3 instructions are added as FPU instructions.

Page 5, Added.

Section No. and Name	Sub-section	Sub-section Name	Changes
7. Memory Management Unit	7.7	32-Bit Address Extended Mode	Newly added.

RENESAS

Item	Page	Revision (See Manual for Details)
Table 1.2 Changes from SH-4 to SH-4A	6	Added.
2.2.4 Control Registers Status Register (SR)	15	Amended.
2.2.5 System Registers Floating-Point Status/Control Register (FPSCR)	18	Amended.
2.7 Usage Notes	22	Added.

Table 1.2 (page 6), added:

Section No. and Name	Sub-section	Sub-section Name	Changes
8. Caches	8.3.6	OC Two-Way Mode	Newly added.
	8.4	Instruction Cache Operation	IC index mode is deleted.
	8.4.3	IC Two-Way Mode	Newly added.
	8.5.1	Coherency between Cache and External Memory	The ICBI, PREFI, and SYNCO instructions are added.
	8.6	Memory-Mapped Cache Configuration	The entry bits and the way bits are modified according to the size modification and changed into 4-way set associative cache.
	8.8	Notes on Using 32-Bit Address Extended Mode	Newly added.
9. L Memory	—	—	Newly added.
10. Instruction Descriptions	—	—	9 instructions are added as CPU instructions.
			3 instructions are added as FPU instructions.

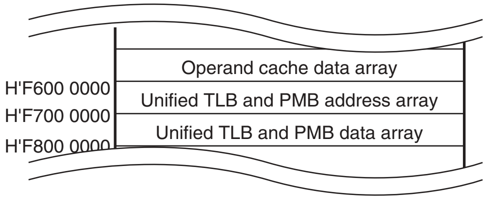
Status Register (SR) (page 15), amended:

Bit	Bit Name	Initial Value	R/W	Description
1	S	0	R/W	S Bit Used by the MAC instruction.
0	T	0	R/W	T Bit Indicates true/false condition, carry/borrow, or overflow/underflow. For details, see section 3, Instruction Set.

Item	Page	Revision (See Manual for Details)
Figure 4.2 Instruction Execution Patterns (7)	51	Amended. (6-3) LDS.L to FPUL: 1 issue cycle (6-5) LDS to FPSCR: 1 issue cycle (6-7) LDS.L to FPSCR: 1 issue cycle
Table 4.2 Instruction Groups	54	Amended.

Instruction Group	Instruction	
LS	MOV.[BWL] @adr,R	STC CR2,Rn
	MOV.[BWL] R,@adr	STC.L CR2,@-Rn
	MOVA	STS SR2,Rn
	MOVCA.L	STS.L SR2,@-Rn
	MOVUA	STS SR1,Rn
	OCBI	STS.L SR1,@-Rn
	OCBP	
	OCBWB	
	PREF	

Item	Page	Revision (See Manual for Details)
6.5.3 FPU Exception Handling	110	Amended. • Division by zero (Z): FPSCR.Enable.Z = 1 and division with a zero divisor or the input of FSRRA is zero
Figure 7.4 P4 Area	118	Amended.


Operand cache data array
H'F600 0000
Unified TLB and PMB address array
H'F700 0000
Unified TLB and PMB data array
H'F800 0000

Item	Page	Revision (See Manual for Details)
7.1.1 Address Spaces • P4 Area	119	Added. The area from H'F610 0000 to H'F61F FFFF is used for direct access to the PMB address array. For details, see section 7.7.5, Memory-Mapped PMB Configuration. The area from H'F700 0000 to H'F70F FFFF is used for direct access to unified TLB data array. For details, see section 7.6.4, UTLB Data Array. The area from H'F710 0000 to H'F71F FFFF is used for direct access to the PMB data array. For details, see section 7.7.5, Memory-Mapped PMB Configuration.

Item	Page	Revision (See Manual for Details)
7.2.2 Page Table Entry Low Register (PTEL)	123	Added. Bit | Bit Name | Initial Value | R/W 0 | WT | — | R/W
7.2.6 Physical Address Space Control Register (PASCR)	128	Amended. Bit | Bit Name | Description 7 to 0 | UB | Buffered Write Control for Each Area (64 Mbytes) When writing is performed without using the cache or in the cache write-through mode, these bits specify whether the next bus access from the CPU waits for the end of writing for each area. 0 : The CPU does not wait for the end of writing bus access and starts the next bus access 1 : The CPU waits for the end of writing bus access and starts the next bus access
7.2.7 Instruction Re-Fetch Inhibit Control Register (IRMCR)	129, 130	Amended. Bit | Bit Name | Initial Value | R/W 4 | R2 | 0 | R/W 3 | R1 | 0 | R/W 2 | LT | 0 | R/W
7.7 32-Bit Address Extended Mode	151 to 158	Added.
Section 8 Caches	159	Note added.
8.5.1 Coherency between Cache and External Memory	175	Deleted. • 1Kbyte page size cannot be used. ~~• In the case of 64KB size operand cache, the bit [13] of virtual address must be same as the bit [13] of the physical address in 4KB page mode.~~ ~~• In the case of 128KB size operand cache, the bits [14:13] of virtual address must be same as the bits [14:13] of the physical address in 4KB page mode.~~
8.6.1 IC Address Array	177	Note added.
8.6.3 OC Address Array	180	Note added.

Item	Page	Revision (See Manual for Details)
8.7.3 Transfer to External Memory • When MMU is enabled (AT = 1 in MMUCR)	183	Deleted. The SQ area (H'E000 0000 to H'E3FF FFFF) is set in VPN of the UTLB, and the transfer destination physical address in PPN. The ASID, V, SZ, SH, PR, and D bits have the same meaning as for normal address translation, but the C and WT bits have no meaning with regard to this page. ~~Transfer to the PCMCIA interface area by means of the SQs is not allowed.~~
8.8 Notes on Using 32-Bit Address Extended Mode	185	Added.
Section 9 L Memory	187	Note added.
Table 9.2 Register Configuration	188	Amended.
9.3.3 Block Transfer	198	Amended. **When MMU is Disabled (MMUCR.AT = 0) or RAMCR.RP = 0:** The transfer source physical address in block transfer to page 0 in the L memory is set in the L0SADR bits of the LSA0 register. And the L0SSZ bits in the LSA0 register choose either the virtual addresses specified through the PRFF instruction or the L0SADR values as bits 15 to 10 of the transfer source physical address. In other words, the transfer source area can be specified in units of 1 Kbyte to 64 Kbytes. The transfer destination physical address in block transfer from page 0 in the L memory is set in the L0DADR bits of the LDA0 register. And the L0DSZ bits in the LDA0 register choose either the virtual addresses specified through the OCBWB instruction or the L0DADR values as bits 15 to 10 of the transfer destination physical address. In other words, the transfer source area can be specified in units of 1 Kbyte to 64 Kbytes. Block transfer to page 1 in the L memory is set to LSA1 and LDA1 as with page 0 in the L memory.
9.6 Note on Using 32-Bit Address Extended Mode	200	Added.

Table 9.2 amendment:

Name	Abbr.	R/W	P4 Address*	Area 7 Address*
L memory transfer destination address register 1	LDA1	R/W	H'FF00005C	H'1F00005C

Item	Page	Revision (See Manual for Details)
10.1.4 AND (AND Logical) • Possible Exceptions	209	Added. Exceptions are checked taking a data access by this instruction as a byte load and a byte store.
10.1.50 OR (OR Logical) • Possible Exceptions	295	Added. Exceptions are checked taking a data access by this instruction as a byte load and a byte store.
10.1.51 PREF (Prefetch Data to Cache) • Description • Possible Exceptions:	296	Amended. This instruction does not generate data address error and MMU exceptions except data TLB multiple-hit exception. In the event of an error, the PREF instruction is treated as an NOP (no operation) instruction. Added. • Data TLB multiple-hit exception
10.1.52 PREFI (Prefetch Instruction Cache Block)	297	Amended. This instruction does not generate data address error and MMU exceptions. In the event of an error, the PREFI instruction is treated as an NOP (no operation) instruction.
10.1.76 SYNCO (Synchronize Data Operation)	333	Amended. **Format** / **Operation** SYNCO: Data accesses invoked by the following instruction are not executed until execution of data accesses which precede this instruction has been completed.
10.1.76 SYNCO (Synchronize Data Operation) • Description	333	Amended. This instruction is used to synchronize data operations. When this instruction is executed, the subsequent bus accesses are not executed until the execution of all preceding bus accesses has been completed.
10.1.76 SYNCO (Synchronize Data Operation) • Notes	333	Changed. The SYNCO instruction can not guarantee the ordering of receipt timing which is notified by the memory-mapped peripheral resources through the method except bus when the register is changed by bus accesses. Refer to the description of each registers to guarantee this ordering.

Item	Page	Revision (See Manual for Details)
10.1.76 SYNCO (Synchronize Data Operation) • Example	333	Deleted. 1. Ordering access to memory areas which are shared with other memory users ~~2. Ordering access to memory-mapped hardware registers~~ 2. Flushing all write buffers 3. Stopping memory-access operations from merging and becoming ineffective 4. Waiting for the completion of cache-control instructions
10.1.77 TAS (Test And Set): Logical Instruction • Possible Exceptions	335	Amended. Exceptions are checked taking a data access by this instruction as a byte load and a byte store.
10.1.79 TST (Test Logical) • Possible Exceptions	338	Added. Exceptions are checked taking a data access by this instruction as a byte load and a byte store.
10.1.80 XOR (Exclusive OR Logical) • Possible Exceptions	340	Added. Exceptions are checked taking a data access by this instruction as a byte load and a byte store.
10.3.19 FSCA (Floating Point Sine And Cosine Approximate) • Description	408	Amended. (absolute error is within ±2^–21)
10.3.22 FSRRA (Floating Point Square Reciprocal Approximate) • Description	414	Added. This instruction takes the approximate inverse of the arithmetic square root (absolute error is within ±2^–21) of the single-precision floating-point in FRn and writes the result to FRn.
Section 11 List of Registers • Register Addresses (by functional module, in order of the corresponding section numbers)	427	Deleted. • Descriptions by functional module, in order of the corresponding section numbers • Access to reserved addresses which are not described in this list is disabled. ~~• When registers consist of 16 or 32 bits, the addresses of the MSBs are given, on the presumption of a big-endian system.~~

Item	Page	Revision (See Manual for Details)
Appendix A	431	Added. The write value to the reserved bits should be the initial value. The operation is not guaranteed if the write value is not the initial value. The CPUOPM register should be updated by the CPU store instruction not the access from SuperHyway bus master except CPU. After the CPUOPM is updated, read CPUOPM once, and execute one of the following two methods. 1. Execute a branch using the RTE instruction. 2. Execute the ICBI instruction for any address (including non-cacheable area). After one of these methods are executed, it is guaranteed that the CPU runs under the updated CPUOPM value.
	432	Amended.

Bit	Bit Name	Initial Value	R/W	Description
31 to 6	—	H'000000F	R	Reserved The write value must be the initial value.
5	RABD	1	R/W	Speculative execution bit for subroutine return 0: Instruction fetch for subroutine return is issued speculatively. When this bit is set to 0, refer to Appendix C, Speculative Execution for Subroutine Return. 1: Instruction fetch for subroutine return is not issued speculatively.
4	—	0	R	Reserved The write value must be the initial value.
3	INTMU	0	R/W	Interrupt mode switch bit 0: SR.IMASK is not changed when an interrupt is accepted. 1: SR.IMASK is changed to the accepted interrupt level.
2 to 0	—	All 0	R	Reserved The write value must be the initial value.

Item	Page	Revision (See Manual for Details)
Appendix C. Speculative Execution for Subroutine Return	434	Added.
Appendix D Version Registers (PVR, PRR)	435, 436	Added.

Index

RENESAS

Renesas 32-Bit RISC Microcomputer
Software Manual
SH-4A

Publication Date:	Rev.1.00, Nov 27, 2003
	Rev.1.50, Oct 29, 2004
Published by:	Sales Strategic Planning Div.
	Renesas Technology Corp.
Edited by:	Technical Documentation & Information Department
	Renesas Kodaira Semiconductor Co., Ltd.

Renesas Technology Corp. Sales Strategic Planning Div. Nippon Bldg., 2-6-2, Ohte-machi, Chiyoda-ku, Tokyo 100-0004, Japan

RENESAS SALES OFFICES

http://www.renesas.com

Refer to "**http://www.renesas.com/en/network**" for the latest and detailed information.

Renesas Technology America, Inc.
450 Holger Way, San Jose, CA 95134-1368, U.S.A
Tel: <1> (408) 382-7500, Fax: <1> (408) 382-7501

Renesas Technology Europe Limited
Dukes Meadow, Millboard Road, Bourne End, Buckinghamshire, SL8 5FH, U.K.
Tel: <44> (1628) 585-100, Fax: <44> (1628) 585-900

Renesas Technology Hong Kong Ltd.
7th Floor, North Tower, World Finance Centre, Harbour City, 1 Canton Road, Tsimshatsui, Kowloon, Hong Kong
Tel: <852> 2265-6688, Fax: <852> 2730-6071

Renesas Technology Taiwan Co., Ltd.
10th Floor, No.99, Fushing North Road, Taipei, Taiwan
Tel: <886> (2) 2715-2888, Fax: <886> (2) 2713-2999

Renesas Technology (Shanghai) Co., Ltd.
Unit2607 Ruijing Building, No.205 Maoming Road (S), Shanghai 200020, China
Tel: <86> (21) 6472-1001, Fax: <86> (21) 6415-2952

Renesas Technology Singapore Pte. Ltd.
1 Harbour Front Avenue, #06-10, Keppel Bay Tower, Singapore 098632
Tel: <65> 6213-0200, Fax: <65> 6278-8001

SH-4A
Software Manual

RENESAS

Renesas Technology Corp.
2-6-2, Ote-machi, Chiyoda-ku, Tokyo, 100-0004, Japan