To all our customers

Regarding the change of names mentioned in the document, such as Hitachi Electric and Hitachi XX, to Renesas Technology Corp.

The semiconductor operations of Mitsubishi Electric and Hitachi were transferred to Renesas Technology Corporation on April 1st 2003. These operations include microcomputer, logic, analog and discrete devices, and memory chips other than DRAMs (flash memory, SRAMs etc.) Accordingly, although Hitachi, Hitachi, Ltd., Hitachi Semiconductors, and other Hitachi brand names are mentioned in the document, these names have in fact all been changed to Renesas Technology Corp. Thank you for your understanding. Except for our corporate trademark, logo and corporate statement, no changes whatsoever have been made to the contents of the document, and these changes do not constitute any alteration to the contents of the document itself.

Renesas Technology Home Page: http://www.renesas.com

Renesas Technology Corp.
Customer Support Dept.
April 1, 2003

Cautions

SuperH™ RISC engine

SH-4

Programming Manual

ADE-602-156D

Rev. 5.0
4/19/2001
Hitachi, Ltd.

Cautions

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

Preface

The SH-4 has been developed as the top-end model in the SuperH™* RISC engine family, featuring a 128-bit graphic engine for multimedia applications and 360 MIPS performance.

The SH-4 CPU has a RISC type instruction set, and features upward-compatibility at the object code level with SH-1, SH-2, SH-3 microcomputers.

In addition to single- and double-precision floating-point operation capability, the on-chip FPU has a 128-bit graphic engine that enables 32-bit floating-point data to be processed 128 bits at a time. It also supports 4×4 array operations and inner product operations.

A superscalar architecture is employed that enables simultaneous execution of two instructions (including FPU instructions), providing performance of up to twice that of conventional architectures at the same frequency.

This programming manual gives details of the SH-4 instructions. For hardware details, refer to the relevant hardware manual.

Related Manual:
SH7750 Series Hardware Manual
SH7751 Hardware Manual

Please consult your Hitachi sales representative for information on development environment systems.

Note: * SuperH™ is a trademark of Hitachi, Ltd.

Main Revisions and Additions in this Version

Page	Item		Revision (See Manual for Details)
13	2.2.4	Control Registers • IMASK: Interrupt mask level	External interrupts of a same level or a lower level than IMASK are masked.
16	2.3	Memory-Mapped Registers	Description amended
26	3.2	Register Descriptions 3. Page table entry assistance register (PTEA)	In the SH7750 series and SH7751, access to a PCMCIA interface area by the DMAC is always performed using the DMAC's CHCRn.SSAn, CHCRn.DSAn, CHCRn.STC, and CHCRn.DTC values.
112	5.6.4	Priority Order with Multiple Exceptions 2. Indivisible delayed branch instruction and delay slot instruction	Descriptions (a) to (f) totally revised
280	9.47	FTRV	1. Multiplies all terms. The results are 28 bits long. 2. Aligns these results, rounding them to fit within 30 bits.

Contents

Section 1 Overview

1.1 SH-4 Features

The SH-4 is a 32-bit RISC (reduced instruction set computer) microprocessor, featuring object code upward-compatibility with SH-1, SH-2, SH-3, and SH-3E microcomputers. 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-4 are summarized in table 1.1.

Table 1.1 SH-4 Features

Item	Features
Architecture	• Original Hitachi SH architecture • 32-bit internal data bus • General register file: — Sixteen 32-bit general registers (and eight 32-bit shadow registers) — Seven 32-bit control registers — Four 32-bit system registers • RISC-type instruction set (upward-compatible with SH Series) — Fixed 16-bit instruction length for improved code efficiency — Load-store architecture — Delayed branch instructions — Conditional execution — C-based instruction set • Superscalar architecture (providing simultaneous execution of two instructions) including FPU • Instruction execution time: Maximum 2 instructions/cycle • Virtual address space: 4 Gbytes (448-Mbyte external memory space) • Space identifier ASIDs: 8 bits, 256 virtual address spaces • On-chip multiplier • Five-stage pipeline

Table 1.1 SH-4 Features (cont)

Item	Features
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 compliance with IEEE754 • 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 (FMAC/FADD/FSUB/FMUL): 3 cycles (single-precision), 8 cycles (double-precision) — Pitch (FMAC/FADD/FSUB/FMUL): 1 cycle (single-precision), 6 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), 7 cycles (latency) — 4-dimensional vector (FIPR) inner product: 1 cycle (pitch), 4 cycles (latency) • Five-stage pipeline
Memory management unit (MMU)	• 4-Gbyte address space, 256 address space identifiers (8-bit ASIDs) • Single virtual mode and multiple virtual memory mode • Supports multiple page sizes: 1 kbyte, 4 kbytes, 64 kbytes, 1 Mbyte • 4-entry fully-associative TLB for instructions • 64-entry fully-associative TLB for instructions and operands • Supports software-controlled replacement and random-counter replacement algorithm • TLB contents can be accessed directly by address mapping

Table 1.1 SH-4 Features (cont)

Item	Features
Cache memory	• Instruction cache (IC) — 8 kbytes, direct mapping — 256 entries, 32-byte block length — Normal mode (8-kbyte cache) — Index mode • Operand cache (OC) — 16 kbytes, direct mapping — 512 entries, 32-byte block length — Normal mode (16-kbyte cache) — Index mode — RAM mode (8-kbyte cache + 8-kbyte RAM) — Choice of write method (copy-back or write-through) • Single-stage copy-back buffer, single-stage write-through buffer • Cache memory contents can be accessed directly by address mapping (usable as on-chip memory) • Store queue (32 bytes × 2 entries)

2.1 Data Formats

The data formats handled by the SH-4 are shown in figure 2.1.

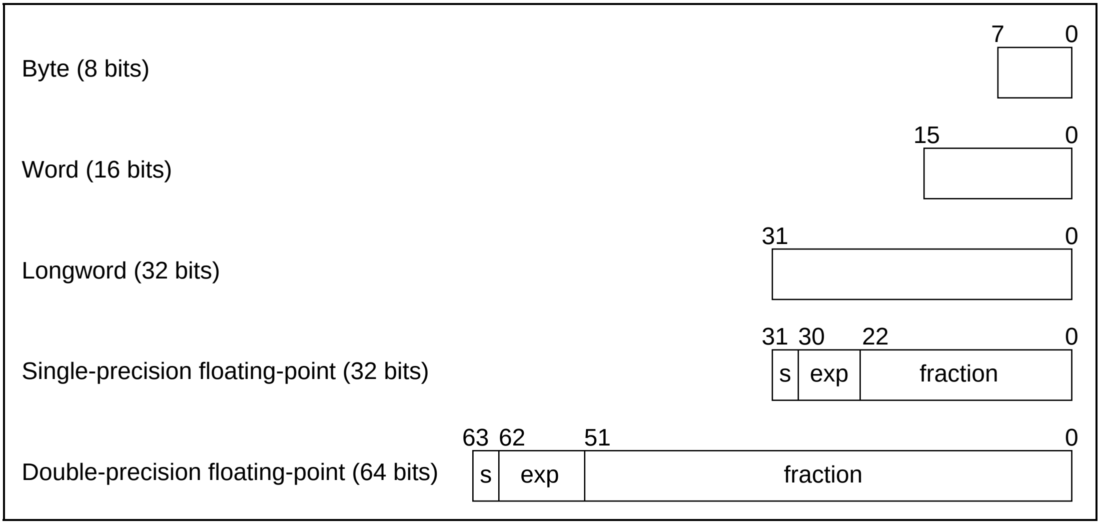


Figure 2.1 Data Formats

2.2 Register Configuration

2.2.1 Privileged Mode and Banks

Processor Modes: The SH-4 has two processor modes, user mode and privileged mode. The SH-4 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 processor 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 processor mode change.

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.

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 processor 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), the program counter (PC), the floating-point status/control register (FPSCR), and the floating-point communication register (FPUL). Access to these registers does not depend on the processor mode.

Floating-Point Registers: 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.

Register values after a reset are shown in table 2.1.

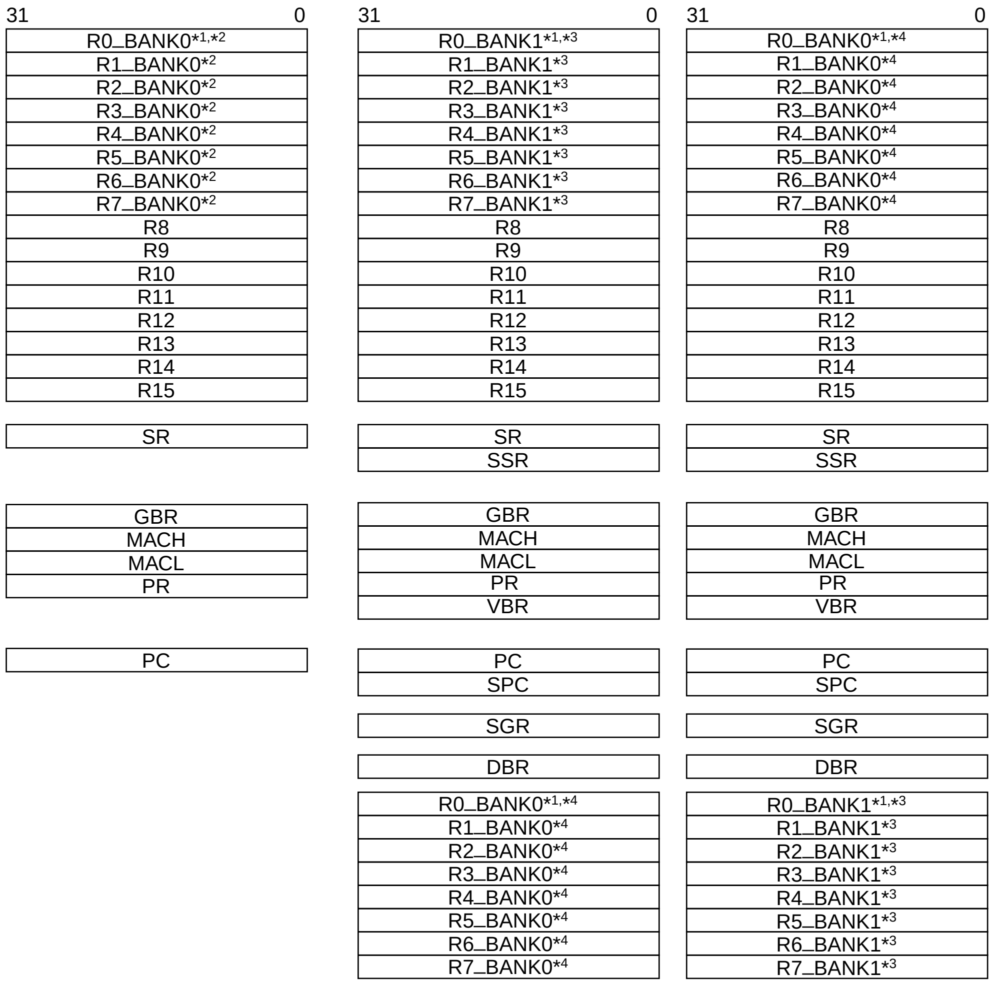
Table 2.1 Initial Register Values

Type	Registers	Initial Value*
General registers	R0_BANK0–R7_BANK0, R0_BANK1–R7_BANK1, R8–R15	Undefined
Control registers	SR	MD bit = 1, RB bit = 1, BL bit = 1, FD bit = 0, I3–I0 = 1111 (H'F), reserved bits = 0, others undefined
	GBR, SSR, SPC, SGR, DBR	Undefined
	VBR	H'00000000
System registers	MACH, MACL, PR, FPUL	Undefined
	PC	H'A0000000
	FPSCR	H'00040001
Floating-point registers	FR0–FR15, XF0–XF15	Undefined

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

The register configuration in each processor is shown in figure 2.2.

Switching between user mode and privileged mode is controlled by the processor mode bit (MD) in the status register.



(a) Register configuration in user mode

(b) Register configuration in privileged mode (RB = 1)

(c) Register configuration in privileged mode (RB = 0)

Notes:
1. The R0 register 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 the SR register. 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 the SR register. Accessed only by LDC/STC instructions when the RB bit is set to 1.

Figure 2.2 CPU Register Configuration in Each Processor Mode

2.2.2 General Registers

Figure 2.3 shows the relationship between the processor modes and general registers. The SH-4 has twenty-four 32-bit general registers (R0_BANK0–R7_BANK0, R0_BANK1–R7_BANK1, and R8–R15). However, only 16 of these can be accessed as general registers R0–R15 in one processor mode. The SH-4 has two processor modes, user mode and privileged mode, in which R0–R7 are assigned as shown below.

- R0_BANK0–R7_BANK0

  In user mode (SR.MD = 0), R0–R7 are always assigned to R0_BANK0–R7_BANK0.

  In privileged mode (SR.MD = 1), R0–R7 are assigned to R0_BANK0–R7_BANK0 only when SR.RB = 0.

- R0_BANK1–R7_BANK1

  In user mode, R0_BANK1–R7_BANK1 cannot be accessed.

  In privileged mode, R0–R7 are assigned to R0_BANK1–R7_BANK1 only when SR.RB = 1.

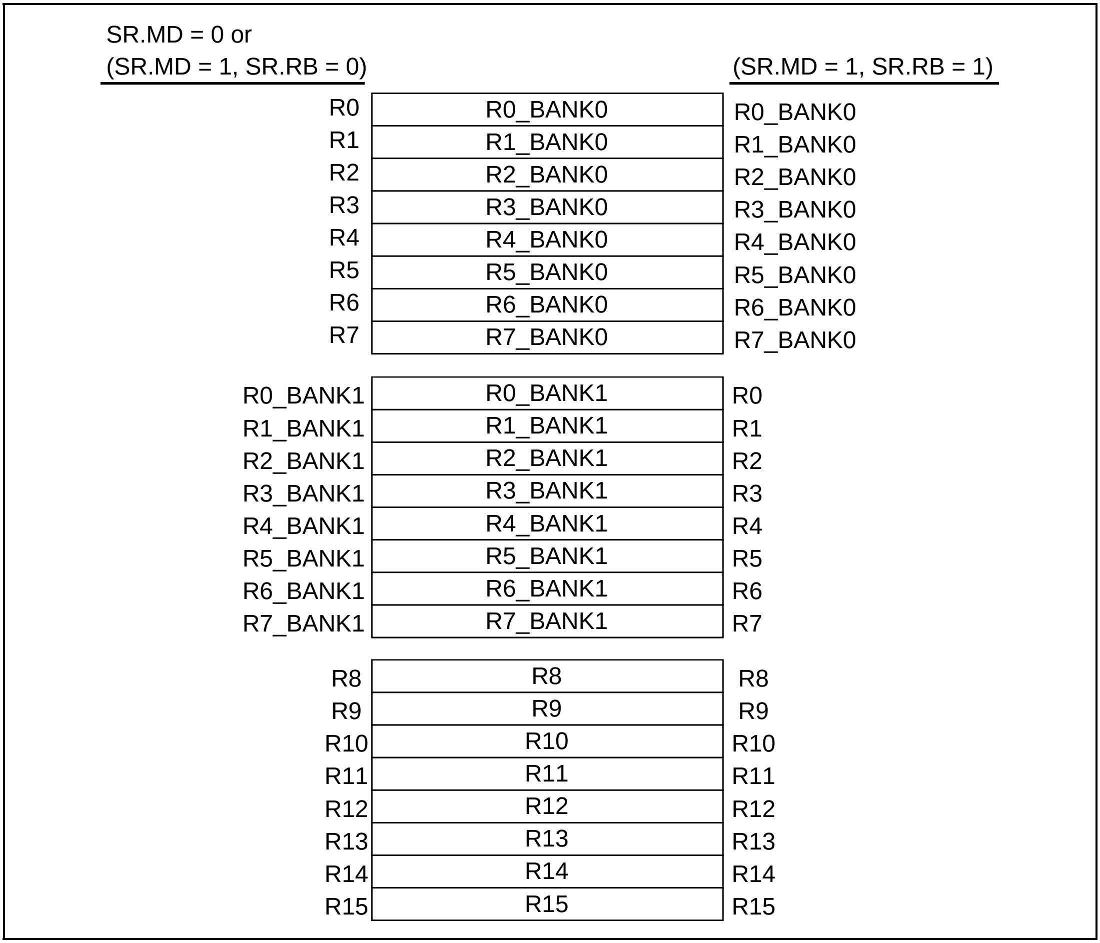


Figure 2.3 General Registers

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

After a reset, the values of R0_BANK0–R7_BANK0, R0_BANK1–R7_BANK1, and R8–R15 are undefined.

2.2.3 Floating-Point Registers

Figure 2.4 shows the floating-point registers. There are thirty-two 32-bit floating-point registers, divided into two banks (FPR0_BANK0–FPR15_BANK0 and FPR0_BANK1–FPR15_BANK1). These 32 registers are referenced as FR0–FR15, DR0/2/4/6/8/10/12/14, FV0/4/8/12, XF0–XF15, XD0/2/4/6/8/10/12/14, or XMTRX. The correspondence between FPRn_BANKi and the reference name is determined by the FR bit in FPSCR (see figure 2.4).

- Floating-point registers, FPRn_BANKi (32 registers)

  FPR0_BANK0, FPR1_BANK0, FPR2_BANK0, FPR3_BANK0, FPR4_BANK0, FPR5_BANK0, FPR6_BANK0, FPR7_BANK0, FPR8_BANK0, FPR9_BANK0, FPR10_BANK0, FPR11_BANK0, FPR12_BANK0, FPR13_BANK0, FPR14_BANK0, FPR15_BANK0

  FPR0_BANK1, FPR1_BANK1, FPR2_BANK1, FPR3_BANK1, FPR4_BANK1, FPR5_BANK1, FPR6_BANK1, FPR7_BANK1, FPR8_BANK1, FPR9_BANK1, FPR10_BANK1, FPR11_BANK1, FPR12_BANK1, FPR13_BANK1, FPR14_BANK1, FPR15_BANK1

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

  When FPSCR.FR = 0, FR0–FR15 are assigned to FPR0_BANK0–FPR15_BANK0.

  When FPSCR.FR = 1, FR0–FR15 are assigned to FPR0_BANK1–FPR15_BANK1.

- Double-precision floating-point registers or single-precision floating-point register pairs, 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}

- 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}

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

  When FPSCR.FR = 0, XF0–XF15 are assigned to FPR0_BANK1–FPR15_BANK1.

  When FPSCR.FR = 1, XF0–XF15 are assigned to FPR0_BANK0–FPR15_BANK0.

- Single-precision floating-point extended register pairs, 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}

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

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

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

Programming Note: After a reset, the values of FPR0_BANK0–FPR15_BANK0 and FPR0_BANK1–FPR15_BANK1 are undefined.

2.2.4 Control Registers

Status register, SR (32 bits, privilege protection, initial value = 0111 0000 0000 0000 0000 00XX 1111 00XX (X: Undefined))

31	30	29	28	27–16	15	14–10	9	8	7–4	3–2	1	0
—	MD	RB	BL	—	FD	—	M	Q	IMASK	—	S	T

Note: —: Reserved. These bits are always read as 0, and should only be written with 0.

- MD: Processor mode

  MD = 0: User mode (some instructions cannot be executed, and some resources cannot be accessed)

  MD = 1: Privileged mode

- RB: General register bank specifier in privileged mode (set to 1 by a reset, exception, or interrupt)

  RB = 0: R0_BANK0–R7_BANK0 are accessed as general registers R0–R7. (R0_BANK1–R7_BANK1 can be accessed using LDC/STC R0_BANK–R7_BANK instructions.)

  RB = 1: R0_BANK1–R7_BANK1 are accessed as general registers R0–R7. (R0_BANK0–R7_BANK0 can be accessed using LDC/STC R0_BANK–R7_BANK instructions.)

- BL: Exception/interrupt block bit (set to 1 by a reset, exception, or interrupt)

  BL = 1: Interrupt requests are masked. If a general exception other than a user break occurs while BL = 1, the processor switches to the reset state.

- FD: FPU disable bit (cleared to 0 by a reset)

  FD = 1: An FPU instruction causes a general FPU disable exception, and if the FPU instruction is in a delay slot, a slot FPU disable exception is generated. (FPU instructions: H'F*** instructions, LDC(.L)/STS(.L) instructions for FPUL/FPSCR)

- M, Q: Used by the DIV0S, DIV0U, and DIV1 instructions.

- IMASK: Interrupt mask level

  External interrupts of a same level or a lower level than IMASK are masked.

- S: Specifies a saturation operation for a MAC instruction.

- T: True/false condition or carry/borrow bit

Saved status register, SSR (32 bits, privilege protection, initial value undefined): The current contents of SR are saved to SSR in the event of an exception or interrupt.

Saved program counter, SPC (32 bits, privilege protection, 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 in a GBR-referencing MOV instruction.

Vector base register, VBR (32 bits, privilege protection, initial value = H'0000 0000): VBR is referenced as the branch destination base address in the event of an exception or interrupt. For details, see section 5, Exceptions.

Saved general register 15, SGR (32 bits, privilege protection, 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, privilege protection, initial value undefined): When the user break debug function is enabled (BRCR.UBDE = 1), DBR is referenced as the user break handler branch destination address instead of VBR.

2.2.5 System Registers

Multiply-and-accumulate register high, MACH (32 bits, initial value undefined)
Multiply-and-accumulate register low, MACL (32 bits, initial value undefined)
MACH/MACL is used for the added value in a MAC instruction, and to store a MAC instruction or MUL operation result.

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, and PR is referenced by the subroutine return instruction (RTS).

Program counter, PC (32 bits, initial value = H'A000 0000): PC indicates the instruction fetch address.

Floating-point status/control register, FPSCR (32 bits, initial value = H'0004 0001)

31–22	21	20	19	18	17–12	11–7	6–2	1–0
—	FR	SZ	PR	DN	Cause	Enable	Flag	RM

Note: —: Reserved. These bits are always read as 0, and should only be written with 0.

- FR: Floating-point register bank

  FR = 0: FPR0_BANK0–FPR15_BANK0 are assigned to FR0–FR15; FPR0_BANK1–FPR15_BANK1 are assigned to XF0–XF15.

  FR = 1: FPR0_BANK0–FPR15_BANK0 are assigned to XF0–XF15; FPR0_BANK1–FPR15_BANK1 are assigned to FR0–FR15.

- SZ: Transfer size mode

  SZ = 0: The data size of the FMOV instruction is 32 bits.

  SZ = 1: The data size of the FMOV instruction is a 32-bit register pair (64 bits).

- PR: Precision mode

  PR = 0: Floating-point instructions are executed as single-precision operations.

  PR = 1: Floating-point instructions are executed as double-precision operations (the result of instructions for which double-precision is not supported is undefined).

  Do not set SZ and PR to 1 simultaneously; this setting is reserved.

  [SZ, PR = 11]: Reserved (FPU operation instruction is undefined.)

- DN: Denormalization mode

  DN = 0: A denormalized number is treated as such.

  DN = 1: A denormalized number is treated as zero.

- Cause: FPU exception cause field
- Enable: FPU exception enable field
- Flag: FPU exception flag field

		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

When an FPU operation instruction is executed, the FPU exception cause field is cleared to zero first. When the next FPU exception is occured, the corresponding bits in the FPU exception cause field and FPU exception flag field are set to 1. The FPU exception flag field holds the status of the exception generated after the field was last cleared.

- RM: Rounding mode

  RM = 00: Round to Nearest

  RM = 01: Round to Zero

  RM = 10: Reserved

  RM = 11: Reserved

- Bits 22 to 31: Reserved

Floating-point communication register, FPUL (32 bits, initial value undefined): Data transfer between FPU registers and CPU registers is carried out via the FPUL register.

Programming Note: When SZ = 1 and big endian mode is selected, FMOV can be used for double-precision floating-point load or store operations. In little endian mode, two 32-bit data size moves must be executed, with SZ = 0, to load or store a double-precision floating-point number.

2.3 Memory-Mapped Registers

The control registers are double-mapped to the following two memory areas. All registers have two addresses.

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

These two areas are used as follows.

- H'1C00 0000–H'1FFF FFFF

  This area must be accessed using the MMU's address translation function. A memory-mapped register can be accessed by setting the page number of this area in the corresponding field of the TLB. Operation is not guaranteed if this area is accessed without using the MMU's address translation function.

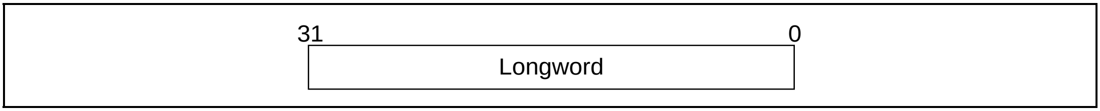
- H'FC00 0000–H'FFFF FFFF

  Access to area H'FC00 0000–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 undefined locations in either area The operation of an access to an undefined location 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 Format 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.



2.5 Data Formats in Memory

Memory data formats are classified into bytes, words, and longwords. Memory can be accessed in 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 MD5 external pin in a power-on reset. Big endian is selected when the MD5 pin is low, and little endian when high. 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 2.5.

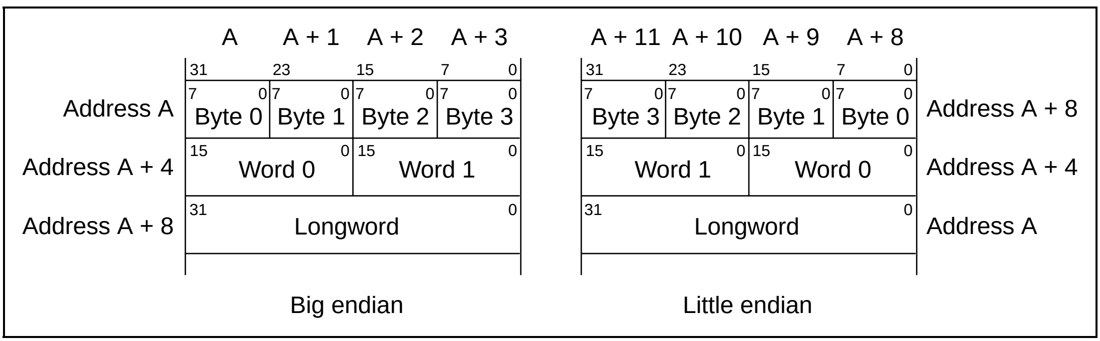


Figure 2.5 Data Formats In Memory

Note: The SH-4 does not support endian conversion for the 64-bit data format. Therefore, if double-precision floating-point format (64-bit) access is performed in little endian mode, the upper and lower 32 bits will be reversed.

2.6 Processor States

The SH-4 has five processor states: the reset state, exception-handling state, bus-released state, program execution state, and power-down state.

Reset State: In this state the CPU is reset. There are two kinds of reset state, power-on reset and manual reset, defined as shown in table 2.6 according to the relevant external pin states.

Table 2.6 Reset State

	Power-On Reset State	Manual Reset State
SH7750 Series	$\overline{\text{RESET}}$ = 0 and $\overline{\text{MRESET}}$ = 1	$\overline{\text{RESET}}$ = 0 and $\overline{\text{MRESET}}$ = 0
SH7751	$\overline{\text{RESET}}$ = 0	$\overline{\text{RESET}}$ = 1 and $\overline{\text{MRESET}}$ = 0

For more information on resets, see section 5, Exceptions.

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 registers of on-chip peripheral modules other than the bus state controller (BSC) are initialized. Since the bus state controller (BSC) is not initialized in the manual reset state, refreshing operations continue. Refer to the register configurations in the relevant sections for further details.

Exception-Handling State: This is a transient state during which the CPU's processor state flow is altered by a reset, general exception, or interrupt exception handling source.

In the case of a reset, the CPU branches to address H'A000 0000 and starts executing the user-coded exception handling program.

In the case of a general exception or interrupt, the program counter (PC) contents are saved in the saved program counter (SPC), the status register (SR) contents are saved in the saved status register (SSR), and the R15 contents are saved in saved general register 15 (SGR). The CPU branches to the start address of the user-coded exception service routine found from the sum of the contents of the vector base address and the vector offset. See section 5, Exceptions, for more information on resets, general exceptions, and interrupts.

Program Execution State: In this state the CPU executes program instructions in sequence.

Power-Down State: In the power-down state, CPU operation halts 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. For details, see hardware manual, Power-Down Modes.

Bus-Released State: In this state the CPU has released the bus to a device that requested it.

SH7750 Series state transitions are shown in figure 2.6, and SH7751 state transitions in figure 2.7.

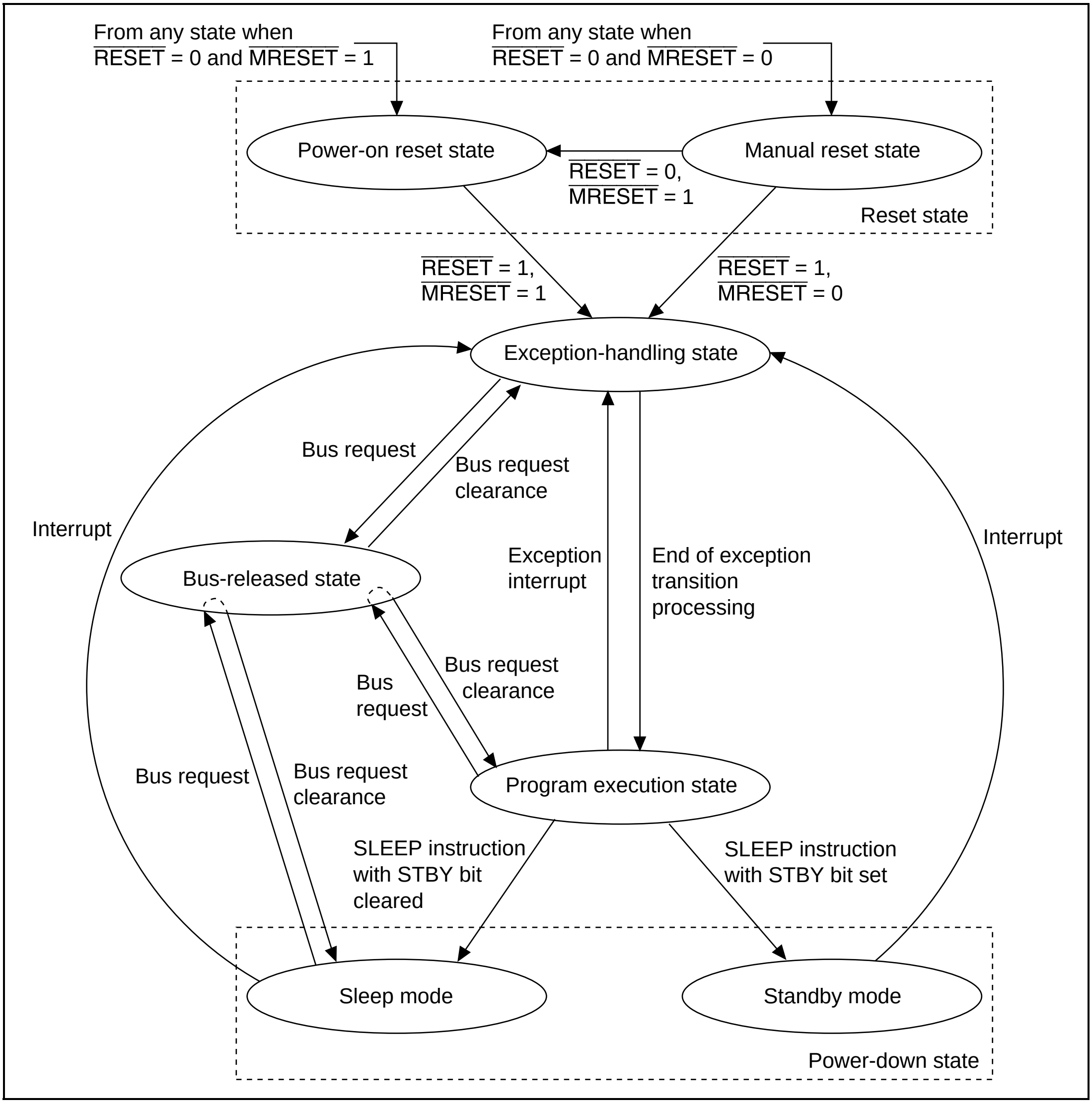


Figure 2.6 Processor State Transitions (SH7750 Series)

RENESAS

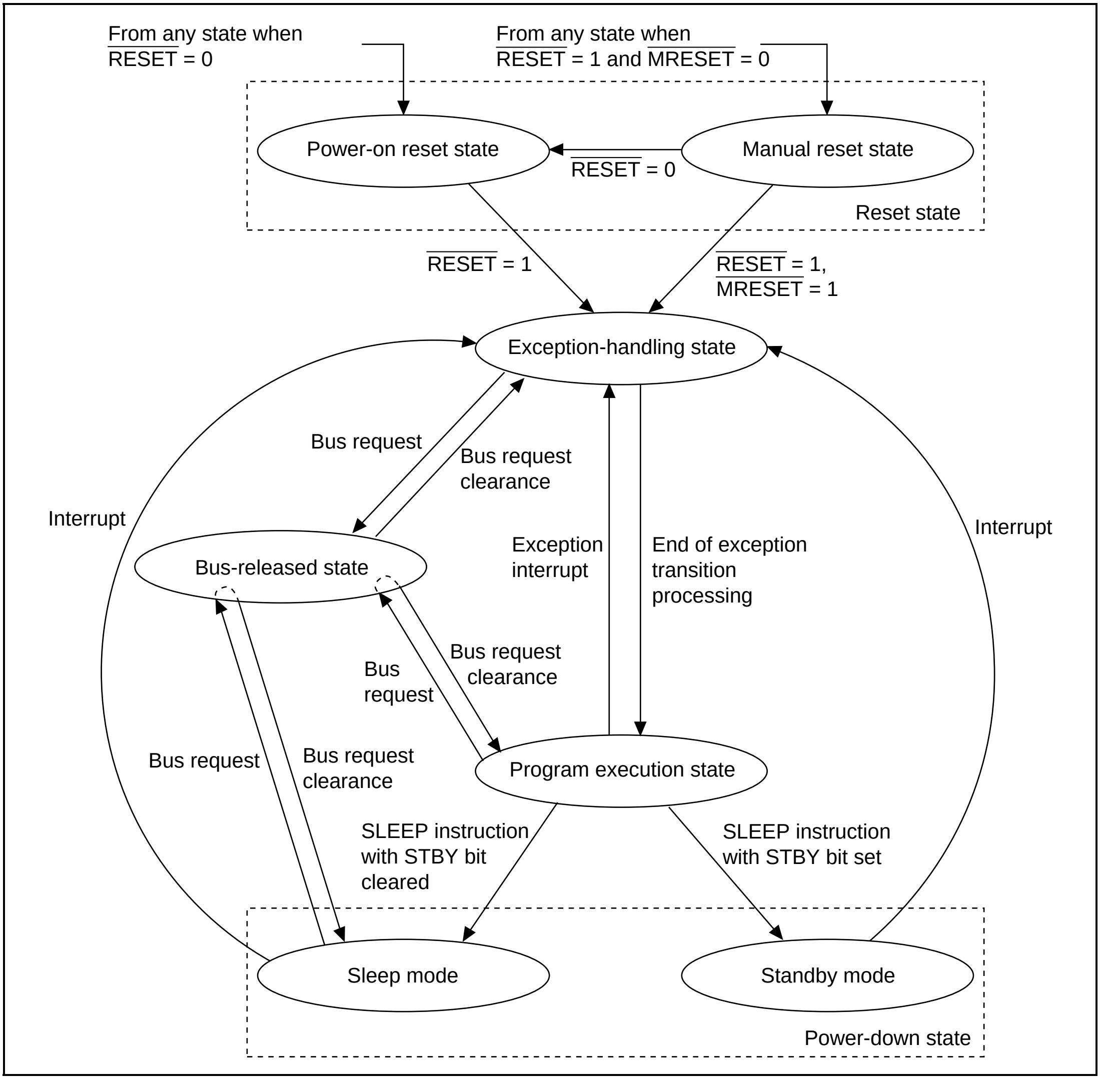


Figure 2.7 Processor State Transitions (SH7751)

2.7 Processor Modes

There are two processor modes: user mode and privileged mode. The processor mode is determined by the processor mode bit (MD) in the status register (SR). User mode is selected when the MD bit is cleared to 0, and privileged mode when the MD bit is set to 1. When the reset state or exception state is entered, the MD bit is set to 1. When exception handling ends, the MD bit is cleared to 0 and user mode is entered. There are certain registers and bits which can only be accessed in privileged mode.

Section 3 Memory Management Unit (MMU)

3.1 Overview

3.1.1 Features

The SH-4 can handle 29-bit external memory space from an 8-bit address space identifier and 32-bit logical (virtual) address space. Address translation from virtual address to physical address is performed using the memory management unit (MMU) built into the SH-4. 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-4 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 support for four page sizes (1, 4, and 64 kbytes, and 1 Mbyte). It is possible to set the virtual address space access right and implement storage protection independently for privileged mode and user mode.

3.1.2 Role of the MMU

The MMU was conceived as a means of making efficient use of physical memory. As shown in figure 3.1, 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 on an ad hoc basis ((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)). 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 task 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)). 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 this 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 in 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 (usually from 1 to 64 kbytes in size).

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

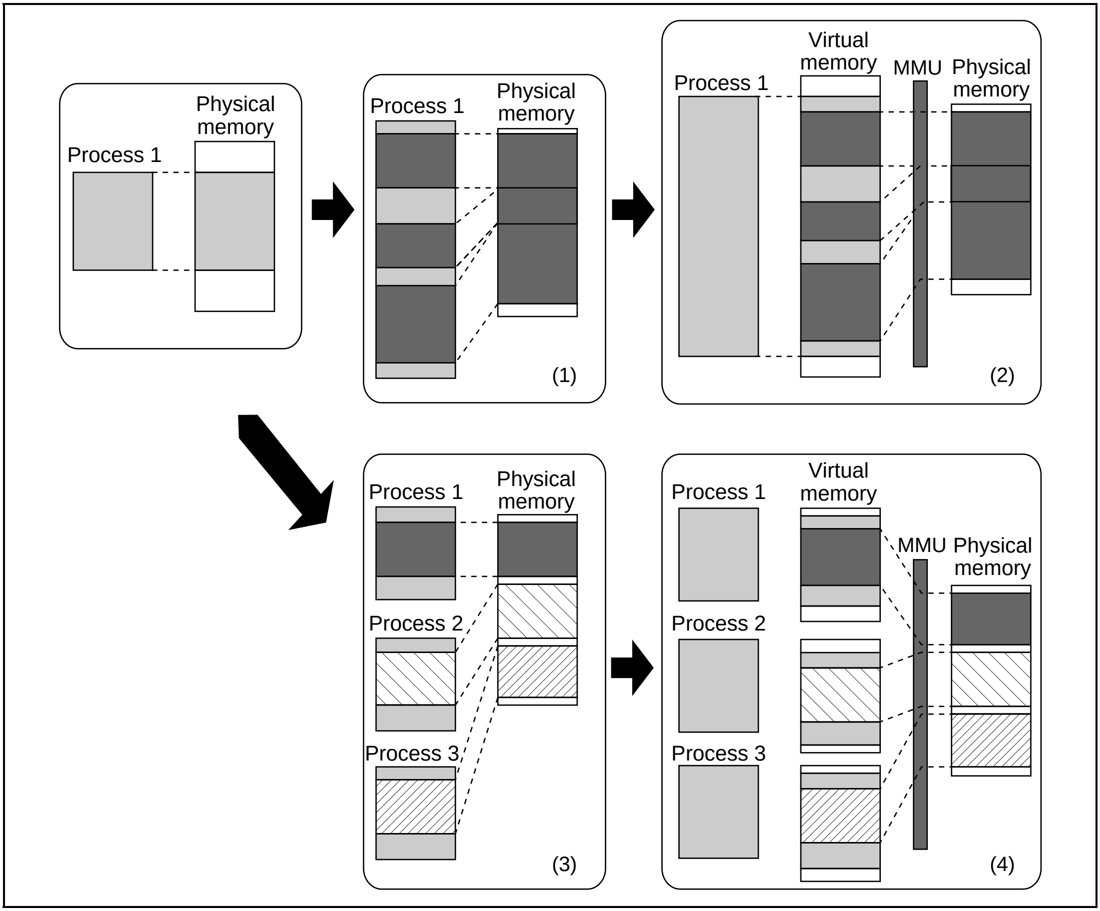


Figure 3.1 Role of the MMU

3.1.3 Register Configuration

The MMU registers are shown in table 3.1.

Table 3.1 MMU Registers

Name	Abbreviation	R/W	Initial Value*[1]	P4 Address*[2]	Area 7 Address*[2]	Access Size
Page table entry high register	PTEH	R/W	Undefined	H'FF00 0000	H'1F00 0000	32
Page table entry low register	PTEL	R/W	Undefined	H'FF00 0004	H'1F00 0004	32
Page table entry assistance register	PTEA	R/W	Undefined	H'FF00 0034	H'1F00 0034	32
Translation table base register	TTB	R/W	Undefined	H'FF00 0008	H'1F00 0008	32
TLB exception address register	TEA	R/W	Undefined	H'FF00 000C	H'1F00 000C	32
MMU control register	MMUCR	R/W	H'0000 0000	H'FF00 0010	H'1F00 0010	32

Notes: 1. The initial value is the value after a power-on reset or manual reset.

2. This is the address when using the virtual/physical address space P4 area. The area 7 address is the address used when making an access from physical address space area 7 using the TLB.

3.1.4 Caution

Operation is not guaranteed if an area designated as a reserved area in this manual is accessed.

3.2 Register Descriptions

There are six MMU-related registers.

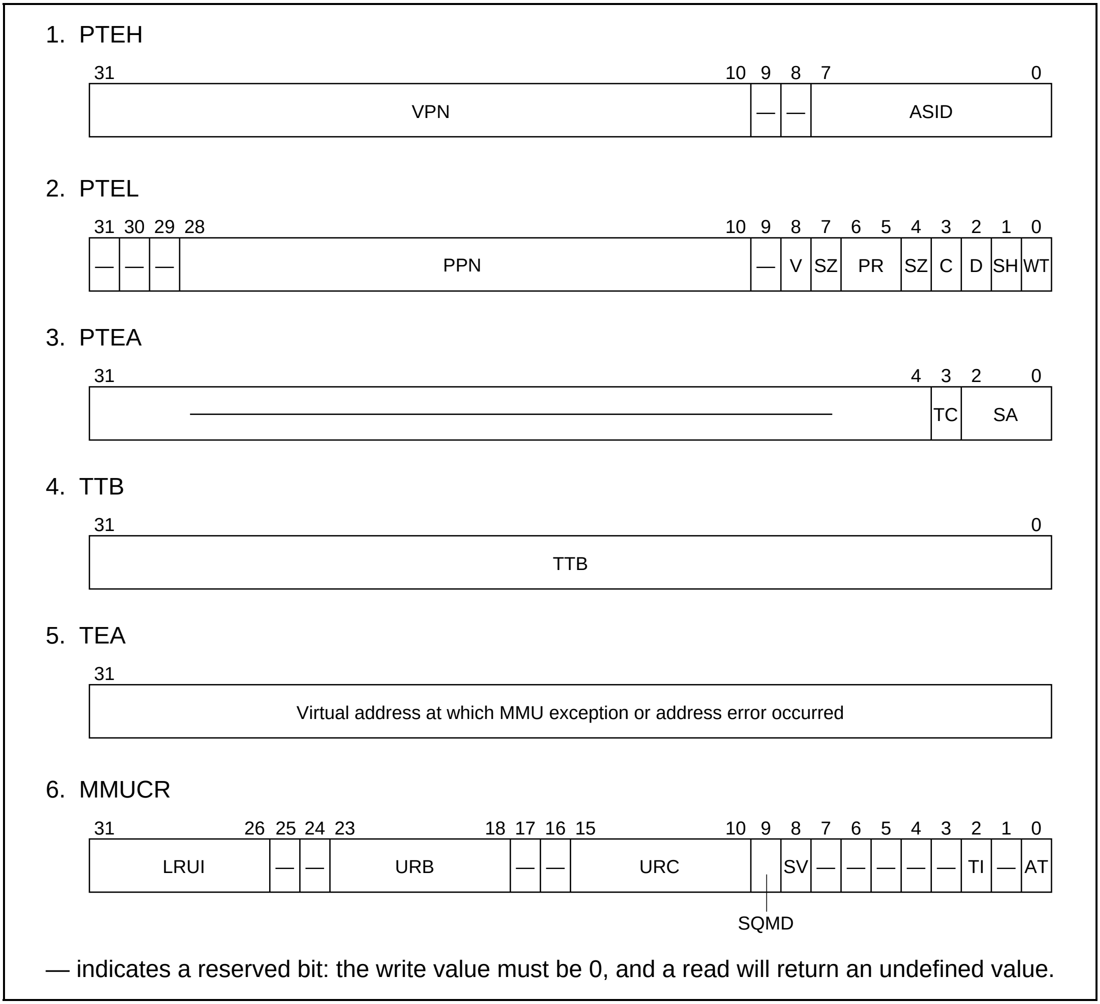


Figure 3.2 MMU-Related Registers

1. **Page table entry high register (PTEH):** Longword access to PTEH can be performed from H'FF00 0000 in the P4 area and H'1F00 0000 in area 7. 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 field 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 field by software. ASID is not updated by hardware. VPN and ASID are recorded in the UTLB by means of the LDLTB instruction.

2. **Page table entry low register (PTEL):** Longword access to PTEL can be performed from H'FF00 0004 in the P4 area and H'1F00 0004 in area 7. 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.

3. **Page table entry assistance register (PTEA):** Longword access to PTEA can be performed from H'FF00 0034 in the P4 area and H'1F00 0034 in area 7. PTEL is used to store assistance bits for PCMCIA access to the UTLB by means of the LDTLB instruction.

In the SH7750S and SH7751, when access to a PCMCIA interface area is performed from the CPU with MMUCR.AT = 0, access is always performed using the values of the SA bit and TC bit in this register. In the SH7750, it is not possible to access a PCMCIA interface area with MMUCR.AT = 0.

In the SH7750 series and SH7751, access to a PCMCIA interface area by the DMAC is always performed using the DMAC’s CHCRn.SSAn, CHCRn.DSAn, CHCRn.STC, and CHCRn.DTC values. See the DMAC section in hardware manual for details.

The contents of this register are not changed unless a software directive is issued.

4. **Translation table base register (TTB):** Longword access to TTB can be performed from H'FF00 0008 in the P4 area and H'1F00 0008 in area 7. TTB is used, for example, to hold the base address of the currently used page table. The contents of TTB are not changed unless a software directive is issued. This register can be freely used by software.

5. **TLB exception address register (TEA):** Longword access to TEA can be performed from H'FF00 000C in the P4 area and H'1F00 000C in area 7. After an MMU exception or address error exception occurs, the virtual address at which the exception occurred is set in TEA by hardware. The contents of this register can be changed by software.

6. **MMU control register (MMUCR):** MMUCR contains the following bits:

LRUI: Least recently used ITLB
URB: UTLB replace boundary
URC: UTLB replace counter
SQMD: Store queue mode bit

SV: Single virtual mode bit
TI: TLB invalidate
AT: Address translation bit

Longword access to MMUCR can be performed from H'FF00 0010 in the P4 area and H'1F00 0010 in area 7. 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 is updated, an instruction that performs data access to the P0, P3, U0, or store queue area should be located at least four instructions after the MMUCR update instruction. Also, a branch instruction to the P0, P3, or U0 area should be located at least eight instructions after the MMUCR update instruction. MMUCR contents can be changed by software. The LRUI bits and URC bits may also be updated by hardware.

- LRUI: 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. A dash in this table means that updating is not performed.

	LRUI					
	[5]	**[4]**	**[3]**	**[2]**	**[1]**	**[0]**
When ITLB entry 0 is used	0	0	0	—	—	—
When ITLB entry 1 is used	1	—	—	0	0	—
When ITLB entry 2 is used	—	1	—	1	—	0
When ITLB entry 3 is used	—	—	1	—	1	1
Other than the above	—	—	—	—	—	—

When the LRUI bit settings are as shown below, the corresponding ITLB entry is updated by an ITLB miss. An asterisk in this table means "don't care".

	LRUI					
	[5]	**[4]**	**[3]**	**[2]**	**[1]**	**[0]**
ITLB entry 0 is updated	1	1	1	*	*	*
ITLB entry 1 is updated	0	*	*	1	1	*
ITLB entry 2 is updated	*	0	*	0	*	1
ITLB entry 3 is updated	*	*	0	*	0	0
Other than the above	Setting prohibited					

Ensure that values for which "Setting prohibited" is indicated in the above table 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.

- URB: Bits that indicate the UTLB entry boundary at which replacement is to be performed. Valid only when URB > 0.

- URC: Random counter for indicating the UTLB entry for which replacement is to be performed with an LDTLB instruction. URC is incremented each time the UTLB is accessed. When URB > 0, URC is reset to 0 when the condition URC = URB occurs. Also note that, if a value is written to URC by software which results in the condition URC > URB, incrementing is first performed in excess of URB until URC = H'3F. URC is not incremented by an LDTLB instruction.

- SQMD: 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)

- SV: Bit that switches between single virtual memory mode and multiple virtual memory mode.

  0: Multiple virtual memory mode

  1: Single virtual memory mode

  When this bit is changed, ensure that 1 is also written to the TI bit.

- TI: Writing 1 to this bit invalidates (clears to 0) all valid UTLB/ITLB bits. This bit always returns 0 when read.

- AT: Specifies MMU enabling or disabling.

  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, therefore, the AT bit should be cleared to 0.

3.3 Memory Space

3.3.1 Physical Memory Space

The SH-4 supports a 32-bit physical memory space, and can access a 4-Gbyte address space. When the MMUCR.AT bit is cleared to 0 and the MMU is disabled, the address space is this physical memory space. The physical memory space is divided into a number of areas, as shown in figure 3.3. The physical memory space is permanently mapped onto 29-bit external memory space; this correspondence can be implemented by ignoring the upper 3 bits of the physical memory space addresses. 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. Accessing the P1 to P4 areas (except the store queue area) in user mode will cause an address error.

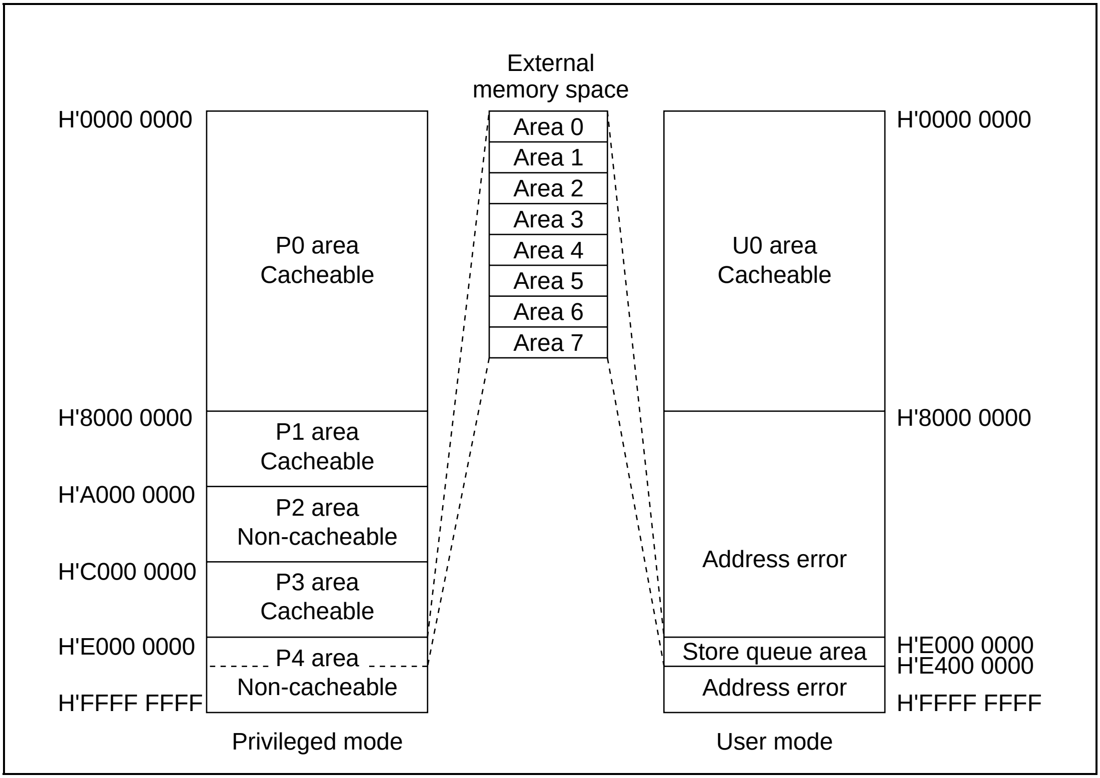


Figure 3.3 Physical Memory Space (MMUCR.AT = 0)

In the SH7750, it is not possible to access a PCMCIA interface area from the CPU.

In the SH7750S and SH7751, when access to a PCMCIA interface area is performed from the CPU, the SA and TC values set in the PTEA register are always used for the access.

Access to a PCMCIA interface area by the DMAC is always performed using the DMAC's CHCRn.SSAn and CHCRn.STCn values. See the DMAC section for details.

P0, P1, P3, U0 Areas: The P0, P1, P3, and U0 areas can be accessed using the cache. Whether or not the cache is used is determined by the cache control register (CCR). When the cache is used, with the exception of the P1 area, switching between the copy-back method and the write-through method for write accesses is specified by the CCR.WT bit. For the P1 area, switching is specified by the CCR.CB bit. Zeroizing the upper 3 bits of an address in these areas gives the corresponding external memory space address. However, since area 7 in the external memory space is a reserved area, a reserved area also appears in these areas.

P2 Area: The P2 area cannot be accessed using the cache. In the P2 area, zeroizing the upper 3 bits of an address gives the corresponding external memory space address. However, since area 7 in the external memory space is a reserved area, a reserved area also appears in this area.

P4 Area: The P4 area is mapped onto SH-4 on-chip I/O channels. This area cannot be accessed using the cache. The P4 area is shown in detail in figure 3.4.

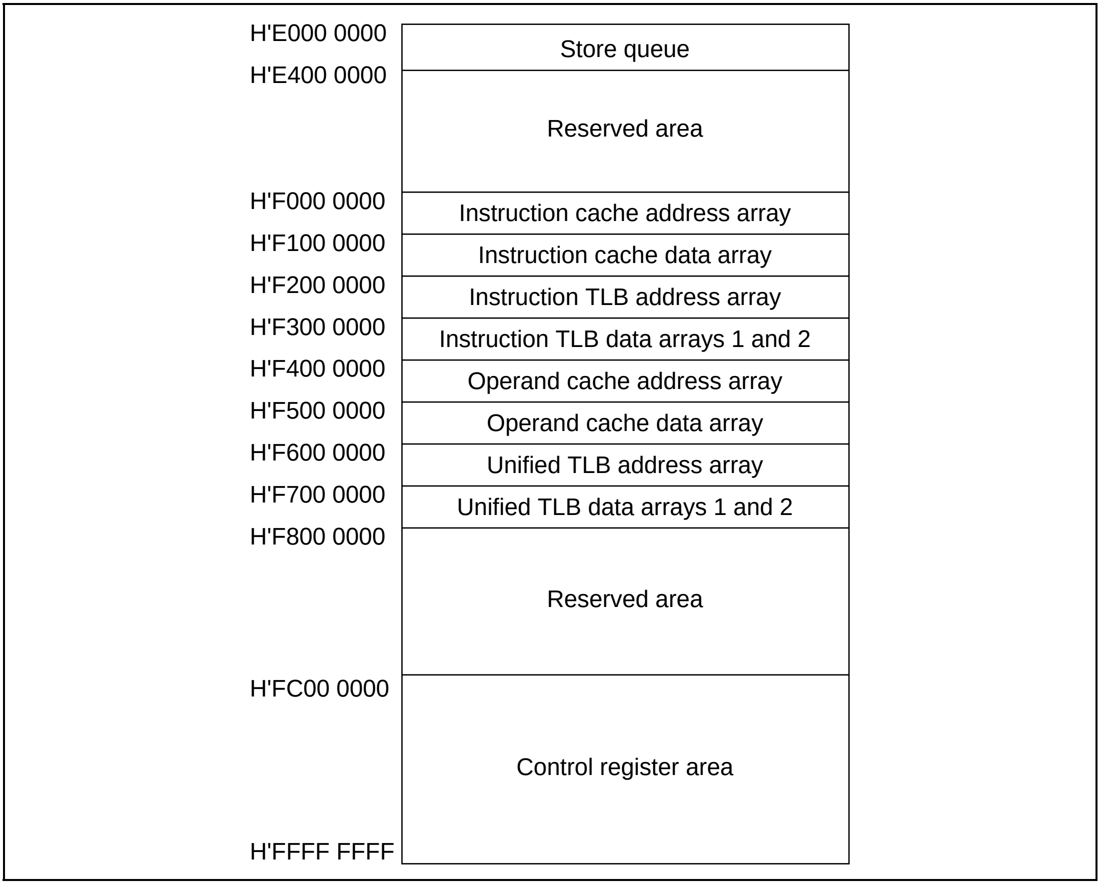


Figure 3.4 P4 Area

The area from H'E000 0000 to H'E3FF FFFF comprises addresses for accessing the store queues (SQs). When the MMU is disabled (MMUCR.AT = 0), the SQ access right is specified by the MMUCR.SQMD bit. For details, see section 4.6, Store Queues.

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 4.5.1, 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 4.5.2, 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 3.7.1, ITLB Address Array.

The area from H'F300 0000 to H'F3FF FFFF is used for direct access to instruction TLB data arrays 1 and 2. For details, see sections 3.7.2, ITLB Data Array 1, and 3.7.3, ITLB Data Array 2.

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 4.5.3, 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 4.5.4, OC Data Array.

The area from H'F600 0000 to H'F6FF FFFF is used for direct access to the unified TLB address array. For details, see section 3.7.4, UTLB Address Array.

The area from H'F700 0000 to H'F7FF FFFF is used for direct access to unified TLB data arrays 1 and 2. For details, see sections 3.7.5, UTLB Data Array 1, and 3.7.6, UTLB Data Array 2.

The area from H'FC00 0000 to H'FFFF FFFF is the control register area.

3.3.2 External Memory Space

The SH-4 supports a 29-bit external memory space. The external memory space is divided into eight areas as shown in figure 3.5. Areas 0 to 6 relate to memory, such as SRAM, synchronous DRAM, DRAM, and PCMCIA. Area 7 is a reserved area. For details, see section 13, Bus State Controller (BSC), in the Hardware Manual.

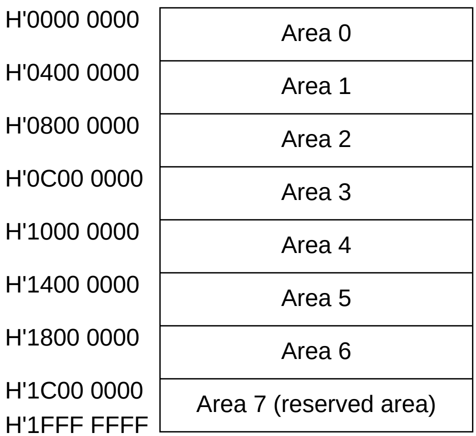


Figure 3.5 External Memory Space

3.3.3 Virtual Memory Space

Setting the MMUCR.AT bit to 1 enables the P0, P3, and U0 areas of the physical memory space in the SH-4 to be mapped onto any external memory space in 1-, 4-, or 64-kbyte, or 1-Mbyte, page units. By using an 8-bit address space identifier, the P0, U0, P3, and store queue areas can be increased to a maximum of 256. This is called the virtual memory space. Mapping from virtual memory space to 29-bit external memory space is carried out using the TLB. Only when area 7 in external memory space is accessed using virtual memory space, addresses H'1C00 0000 to H'1FFF FFFF of area 7 are not designated as a reserved area, but are equivalent to the P4 area control register area in the physical memory space. Virtual memory space is illustrated in figure 3.6.

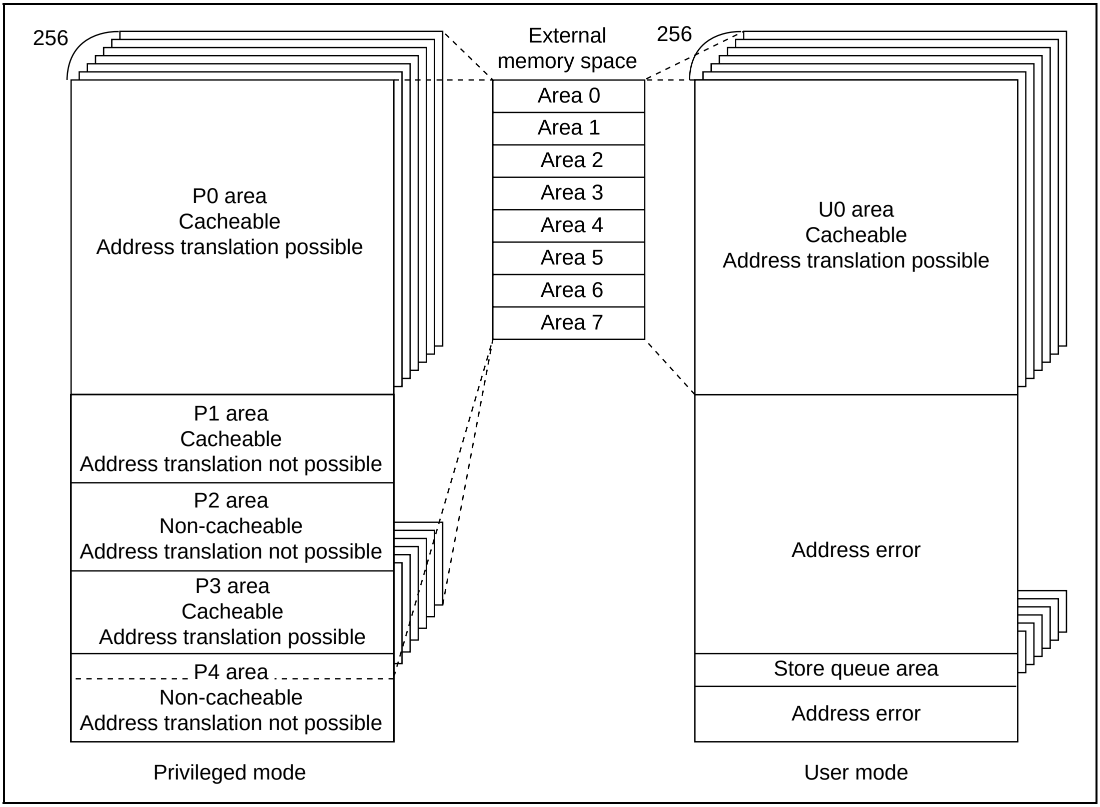


Figure 3.6 Virtual Memory Space (MMUCR.AT = 1)

When areas P0, P3, and U0 are mapped onto PCMCIA interface areas by the TLB in the cache-enabled state, it is necessary to specify 1 for the WT bit of that page, or to clear the C bit to 0. Access is performed using the SA and TC values set for individual TLB pages.

It is not possible to access a PCMCIA interface area from the CPU by access to area P1, P2, or P4.

Access to a PCMCIA interface area by the DMAC is always performed using the DMAC's CHCRn.SSAn and CHCRn.STCn values. See the DMAC section for details.

P0, P3, U0 Areas: The P0 area (excluding addresses H'7C00 0000 to H'7FFF FFFF), P3 area, and U0 area allow access using the cache and address translation using the TLB. These areas can be mapped onto any external memory space in 1-, 4-, or 64-kbyte, or 1-Mbyte, page units. When CCR is in the cache-enabled state and the TLB enable bit (C bit) is 1, accesses can be performed using the cache. In write accesses to the cache, switching between the copy-back method and the write-through method is indicated by the TLB write-through bit (WT bit), and is specified in page units.

Only when the P0, P3, and U0 areas are mapped onto external memory space by means of the TLB, addresses H'1C00 0000 to H'1FFF FFFF of area 7 in external memory space are allocated to the control register area. This enables control registers to be accessed from the U0 area in user mode. In this case, the C bit for the corresponding page must be cleared to 0.

P1, P2, P4 Areas: Address translation using the TLB cannot be performed for the P1, P2, or P4 area (except for the store queue area). Accesses to these areas are the same as for physical memory space. The store queue area can be mapped onto any external memory space by the MMU. However, operation in the case of an exception differs from that for normal P0, U0, and P3 spaces. For details, see section 4.6, Store Queues.

3.3.4 On-Chip RAM Space

In the SH-4, half (8 kbytes) of the instruction cache (16 kbytes) can be used as on-chip RAM. This can be done by changing the CCR settings.

When the operand cache is used as on-chip RAM (CCR.ORA = 1), P0 area addresses H'7C00 0000 to H'7FFF FFFF are an on-chip RAM area. Data accesses (byte/word/longword/quadword) can be used in this area. This area can only be used in RAM mode.

3.3.5 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-4, 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 routine. In the TLB miss exception 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.

3.3.6 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 MMUCR.SV bit. 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 a particular virtual address 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 3.4.3, Address Translation Method).

3.3.7 Address Space Identifier (ASID)

In multiple virtual memory mode, the 8-bit address space identifier (ASID) is used to distinguish between processes running simultaneously while sharing the virtual address space. Software can set the 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 processes running simultaneously while using the virtual memory space on an exclusive basis.

3.4 TLB Functions

3.4.1 Unified TLB (UTLB) Configuration

The unified TLB (UTLB) is so called because of its use 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 instruction TLB in the event of an ITLB miss

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 3.7 shows the overall configuration of the UTLB. The UTLB consists of 64 fully-associative type entries. Figure 3.8 shows the relationship between the address format and page size.

Entry 0	ASID [7:0]	VPN [31:10]	V	PPN [28:10]	SZ [1:0]	SH	C	PR [1:0]	D	WT	SA [2:0]	TC
Entry 1	ASID [7:0]	VPN [31:10]	V	PPN [28:10]	SZ [1:0]	SH	C	PR [1:0]	D	WT	SA [2:0]	TC
Entry 2	ASID [7:0]	VPN [31:10]	V	PPN [28:10]	SZ [1:0]	SH	C	PR [1:0]	D	WT	SA [2:0]	TC
	⋮					⋮						
Entry 63	ASID [7:0]	VPN [31:10]	V	PPN [28:10]	SZ [1:0]	SH	C	PR [1:0]	D	WT	SA [2:0]	TC

Figure 3.7 UTLB Configuration

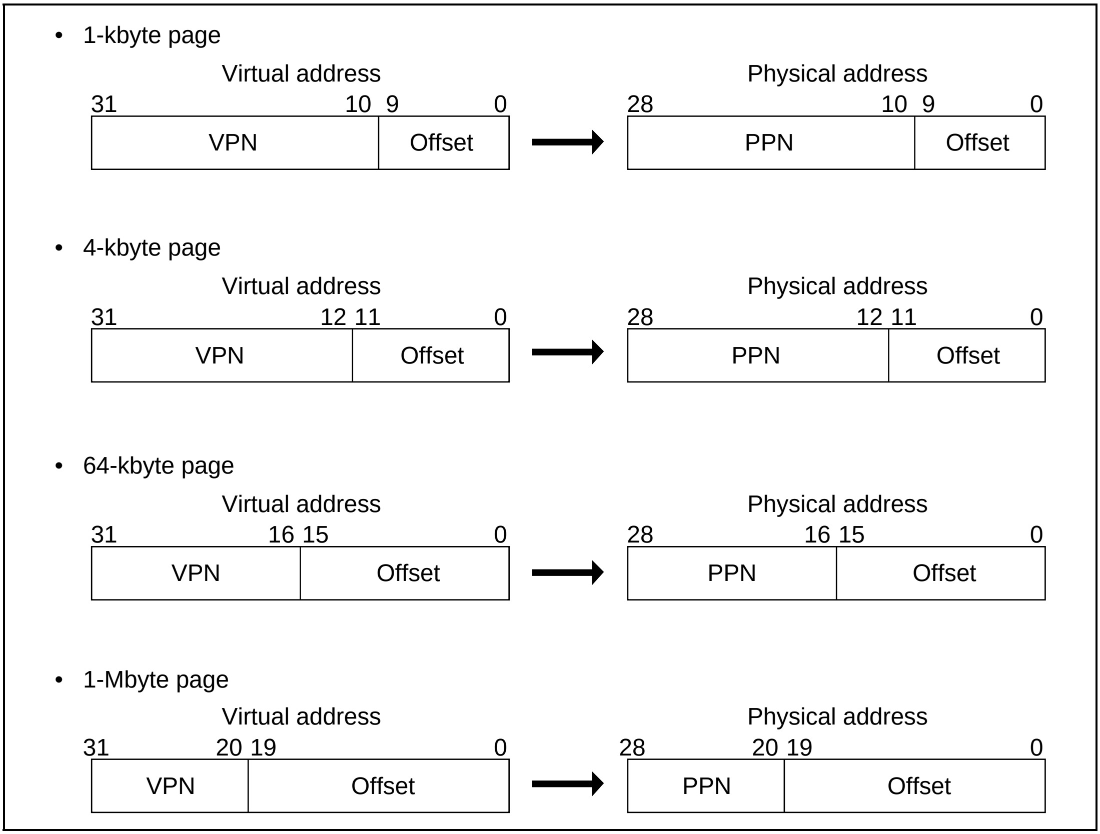


Figure 3.8 Relationship between Page Size and Address Format

- 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: 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.

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

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

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

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

  The synonym problem must be taken into account when setting the PPN (see section 3.5.5, Avoiding Synonym Problems).

- PR: Protection key data

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

  00: Can be read only, in privileged mode

  01: Can be read and written in privileged mode

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

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

- C: Cacheability bit

  Indicates whether a page is cacheable.

  0: Not cacheable

  1: Cacheable

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

  When performing PCMCIA space mapping in the cache enabled state, either clear this bit to 0 or set the WT bit to 1.

- 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

  When performing PCMCIA space mapping in the cache enabled state, either set this bit to 1 or clear the C bit to 0.

- SA: Space attribute bits

  Valid only when the page is mapped onto PCMCIA connected to area 5 or 6.

  000: Undefined

  001: Variable-size I/O space (base size according to $\overline{\text{IOIS16}}$ signal)

  010: 8-bit I/O space

  011: 16-bit I/O space

  100: 8-bit common memory space

  101: 16-bit common memory space

  110: 8-bit attribute memory space

  111: 16-bit attribute memory space

- TC: Timing control bit

  Used to select wait control register bits in the bus control unit for areas 5 and 6.

  0: WCR2 (A5W2–A5W0) and PCR (A5PCW1–A5PCW0, A5TED2–A5TED0, A5TEH2–A5TEH0) are used

  1: WCR2 (A6W2–A6W0) and PCR (A6PCW1–A6PCW0, A6TED2–A6TED0, A6TEH2–A6TEH0) are used

3.4.2 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 3.9 shows the overall configuration of the ITLB. The ITLB consists of 4 fully-associative type entries.

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

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

Figure 3.9 ITLB Configuration

3.4.3 Address Translation Method

Figures 3.10 and 3.11 show flowcharts of memory accesses using the UTLB and ITLB.

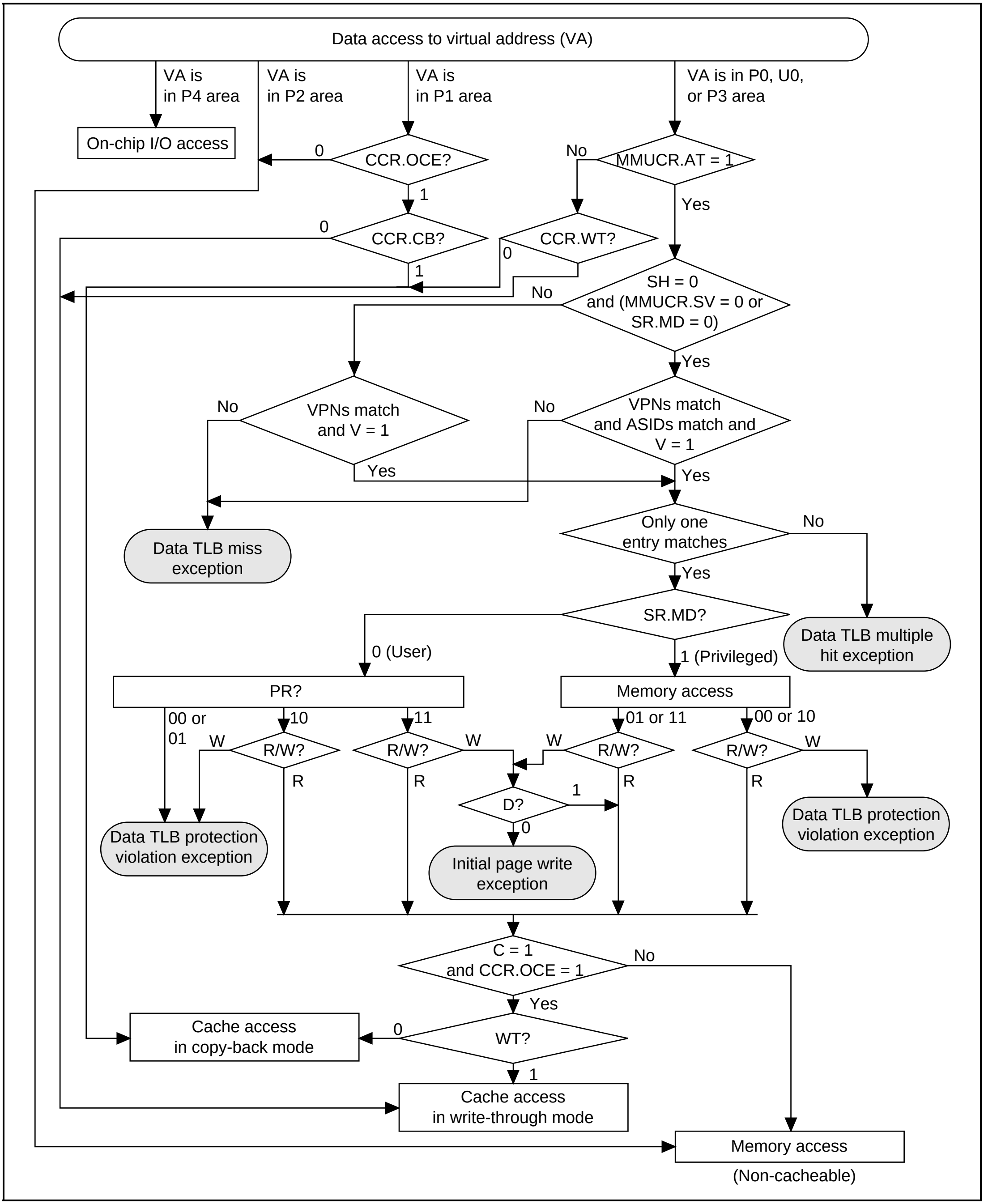


Figure 3.10 Flowchart of Memory Access Using UTLB

RENESAS

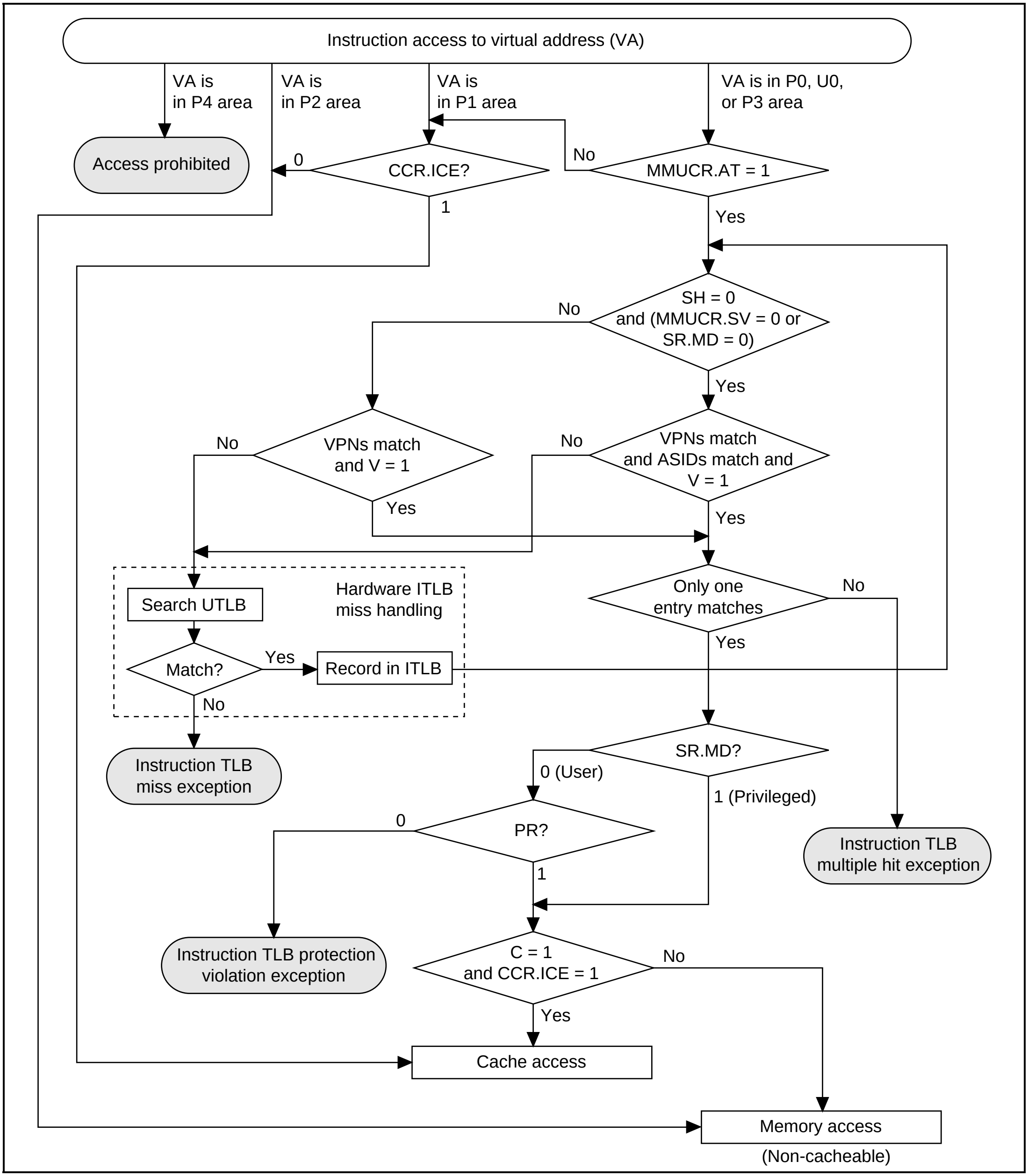


Figure 3.11 Flowchart of Memory Access Using ITLB

3.5 MMU Functions

3.5.1 MMU Hardware Management

The SH-4 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, WT, SA, and TC 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, and if the necessary address translation information is recorded in the UTLB, the MMU copies this information into the ITLB in accordance with MMUCR.LRUI.

3.5.2 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. For deleting or reading UTLB/ITLB entries, it is possible to access the memory-mapped UTLB/ITLB.
3. MMU exception handling. When an MMU exception occurs, processing is performed based on information set by hardware.

3.5.3 MMU Instruction (LDTLB)

A TLB load instruction (LDTLB) is provided for recording UTLB entries. When an LDTLB instruction is issued, the SH-4 copies the contents of PTEH, PTEL, and PTEA to the UTLB entry indicated by MMUCR.URC. 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. The operation of the LDTLB instruction is shown in figure 3.12.

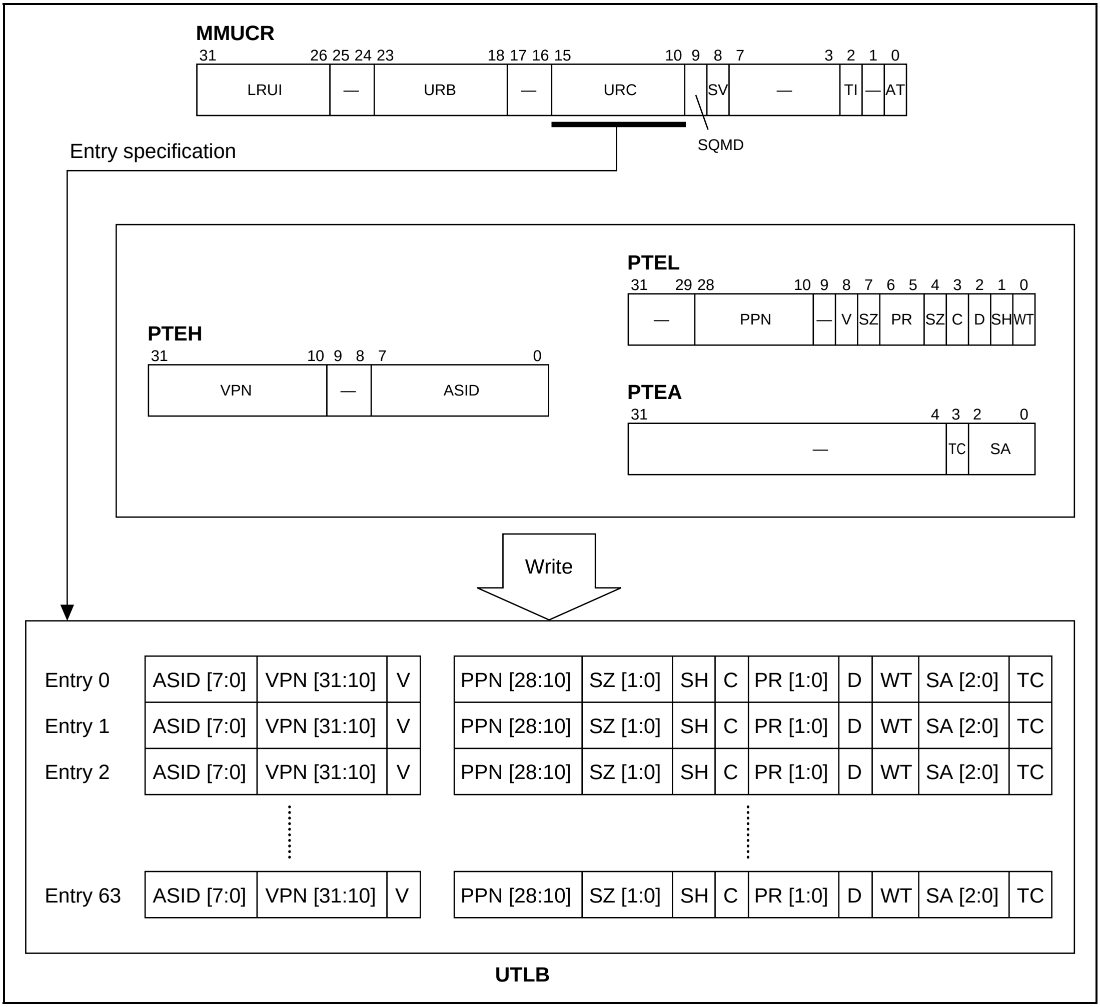


Figure 3.12 Operation of LDTLB Instruction

3.5.4 Hardware ITLB Miss Handling

In an instruction access, the SH-4 searches the ITLB. If it cannot find the necessary address translation information (i.e. in the event of an ITLB miss), 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.

3.5.5 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 or instruction cache . In the SH-4, entry specification is performed using bits [13:5] of the virtual address in order to achieve fast operand cache operation. However, bits [13:10] of the virtual address in the case of a 1-kbyte page, and bits [13:12] of the virtual address in the case of a 4-kbyte page, are subject to address translation. As a result, bits [13:10] of the physical address after translation may differ from bits [13:10] of the virtual address.

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

1. 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 [13:10] values are the same.
2. 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 [13:12] values are the same.
3. Do not use 1-kbyte page UTLB entry physical addresses with UTLB entries of a different page size.
4. 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. When cache index mode is used, VPN [25] is used for the entry address instead of VPN [13], and therefore the above restrictions apply to VPN [25].

Note: When multiple items of address translation information use the same physical memory to provide for future SuperH RISC engine family expansion, 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.

3.6 MMU Exceptions

There are seven MMU exceptions: the 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 3.10 and 3.11 for the conditions under which each of these exceptions occurs.

3.6.1 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, a data 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.

3.6.2 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 procedure. 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. If necessary, the values of the SA and TC bits should be written to PTEA.
2. When the entry to be replaced in entry replacement is specified by software, write that value to URC in the MMUCR register. 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, PTEL, and PTEA 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.

3.6.3 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.

3.6.4 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. A data TLB multiple hit exception is also generated if multiple hits occur when the UTLB is searched in hardware ITLB miss handling.

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.

3.6.5 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. If necessary, the values of the SA and TC bits should be written to PTEA.
2. When the entry to be replaced in entry replacement is specified by software, write that value to URC in the MMUCR register. 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, PTEL, and PTEA 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.

3.6.6 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.

3.6.7 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): The following processing should be carried out as the responsibility of software:

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. If necessary, the values of the SA and TC bits should be written to PTEA.
4. When the entry to be replaced in entry replacement is specified by software, write that value to URC in the MMUCR register. 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, PTEL, and PTEA 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.

3.7 Memory-Mapped TLB Configuration

To enable the ITLB and UTLB to be managed by software, their contents can be read and written by a P2 area program with a MOV instruction in privileged mode. Operation is not guaranteed if access is made from a program in another area. A branch to an area other than the P2 area should be made at least 8 instructions after this MOV instruction. The ITLB and UTLB are allocated to the P4 area in physical memory space. VPN, V, and ASID in the ITLB can be accessed as an address array, PPN, V, SZ, PR, C, and SH as data array 1, and SA and TC as data array 2. 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 data array 1, and SA and TC as data array 2. 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.

3.7.1 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 selected by bits [9:8]. As longword access is used, 0 should be specified for address field bits [1:0].

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

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.

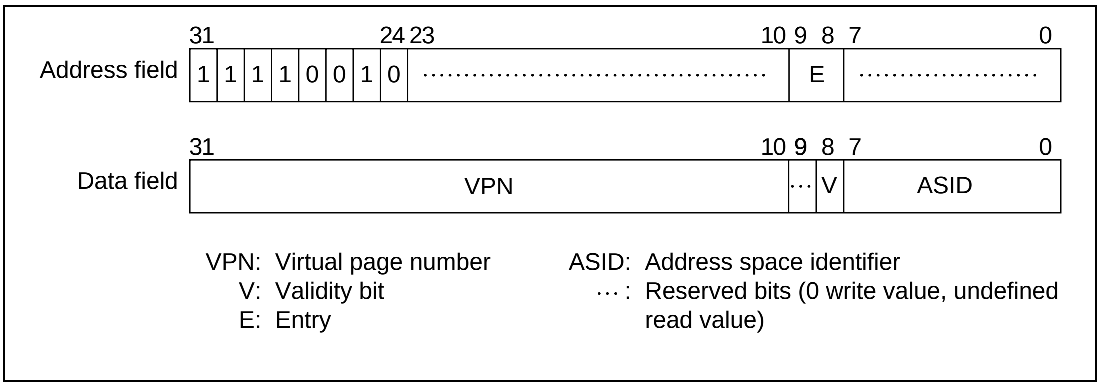


Figure 3.13 Memory-Mapped ITLB Address Array

3.7.2 ITLB Data Array 1

ITLB data array 1 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 1, and the entry is selected by bits [9:8].

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

The following two kinds of operation can be used on ITLB data array 1:

1. ITLB data array 1 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 1 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.

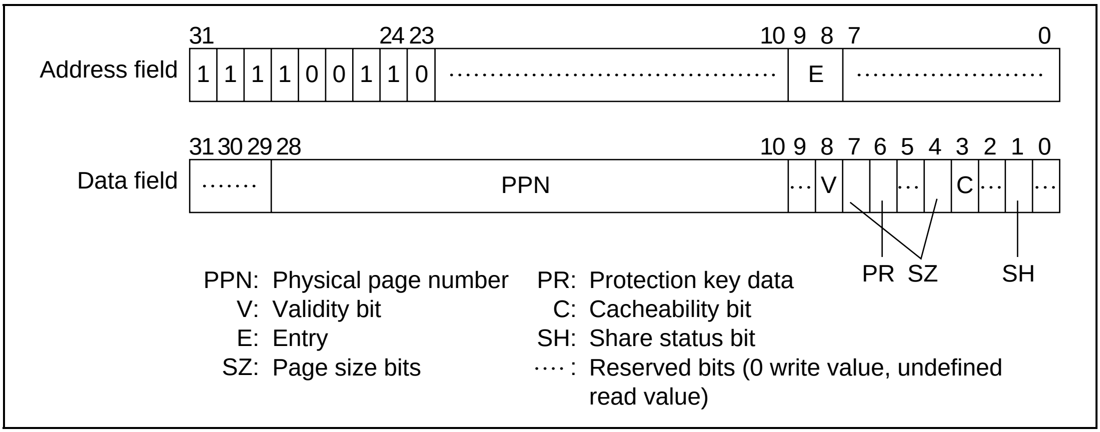


Figure 3.14 Memory-Mapped ITLB Data Array 1

3.7.3 ITLB Data Array 2

ITLB data array 2 is allocated to addresses H'F380 0000 to H'F3FF 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 SA and TC to be written to data array 2 are specified in the data field.

In the address field, bits [31:23] have the value H'F38 indicating ITLB data array 2, and the entry is selected by bits [9:8].

In the data field, SA is indicated by bits [2:0], and TC by bit [3].

The following two kinds of operation can be used on ITLB data array 2:

1. ITLB data array 2 read

   SA and TC are read into the data field from the ITLB entry corresponding to the entry set in the address field.
2. ITLB data array 2 write

   SA and TC specified in the data field are written to the ITLB entry corresponding to the entry set in the address field.

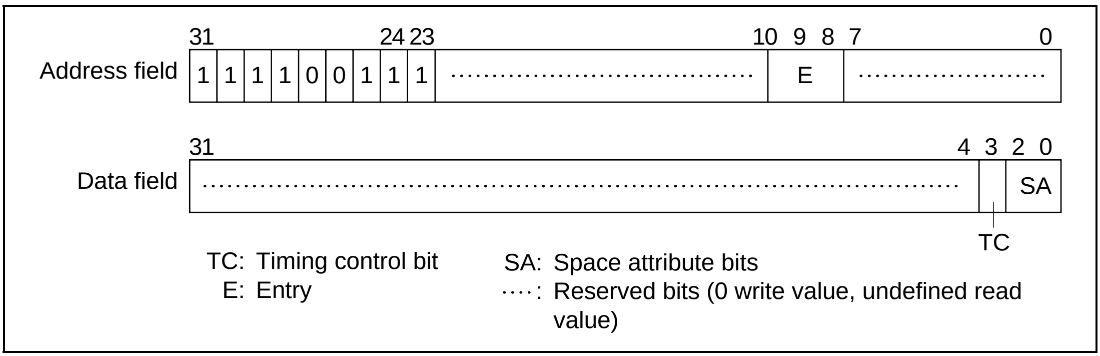


Figure 3.15 Memory-Mapped ITLB Data Array 2

3.7.4 UTLB Address Array

The UTLB address array is allocated to addresses H'F600 0000 to H'F6FF 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 [31:24] have the value H'F6 indicating the UTLB address array, and the entry is selected by bits [13:8]. The address array bit [7] association bit (A bit) specifies whether or not address comparison is performed when writing to the UTLB address array.

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

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 PTEH.ASID. 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. If there is more than one matching entry, a data TLB multiple hit exception results. 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 side only is performed as long as there is an ITLB match. If there is a match in both the UTLB and ITLB, the UTLB information is also written to the ITLB.

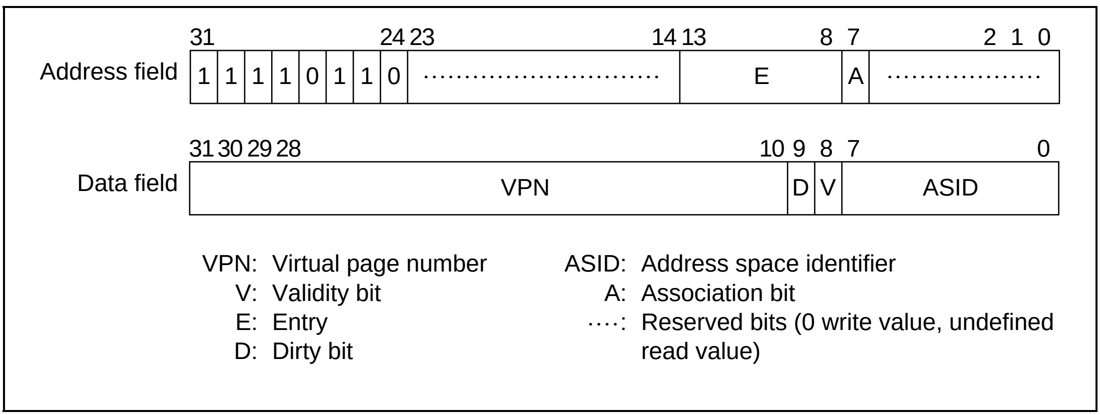


Figure 3.16 Memory-Mapped UTLB Address Array

3.7.5 UTLB Data Array 1

UTLB data array 1 is allocated to addresses H'F700 0000 to H'F77F 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 the data array are specified in the data field.

In the address field, bits [31:23] have the value H'F70 indicating UTLB data array 1, and the entry is selected by bits [13:8].

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

The following two kinds of operation can be used on UTLB data array 1:

1. UTLB data array 1 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 1 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.

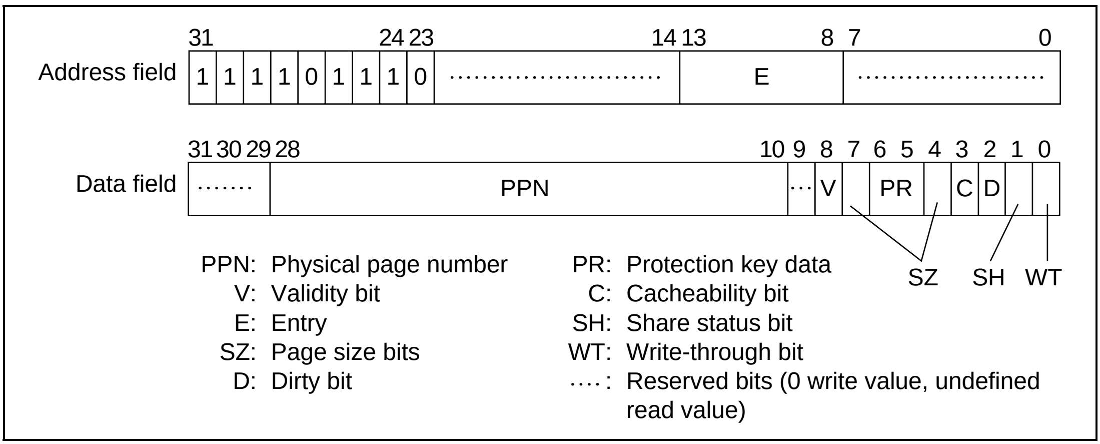


Figure 3.17 Memory-Mapped UTLB Data Array 1

3.7.6 UTLB Data Array 2

UTLB data array 2 is allocated to addresses H'F780 0000 to H'F7FF 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 SA and TC to be written to data array 2 are specified in the data field.

In the address field, bits [31:23] have the value H'F78 indicating UTLB data array 2, and the entry is selected by bits [13:8].

In the data field, TC is indicated by bit [3], and SA by bits [2:0].

The following two kinds of operation can be used on UTLB data array 2:

1. UTLB data array 2 read
   SA and TC are read into the data field from the UTLB entry corresponding to the entry set in the address field.
2. UTLB data array 2 write
   SA and TC specified in the data field are written to the UTLB entry corresponding to the entry set in the address field.

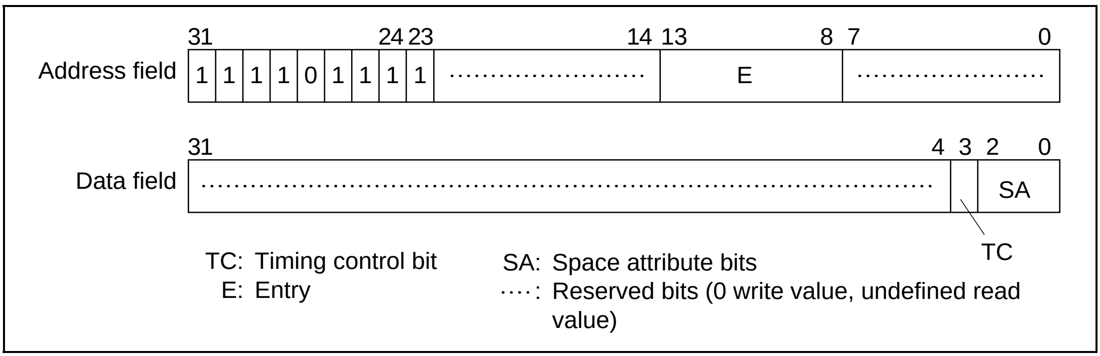


Figure 3.18 Memory-Mapped UTLB Data Array 2

Section 4 Caches

4.1 Overview

4.1.1 Features

The SH-4 has an on-chip 8-kbyte instruction cache (IC) for instructions and 16-kbyte operand cache (OC) for data. Half of the memory of the operand cache (8 kbytes) can also be used as on-chip RAM. The features of these caches are summarized in table 4.1.

Table 4.1 Cache Features

Item	Instruction Cache	Operand Cache
Capacity	8-kbyte cache	16-kbyte cache or 8-kbyte cache + 8-kbyte RAM
Type	Direct mapping	Direct mapping
Line size	32 bytes	32 bytes
Entries	256	512
Write method		Copy-back/write-through selectable

Item	Store Queues
Capacity	2 × 32 bytes
Addresses	H'E000 0000 to H'E3FF FFFF
Write	Store instruction (1-cycle write)
Write-back	Prefetch instruction
Access right	MMU off: according to MMUCR.SQMD MMU on: according to individual page PR

4.1.2 Register Configuration

Table 4.2 shows the cache control registers.

Table 4.2 Cache Control Registers

Name	Abbreviation	R/W	Initial Value*1	P4 Address*2	Area 7 Address*2	Access Size
Cache control register	CCR	R/W	H'0000 0000	H'FF00 001C	H'1F00 001C	32
Queue address control register 0	QACR0	R/W	Undefined	H'FF00 0038	H'1F00 0038	32
Queue address control register 1	QACR1	R/W	Undefined	H'FF00 003C	H'1F00 003C	32

Notes: 1. The initial value is the value after a power-on or manual reset.

2. This is the address when using the virtual/physical address space P4 area. When making an access from physical address space area 7 using the TLB, the upper 3 bits of the address are ignored.

4.2 Register Descriptions

There are three cache and store queue related control registers, as shown in figure 4.1.

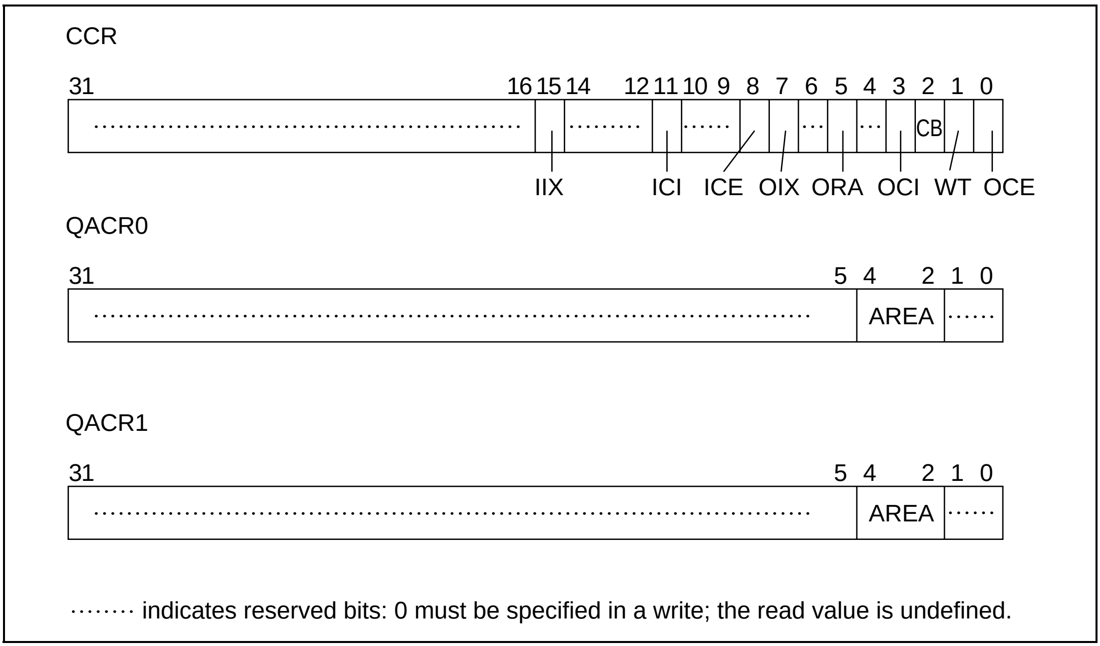


Figure 4.1 Cache and Store Queue Control Registers

(1) Cache Control Register (CCR): CCR contains the following bits:

IIX: IC index enable
ICI: IC invalidation
ICE: IC enable
OIX: OC index enable
ORA: OC RAM enable
OCI: OC invalidation
CB: Copy-back enable
WT: Write-through enable
OCE: OC enable

Longword access to CCR can be performed from H'FF00 001C in the P4 area and H'1F00 001C in area 7. The CCR bits are used for the cache settings described below. Consequently, CCR modifications must only be made by a program in the non-cached P2 area. After CCR is updated, an instruction that performs data access to the P0, P1, P3, or U0 area should be located at least four instructions after the CCR update instruction. Also, a branch instruction to the P0, P1, P3, or U0 area should be located at least eight instructions after the CCR update instruction.

- IIX: IC index enable bit

  0: Address bits [12:5] used for IC entry selection

  1: Address bits [25] and [11:5] used for IC entry selection

- ICI: IC invalidation bit

  When 1 is written to this bit, the V bits of all IC entries are cleared to 0. This bit always returns 0 when read.

- ICE: IC enable bit

  Indicates whether or not the IC is to be used. 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

- OIX: OC index enable bit

  0: Address bits [13:5] used for OC entry selection

  1: Address bits [25] and [12:5] used for OC entry selection

- ORA: OC RAM enable bit

  When the OC is enabled (OCE = 1), the ORA bit specifies whether the 8 kbytes from entry 128 to entry 255 and from entry 384 to entry 511 of the OC are to be used as RAM. When the OC is not enabled (OCE = 0), the ORA bit should be cleared to 0.

  0: 16 kbytes used as cache

  1: 8 kbytes used as cache, and 8 kbytes as RAM

- OCI: 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 always returns 0 when read.

- CB: Copy-back bit

  Indicates the P1 area cache write mode.

  0: Write-through mode

  1: Copy-back mode

- WT: Write-through bit

  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

- OCE: OC enable bit

  Indicates whether or not the OC is to be used. 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

(2) Queue Address Control Register 0 (QACR0): Longword access to QACR0 can be performed from H'FF00 0038 in the P4 area and H'1F00 0038 in area 7. QACR0 specifies the area onto which store queue 0 (SQ0) is mapped when the MMU is off.

(3) Queue Address Control Register 1 (QACR1): Longword access to QACR1 can be performed from H'FF00 003C in the P4 area and H'1F00 003C in area 7. QACR1 specifies the area onto which store queue 1 (SQ1) is mapped when the MMU is off.

4.3 Operand Cache (OC)

4.3.1 Configuration

Figure 4.2 shows the configuration of the operand cache.

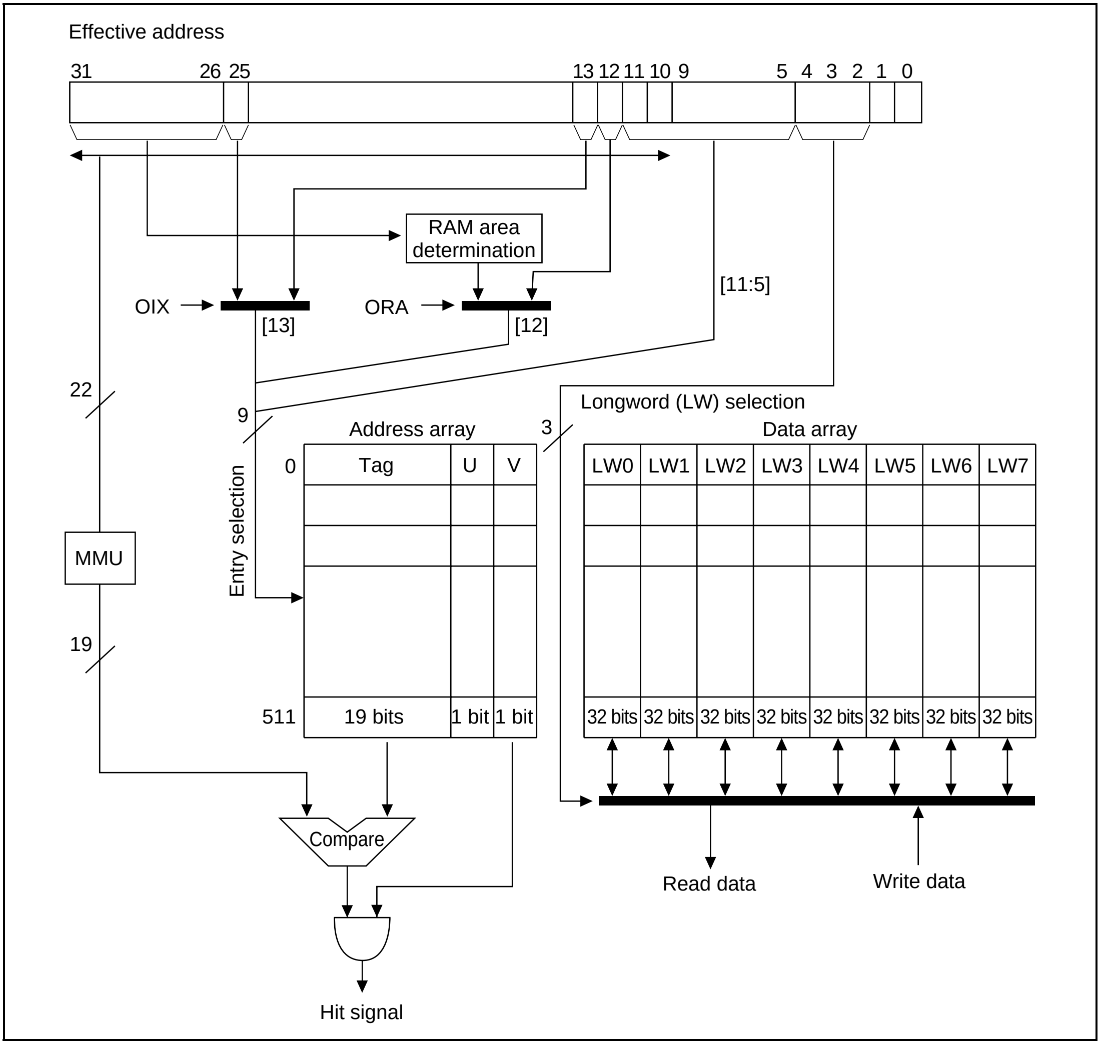


Figure 4.2 Configuration of Operand Cache

The operand cache consists of 512 cache lines, each composed of a 19-bit tag, V bit, U bit, and 32-byte data.

- Tag

  Stores the upper 19 bits of the 29-bit external memory 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 4.5, 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 field

  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.

4.3.2 Read Operation

When the OC is enabled (CCR.OCE = 1) and data is read by means of an effective address from a cacheable area, the cache operates as follows:

1. The tag, V bit, and U bit are read from the cache line indexed by effective address bits [13:5].
2. The tag is compared with bits [28:10] of the address resulting from effective address translation by the MMU:
   - If the tag matches and the V bit is 1 → (3a)
   - If the tag matches and the V bit is 0 → (3b)
   - If the tag does not match and the V bit is 0 → (3b)
   - If the tag does not match, the V bit is 1, and the U bit is 0 → (3b)
   - If the tag does not match, the V bit is 1, and the U bit is 1 → (3c)

3a. Cache hit

The data indexed by effective address bits [4:0] is read from the data field of the cache line indexed by effective address bits [13:5] in accordance with the access size (quadword/longword/word/byte).

3b. Cache miss (no write-back)

Data is read into the cache line from the external memory space corresponding to the effective address. Data reading is performed, using the wraparound method, in order from the longword data corresponding to the effective address, 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 effective address is recorded in the cache, and 1 is written to the V bit.

3c. Cache miss (with write-back)

The tag and data field of the cache line indexed by effective address bits [13:5] are saved in the write-back buffer. Then data is read into the cache line from the external memory space corresponding to the effective address. Data reading is performed, using the wraparound method, in order from the longword data corresponding to the effective address, 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 effective address is recorded in the cache, 1 is written to the V bit, and 0 to the U bit. The data in the write-back buffer is then written back to external memory.

4.3.3 Write Operation

When the OC is enabled (CCR.OCE = 1) and data is written by means of an effective address to a cacheable area, the cache operates as follows:

1. The tag, V bit, and U bit are read from the cache line indexed by effective address bits [13:5].
2. The tag is compared with bits [28:10] of the address resulting from effective address translation by the MMU:

	Copy-back	Write-through
• If the tag matches and the V bit is 1	→ (3a)	→ (3b)
• If the tag matches and the V bit is 0	→ (3c)	→ (3d)
• If the tag does not match and the V bit is 0	→ (3c)	→ (3d)
• If the tag does not match, the V bit is 1, and the U bit is 0	→ (3c)	→ (3d)
• If the tag does not match, the V bit is 1, and the U bit is 1	→ (3e)	→ (3d)

3a. Cache hit (copy-back)

A data write in accordance with the access size (quadword/longword/word/byte) is performed for the data indexed by bits [4:0] of the effective address of the data field of the cache line indexed by effective address bits [13:5]. Then 1 is set in the U bit.

3b. Cache hit (write-through)

A data write in accordance with the access size (quadword/longword/word/byte) is performed for the data indexed by bits [4:0] of the effective address of the data field of the cache line indexed by effective address bits [13:5]. A write is also performed to the corresponding external memory using the specified access size.

3c. Cache miss (no copy-back/write-back)

A data write in accordance with the access size (quadword/longword/word/byte) is performed for the data indexed by bits [4:0] of the effective address of the data field of the cache line indexed by effective address bits [13:5]. Then, data is read into the cache line from the external memory space corresponding to the effective address. Data reading is performed, using the wraparound method, in order from the longword data corresponding to the effective address, and one cache line of data is read excluding the written data. During this time, the CPU can execute the next processing. When reading of one line of data is completed, the tag corresponding to the effective address is recorded in the cache, and 1 is written to the V bit and U bit.

3d. Cache miss (write-through)

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

3e. Cache miss (with copy-back/write-back)

The tag and data field of the cache line indexed by effective address bits [13:5] are first saved in the write-back buffer, and then a data write in accordance with the access size (quadword/longword/word/byte) is performed for the data indexed by bits [4:0] of the effective address of the data field of the cache line indexed by effective address bits [13:5]. Then, data is read into the cache line from the external memory space corresponding to the effective address. Data reading is performed, using the wraparound method, in order from the longword data corresponding to the effective address, and one cache line of data is read excluding the written data. During this time, the CPU can execute the next processing. When reading of one line of data is completed, the tag corresponding to the effective address is recorded in the cache, and 1 is written to the V bit and U bit. The data in the write-back buffer is then written back to external memory.

4.3.4 Write-Back Buffer

In order to give priority to data reads to the cache and improve performance, the SH-4 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 4.3 Configuration of Write-Back Buffer

4.3.5 Write-Through Buffer

The SH-4 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 4.4 Configuration of Write-Through Buffer

4.3.6 RAM Mode

Setting CCR.ORA to 1 enables 8 kbytes of the operand cache to be used as RAM. The operand cache entries used as RAM are entries 128 to 255 and 384 to 511 . Other entries can still be used as cache. RAM can be accessed using addresses H'7C00 0000 to H'7FFF FFFF. Byte-, word-, longword-, and quadword-size data reads and writes can be performed in the operand cache RAM area. Instruction fetches cannot be performed in this area.

An example of RAM use is shown below. Here, the 4 kbytes comprising OC entries 128 to 256 are designated as RAM area 1, and the 4 kbytes comprising OC entries 384 to 511 as RAM area 2.

- When OC index mode is off (CCR.OIX = 0)

  H'7C00 0000 to H'7C00 0FFF (4 kB): Corresponds to RAM area 1

  H'7C00 1000 to H'7C00 1FFF (4 kB): Corresponds to RAM area 1

  H'7C00 2000 to H'7C00 2FFF (4 kB): Corresponds to RAM area 2

  H'7C00 3000 to H'7C00 3FFF (4 kB): Corresponds to RAM area 2

  H'7C00 4000 to H'7C00 4FFF (4 kB): Corresponds to RAM area 1

  : : :

  RAM areas 1 and 2 then repeat every 8 kbytes up to H'7FFF FFFF.

  Thus, to secure a continuous 8-kbyte RAM area, the area from H'7C00 1000 to H'7C00 2FFF can be used, for example.

- When OC index mode is on (CCR.OIX = 1)

  H'7C00 0000 to H'7C00 0FFF (4 kB): Corresponds to RAM area 1

  H'7C00 1000 to H'7C00 1FFF (4 kB): Corresponds to RAM area 1

  H'7C00 2000 to H'7C00 2FFF (4 kB): Corresponds to RAM area 1

  : : :

  H'7DFF F000 to H'7DFF FFFF (4 kB): Corresponds to RAM area 1

  H'7E00 0000 to H'7E00 0FFF (4 kB): Corresponds to RAM area 2

  H'7E00 1000 to H'7E00 1FFF (4 kB): Corresponds to RAM area 2

  : : :

  H'7FFF F000 to H'7FFF FFFF (4 kB): Corresponds to RAM area 2

  As the distinction between RAM areas 1 and 2 is indicated by address bit [25], the area from H'7DFF F000 to H'7E00 0FFF should be used to secure a continuous 8-kbyte RAM area.

4.3.7 OC Index Mode

Setting CCR.OIX to 1 enables OC indexing to be performed using bit [25] of the effective address. This is called OC index mode. In normal mode, with CCR.OIX cleared to 0, OC indexing is performed using bits [13:5] of the effective address; therefore, when 16 kbytes or more of consecutive data is handled, the OC is fully used by this data. This results in frequent cache misses. Using index mode allows the OC to be handled as two 8-kbyte areas by means of effective address bit [25], providing efficient use of the cache.

4.3.8 Coherency between Cache and External Memory

Coherency between cache and external memory should be assured by software. In the SH-4, the following four new instructions are supported for cache operations. For details of these instructions, see section 9, Instruction Descriptions.

Invalidate instruction:	OCBI @Rn	Cache invalidation (no write-back)
Purge instruction:	OCBP @Rn	Cache invalidation (with write-back)
Write-back instruction:	OCBWB @Rn	Cache write-back
Allocate instruction:	MOVCA.L R0,@Rn	Cache allocation

4.3.9 Prefetch Operation

The SH-4 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. For details of the prefetch instruction, see section 9.74, PREF.

Prefetch instruction: PREF @Rn

4.4 Instruction Cache (IC)

4.4.1 Configuration

Figure 4.5 shows the configuration of the instruction cache.

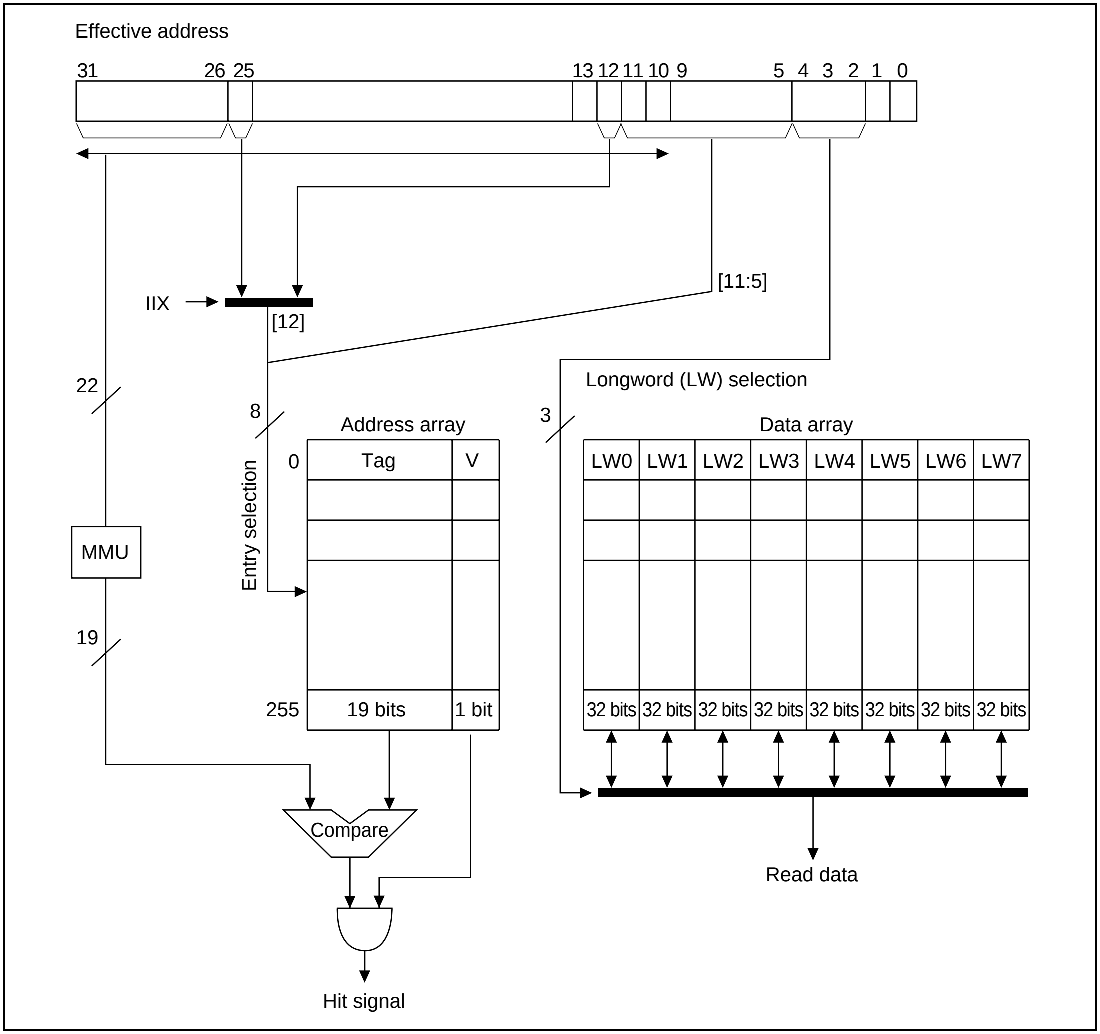


Figure 4.5 Configuration of Instruction Cache

The instruction cache consists of 256 cache lines, each composed of a 19-bit tag, V bit, and 32-byte data (16 instructions).

- Tag

  Stores the upper 19 bits of the 29-bit external 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.
- 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.

4.4.2 Read Operation

When the IC is enabled (CCR.ICE = 1) and instruction fetches are performed by means of an effective address from a cacheable area, the instruction cache operates as follows:

1. The tag and V bit are read from the cache line indexed by effective address bits [12:5].
2. The tag is compared with bits [28:10] of the address resulting from effective address translation by the MMU:
   - If the tag matches and the V bit is 1 → (3a)
   - If the tag matches and the V bit is 0 → (3b)
   - If the tag does not match and the V bit is 0 → (3b)
   - If the tag does not match and the V bit is 1 → (3b)

3a. Cache hit

The data indexed by effective address bits [4:2] is read as an instruction from the data field of the cache line indexed by effective address bits [12:5].

3b. Cache miss

Data is read into the cache line from the external memory space corresponding to the effective address. Data reading is performed, using the wraparound method, in order from the longword data corresponding to the effective address, and when the corresponding data arrives in the cache, the read data is returned to the CPU as an instruction. When reading of one line of data is completed, the tag corresponding to the effective address is recorded in the cache, and 1 is written to the V bit.

4.4.3 IC Index Mode

Setting CCR.IIX to 1 enables IC indexing to be performed using bit [25] of the effective address. This is called IC index mode. In normal mode, with CCR.IIX cleared to 0, IC indexing is performed using bits [12:5] of the effective address; therefore, when 8 kbytes or more of consecutive program instructions are handled, the IC is fully used by this program. This results in frequent cache misses. Using index mode allows the IC to be handled as two 4-kbyte areas by means of effective address bit [25], providing efficient use of the cache.

4.5 Memory-Mapped Cache Configuration

In the SH7750 Series, to enable the IC and OC to be managed by software, their contents can be read and written by a P2 area program with a MOV instruction in privileged mode.

In privileged mode in the SH7751, the contents of OC can be read and written by a P1 or P2 area program with a MOV instruction, and the contents of IC can be read and written by a P2 area program with a MOV instruction.

Operation is not guaranteed if access is made from a program in another area. In this case, a branch to the other area should be made at least 8 instructions after this MOV instruction. The IC and OC are allocated to the P4 area in physical memory 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; their read value is undefined.

4.5.1 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 entry to be accessed is 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 entry is specified by bits [12:5]. CCR.IIX has no effect on this entry specification. The address array bit [3] association bit (A bit) 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 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 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 stored in the entry specified in the address field is compared with the tag specified in the data field. 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 is 1, the V bit specified in the data field is written into the IC entry. This operation is used to invalidate a specific IC entry. If an ITLB miss occurs during address translation, or the comparison shows a mismatch, no operation results and the write is not performed. If an instruction TLB multiple hit exception occurs during address translation, processing switches to the instruction TLB multiple hit exception handling routine.

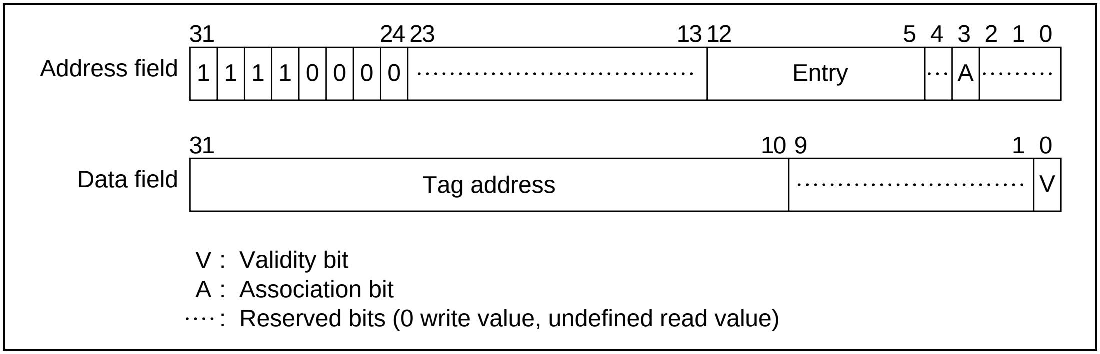


Figure 4.6 Memory-Mapped IC Address Array

4.5.2 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 entry to be accessed is 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 entry is specified by bits [12:5]. CCR.IIX has no effect on this entry specification. 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 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 entry set in the address field.

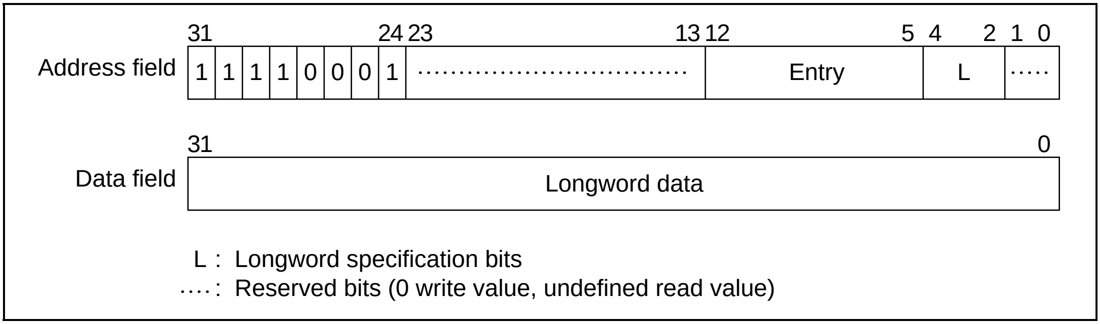


Figure 4.7 Memory-Mapped IC Data Array

4.5.3 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 entry to be accessed is 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 entry is specified by bits [13:5]. CCR.OIX and CCR.ORA have no effect on this entry specification. The address array bit [3] association bit (A bit) 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 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 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 stored in the entry specified in the address field is compared with the tag specified in the data field. 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 is 1, the U bit and V bit specified in the data field are written into the OC entry. 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 an UTLB miss occurs during address translation, or the comparison shows a mismatch, no operation results and the write is not performed. If a data TLB multiple hit exception occurs during address translation, processing switches to the data TLB multiple hit exception handling routine.

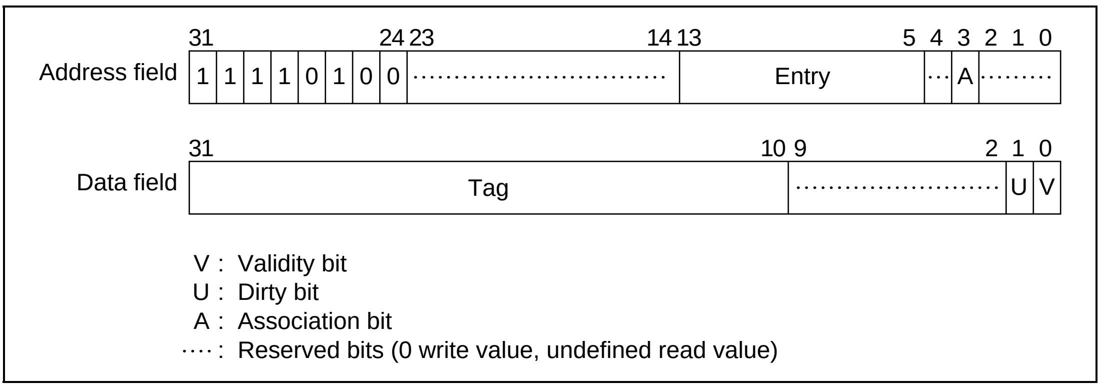


Figure 4.8 Memory-Mapped OC Address Array

4.5.4 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 entry to be accessed is 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 entry is specified by bits [13:5]. CCR.OIX and CCR.ORA have no effect on this entry specification. 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 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 the entry set in the address field. This write does not set the U bit to 1 on the address array side.

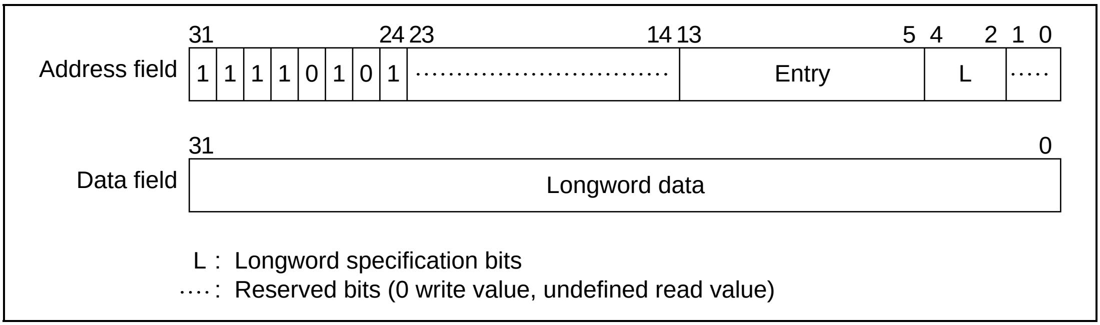


Figure 4.9 Memory-Mapped OC Data Array

4.6 Store Queues

Two 32-byte store queues (SQs) are supported to perform high-speed writes to external memory. In the SH7750S and SH7751, when not using the SQs, the low power dissipation power-down modes, in which SQ functions are stopped, can be used. The queue address control registers (QACR0 and QACR1) cannot be accessed while SQ functions are stopped. See section 9, Power-Down Modes, for the procedure for stopping SQ functions.

4.6.1 SQ Configuration

There are two 32-byte store queues, SQ0 and SQ1, as shown in figure 4.10. These two store queues can be set independently.

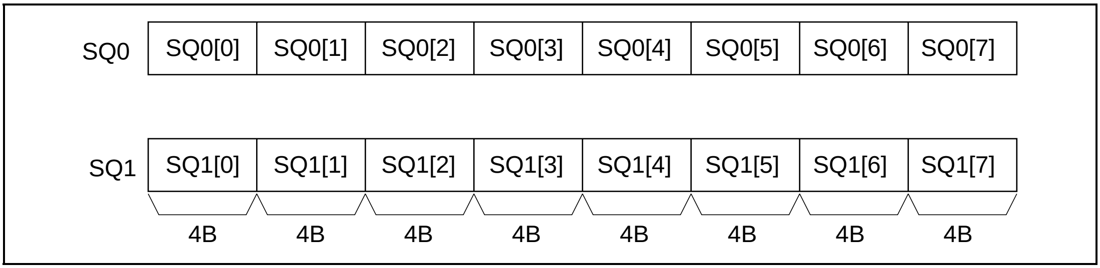


Figure 4.10 Store Queue Configuration

4.6.2 SQ Writes

A write to the SQs can be performed using a store instruction on P4 area H'E000 0000 to H'E3FF FFFC. A longword or quadword access size can be used. The meaning of the address bits is 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

4.6.3 Transfer to External Memory

Transfer from the SQs to external memory can be performed with a prefetch instruction (PREF). Issuing a PREF instruction for P4 area H'E000 0000 to H'E3FF FFFC 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, but writing to the SQ involved in the transfer to external memory is deferred until the transfer is completed.

The SQ transfer destination external memory address bit [28:0] specification is as shown below, according to whether the MMU is on or off.

- When MMU is on (MMUCR.AT = 1)

  The SQ area (H'E000 0000 to H'E3FF FFFF) is set in VPN of the UTLB, and the transfer destination external memory 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. It is not possible to perform data transfer to a PCMCIA interface area using the SQs.

  When a prefetch instruction is issued for the SQ area, address translation is performed and external memory address bits [28:10] are generated in accordance with the SZ bit specification. For external memory address bits [9:5], the address prior to address translation is generated in the same way as when the MMU is off. External memory address bits [4:0] are fixed at 0. Transfer from the SQs to external memory is performed to this address.

- When MMU is off (MMUCR.AT = 0)

  The SQ area (H'E000 0000 to H'E3FF FFFF) is specified as the address at which a prefetch is performed. The meaning of address bits [31:0] is as follows:

[31:26]:	111000	Store queue specification
[25:6]:	Address	External memory address bits [25:6]
[5]:	0/1	0: SQ0 specification 1: SQ1 specification and external memory address bit [5]
[4:2]:	Don’t care	No meaning in a prefetch
[1:0]	00	Fixed at 0

  External memory address bits [28:26], which cannot be generated from the above address, are generated from the QACR0/1 registers.

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

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

  In the SH7750, it is not possible to perform data transfer to a PCMCIA interface area using the SQs.

  In the SH7750S and SH7751, data transfer to a PCMCIA interface area is always performed using the values of the SA bit and TC bit in PTEA.

4.6.4 SQ Protection

It is possible to set protection against SQ writes and transfers to external memory. If an SQ write violates the protection setting, an exception will be generated but the SQ contents will be corrupted. If a transfer from the SQs to external memory (prefetch instruction) violates the protection setting, the transfer to external memory will be inhibited and an exception will be generated.

- When MMU is on

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

- When MMU is off

  Operation is in accordance with MMUCR.SQMD.

  0: Privileged/user access possible

  1: Privileged access possible

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

Section 5 Exceptions

5.1 Overview

5.1.1 Features

Exception handling is processing handled by a special routine, separate from normal program processing, that is executed by the CPU in case of abnormal events. 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.

SH-4 exception handling is of three kinds: for resets, general exceptions, and interrupts.

5.1.2 Register Configuration

The registers used in exception handling are shown in table 5.1.

Table 5.1 Exception-Related Registers

Name	Abbreviation	R/W	Initial Value*[1]	P4 Address*[2]	Area 7 Address*[2]	Access Size
TRAPA exception register	TRA	R/W	Undefined	H'FF00 0020	H'1F00 0020	32
Exception event register	EXPEVT	R/W	H'0000 0000/ H'0000 0020*[1]	H'FF00 0024	H'1F00 0024	32
Interrupt event register	INTEVT	R/W	Undefined	H'FF00 0028	H'1F00 0028	32

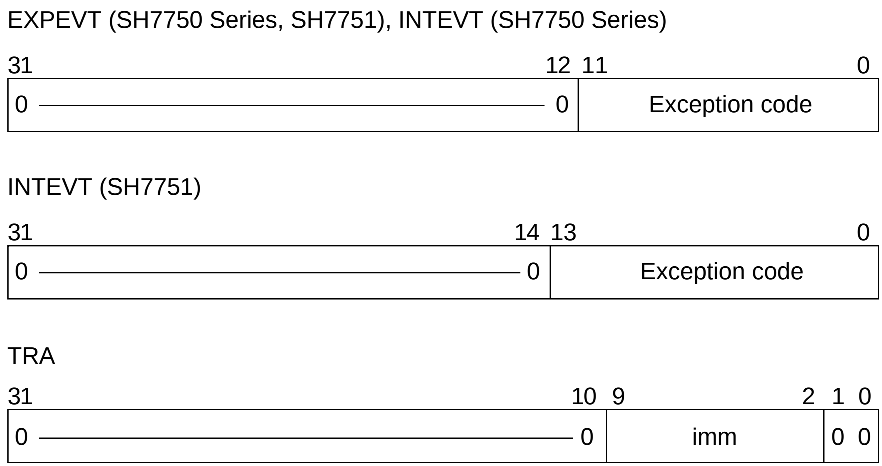
Notes: 1. H'0000 0000 is set in a power-on reset, and H'0000 0020 in a manual reset.
2. This is the address when using the virtual/physical address space P4 area. When making an access from physical address space area 7 using the TLB, the upper 3 bits of the address are ignored.

5.2 Register Descriptions

There are three registers related to exception handling. These are allocated to memory, and can be accessed by specifying the P4 address or area 7 address.

1. The exception event register (EXPEVT) resides at P4 address H'FF00 0024, and contains 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.
2. The interrupt event register (INTEVT) resides at P4 address H'FF00 0028, and contains a 12-bit (SH7750 Series) or 14-bit (SH7751) exception code. The exception code set in INTEVT is that for an interrupt request. The exception code is set automatically by hardware when an exception occurs. INTEVT can also be modified by software.
3. The TRAPA exception register (TRA) resides at P4 address H'FF00 0020, and contains 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.

The bit configurations of EXPEVT, INTEVT, and TRA are shown in figure 5.1.



0: Reserved bits. These bits are always read as 0, and should only be written with 0.
imm: 8-bit immediate data of the TRAPA instruction

Figure 5.1 Register Bit Configurations

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 by an RTE instruction.

The basic processing flow is as follows. See section 2, Data Formats and Registers, for the meaning of the individual SR bits.

1. The PC, SR, and R15 contents are saved in SPC, SSR, and SGR.
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 11–0 of the exception event register (EXPEVT): SH7750 Series, bits 13–0 of the exception event register (EXPEVT): SH7751 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'A000 0000. 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'0000 0400, so if H'9C08 0000 is set in VBR, the exception handling vector address will be H'9C08 0400. If a further exception occurs at the exception handling vector address, a duplicate exception will result, and recovery will be difficult; therefore, fixed physical addresses (P1, P2) should be specified for vector addresses.

5.4 Exception Types and Priorities

Table 5.2 shows the types of exceptions, with their relative priorities, vector addresses, and exception/interrupt codes.

Table 5.2 Exceptions

Exception Category	Execution Mode	Exception	Priority Level	Priority Order	Vector Address	Offset	Exception Code
Reset	Abort type	Power-on reset	1	1	H'A000 0000	—	H'000
		Manual reset	1	2	H'A000 0000	—	H'020
		Hitachi-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*[1]	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*[1]	2	10	(VBR/DBR)	H'100/—	H'1E0

Table 5.2 Exceptions (cont)

Exception Category	Execution Mode	Exception			Priority Level	Priority Order	Vector Address	Offset	Exception Code
Interrupt	Completion type	Nonmaskable interrupt			3	—	(VBR)	H'600	H'1C0
		External interrupts	IRL3–IRL0	0	4	*2	(VBR)	H'600	H'200
				1					H'220
				2					H'240
				3					H'260
				4					H'280
				5					H'2A0
				6					H'2C0
				7					H'2E0
				8					H'300
				9					H'320
				A					H'340
				B					H'360
				C					H'380
				D					H'3A0
				E					H'3C0
		Peripheral module interrupt (module/ source)	TMU0	TUNI0	4	*2	(VBR)	H'600	H'400
			TMU1	TUNI1					H'420
			TMU2	TUNI2					H'440
				TICPI2					H'460
			TMU3*3	TUNI3					H'B00
			TMU4*3	TUNI4					H'B80
			RTC	ATI					H'480
				PRI					H'4A0
				CUI					H'4C0
			SCI	ERI					H'4E0
				RXI					H'500
				TXI					H'520
				TEI					H'540
			WDT	ITI					H'560
			REF	RCMI					H'580
				ROVI					H'5A0
			H-UDI	H-UDI					H'600
			GPIO	GPIOI					H'620

Table 5.2 Exceptions (cont)

Exception Category	Execution Mode	Exception			Priority Level	Priority Order	Vector Address	Offset	Exception Code
Interrupt	Completion type	Peripheral module interrupt (module/ source)	DMAC	DMTE0	4	*2	(VBR)	H'600	H'640
				DMTE1					H'660
				DMTE2					H'680
				DMTE3					H'6A0
				DMAE					H'6C0
			SCIF	ERI					H'700
				RXI					H'720
				BRI					H'740
				TXI					H'760
			PCIC*3	PCISERR					H'A00
				PCIERR					H'AE0
				PCIPWDWN					H'AC0
				PCIPWON					H'AA0
				PCIDMA0					H'A80
				PCIDMA1					H'A60
				PCIDMA2					H'A40
				PCIDMA3					H'A20

Priority: Priority is first assigned by priority level, then by priority order within each level (the lowest number represents the highest priority).

Exception transition destination: Control passes to H'A000 0000 in a reset, and to [VBR + offset] in other cases.

Exception code: Stored in EXPEVT for a reset or general exception, and in INTEVT for an interrupt.

IRL: Interrupt request level (pins IRL3–IRL0).

Module/source: See the sections on the relevant peripheral modules.

Notes: 1. When BRCR.UBDE = 1, PC = DBR. In other cases, PC = VBR + H'100.

2. The priority order of external interrupts and peripheral module interrupts can be set by software.

3. SH7751 exceptions only. Not provided in the SH7750 Series.

5.5 Exception Flow

5.5.1 Exception Flow

Figure 5.2 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 5.2 shows the relative priority order of the different kinds of exceptions (reset/general exception/interrupt). Register settings in the event of an exception are shown only for SSR, SPC, SGR, EXPEVT/INTEVT, SR, and PC, but 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, and in the case of instructions in which two data accesses are performed.

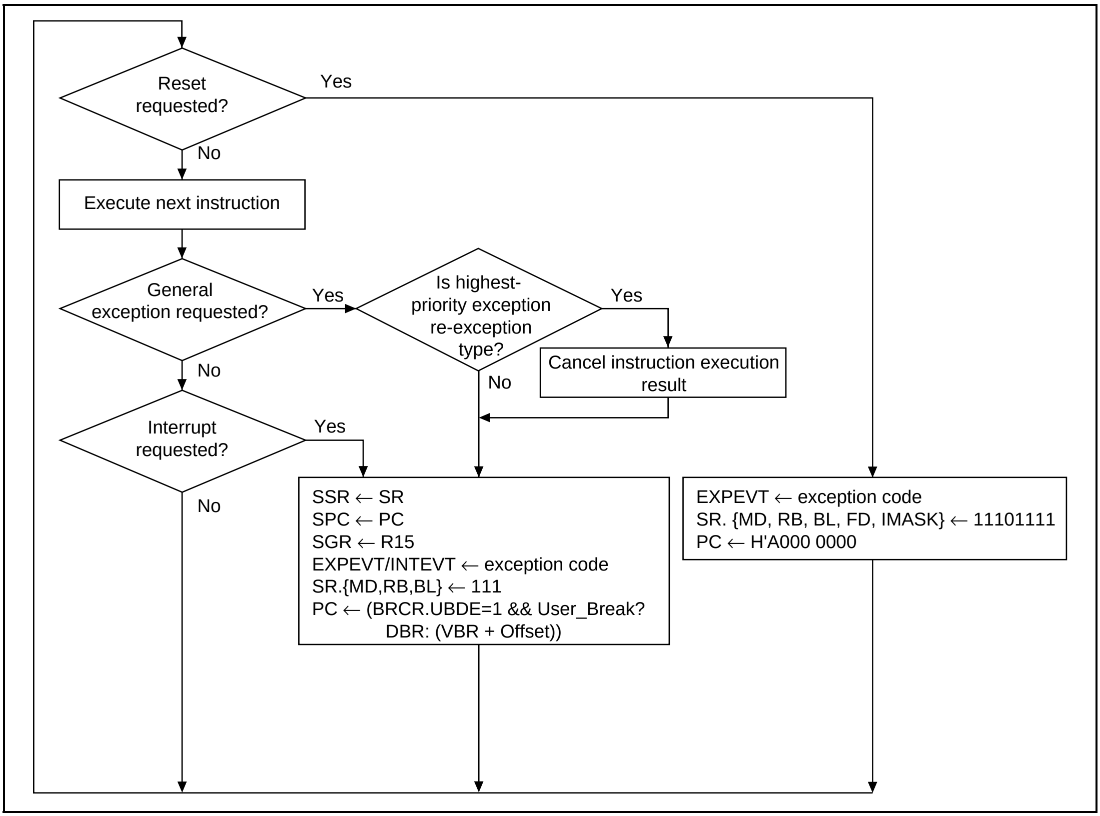


Figure 5.2 Instruction Execution and Exception Handling

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—the 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 5.3.

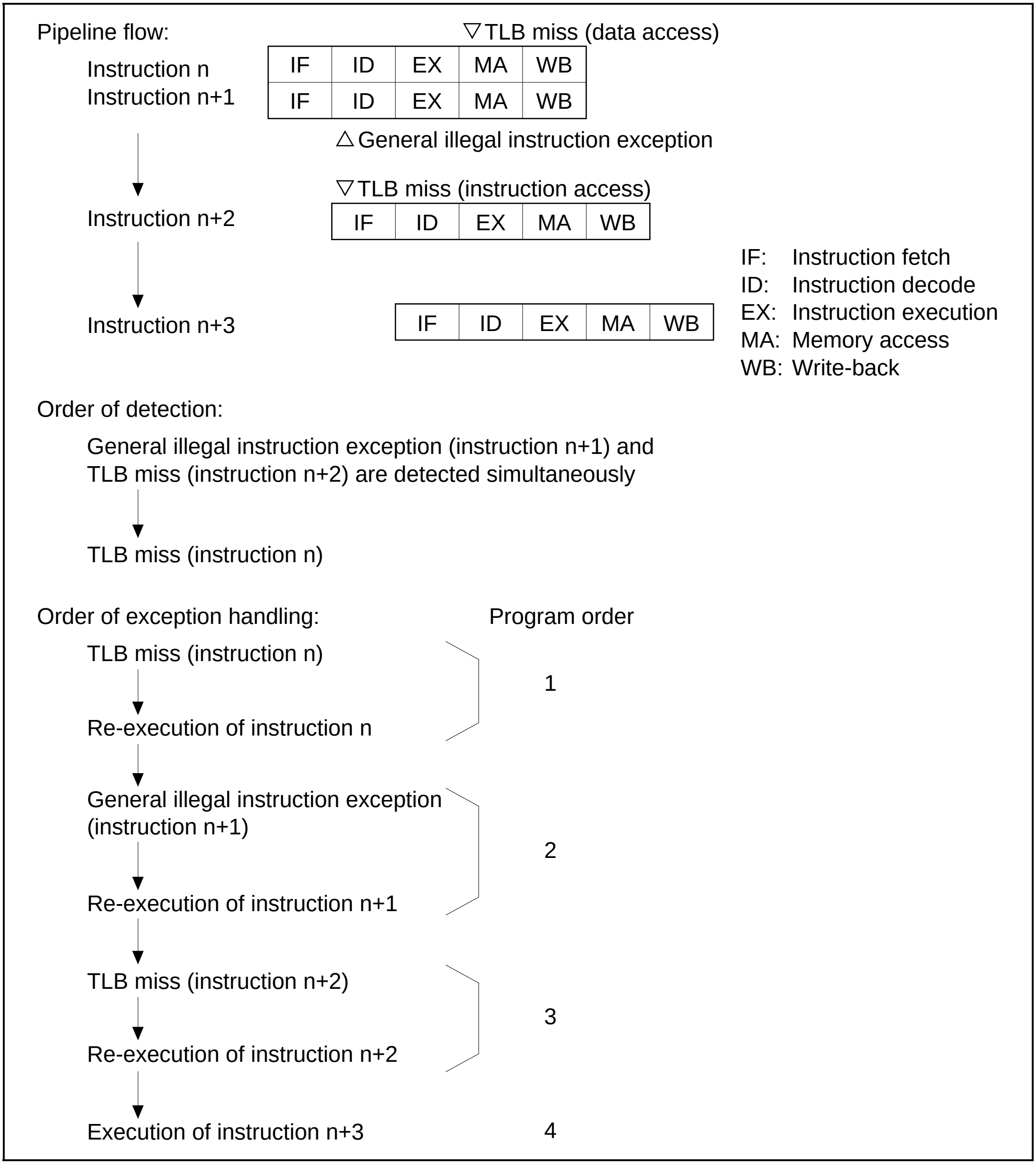


Figure 5.3 Example of General Exception Acceptance Order

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'A000 0000). For the operation in the event of a user break, see User Break Controller in the hardware manual. 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 are described here, covering exception sources, transition addresses, and processor operation when a transition is made.

5.6.1 Resets

(1) Power-On Reset

- Sources:
  - $\overline{\text{SCK2}}$ pin high level and $\overline{\text{RESET}}$ pin low level (SH7750 Series)/$\overline{\text{RESET}}$ pin low level (SH7751)
  - When the watchdog timer overflows while the WT/$\overline{\text{IT}}$ bit is set to 1 and the RSTS bit is cleared to 0 in WTCSR. For details, see Clock Oscillation Circuits in hardware manual.
- Transition address: H'A000 0000
- 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'A000 0000.

In the initialization processing, the VBR register is set to H'0000 0000, and in SR, the MD, RB, and BL bits are set to 1, the FD bit is cleared to 0, and the interrupt mask bits (I3–I0) are set to B’1111.

CPU and on-chip peripheral module initialization is performed. For details, see the register descriptions in the relevant sections. For some CPU functions, the $\overline{\text{TRST}}$ pin and $\overline{\text{RESET}}$ pin must be driven low. It is therefore essential to execute a power-on reset and drive the $\overline{\text{TRST}}$ pin low when powering on.

If the SCK2 pin is changed to the low level while the $\overline{\text{RESET}}$ pin is low, a manual reset may occur after the power-on reset operation. Do not drive the SCK2 pin low during this interval (see Electrical Characteristics in the hardware manual).

In the SH7750 Series, if the $\overline{\text{SCK2}}$ pin is changed to the low level while the $\overline{\text{RESET}}$ pin is low, a manual reset may occur after the power-on reset operation. Do not drive the $\overline{\text{SCK2}}$ pin low during this interval. For details, see Electrical Characteristics in the hardware manual.

In the SH7751, if the $\overline{\text{RESET}}$ pin is driven high before the $\overline{\text{MRESET}}$ pin while both these pins are low, a manual reset may occur after the power-on reset operation. The $\overline{\text{RESET}}$ pin must be driven high at the same time as, or after, the $\overline{\text{MRESET}}$ pin.

```
Power_on_reset()
{
    EXPEVT = H'00000000;
    VBR = H'00000000;
    SR.MD = 1;
    SR.RB = 1;
    SR.BL = 1;
    SR.(I0-I3) = B'1111;
    SR.FD=0;
    Initialize_CPU();
    Initialize_Module(PowerOn);
    PC = H'A0000000;
}
```

(2) Manual Reset

- Sources:
  - $\overline{\text{SCK2}}$ pin low level and $\overline{\text{RESET}}$ pin low level (SH7750 Series)/$\overline{\text{MRESET}}$ pin low level and $\overline{\text{RESET}}$ pin high level (SH7751)
  - When a general exception other than a user break occurs while the BL bit is set to 1 in SR
  - When the watchdog timer overflows while the WT/$\overline{\text{IT}}$ bit is set to 1 and the RSTS bit is set to 1 in WTCSR. For details, see Clock Oscillation Circuits in the hardware manual.
- Transition address: H'A000 0000
- Transition operations:

  Exception code H'020 is set in EXPEVT, initialization of VBR and SR is performed, and a branch is made to PC = H'A000 0000.

  In the initialization processing, the VBR register is set to H'0000 0000, and in SR, the MD, RB, and BL bits are set to 1, the FD bit is cleared to 0, and the interrupt mask bits (I3–I0) are set to B'1111.

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

```
Manual_reset()
{
    EXPEVT = H'00000020;
    VBR = H'00000000;
    SR.MD = 1;
    SR.RB = 1;
    SR.BL = 1;


    SR.(I0-I3) = B'1111;
    SR.FD = 0;
    Initialize_CPU();
    Initialize_Module(Manual);
    PC = H'A0000000;
}
```

Table 5.3 Types of Reset (SH7750 Series)

Type	Reset State Transition Conditions		Internal States	
	$\overline{SCK2}$	$\overline{RESET}$	CPU	On-Chip Peripheral Modules
Power-on reset	High	Low	Initialized	See Register Configuration in individual sections of the hardware manual
Manual reset	Low	Low	Initialized	

Table 5.4 Types of Reset (SH7751)

Type	Reset State Transition Conditions		Internal States	
	$\overline{MRESET}$	$\overline{RESET}$	CPU	On-Chip Peripheral Modules
Power-on reset	—	Low	Initialized	See Register Configuration in individual sections of the hardware manual
Manual reset	Low	High	Initialized	

(3) H-UDI Reset

- Source: SDIR.TI3–TI0 = B'0110 (negation) or B'0111 (assertion)
- Transition address: H'A000 0000
- 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'A000 0000.

  In the initialization processing, the VBR register is set to H'0000 0000, and in SR, the MD, RB, and BL bits are set to 1, the FD bit is cleared to 0, and the interrupt mask bits (I3–I0) are set to B'1111.

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

```
H-UDI_reset()
{
    EXPEVT = H'00000000;
    VBR = H'00000000;
    SR.MD = 1;
    SR.RB = 1;
    SR.BL = 1;
    SR.(I0-I3) = B'1111;
    SR.FD = 0;
    Initialize_CPU();
    Initialize_Module(PowerOn);
    PC = H'A0000000;
}
```

(4) Instruction TLB Multiple-Hit Exception

- Source: Multiple ITLB address matches
- Transition address: H'A000 0000
- 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'A000 0000.

  In the initialization processing, the VBR register is set to H'0000 0000, and in SR, the MD, RB, and BL bits are set to 1, the FD bit is cleared to 0, and the interrupt mask bits (I3–I0) are set to B'1111.

  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.

```
TLB_multi_hit()
{
    TEA = EXCEPTION_ADDRESS;
    PTEH.VPN = PAGE_NUMBER;
    EXPEVT = H'00000140;
    VBR = H'00000000;
    SR.MD = 1;
    SR.RB = 1;
    SR.BL = 1;
    SR.(I0-I3) = B'1111;
    SR.FD = 0;
    Initialize_CPU();
    Initialize_Module(Manual);
    PC = H'A0000000;
}
```

(5) Operand TLB Multiple-Hit Exception

- Source: Multiple UTLB address matches
- Transition address: H'A000 0000
- 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'A000 0000.

  In the initialization processing, the VBR register is set to H'0000 0000, and in SR, the MD, RB, and BL bits are set to 1, the FD bit is cleared to 0, and the interrupt mask bits (I3–I0) are set to B'1111.

  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.

```
TLB_multi_hit()
{
    TEA = EXCEPTION_ADDRESS;
    PTEH.VPN = PAGE_NUMBER;
    EXPEVT = H'00000140;
    VBR = H'00000000;
    SR.MD = 1;
    SR.RB = 1;
    SR.BL = 1;
    SR.(I0-I3) = B'1111;
    SR.FD = 0;
    Initialize_CPU();
    Initialize_Module(Manual);
    PC = H'A0000000;
}
```

5.6.2 General Exceptions

(1) Data TLB Miss Exception

- Source: Address mismatch in UTLB address comparison
- Transition address: VBR + H'0000 0400
- 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, and the contents of R15 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'00000040 : H'00000060;
    SR.MD = 1;
    SR.RB = 1;
    SR.BL = 1;
    PC = VBR + H'00000400;
}
```

(2) Instruction TLB Miss Exception

- Source: Address mismatch in ITLB address comparison
- Transition address: VBR + H'0000 0400
- 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, and the contents of R15 are saved in SGR.

  Exception code H'040 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'00000040;
    SR.MD = 1;
    SR.RB = 1;
    SR.BL = 1;
    PC = VBR + H'00000400;
}
```

(3) Initial Page Write Exception

- Source: TLB is hit in a store access, but dirty bit D = 0
- Transition address: VBR + H'0000 0100
- 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, and the contents of R15 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'00000080;
    SR.MD = 1;
    SR.RB = 1;
    SR.BL = 1;
    PC = VBR + H'00000100;
}
```

(4) 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'0000 0100
- 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, and the contents of R15 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'000000A0 : H'000000C0;
    SR.MD = 1;
    SR.RB = 1;
    SR.BL = 1;
    PC = VBR + H'00000100;
}
```

(5) 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'0000 0100
- 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, and the contents of R15 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'000000A0;
   SR.MD = 1;
   SR.RB = 1;
   SR.BL = 1;
   PC = VBR + H'00000100;
}
```

(6) 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'8000 0000–H'FFFF FFFF in user mode
- Transition address: VBR + H'0000 0100
- 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, and the contents of R15 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 3, Memory Management Unit (MMU).

```
Data_address_error()
{
    TEA = EXCEPTION_ADDRESS;
    PTEN.VPN = PAGE_NUMBER;
    SPC = PC;
    SSR = SR;
    SGR = R15;
    EXPEVT = read_access? H'000000E0: H'00000100;
    SR.MD = 1;
    SR.RB = 1;
    SR.BL = 1;
    PC = VBR + H'00000100;
}
```

(7) Instruction Address Error

- Sources:
  - Instruction fetch from other than a word boundary (2n +1)
  - Instruction fetch from area H'8000 0000–H'FFFF FFFF in user mode
- Transition address: VBR + H'0000 0100
- 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, and the contents of R15 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 3, Memory Management Unit (MMU).

```
Instruction_address_error()
{
   TEA = EXCEPTION_ADDRESS;
   PTEN.VPN = PAGE_NUMBER;
   SPC = PC;
   SSR = SR;
   SGR = R15;
   EXPEVT = H'000000E0;
   SR.MD = 1;
   SR.RB = 1;
   SR.BL = 1;
   PC = VBR + H'00000100;
}
```

(8) Unconditional Trap

- Source: Execution of TRAPA instruction
- Transition address: VBR + H'0000 0100
- 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 values 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'00000160;
    SR.MD = 1;
    SR.RB = 1;
    SR.BL = 1;
    PC = VBR + H'00000100;
}
```

(9) 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'0000 0100
- Transition operations:

  The PC and SR contents for the instruction at which this exception occurred are saved in SPC and SSR, and the contents of R15 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'00000180;
   SR.MD = 1;
   SR.RB = 1;
   SR.BL = 1;
   PC = VBR + H'00000100;
}
```

(10) 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
  - 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'0000 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'000001A0;
    SR.MD = 1;
    SR.RB = 1;
    SR.BL = 1;
    PC = VBR + H'00000100;
}
```

(11) General FPU Disable Exception

- Source: Decoding of an FPU instruction* not in a delay slot with SR.FD =1
- Transition address: VBR + H'0000 0100
- Transition operations:

  The PC and SR contents for the instruction at which this exception occurred are saved in SPC and SSR, and the contents of R15 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'00000800;
    SR.MD = 1;
    SR.RB = 1;
    SR.BL = 1;
    PC = VBR + H'00000100;
}
```

(12) Slot FPU Disable Exception

- Source: Decoding of an FPU instruction in a delay slot with SR.FD =1
- Transition address: VBR + H'0000 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'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'00000820;
    SR.MD = 1;
    SR.RB = 1;
    SR.BL = 1;
    PC = VBR + H'00000100;
}
```

(13) User Breakpoint Trap

- Source: Fulfilling of a break condition set in the user break controller
- Transition address: VBR + H'0000 0100, 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 User Break Controller in the hardware manual.

```
User_break_exception()
{
   SPC = (pre_execution break? PC : PC + 2);
   SSR = SR;
   SGR = R15;
   EXPEVT = H'000001E0;
   SR.MD = 1;
   SR.RB = 1;
   SR.BL = 1;
   PC = (BRCR.UBDE==1 ? DBR : VBR + H’00000100);
}
```

(14) FPU Exception

- Source: Exception due to execution of a floating-point operation
- Transition address: VBR + H'0000 0100
- Transition operations:

  The PC and SR contents for the instruction at which this exception occurred are saved in SPC and SSR, and the contents of R15 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'00000120;
    SR.MD = 1;
    SR.RB = 1;
    SR.BL = 1;
    PC = VBR + H'00000100;
}
```

5.6.3 Interrupts

(1) NMI

- Source: NMI pin edge detection
- Transition address: VBR + H'0000 0600
- Transition operations:

  The contents of PC and SR immediately after the instruction at which this interrupt was accepted are saved in SPC and SSR, and the contents of R15 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 Interrupt Controller in the hardware manual.

```
NMI()
{
    SPC = PC;
    SSR = SR;
    SGR = R15;
    INTEVT = H'000001C0;
    SR.MD = 1;
    SR.RB = 1;
    SR.BL = 1;
    PC = VBR + H'00000600;
}
```

(2) IRL Interrupts

- Source: The interrupt mask bit setting in SR is smaller than the IRL (3–0) level, and the BL bit in SR is 0 (accepted at instruction boundary).
- Transition address: VBR + H'0000 0600
- 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 IRL (3–0) level is set in INTEVT. See table 19.5, Interrupt Exception Handling Sources and Priority Order, for the corresponding codes. The BL, MD, and RB bits are set to 1 in SR, and a branch is made to VBR + H'0600. The acceptance level is not set in the interrupt mask bits in SR. When the BL bit in SR is 1, the interrupt is masked. For details, see Interrupt Controller in the hardware manual.

```
IRL()
{
   SPC = PC;
   SSR = SR;
   SGR = R15;
   INTEVT = H'00000200 ~ H'000003C0;
   SR.MD = 1;
   SR.RB = 1;
   SR.BL = 1;
   PC = VBR + H'00000600;
}
```

(3) Peripheral Module Interrupts

- Source: The interrupt mask bit setting in SR is smaller than the peripheral module (H-UDI, GPIO, DMAC, PCIC*, TMU, RTC, SCI, SCIF, WDT, or REF) interrupt level, and the BL bit in SR is 0 (accepted at instruction boundary).

Note: * SH7751 only

- Transition address: VBR + H'0000 0600
- 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 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. The module interrupt levels should be set as values between B’0000 and B’1111 in the interrupt priority registers (IPRA–IPRC) in the interrupt controller. For details, see Interrupt Controller in the hardware manual.

```
Module_interruption()
{
    SPC = PC;
    SSR = SR;
    SGR = R15;
    INTEVT = H'00000400 ~ H'00000760;
    SR.MD = 1;
    SR.RB = 1;
    SR.BL = 1;
    PC = VBR + H'00000600;
}
```

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.

1. Instructions that make two accesses to memory

   With MAC instructions, memory-to-memory arithmetic/logic instructions, and TAS 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.

   a. Data address error in first data transfer
   b. TLB miss in first data transfer
   c. TLB protection violation in first data transfer
   d. Initial page write exception in first data transfer
   e. Data address error in second data transfer
   f. TLB miss in second data transfer
   g. TLB protection violation in second data transfer
   h. Initial page write exception in second data transfer

2. 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.

   a. The delayed branch instruction is checked for priority level 1 and 2 abort type and re-execution type exceptions.
   b. The delay slot instruction is checked for priority level 1 and 2 abort type and re-execution type exceptions.
   c. The delayed branch instruction is checked for a priority level 2 completion type exception.
   d. The delay slot instruction is checked for a priority level 2 completion type exception.
   e. 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.)
   f. 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 b, as in 1 above.

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 → PR operation performed in BSR, BSRF, JSR) is inhibited.

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 external 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 set in PC, the SSR contents are set 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 SR.BL = 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'0000 0020; the value of the SPC and SSR registers is 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 the sleep or standby state, 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 in which the exception occurred is set in SPC, and the instruction is re-executed after returning from exception handling. If an exception occurs in a delay slot instruction, however, the PC value for the delay slot 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 in 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. An exception must not be generated in an RTE instruction delay slot, as the operation will be undefined in this case.

5.8 Restrictions

1. Restrictions on first instruction of exception handling routine

- Do not locate a BT, BF, BT/S, BF/S, BRA, or BSR instruction at address VBR + H'100, VBR + H'400, or VBR + H'600.

- When the UBDE bit in the BRCR register is set to 1 and the user break debug support function* is used, do not locate a BT, BF, BT/S, BF/S, BRA, or BSR instruction at the address indicated by the DBR register.

Note: * See User Break Debug Support Function in the hardware manual.

Section 6 Floating-Point Unit

6.1 Overview

The floating-point unit (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, system control

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.

6.2 Data Formats

6.2.1 Floating-Point Format

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

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

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

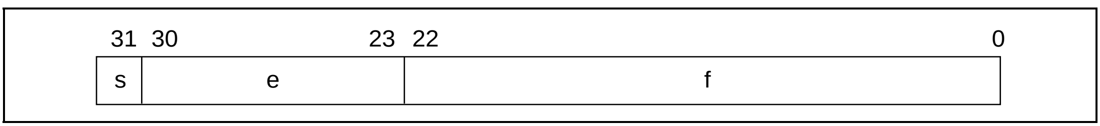


Figure 6.1 Format of Single-Precision Floating-Point Number

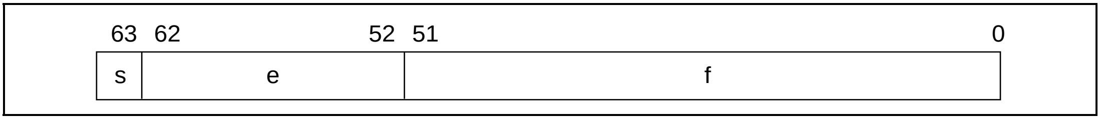


Figure 6.2 Format of Double-Precision Floating-Point Number

The exponent is expressed in biased form, as follows:

$$e = E + bias$$

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 bias, E_{min}, and E_{max} values.

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.

Table 6.2 Floating-Point Ranges

Type	Single-Precision	Double-Precision
Signaling non-number	H'7FFFFFFF to H'7FC00000	H'7FFFFFFF FFFFFFFF to H'7FF80000 00000000
Quiet non-number	H'7FBFFFFF to H'7F800001	H'7FF7FFFF FFFFFFFF to H'7FF00000 00000001
Positive infinity	H'7F800000	H'7FF00000 00000
Positive normalized number	H'7F7FFFFF to H'00800000	H'7FEFFFFF FFFFFFFF to H'00100000 00000000
Positive denormalized number	H'007FFFFF to H'00000001	H'000FFFFF FFFFFFFF to H'00000000 00000001
Positive zero	H'00000000	H'00000000 00000000
Negative zero	H'80000000	H'80000000 00000000
Negative denormalized number	H'80000001 to H'807FFFFF	H'80000000 00000001 to H'800FFFFF FFFFFFFF
Negative normalized number	H'80800000 to H'FF7FFFFF	H'80100000 00000000 to H'FFEFFFFF FFFFFFFF
Negative infinity	H'FF800000	H'FFF00000 00000000
Quiet non-number	H'FF800001 to H'FFBFFFFF	H'FFF00000 00000001 to H'FFF7FFFF FFFFFFFF
Signaling non-number	H'FFC00000 to H'FFFFFFFF	H'FFF80000 00000000 to H'FFFFFFFF FFFFFFFF

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.

31	30 … 23	22 … 0
x	11111111	Nxxxxxxxxxxxxxxxxxxxxxx

N = 1: sNaN
N = 0: qNaN

Figure 6.3 Single-Precision NaN Bit Pattern

An sNAN is input in an operation, except copy, FABS, and FNEG, that generates a floating-point value.

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

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 the FPSCR register. An exception will not be generated in this case.

The qNAN values generated by the SH-4 as operation results are as follows:

- Single-precision qNaN: H'7FBFFFFF
- Double-precision qNaN: H'7FF7FFFF FFFFFFFF

See section 9, 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 the FPU’s status register FPSCR is 1, a denormalized number (source operand or operation result) is always flushed to 0 in a floating-point operation that generates a value (an operation other than copy, 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 9, Description of Instructions, for details of floating-point operations when a denormalized number is input.

6.3 Registers

6.3.1 Floating-Point Registers

Figure 6.4 shows the floating-point register configuration. There are thirty-two 32-bit floating-point registers, referenced by specifying FR0–FR15, DR0/2/4/6/8/10/12/14, FV0/4/8/12, XF0–XF15, XD0/2/4/6/8/10/12/14, or XMTRX.

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

   FPR0_BANK0–FPR15_BANK0

   FPR0_BANK1–FPR15_BANK1

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

   When FPSCR.FR = 0, FR0–FR15 indicate FPR0_BANK0–FPR15_BANK0;

   when FPSCR.FR = 1, FR0–FR15 indicate FPR0_BANK1–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–XF15 indicate FPR0_BANK1–FPR15_BANK1;

   when FPSCR.FR = 1, XF0–XF15 indicate FPR0_BANK0–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

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

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)

Floating-point status/control register, FPSCR (32 bits, initial value = H'0004 0001)

31–22	21	20	19	18	17–12	11–7	6–2	1–0
—	FR	SZ	PR	DN	Cause	Enable	Flag	RM

Note: —: Reserved. These bits are always read as 0, and should only be written with 0.

- FR: Floating-point register bank

  FR = 0: FPR0_BANK0–FPR15_BANK0 are assigned to FR0–FR15; FPR0_BANK1–FPR15_BANK1 are assigned to XF0–XF15.

  FR = 1: FPR0_BANK0–FPR15_BANK0 are assigned to XF0–XF15; FPR0_BANK1–FPR15_BANK1 are assigned to FR0–FR15.

- SZ: Transfer size mode

  SZ = 0: The data size of the FMOV instruction is 32 bits.

  SZ = 1: The data size of the FMOV instruction is a 32-bit register pair (64 bits).

- PR: Precision mode

  PR = 0: Floating-point instructions are executed as single-precision operations.

  PR = 1: Floating-point instructions are executed as double-precision operations (graphics support instructions are undefined).

  Do not set SZ and PR to 1 simultaneously; this setting is reserved.

  [SZ, PR = 11]: Reserved (FPU operation instruction is undefined.)

- DN: Denormalization mode

  DN = 0: A denormalized number is treated as such.

  DN = 1: A denormalized number is treated as zero.

- Cause: FPU exception cause field
- Enable: FPU exception enable field
- Flag: FPU exception flag field

		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

When an FPU operation instruction is executed, the FPU exception cause field is cleared to zero first. When the next FPU exception is occured, the corresponding bits in the FPU exception cause field and FPU exception flag field are set to 1. The FPU exception flag field holds the status of the exception generated after the field was last cleared.

- RM: Rounding mode
  RM = 00: Round to Nearest
  RM = 01: Round to Zero
  RM = 10: Reserved
  RM = 11: Reserved

- Bits 22 to 31: Reserved
  These bits are always read as 0, and should only be written with 0.

Notes: The following functions have been added to the FPU of the SH-4 (not provided in the FPU of the SH7718):

1. The FR, SZ, and PR bits have been added.
2. Exception O (overflow), U (underflow), and I (inexact) bits have been added to the cause, enable, and flag fields.
3. An exception E (FPU error) bit has been added to the cause field.

6.3.3 Floating-Point Communication Register (FPUL)

Information is transferred between the FPU and CPU via the FPUL register. The 32-bit FPUL register is a system register, and 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.

There are two rounding methods, the method to be used being determined by the RM field in FPSCR.

- RM = 00: Round to Nearest
- RM = 01: Round to Zero

Round to Nearest: The value 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 be the maximum expressible absolute value.

6.5 Floating-Point Exceptions

FPU-related exceptions are as follows:

- General illegal instruction/slot illegal instruction exception

  The exception occurs if an FPU instruction is executed when SR.FD = 1.

- FPU exceptions

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

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

- FPU exception handling

  FPU exception occurs in the following cases:

  — FPU error (E): FPSCR.DN = 0 and a denormalized number is input

  — Invalid operation (V): FPSCR.EN.V = 1 and (instruction = FTRV or invalid operation)

  — Division by zero (Z): FPSCR.EN.Z = 1 and division with a zero divisor

  — Overflow (O): FPSCR.EN.O = 1 and instruction with possibility of operation result overflow

— Underflow (U): FPSCR.EN.U = 1 and instruction with possibility of operation result underflow
— Inexact exception (I): FPSCR.EN.I = 1 and instruction with possibility of inexact operation result

These possibilities are shown in the individual instruction descriptions. 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 system register FPSCR and interpreting the information it contains. If no bits are set in the cause field of FPSCR when one or more of bits O, U, I, and V (in case of FTRV only) are set in the enable field, this indicates that an actual FPU exception is not generated. Also, the destination register is not changed by any FPU exception handling operation.

Except for the above, 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 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-4 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 future version of SH series, the above error is guaranteed, but the same result as SH-4 is not guaranteed.

FIPR FVm, FVn (m, n: 0, 4, 8, 12): Examples of the use of this instruction are shown below.

- 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 approximate-value computations are performed to enable high-speed computation, the inexact exception (I) bit in the cause field and flag field is always set to 1 when an FIPR instruction is executed. Therefore, if the corresponding bit is set in the enable field, enable exception handling will be executed.

FTRV XMTRX, FVn (n: 0, 4, 8, 12): Examples of the use of this instruction are shown below.

- 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-4 supports 4-dimensional operations.
- Matrix (4×4) $\times$ matrix (4×4):

  This operation requires the execution of four FTRV instructions.

Since approximate-value computations are performed to enable high-speed computation, the inexact exception (I) bit in the cause field and flag field is always set to 1 when an FTRV instruction is executed. Therefore, if the corresponding bit is set in the enable field, FPU exception handling will be executed. For the same reason, 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 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 LDC instruction is used on FPSCR, this instruction expends 4 to 5 cycles in order to maintain the FPU state. With the FRCHG instruction, an FPSCR.FR bit modification can be performed in one cycle.

6.6.2 Pair Single-Precision Data Transfer

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

When FPSCR.SZ = 1, the SH-4 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.

Programming Note

When FPSCR.SZ = 1 and big-endian mode is used, FMOV can be used for a double-precision floating-point load or store. In little-endian mode, a double-precision floating-point load or store requires execution of two 32-bit data size operations with FPSCR.SZ = 0.

7.1 Execution Environment

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

Data sizes and data types: The SH-4's instruction set is implemented with 16-bit fixed-length instructions. The SH-4 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 a double-precision floating-point operation is specified (FPSCR.PR = 1), the result of an operation using quadword access will be undefined. When the SH-4 moves byte-size or word-size data from memory to a register, the data is sign-extended.

Load-Store Architecture: The SH-4 features 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-4's branch instructions and RTE are delayed branches. In a delayed branch, the instruction following the branch is executed before the branch destination instruction. This execution slot following a delayed branch is called a delay slot. For example, the BRA execution sequence is as follows:

Static Sequence		Dynamic Sequence		
BRA	TARGET	BRA	TARGET	
ADD next_2	R1, R0	ADD target_instr	R1, R0	ADD in delay slot is executed before branching to TARGET

Delay Slot: An illegal instruction exception may occur when a specific instruction is executed in a delay slot. See section 5, Exceptions. 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 the status register (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, status register (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.

7.2 Addressing Modes

Addressing modes and effective address calculation methods are shown in table 7.1. When a location in virtual memory space is accessed (MMUCR.AT = 1), the effective address is translated into a physical memory address. If multiple virtual memory space systems are selected (MMUCR.SV = 0), the least significant bit of PTEH is also referenced as the access ASID. See section 3, Memory Management Unit (MMU).

Table 7.1 Addressing Modes and Effective Addresses

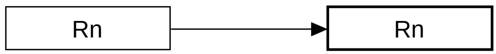
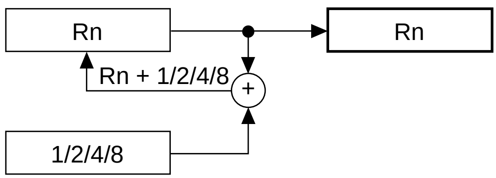
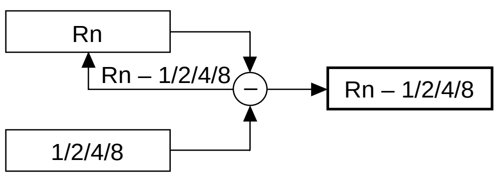
Addressing Mode	Instruction Format	Effective Address Calculation Method	Calculation Formula
Register direct	Rn	Effective address is register Rn. (Operand is register Rn contents.)	—
Register indirect	@Rn	Effective address is register Rn contents. 	Rn → EA (EA: effective address)
Register indirect with 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)

Table 7.1 Addressing Modes and Effective Addresses (cont)

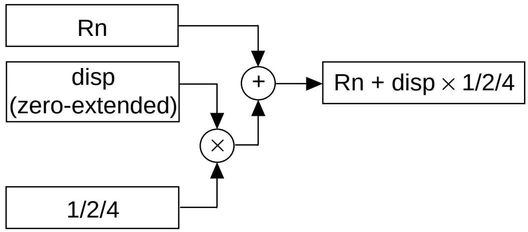
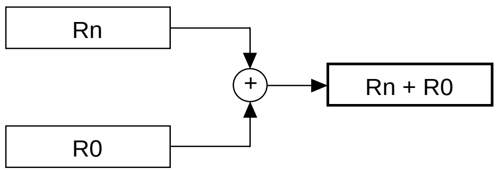
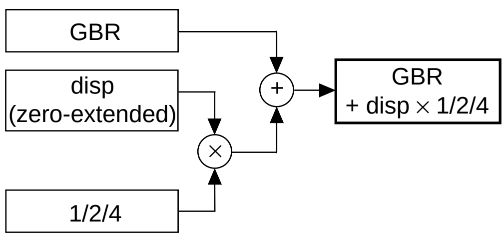
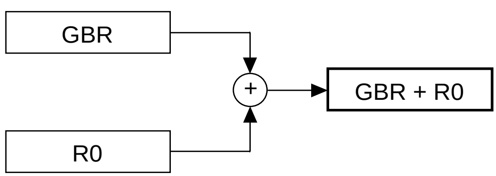
Addressing Mode	Instruction Format	Effective Address Calculation Method	Calculation Formula
Register indirect with displacement	@(disp:4, Rn)	Effective address is register Rn contents with 4-bit displacement disp added. After disp is zero-extended, it is multiplied by 1 (byte), 2 (word), or 4 (longword), according to the operand size. 	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 displacement	@(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. 	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

Table 7.1 Addressing Modes and Effective Addresses (cont)

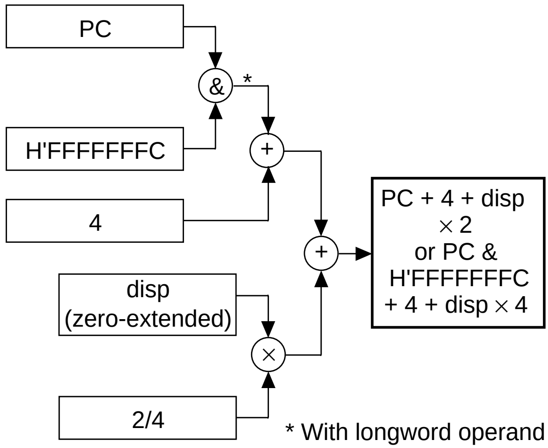
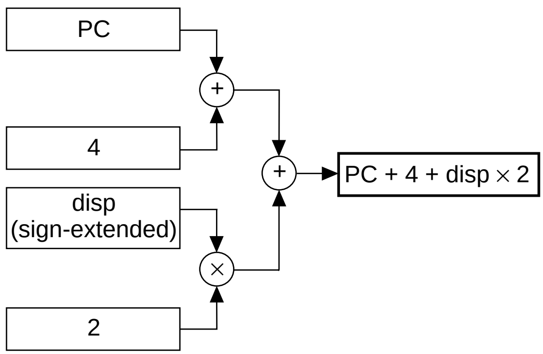
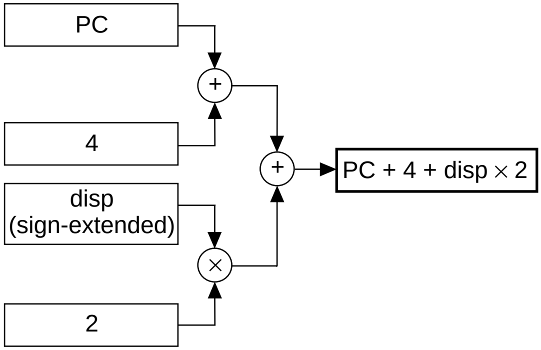
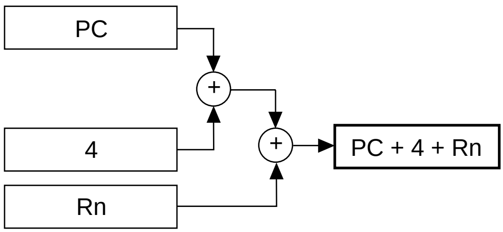
Addressing Mode	Instruction Format	Effective Address Calculation Method	Calculation Formula
PC-relative with displacement	@(disp:8, PC)	Effective address is PC+4 with 8-bit displacement disp added. After disp is zero-extended, it is multiplied by 2 (word), or 4 (longword), according to the operand size.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

Table 7.1 Addressing Modes and Effective Addresses (cont)

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 + 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 chip. 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

7.3 Instruction Set

Table 7.2 shows the notation used in the following SH instruction list.

Table 7.2 Notation Used in Instruction List

<table>
<tr><th>Item</th><th>Format</th><th colspan="2">Description</th></tr>
<tr><td>Instruction mnemonic</td><td>OP.Sz SRC, DEST</td><td>OP:
Sz:
SRC:
DEST:</td><td>Operation code
Size
Source
Source and/or destination operand</td></tr>
<tr><td>Summary of operation</td><td></td><td>→, ←
(xx)
M/Q/T
&
|
^
~
<<n, >>n</td><td>Transfer direction
Memory operand
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-bit shift</td></tr>
<tr><td>Instruction code</td><td>MSB ↔ LSB</td><td>mmmm:
nnnn:
0000:
0001:
:
1111:
mmm:
nnn:
000:
001:
:
111:
mm:
nn:
00:
01:
10:
11:
iiii:
dddd:</td><td>Register number (Rm, FRm)
Register number (Rn, FRn)
R0, FR0
R1, FR1

R15, FR15
Register number (DRm, XDm, Rm_BANK)
Register number (DRm, XDm, Rn_BANK)
DR0, XD0, R0_BANK
DR2, XD2, R1_BANK

DR14, XD14, R7_BANK
Register number (FVm)
Register number (FVn)
FV0
FV4
FV8
FV12
Immediate data
Displacement</td></tr>
<tr><td>Privileged mode</td><td></td><td colspan="2">“Privileged” means the instruction can only be executed in privileged mode.</td></tr>
<tr><td>T bit</td><td>Value of T bit after instruction execution</td><td colspan="2">—: No change</td></tr>
</table>

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

Table 7.3 Fixed-Point Transfer Instructions

Instruction		Operation	Instruction Code	Privileged	T Bit
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'FFFFFFFC + 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	—	—

Table 7.3 Fixed-Point Transfer Instructions (cont)

Instruction		Operation	Instruction Code	Privileged	T Bit
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'FFFFFFFC + 4 → R0	11000111dddddddd	—	—
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	—	—

Table 7.4 Arithmetic Operation Instructions

Instruction		Operation	Instruction Code	Privileged	T Bit
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	—	—
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	—	—

Table 7.4 Arithmetic Operation Instructions (cont)

Instruction		Operation	Instruction Code	Privileged	T Bit
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 7.5 Logic Operation Instructions

Instruction		Operation	Instruction Code	Privileged	T Bit
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 7.6 Shift Instructions

Instruction		Operation	Instruction Code	Privileged	T Bit
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 Rn ≥ 0, Rn << Rm → Rn When Rn < 0, Rn >> Rm → [MSB → Rn]	0100nnnnmmmm1100	—	—
SHAL	Rn	T ← Rn ← 0	0100nnnn00100000	—	MSB
SHAR	Rn	MSB → Rn → T	0100nnnn00100001	—	LSB
SHLD	Rm,Rn	When Rn ≥ 0, Rn << Rm → Rn When Rn < 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 7.7 Branch Instructions

Instruction		Operation	Instruction Code	Privileged	T Bit
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	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 7.8 System Control Instructions

Instruction		Operation	Instruction Code	Privileged	T Bit
CLRMAC		0 → MACH, MACL	0000000000101000	—	—
CLRS		0 → S	0000000001001000	—	—
CLRT		0 → T	0000000000001000	—	0
LDC	Rm,SR	Rm → SR	0100mmmm00001110	Privileged	LSB
LDC	Rm,GBR	Rm → GBR	0100mmmm00011110	—	—
LDC	Rm,VBR	Rm → VBR	0100mmmm00101110	Privileged	—
LDC	Rm,SSR	Rm → SSR	0100mmmm00111110	Privileged	—
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+,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	—	—
RTE		Delayed branch, SSR/SPC → SR/PC	0000000000101011	Privileged	—

Table 7.8 System Control Instructions (cont)

Instruction		Operation	Instruction Code	Privileged	T Bit
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	—	—
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	—	—
TRAPA	#imm	PC + 2 → SPC, SR → SSR, #imm << 2 → TRA, H'160 → EXPEVT, VBR + H'0100 → PC	11000011iiiiiiii	—	—

Table 7.9 Floating-Point Single-Precision Instructions

Instruction		Operation	Instruction Code	Privileged	T Bit
FLDI0	FRn	H'00000000 → FRn	1111nnnn10001101	—	—
FLDI1	FRn	H'3F800000 → 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	—	Comparison result
FCMP/GT	FRm,FRn	When FRn > FRm, 1 → T Otherwise, 0 → T	1111nnnnmmmm0101	—	Comparison 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'80000000 → FRn	1111nnnn01001101	—	—
FSQRT	FRn	√FRn → FRn	1111nnnn01101101	—	—
FSUB	FRm,FRn	FRn – FRm → FRn	1111nnnnmmmm0001	—	—
FTRC	FRm,FPUL	(long) FRm → FPUL	1111mmmm00111101	—	—

Table 7.10 Floating-Point Double-Precision Instructions

Instruction		Operation	Instruction Code	Privileged	T Bit
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 7.11 Floating-Point Control Instructions

Instruction		Operation	Instruction Code	Privileged	T Bit
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 7.12 Floating-Point Graphics Acceleration Instructions

Instruction		Operation	Instruction Code	Privileged	T Bit
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),DRn	(R0 + Rm) → DRn	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	—	—

Section 8 Pipelining

The SH-4 is a 2-ILP (instruction-level-parallelism) superscalar pipelining microprocessor. Instruction execution is pipelined, and two instructions can be executed in parallel. The execution cycles depend on the implementation of a processor. Definitions in this section may not be applicable to SH-4 Series models other than the SH-4.

8.1 Pipelines

Figure 8.1 shows the basic pipelines. Normally, a pipeline consists of five or six stages: instruction fetch (I), decode and register read (D), execution (EX/SX/F0/F1/F2/F3), data access (NA/MA), and write-back (S/FS). An instruction is executed as a combination of basic pipelines. Figure 8.2 shows the instruction execution patterns.

1. General Pipeline

I	D	EX	NA	S
• Instruction fetch	• Instruction decode • Issue • Register read • Destination address calculation for PC-relative branch	• Operation	• Non-memory data access	• Write-back

2. General Load/Store Pipeline

I	D	EX	MA	S
• Instruction fetch	• Instruction decode • Issue • Register read	• Address calculation	• Memory data access	• Write-back

3. Special Pipeline

I	D	SX	NA	S
• Instruction fetch	• Instruction decode • Issue • Register read	• Operation	• Non-memory data access	• Write-back

4. Special Load/Store Pipeline

I	D	SX	MA	S
• Instruction fetch	• Instruction decode • Issue • Register read	• Address calculation	• Memory data access	• Write-back

5. Floating-Point Pipeline

I	D	F1	F2	FS
• Instruction fetch	• Instruction decode • Issue • Register read	• Computation 1	• Computation 2	• Computation 3 • Write-back

6. Floating-Point Extended Pipeline

I	D	F0	F1	F2	FS
• Instruction fetch	• Instruction decode • Issue • Register read	• Computation 0	• Computation 1	• Computation 2	• Computation 3 • Write-back

7. FDIV/FSQRT Pipeline

F3
Computation: Takes several cycles

Figure 8.1 Basic Pipelines

1. 1-step operation: 1 issue cycle
EXT[SU].[BW], MOV, MOV#, MOVA, MOVT, SWAP.[BW], XTRCT, ADD*, CMP*, DIV*, DT, NEG*, SUB*, AND, AND#, NOT, OR, OR#, TST, TST#, XOR, XOR#, ROT*, SHA*, SHL*, BF*, BT*, BRA, NOP, CLRS, CLRT, SETS, SETT, LDS to FPUL, STS from FPUL/FPSCR, FLDI0, FLDI1, FMOV, FLDS, FSTS, single-/double-precision FABS/FNEG

I	D	EX	NA	S

2. Load/store: 1 issue cycle
MOV.[BWL]. FMOV*@, LDS.L to FPUL, LDTLB, PREF, STS.L from FPUL/FPSCR

I	D	EX	MA	S

3. GBR-based load/store: 1 issue cycle
MOV.[BWL]@(d,GBR)

I	D	SX	MA	S

4. JMP, RTS, BRAF: 2 issue cycles

I	D	EX	NA	S	
		D	EX	NA	S

5. TST.B: 3 issue cycles

I	D	SX	MA	S		
		D	SX	NA	S	
			D	SX	NA	S

6. AND.B, OR.B, XOR.B: 4 issue cycles

I	D	SX	MA	S			
		D	SX	NA	S		
			D	SX	NA	S	
				D	SX	MA	S

7. TAS.B: 5 issue cycles

I	D	EX	MA	S				
		D	EX	MA	S			
			D	EX	NA	S		
				D	EX	NA	S	
					D	EX	MA	S

8. RTE: 5 issue cycles

I	D	EX	NA	S				
		D	EX	NA	S			
			D	EX	NA	S		
				D	EX	NA	S	
					D	EX	NA	S

9. SLEEP: 4 issue cycles

I	D	EX	NA	S			
		D	EX	NA	S		
			D	EX	NA	S	
				D	EX	NA	S

Figure 8.2 Instruction Execution Patterns

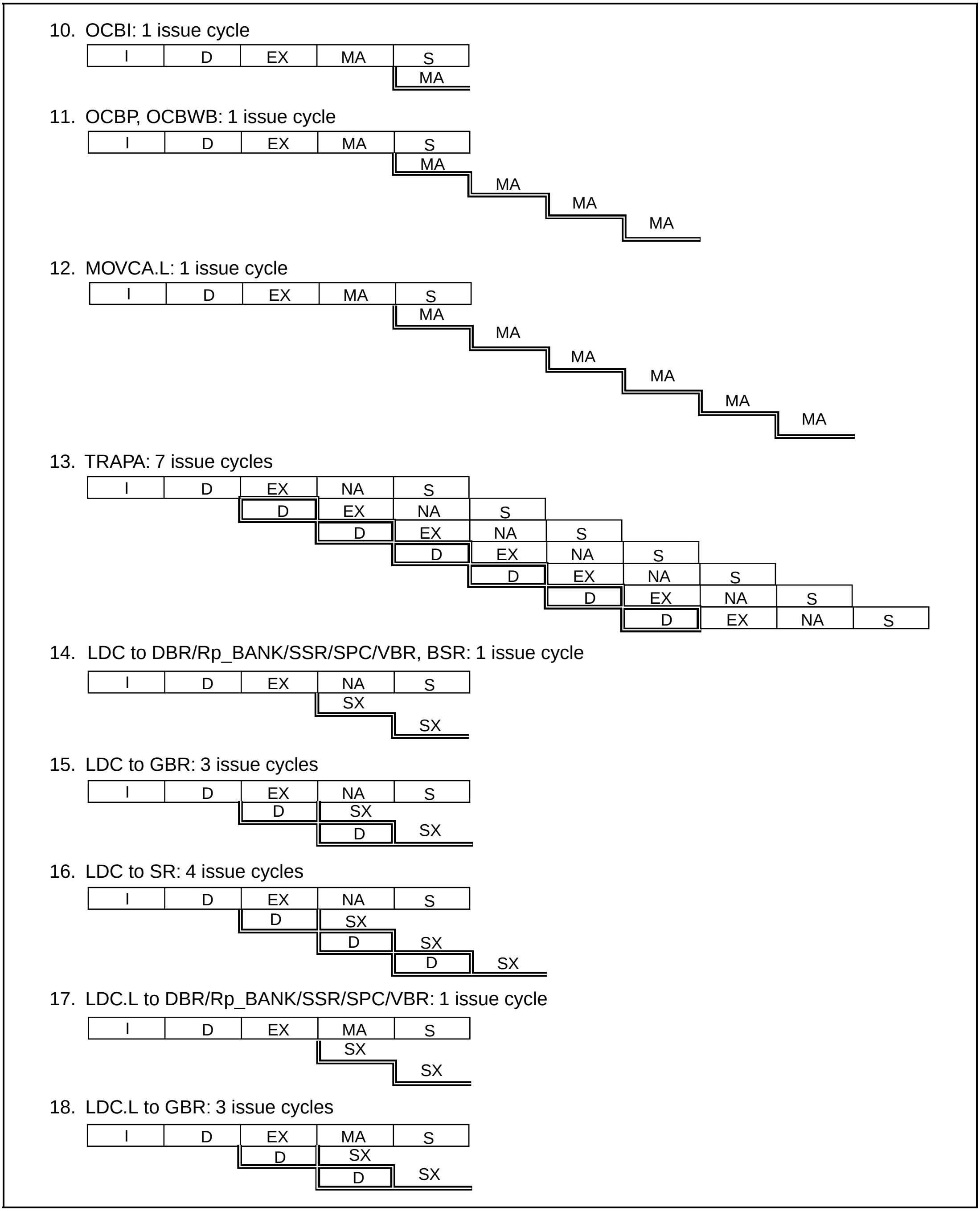


Figure 8.2 Instruction Execution Patterns (cont)

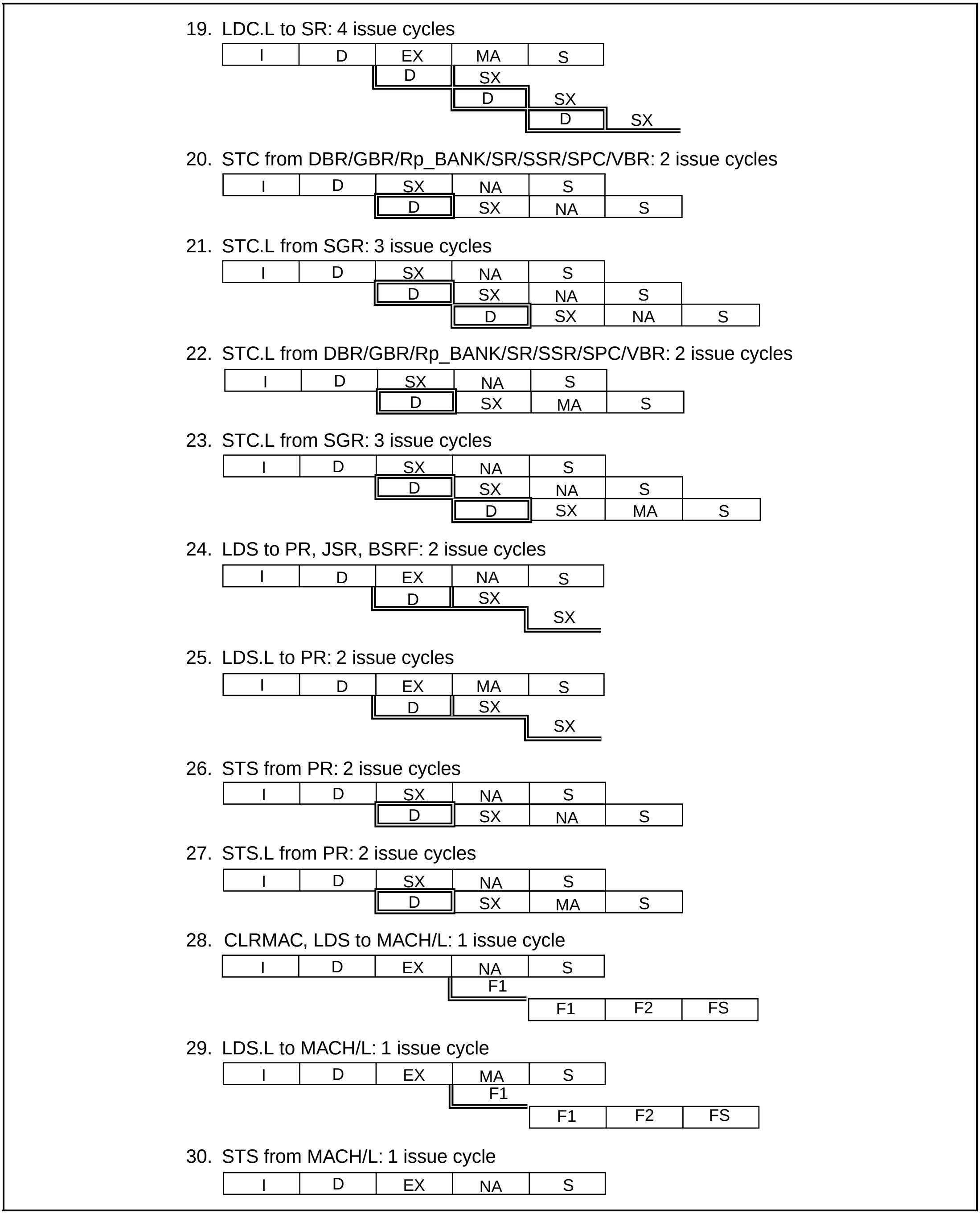


Figure 8.2 Instruction Execution Patterns (cont)

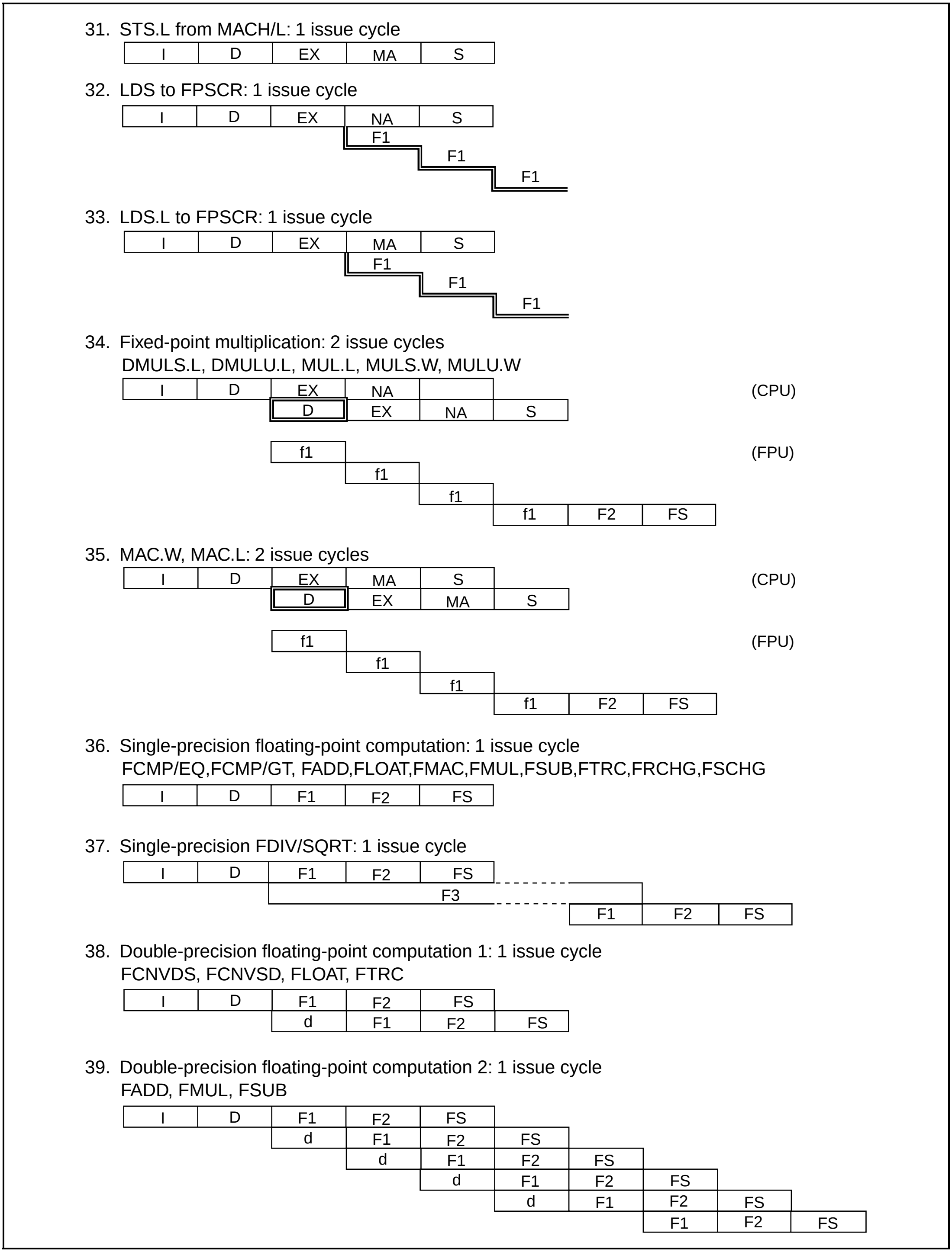
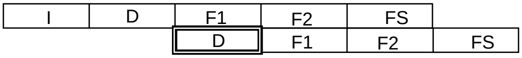
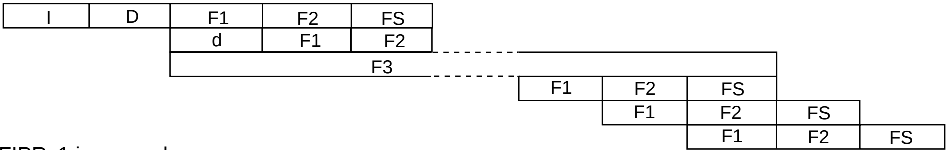


Figure 8.2 Instruction Execution Patterns (cont)

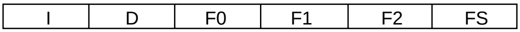
40. Double-precision FCMP: 2 issue cycles
FCMP/EQ,FCMP/GT



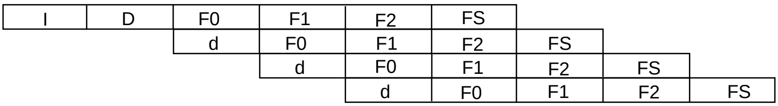
41. Double-precision FDIV/SQRT: 1 issue cycle
FDIV, FSQRT



42. FIPR: 1 issue cycle



43. FTRV: 1 issue cycle



Notes:

??	: Cannot overlap a stage of the same kind, except when two instructions are executed in parallel.
D	: Locks D-stage
d	: Register read only
??	: Locks, but no operation is executed.
f1	: Can overlap another f1, but not another F1.

Figure 8.2 Instruction Execution Patterns (cont)

8.2 Parallel-Executability

Instructions are categorized into six groups according to the internal function blocks used, as shown in table 8.1. Table 8.2 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 8.1 Instruction Groups

1. MT Group

CLRT		CMP/HI	Rm,Rn	MOV	Rm,Rn
CMP/EQ	#imm,R0	CMP/HS	Rm,Rn	NOP	
CMP/EQ	Rm,Rn	CMP/PL	Rn	SETT	
CMP/GE	Rm,Rn	CMP/PZ	Rn	TST	#imm,R0
CMP/GT	Rm,Rn	CMP/STR	Rm,Rn	TST	Rm,Rn

2. EX Group

ADD	#imm,Rn	MOVT	Rn	SHLL2	Rn
ADD	Rm,Rn	NEG	Rm,Rn	SHLL8	Rn
ADDC	Rm,Rn	NEGC	Rm,Rn	SHLR	Rn
ADDV	Rm,Rn	NOT	Rm,Rn	SHLR16	Rn
AND	#imm,R0	OR	#imm,R0	SHLR2	Rn
AND	Rm,Rn	OR	Rm,Rn	SHLR8	Rn
DIV0S	Rm,Rn	ROTCL	Rn	SUB	Rm,Rn
DIV0U		ROTCR	Rn	SUBC	Rm,Rn
DIV1	Rm,Rn	ROTL	Rn	SUBV	Rm,Rn
DT	Rn	ROTR	Rn	SWAP.B	Rm,Rn
EXTS.B	Rm,Rn	SHAD	Rm,Rn	SWAP.W	Rm,Rn
EXTS.W	Rm,Rn	SHAL	Rn	XOR	#imm,R0
EXTU.B	Rm,Rn	SHAR	Rn	XOR	Rm,Rn
EXTU.W	Rm,Rn	SHLD	Rm,Rn	XTRCT	Rm,Rn
MOV	#imm,Rn	SHLL	Rn		
MOVA	@(disp,PC),R0	SHLL16	Rn		

3. BR Group

BF	disp	BRA	disp	BT	disp
BF/S	disp	BSR	disp	BT/S	disp

Table 8.1 Instruction Groups (cont)

4. LS Group

FABS	DRn	FMOV.S	@Rm+,FRn	MOV.L	R0,@(disp,GBR)
FABS	FRn	FMOV.S	FRm,@(R0,Rn)	MOV.L	Rm,@(disp,Rn)
FLDI0	FRn	FMOV.S	FRm,@-Rn	MOV.L	Rm,@(R0,Rn)
FLDI1	FRn	FMOV.S	FRm,@Rn	MOV.L	Rm,@-Rn
FLDS	FRm,FPUL	FNEG	DRn	MOV.L	Rm,@Rn
FMOV	@(R0,Rm),DRn	FNEG	FRn	MOV.W	@(disp,GBR),R0
FMOV	@(R0,Rm),XDn	FSTS	FPUL,FRn	MOV.W	@(disp,PC),Rn
FMOV	@Rm,DRn	LDS	Rm,FPUL	MOV.W	@(disp,Rm),R0
FMOV	@Rm,XDn	MOV.B	@(disp,GBR),R0	MOV.W	@(R0,Rm),Rn
FMOV	@Rm+,DRn	MOV.B	@(disp,Rm),R0	MOV.W	@Rm,Rn
FMOV	@Rm+,XDn	MOV.B	@(R0,Rm),Rn	MOV.W	@Rm+,Rn
FMOV	DRm,@(R0,Rn)	MOV.B	@Rm,Rn	MOV.W	R0,@(disp,GBR)
FMOV	DRm,@-Rn	MOV.B	@Rm+,Rn	MOV.W	R0,@(disp,Rn)
FMOV	DRm,@Rn	MOV.B	R0,@(disp,GBR)	MOV.W	Rm,@(R0,Rn)
FMOV	DRm,DRn	MOV.B	R0,@(disp,Rn)	MOV.W	Rm,@-Rn
FMOV	DRm,XDn	MOV.B	Rm,@(R0,Rn)	MOV.W	Rm,@Rn
FMOV	FRm,FRn	MOV.B	Rm,@-Rn	MOVCA.L	R0,@Rn
FMOV	XDm,@(R0,Rn)	MOV.B	Rm,@Rn	OCBI	@Rn
FMOV	XDm,@-Rn	MOV.L	@(disp,GBR),R0	OCBP	@Rn
FMOV	XDm,@Rn	MOV.L	@(disp,PC),Rn	OCBWB	@Rn
FMOV	XDm,DRn	MOV.L	@(disp,Rm),Rn	PREF	@Rn
FMOV	XDm,XDn	MOV.L	@(R0,Rm),Rn	STS	FPUL,Rn
FMOV.S	@(R0,Rm),FRn	MOV.L	@Rm,Rn		
FMOV.S	@Rm,FRn	MOV.L	@Rm+,Rn		

Table 8.1 Instruction Groups (cont)

5. FE Group

FADD	DRm,DRn	FIPR	FVm,FVn	FSQRT	DRn
FADD	FRm,FRn	FLOAT	FPUL,DRn	FSQRT	FRn
FCMP/EQ	FRm,FRn	FLOAT	FPUL,FRn	FSUB	DRm,DRn
FCMP/GT	FRm,FRn	FMAC	FR0,FRm,FRn	FSUB	FRm,FRn
FCNVDS	DRm,FPUL	FMUL	DRm,DRn	FTRC	DRm,FPUL
FCNVSD	FPUL,DRn	FMUL	FRm,FRn	FTRC	FRm,FPUL
FDIV	DRm,DRn	FRCHG		FTRV	XMTRX,FVn
FDIV	FRm,FRn	FSCHG			

Table 8.1 Instruction Groups (cont)

6. CO Group

AND.B	#imm,@(R0,GBR)	LDS	Rm,FPSCR	STC	SR,Rn
BRAF	Rn	LDS	Rm,MACH	STC	SSR,Rn
BSRF	Rn	LDS	Rm,MACL	STC	VBR,Rn
CLRMAC		LDS	Rm,PR	STC.L	DBR,@-Rn
CLRS		LDS.L	@Rm+,FPSCR	STC.L	GBR,@-Rn
DMULS.L	Rm,Rn	LDS.L	@Rm+,FPUL	STC.L	Rp_BANK,@-Rn
DMULU.L	Rm,Rn	LDS.L	@Rm+,MACH	STC.L	SGR,@-Rn
FCMP/EQ	DRm,DRn	LDS.L	@Rm+,MACL	STC.L	SPC,@-Rn
FCMP/GT	DRm,DRn	LDS.L	@Rm+,PR	STC.L	SR,@-Rn
JMP	@Rn	LDTLB		STC.L	SSR,@-Rn
JSR	@Rn	MAC.L	@Rm+,@Rn+	STC.L	VBR,@-Rn
LDC	Rm,DBR	MAC.W	@Rm+,@Rn+	STS	FPSCR,Rn
LDC	Rm,GBR	MUL.L	Rm,Rn	STS	MACH,Rn
LDC	Rm,Rp_BANK	MULS.W	Rm,Rn	STS	MACL,Rn
LDC	Rm,SPC	MULU.W	Rm,Rn	STS	PR,Rn
LDC	Rm,SR	OR.B	#imm,@(R0,GBR)	STS.L	FPSCR,@-Rn
LDC	Rm,SSR	RTE		STS.L	FPUL,@-Rn
LDC	Rm,VBR	RTS		STS.L	MACH,@-Rn
LDC.L	@Rm+,DBR	SETS		STS.L	MACL,@-Rn
LDC.L	@Rm+,GBR	SLEEP		STS.L	PR,@-Rn
LDC.L	@Rm+,Rp_BANK	STC	DBR,Rn	TAS.B	@Rn
LDC.L	@Rm+,SPC	STC	GBR,Rn	TRAPA	#imm
LDC.L	@Rm+,SR	STC	Rp_BANK,Rn	TST.B	#imm,@(R0,GBR)
LDC.L	@Rm+,SSR	STC	SGR,Rn	XOR.B	#imm,@(R0,GBR)
LDC.L	@Rm+,VBR	STC	SPC,Rn		

Table 8.2 Parallel-Executability

		2nd Instruction					
		MT	EX	BR	LS	FE	CO
1st Instruction	MT	O	O	O	O	O	X
	EX	O	X	O	O	O	X
	BR	O	O	X	O	O	X
	LS	O	O	O	X	O	X
	FE	O	O	O	O	X	X
	CO	X	X	X	X	X	X

O: Can be executed in parallel

X: Cannot be executed in parallel

8.3 Execution Cycles and Pipeline Stalling

There are three basic clocks in this processor: the I-clock, B-clock, and P-clock. Each hardware unit operates on one of these clocks, as follows:

- I-clock: CPU, FPU, MMU, caches
- B-clock: External bus controller
- P-clock: Peripheral units

The frequency ratios of the three clocks are determined with the frequency control register (FRQCR). In this section, machine cycles are based on the I-clock unless otherwise specified. For details of FRQCR, see Clock Oscillation Circuits in the hardware manual.

Instruction execution cycles are summarized in table 8.3. Penalty cycles due to a pipeline stall or freeze are not considered in this table.

- Issue rate: Interval between the issue of an instruction and that of the next instruction
- Latency: Interval between the issue of an instruction and the generation of its result (completion)
- Instruction execution pattern (see figure 8.2)
- Locked pipeline stages
- Interval between the issue of an instruction and the start of locking
- Lock time: Period of locking in machine cycle units

The instruction execution sequence is expressed as a combination of the execution patterns shown in figure 8.2. One instruction is separated from the next by the number of machine cycles for its issue rate. Normally, execution, data access, and write-back stages cannot be overlapped onto the same stages of another instruction; the only exception is when two instructions are executed in parallel under parallel-executability conditions. Refer to (a) through (d) in figure 8.3 for some simple examples.

Latency is the interval between issue and completion of an instruction, and is also the interval between the execution of two instructions with an interdependent relationship. When there is interdependency between two instructions fetched simultaneously, the latter of the two is stalled for the following number of cycles:

- (Latency) cycles when there is flow dependency (read-after-write)
- (Latency – 1) or (latency – 2) cycles when there is output dependency (write-after-write)
  - Single/double-precision FDIV, FSQRT is the preceding instruction (latency – 1) cycles
  - The other FE group is the preceding instruction (latency – 2) cycles
- 5 or 2 cycles when there is anti-flow dependency (write-after-read), as in the following cases:
  - FTRV is the preceding instruction (5 cycle)
  - A double-precision FADD, FSUB, or FMUL is the preceding instruction (2 cycles)

In the case of flow dependency, latency may be exceptionally increased or decreased, depending on the combination of sequential instructions (figure 8.3 (e)).

- When a floating-point (FPU) computation is followed by an FPU register store, the latency of the floating-point computation may be decreased by 1 cycle.
- If there is a load of the shift amount immediately before an SHAD/SHLD instruction, the latency of the load is increased by 1 cycle.
- If an instruction with a latency of less than 2 cycles, including write-back to an FPU register, is followed by a double-precision FPU instruction, FIPR, or FTRV, the latency of the first instruction is increased to 2 cycles.

The number of cycles in a pipeline stall due to flow dependency will vary depending on the combination of interdependent instructions or the fetch timing (see figure 8.3. (e)).

Output dependency occurs when the destination operands are the same in a preceding FE group instruction and a following LS group instruction.

For the stall cycles of an instruction with output dependency, the longest latency to the last write-back among all the destination operands must be applied instead of “latency” (see figure 8.3 (f)). A stall due to output dependency with respect to FPSCR, which reflects the result of a floating-point operation, never occurs. For example, when FADD follows FDIV with no dependency between FPU registers, FADD is not stalled even if both instructions update the cause field of FPSCR.

Anti-flow dependency can occur only between a preceding double-precision FADD, FMUL, FSUB, or FTRV and a following FMOV, FLDI0, FLDI1, FABS, FNEG, or FSTS. See figure 8.3 (g).

If an executing instruction locks any resource—i.e. a function block that performs a basic operation—a following instruction that attempts to use the locked resource must be stalled (figure 8.3 (h)). This kind of stall can be compensated by inserting one or more instructions independent of the locked resource to separate the interfering instructions. For example, when a load instruction and an ADD instruction that references the loaded value are consecutive, the 2-cycle stall of the ADD is eliminated by inserting three instructions without dependency. Software performance can be improved by such instruction scheduling.

Other penalties arise in the event of exceptions or external data accesses, as follows.

- Instruction TLB miss
- Instruction access to external memory (instruction cache miss, etc.)
- Data access to external memory (operand cache miss, etc.)
- Data access to a memory-mapped control register

During the penalty cycles of an instruction TLB miss or external instruction access, no instruction is issued, but execution of instructions that have already been issued continues. The penalty for a data access is a pipeline freeze: that is, the execution of uncompleted instructions is interrupted until the arrival of the requested data. The number of penalty cycles for instruction and data accesses is largely dependent on the user’s memory subsystems.

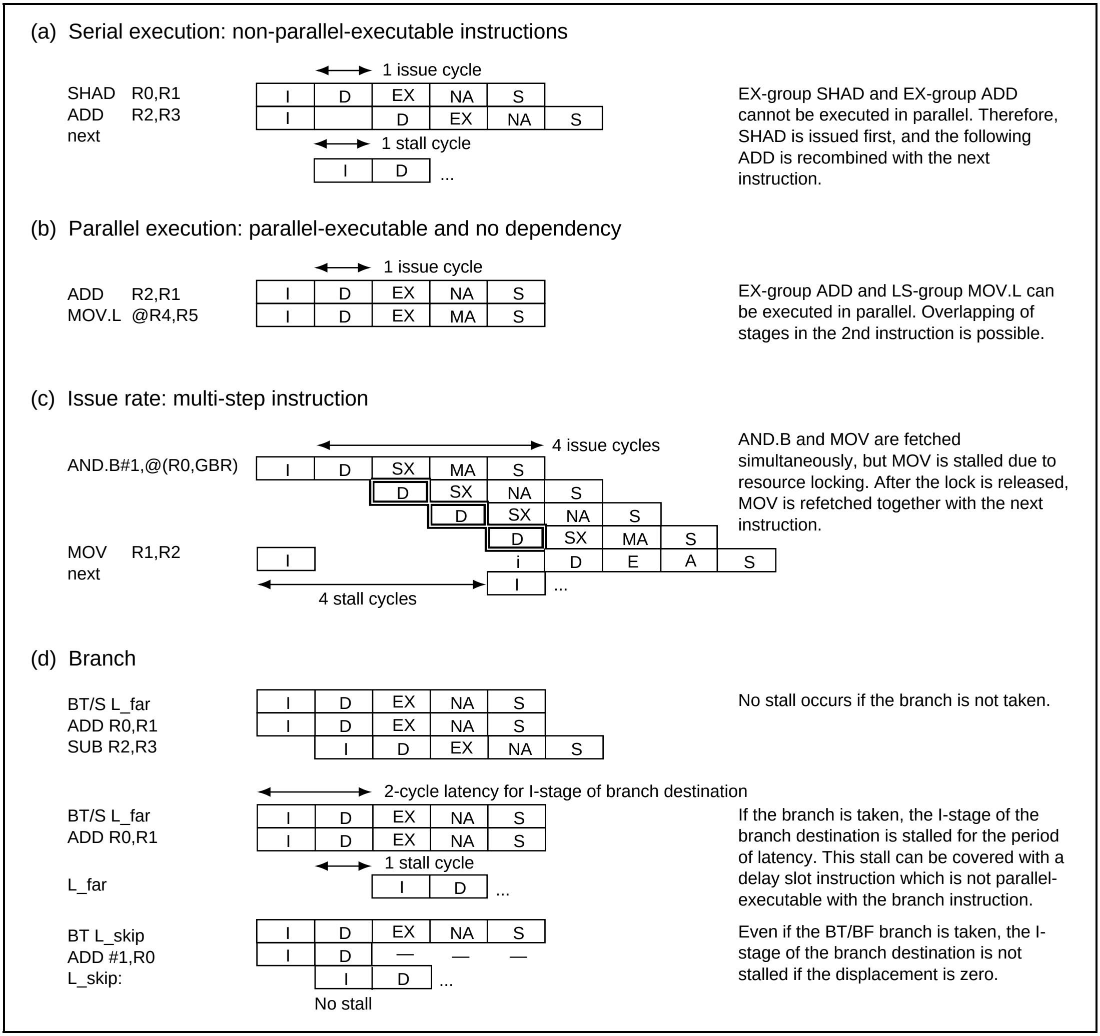


Figure 8.3 Examples of Pipelined Execution

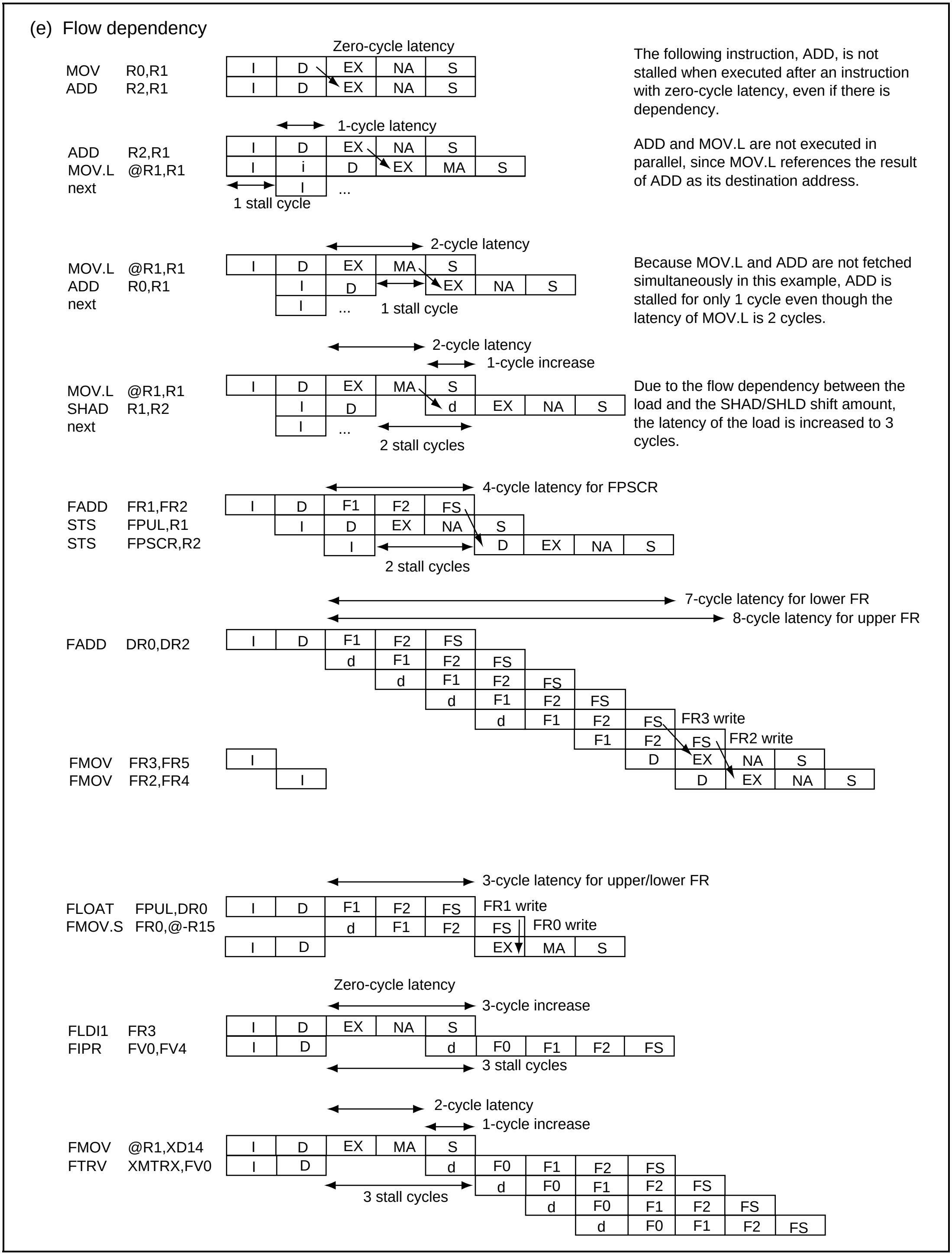


Figure 8.3 Examples of Pipelined Execution (cont)

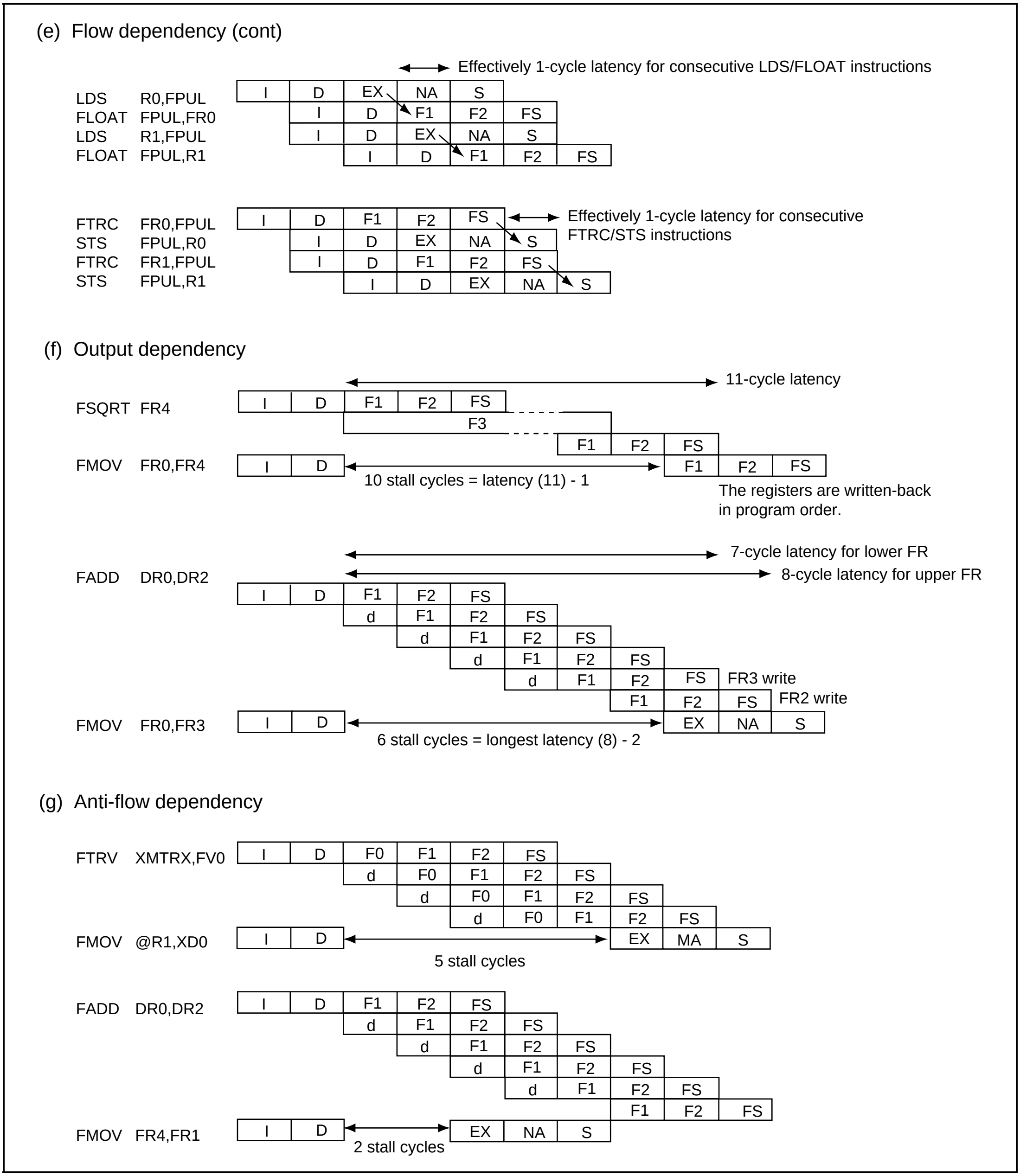


Figure 8.3 Examples of Pipelined Execution (cont)

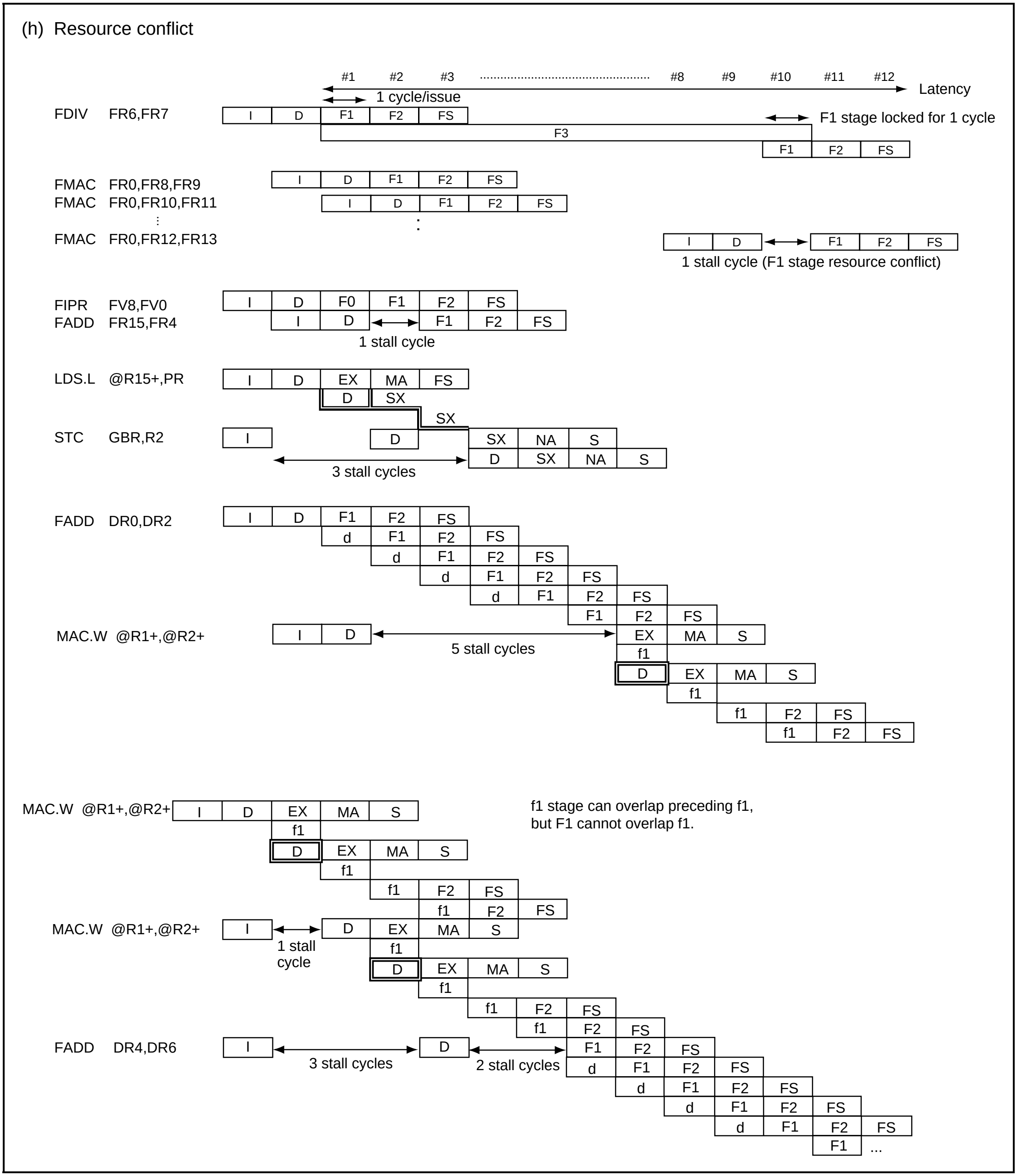


Figure 8.3 Examples of Pipelined Execution (cont)

Table 8.3 Execution Cycles

Functional Category	No.	Instruction		Instruction Group	Issue Rate	Latency	Execution Pattern	Lock Stage	Lock Start	Lock Cycles
Data transfer instructions	1	EXTS.B	Rm,Rn	EX	1	1	#1	—	—	—
	2	EXTS.W	Rm,Rn	EX	1	1	#1	—	—	—
	3	EXTU.B	Rm,Rn	EX	1	1	#1	—	—	—
	4	EXTU.W	Rm,Rn	EX	1	1	#1	—	—	—
	5	MOV	Rm,Rn	MT	1	0	#1	—	—	—
	6	MOV	#imm,Rn	EX	1	1	#1	—	—	—
	7	MOVA	@(disp,PC),R0	EX	1	1	#1	—	—	—
	8	MOV.W	@(disp,PC),Rn	LS	1	2	#2	—	—	—
	9	MOV.L	@(disp,PC),Rn	LS	1	2	#2	—	—	—
	10	MOV.B	@Rm,Rn	LS	1	2	#2	—	—	—
	11	MOV.W	@Rm,Rn	LS	1	2	#2	—	—	—
	12	MOV.L	@Rm,Rn	LS	1	2	#2	—	—	—
	13	MOV.B	@Rm+,Rn	LS	1	1/2	#2	—	—	—
	14	MOV.W	@Rm+,Rn	LS	1	1/2	#2	—	—	—
	15	MOV.L	@Rm+,Rn	LS	1	1/2	#2	—	—	—
	16	MOV.B	@(disp,Rm),R0	LS	1	2	#2	—	—	—
	17	MOV.W	@(disp,Rm),R0	LS	1	2	#2	—	—	—
	18	MOV.L	@(disp,Rm),Rn	LS	1	2	#2	—	—	—
	19	MOV.B	@(R0,Rm),Rn	LS	1	2	#2	—	—	—
	20	MOV.W	@(R0,Rm),Rn	LS	1	2	#2	—	—	—
	21	MOV.L	@(R0,Rm),Rn	LS	1	2	#2	—	—	—
	22	MOV.B	@(disp,GBR),R0	LS	1	2	#3	—	—	—
	23	MOV.W	@(disp,GBR),R0	LS	1	2	#3	—	—	—
	24	MOV.L	@(disp,GBR),R0	LS	1	2	#3	—	—	—
	25	MOV.B	Rm,@Rn	LS	1	1	#2	—	—	—
	26	MOV.W	Rm,@Rn	LS	1	1	#2	—	—	—
	27	MOV.L	Rm,@Rn	LS	1	1	#2	—	—	—
	28	MOV.B	Rm,@-Rn	LS	1	1/1	#2	—	—	—
	29	MOV.W	Rm,@-Rn	LS	1	1/1	#2	—	—	—
	30	MOV.L	Rm,@-Rn	LS	1	1/1	#2	—	—	—
	31	MOV.B	R0,@(disp,Rn)	LS	1	1	#2	—	—	—

Table 8.3 Execution Cycles (cont)

Functional Category	No.	Instruction		Instruction Group	Issue Rate	Latency	Execution Pattern	Lock		
								Stage	Start	Cycles
Data transfer instructions	32	MOV.W	R0,@(disp,Rn)	LS	1	1	#2	—	—	—
	33	MOV.L	Rm,@(disp,Rn)	LS	1	1	#2	—	—	—
	34	MOV.B	Rm,@(R0,Rn)	LS	1	1	#2	—	—	—
	35	MOV.W	Rm,@(R0,Rn)	LS	1	1	#2	—	—	—
	36	MOV.L	Rm,@(R0,Rn)	LS	1	1	#2	—	—	—
	37	MOV.B	R0,@(disp,GBR)	LS	1	1	#3	—	—	—
	38	MOV.W	R0,@(disp,GBR)	LS	1	1	#3	—	—	—
	39	MOV.L	R0,@(disp,GBR)	LS	1	1	#3	—	—	—
	40	MOVCA.L	R0,@Rn	LS	1	3–7	#12	MA	4	3–7
	41	MOVT	Rn	EX	1	1	#1	—	—	—
	42	OCBI	@Rn	LS	1	1–2	#10	MA	4	1–2
	43	OCBP	@Rn	LS	1	1–5	#11	MA	4	1–5
	44	OCBWB	@Rn	LS	1	1–5	#11	MA	4	1–5
	45	PREF	@Rn	LS	1	1	#2	—	—	—
	46	SWAP.B	Rm,Rn	EX	1	1	#1	—	—	—
	47	SWAP.W	Rm,Rn	EX	1	1	#1	—	—	—
	48	XTRCT	Rm,Rn	EX	1	1	#1	—	—	—
Fixed-point arithmetic instructions	49	ADD	Rm,Rn	EX	1	1	#1	—	—	—
	50	ADD	#imm,Rn	EX	1	1	#1	—	—	—
	51	ADDC	Rm,Rn	EX	1	1	#1	—	—	—
	52	ADDV	Rm,Rn	EX	1	1	#1	—	—	—
	53	CMP/EQ	#imm,R0	MT	1	1	#1	—	—	—
	54	CMP/EQ	Rm,Rn	MT	1	1	#1	—	—	—
	55	CMP/GE	Rm,Rn	MT	1	1	#1	—	—	—
	56	CMP/GT	Rm,Rn	MT	1	1	#1	—	—	—
	57	CMP/HI	Rm,Rn	MT	1	1	#1	—	—	—
	58	CMP/HS	Rm,Rn	MT	1	1	#1	—	—	—
	59	CMP/PL	Rn	MT	1	1	#1	—	—	—
	60	CMP/PZ	Rn	MT	1	1	#1	—	—	—
	61	CMP/STR	Rm,Rn	MT	1	1	#1	—	—	—
	62	DIV0S	Rm,Rn	EX	1	1	#1	—	—	—

Table 8.3 Execution Cycles (cont)

Functional Category	No.	Instruction		Instruction Group	Issue Rate	Latency	Execution Pattern	Lock Stage	Lock Start	Lock Cycles
Fixed-point arithmetic instructions	63	DIV0U		EX	1	1	#1	—	—	—
	64	DIV1	Rm,Rn	EX	1	1	#1	—	—	—
	65	DMULS.L	Rm,Rn	CO	2	4/4	#34	F1	4	2
	66	DMULU.L	Rm,Rn	CO	2	4/4	#34	F1	4	2
	67	DT	Rn	EX	1	1	#1	—	—	—
	68	MAC.L	@Rm+,@Rn+	CO	2	2/2/4/4	#35	F1	4	2
	69	MAC.W	@Rm+,@Rn+	CO	2	2/2/4/4	#35	F1	4	2
	70	MUL.L	Rm,Rn	CO	2	4/4	#34	F1	4	2
	71	MULS.W	Rm,Rn	CO	2	4/4	#34	F1	4	2
	72	MULU.W	Rm,Rn	CO	2	4/4	#34	F1	4	2
	73	NEG	Rm,Rn	EX	1	1	#1	—	—	—
	74	NEGC	Rm,Rn	EX	1	1	#1	—	—	—
	75	SUB	Rm,Rn	EX	1	1	#1	—	—	—
	76	SUBC	Rm,Rn	EX	1	1	#1	—	—	—
	77	SUBV	Rm,Rn	EX	1	1	#1	—	—	—
Logical instructions	78	AND	Rm,Rn	EX	1	1	#1	—	—	—
	79	AND	#imm,R0	EX	1	1	#1	—	—	—
	80	AND.B	#imm,@(R0,GBR)	CO	4	4	#6	—	—	—
	81	NOT	Rm,Rn	EX	1	1	#1	—	—	—
	82	OR	Rm,Rn	EX	1	1	#1	—	—	—
	83	OR	#imm,R0	EX	1	1	#1	—	—	—
	84	OR.B	#imm,@(R0,GBR)	CO	4	4	#6	—	—	—
	85	TAS.B	@Rn	CO	5	5	#7	—	—	—
	86	TST	Rm,Rn	MT	1	1	#1	—	—	—
	87	TST	#imm,R0	MT	1	1	#1	—	—	—
	88	TST.B	#imm,@(R0,GBR)	CO	3	3	#5	—	—	—
	89	XOR	Rm,Rn	EX	1	1	#1	—	—	—
	90	XOR	#imm,R0	EX	1	1	#1	—	—	—
	91	XOR.B	#imm,@(R0,GBR)	CO	4	4	#6	—	—	—

Table 8.3 Execution Cycles (cont)

Functional Category	No.	Instruction		Instruction Group	Issue Rate	Latency	Execution Pattern	Lock Stage	Lock Start	Lock Cycles
Shift instructions	92	ROTL	Rn	EX	1	1	#1	—	—	—
	93	ROTR	Rn	EX	1	1	#1	—	—	—
	94	ROTCL	Rn	EX	1	1	#1	—	—	—
	95	ROTCR	Rn	EX	1	1	#1	—	—	—
	96	SHAD	Rm,Rn	EX	1	1	#1	—	—	—
	97	SHAL	Rn	EX	1	1	#1	—	—	—
	98	SHAR	Rn	EX	1	1	#1	—	—	—
	99	SHLD	Rm,Rn	EX	1	1	#1	—	—	—
	100	SHLL	Rn	EX	1	1	#1	—	—	—
	101	SHLL2	Rn	EX	1	1	#1	—	—	—
	102	SHLL8	Rn	EX	1	1	#1	—	—	—
	103	SHLL16	Rn	EX	1	1	#1	—	—	—
	104	SHLR	Rn	EX	1	1	#1	—	—	—
	105	SHLR2	Rn	EX	1	1	#1	—	—	—
	106	SHLR8	Rn	EX	1	1	#1	—	—	—
	107	SHLR16	Rn	EX	1	1	#1	—	—	—
Branch instructions	108	BF	disp	BR	1	2 (or 1)	#1	—	—	—
	109	BF/S	disp	BR	1	2 (or 1)	#1	—	—	—
	110	BT	disp	BR	1	2 (or 1)	#1	—	—	—
	111	BT/S	disp	BR	1	2 (or 1)	#1	—	—	—
	112	BRA	disp	BR	1	2	#1	—	—	—
	113	BRAF	Rn	CO	2	3	#4	—	—	—
	114	BSR	disp	BR	1	2	#14	SX	3	2
	115	BSRF	Rn	CO	2	3	#24	SX	3	2
	116	JMP	@Rn	CO	2	3	#4	—	—	—
	117	JSR	@Rn	CO	2	3	#24	SX	3	2
	118	RTS		CO	2	3	#4	—	—	—

Table 8.3 Execution Cycles (cont)

Functional Category	No.	Instruction		Instruction Group	Issue Rate	Latency	Execution Pattern	Lock Stage	Lock Start	Lock Cycles
System control instructions	119	NOP		MT	1	0	#1	—	—	—
	120	CLRMAC		CO	1	3	#28	F1	3	2
	121	CLRS		CO	1	1	#1	—	—	—
	122	CLRT		MT	1	1	#1	—	—	—
	123	SETS		CO	1	1	#1	—	—	—
	124	SETT		MT	1	1	#1	—	—	—
	125	TRAPA	#imm	CO	7	7	#13	—	—	—
	126	RTE		CO	5	5	#8	—	—	—
	127	SLEEP		CO	4	4	#9	—	—	—
	128	LDTLB		CO	1	1	#2	—	—	—
	129	LDC	Rm,DBR	CO	1	3	#14	SX	3	2
	130	LDC	Rm,GBR	CO	3	3	#15	SX	3	2
	131	LDC	Rm,Rp_BANK	CO	1	3	#14	SX	3	2
	132	LDC	Rm,SR	CO	4	4	#16	SX	3	2
	133	LDC	Rm,SSR	CO	1	3	#14	SX	3	2
	134	LDC	Rm,SPC	CO	1	3	#14	SX	3	2
	135	LDC	Rm,VBR	CO	1	3	#14	SX	3	2
	136	LDC.L	@Rm+,DBR	CO	1	1/3	#17	SX	3	2
	137	LDC.L	@Rm+,GBR	CO	3	3/3	#18	SX	3	2
	138	LDC.L	@Rm+,Rp_BANK	CO	1	1/3	#17	SX	3	2
	139	LDC.L	@Rm+,SR	CO	4	4/4	#19	SX	3	2
	140	LDC.L	@Rm+,SSR	CO	1	1/3	#17	SX	3	2
	141	LDC.L	@Rm+,SPC	CO	1	1/3	#17	SX	3	2
	142	LDC.L	@Rm+,VBR	CO	1	1/3	#17	SX	3	2
	143	LDS	Rm,MACH	CO	1	3	#28	F1	3	2
	144	LDS	Rm,MACL	CO	1	3	#28	F1	3	2
	145	LDS	Rm,PR	CO	2	3	#24	SX	3	2
	146	LDS.L	@Rm+,MACH	CO	1	1/3	#29	F1	3	2
	147	LDS.L	@Rm+,MACL	CO	1	1/3	#29	F1	3	2
	148	LDS.L	@Rm+,PR	CO	2	2/3	#25	SX	3	2
	149	STC	DBR,Rn	CO	2	2	#20	—	—	—
	150	STC	SGR,Rn	CO	3	3	#21	—	—	—

Table 8.3 Execution Cycles (cont)

Functional Category	No.	Instruction		Instruction Group	Issue Rate	Latency	Execution Pattern	Lock Stage	Lock Start	Lock Cycles
System control instructions	151	STC	GBR,Rn	CO	2	2	#20	—	—	—
	152	STC	Rp_BANK,Rn	CO	2	2	#20	—	—	—
	153	STC	SR,Rn	CO	2	2	#20	—	—	—
	154	STC	SSR,Rn	CO	2	2	#20	—	—	—
	155	STC	SPC,Rn	CO	2	2	#20	—	—	—
	156	STC	VBR,Rn	CO	2	2	#20	—	—	—
	157	STC.L	DBR,@-Rn	CO	2	2/2	#22	—	—	—
	158	STC.L	SGR,@-Rn	CO	3	3/3	#23	—	—	—
	159	STC.L	GBR,@-Rn	CO	2	2/2	#22	—	—	—
	160	STC.L	Rp_BANK,@-Rn	CO	2	2/2	#22	—	—	—
	161	STC.L	SR,@-Rn	CO	2	2/2	#22	—	—	—
	162	STC.L	SSR,@-Rn	CO	2	2/2	#22	—	—	—
	163	STC.L	SPC,@-Rn	CO	2	2/2	#22	—	—	—
	164	STC.L	VBR,@-Rn	CO	2	2/2	#22	—	—	—
	165	STS	MACH,Rn	CO	1	3	#30	—	—	—
	166	STS	MACL,Rn	CO	1	3	#30	—	—	—
	167	STS	PR,Rn	CO	2	2	#26	—	—	—
	168	STS.L	MACH,@-Rn	CO	1	1/1	#31	—	—	—
	169	STS.L	MACL,@-Rn	CO	1	1/1	#31	—	—	—
	170	STS.L	PR,@-Rn	CO	2	2/2	#27	—	—	—
Single-precision floating-point instructions	171	FLDI0	FRn	LS	1	0	#1	—	—	—
	172	FLDI1	FRn	LS	1	0	#1	—	—	—
	173	FMOV	FRm,FRn	LS	1	0	#1	—	—	—
	174	FMOV.S	@Rm,FRn	LS	1	2	#2	—	—	—
	175	FMOV.S	@Rm+,FRn	LS	1	1/2	#2	—	—	—
	176	FMOV.S	@(R0,Rm),FRn	LS	1	2	#2	—	—	—
	177	FMOV.S	FRm,@Rn	LS	1	1	#2	—	—	—
	178	FMOV.S	FRm,@-Rn	LS	1	1/1	#2	—	—	—
	179	FMOV.S	FRm,@(R0,Rn)	LS	1	1	#2	—	—	—
	180	FLDS	FRm,FPUL	LS	1	0	#1	—	—	—
	181	FSTS	FPUL,FRn	LS	1	0	#1	—	—	—

Table 8.3 Execution Cycles (cont)

Functional Category	No.	Instruction		Instruction Group	Issue Rate	Latency	Execution Pattern	Lock Stage	Lock Start	Lock Cycles
Single-precision floating-point instructions	182	FABS	FRn	LS	1	0	#1	—	—	—
	183	FADD	FRm,FRn	FE	1	3/4	#36	—	—	—
	184	FCMP/EQ	FRm,FRn	FE	1	2/4	#36	—	—	—
	185	FCMP/GT	FRm,FRn	FE	1	2/4	#36	—	—	—
	186	FDIV	FRm,FRn	FE	1	12/13	#37	F3	2	10
								F1	11	1
	187	FLOAT	FPUL,FRn	FE	1	3/4	#36	—	—	—
	188	FMAC	FR0,FRm,FRn	FE	1	3/4	#36	—	—	—
	189	FMUL	FRm,FRn	FE	1	3/4	#36	—	—	—
	190	FNEG	FRn	LS	1	0	#1	—	—	—
	191	FSQRT	FRn	FE	1	11/12	#37	F3	2	9
								F1	10	1
	192	FSUB	FRm,FRn	FE	1	3/4	#36	—	—	—
	193	FTRC	FRm,FPUL	FE	1	3/4	#36	—	—	—
	194	FMOV	DRm,DRn	LS	1	0	#1	—	—	—
	195	FMOV	@Rm,DRn	LS	1	2	#2	—	—	—
	196	FMOV	@Rm+,DRn	LS	1	1/2	#2	—	—	—
	197	FMOV	@(R0,Rm),DRn	LS	1	2	#2	—	—	—
	198	FMOV	DRm,@Rn	LS	1	1	#2	—	—	—
	199	FMOV	DRm,@-Rn	LS	1	1/1	#2	—	—	—
	200	FMOV	DRm,@(R0,Rn)	LS	1	1	#2	—	—	—
Double-precision floating-point instructions	201	FABS	DRn	LS	1	0	#1	—	—	—
	202	FADD	DRm,DRn	FE	1	(7, 8)/9	#39	F1	2	6
	203	FCMP/EQ	DRm,DRn	CO	2	3/5	#40	F1	2	2
	204	FCMP/GT	DRm,DRn	CO	2	3/5	#40	F1	2	2
	205	FCNVDS	DRm,FPUL	FE	1	4/5	#38	F1	2	2
	206	FCNVSD	FPUL,DRn	FE	1	(3, 4)/5	#38	F1	2	2
	207	FDIV	DRm,DRn	FE	1	(24, 25)/26	#41	F3	2	23
								F1	22	3
								F1	2	2
	208	FLOAT	FPUL,DRn	FE	1	(3, 4)/5	#38	F1	2	2
	209	FMUL	DRm,DRn	FE	1	(7, 8)/9	#39	F1	2	6

Table 8.3 Execution Cycles (cont)

Functional Category	No.	Instruction		Instruction Group	Issue Rate	Latency	Execution Pattern	Lock Stage	Lock Start	Lock Cycles
Double-precision floating-point instructions	210	FNEG	DRn	LS	1	0	#1	—	—	—
	211	FSQRT	DRn	FE	1	(23, 24)/ 25	#41	F3	2	22
								F1	21	3
								F1	2	2
	212	FSUB	DRm,DRn	FE	1	(7, 8)/9	#39	F1	2	6
	213	FTRC	DRm,FPUL	FE	1	4/5	#38	F1	2	2
FPU system control instructions	214	LDS	Rm,FPUL	LS	1	1	#1	—	—	—
	215	LDS	Rm,FPSCR	CO	1	4	#32	F1	3	3
	216	LDS.L	@Rm+,FPUL	CO	1	1/2	#2	—	—	—
	217	LDS.L	@Rm+,FPSCR	CO	1	1/4	#33	F1	3	3
	218	STS	FPUL,Rn	LS	1	3	#1	—	—	—
	219	STS	FPSCR,Rn	CO	1	3	#1	—	—	—
	220	STS.L	FPUL,@-Rn	CO	1	1/1	#2	—	—	—
	221	STS.L	FPSCR,@-Rn	CO	1	1/1	#2	—	—	—
Graphics acceleration instructions	222	FMOV	DRm,XDn	LS	1	0	#1	—	—	—
	223	FMOV	XDm,DRn	LS	1	0	#1	—	—	—
	224	FMOV	XDm,XDn	LS	1	0	#1	—	—	—
	225	FMOV	@Rm,XDn	LS	1	2	#2	—	—	—
	226	FMOV	@Rm+,XDn	LS	1	1/2	#2	—	—	—
	227	FMOV	@(R0,Rm),XDn	LS	1	2	#2	—	—	—
	228	FMOV	XDm,@Rn	LS	1	1	#2	—	—	—
	229	FMOV	XDm,@-Rm	LS	1	1/1	#2	—	—	—
	230	FMOV	XDm,@(R0,Rn)	LS	1	1	#2	—	—	—
	231	FIPR	FVm,FVn	FE	1	4/5	#42	F1	3	1
	232	FRCHG		FE	1	1/4	#36	—	—	—
	233	FSCHG		FE	1	1/4	#36	—	—	—
	234	FTRV	XMTRX,FVn	FE	1	(5, 5, 6, 7)/8	#43	F0	2	4
								F1	3	4

Notes: 1. See table 8.1 for the instruction groups.

2. Latency "L1/L2...": Latency corresponding to a write to each register, including MACH/MACL/FPSCR.

Example: MOV.B @Rm+, Rn "1/2": The latency for Rm is 1 cycle, and the latency for Rn is 2 cycles.

3. Branch latency: Interval until the branch destination instruction is fetched

4. Conditional branch latency “2 (or 1)”: The latency is 2 for a nonzero displacement, and 1 for a zero displacement.
5. Double-precision floating-point instruction latency “(L1, L2)/L3”: L1 is the latency for FR [n+1], L2 that for FR [n], and L3 that for FPSCR.
6. FTRV latency “(L1, L2, L3, L4)/L5”: L1 is the latency for FR [n], L2 that for FR [n+1], L3 that for FR [n+2], L4 that for FR [n+3], and L5 that for FPSCR.
7. Latency “L1/L2/L3/L4” of MAC.L and MAC.W instructions: L1 is the latency for Rm, L2 that for Rn, L3 that for MACH, and L4 that for MACL.
8. Latency “L1/L2” of MUL.L, MULS.W, MULU.W, DMULS.L, and DMULU.L instructions: L1 is the latency for MACH, and L2 that for MACL.
9. Execution pattern: The instruction execution pattern number (see figure 8.2)
10. Lock/stage: Stage locked by the instruction
11. Lock/start: Locking start cycle; 1 is the first D-stage of the instruction.
12. Lock/cycles: Number of cycles locked

Exceptions:

1. When a floating-point computation instruction is followed by an FMOV store, an STS FPUL, Rn instruction, or an STS.L FPUL, @-Rn instruction, the latency of the floating-point computation is decreased by 1 cycle.
2. When the preceding instruction loads the shift amount of the following SHAD/SHLD, the latency of the load is increased by 1 cycle.
3. When an LS group instruction with a latency of less than 3 cycles is followed by a double-precision floating-point instruction, FIPR, or FTRV, the latency of the first instruction is increased to 3 cycles.

   Example: In the case of FMOV FR4,FR0 and FIPR FV0,FV4, FIPR is stalled for 2 cycles.
4. When MAC*/MUL*/DMUL* is followed by an STS.L MAC*, @-Rn instruction, the latency of MAC*/MUL*/DMUL* is 5 cycles.
5. In the case of consecutive executions of MAC*/MUL*/DMUL*, the latency is decreased to 2 cycles.
6. When an LDS to MAC* is followed by an STS.L MAC*, @-Rn instruction, the latency of the LDS to MAC* is 4 cycles.
7. When an LDS to MAC* is followed by MAC*/MUL*/DMUL*, the latency of the LDS to MAC* is 1 cycle.
8. When an FSCHG or FRCHG instruction is followed by an LS group instruction that reads or writes to a floating-point register, the aforementioned LS group instruction[s] cannot be executed in parallel.
9. When a single-precision FTRC instruction is followed by an STS FPUL, Rn instruction, the latency of the single-precision FTRC instruction is 1 cycle.

Section 9 Instruction Descriptions

Instructions are listed in this section in alphabetical order. The following format is used for the instruction descriptions.

Instruction Name	**Full Name**	**Instruction Type**
Function		(Indication of delayed branch instruction or interrupt-disabling instruction)

Format	Summary of Operation	Instruction Code	Execution States	T Bit
The assembler input format is shown. imm and disp are numeric values, expressions, or symbols.	Summarizes the operation of the instruction.	Shown in MSB ←→ LSB order.	The no-wait value is shown.	Shows the T bit value after execution of the instruction.

Description

Describes the operation of the instruction.

Notes

Identifies points to be noted when using the instruction.

Operation

Shows the operation in C. This is given as reference material to help understand the operation of the instruction. Use of the following resources is assumed.

```
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

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           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;
     }
}
```

Example

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 Cross-Assembler User's Manual.

.org	Location counter setting
.data.w	Word integer data allocation
.data.l	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
.arepea t 32	32-times repeat expansion
.aendr	Count-specification repeat expansion end

Note: SH Series cross-assembler version 1.0 does not support conditional assembler functions.

Format	Summary of Operation	Instruction Code	Execution States	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.

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
```

9.2 ADDC ADD with Carry — Arithmetic Instruction

Binary Addition with Carry

Format	Summary of Operation	Instruction Code	Execution States	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.

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
```

9.3 ADDV ADD with (V flag) overflow check Arithmetic Instruction

Binary Addition
with Overflow Check

Format	Summary of Operation	Instruction Code	Execution States	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.

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
```

Format	Summary of Operation	Instruction Code	Execution States	T Bit
AND Rm,Rn	Rm & Rm → Rn	0010nnnnmmmm1001	1	—
AND #imm,R0	R0 & imm → R0	11001001iiiiiiii	1	—
AND.B #imm,@(R0,GBR)	(R0+GBR) & imm → (R0+GBR)	11001101iiiiiiii	4	—

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
```

Format	Summary of Operation	Instruction Code	Execution States	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
```

Format	Summary of Operation	Instruction Code	Execution States	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 BF, 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
```

Format	Summary of Operation	Instruction Code	Execution States	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
```

Format	Summary of Operation	Instruction Code	Execution States	T Bit
BRAF Rn	PC+4+Rn → PC	0000nnnn00100011	2	—

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
```

Format	Summary of Operation	Instruction Code	Execution States	T Bit
BSR label	PC+4 → PR, PC+4+disp×2 → 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.
```

9.10 BSRF — Branch to SubRoutine Far

Format	Summary of Operation	Instruction Code	Execution States	T Bit
BSRF Rn	PC+4 → PR, PC+4+Rn → PC	0000nnnn00000011	2	—

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.
```

9.11 BT Branch if True Branch Instruction

Conditional Branch

Format	Summary of Operation	Instruction Code	Execution States	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
```

9.12 BT/S Branch if True with delay Slot — Branch Instruction

Conditional Branch with Delay — Delayed Branch Instruction

Format	Summary of Operation	Instruction Code	Execution States	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. 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.

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.

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
```

Format	Summary of Operation	Instruction Code	Execution States	T Bit
CLRMAC	0 → MACH, MACL	0000000000101000	1	—

Description

This instruction clears the MACH and MACL registers.

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+       ;
```

Format	Summary of Operation	Instruction Code	Execution States	T Bit
CLRS	0 → S	0000000001001000	1	—

Description

This instruction clears the S bit to 0.

Operation

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

Example

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

9.15 CLRT CleaR T bit

T Bit Clear

Format	Summary of Operation	Instruction Code	Execution States	T Bit
CLRT	0 → T	0000000000001000	1	0

Description

This instruction clears the T bit.

Operation

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

Example

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

9.16 CMP/cond CoMPare conditionally Arithmetic Instruction

Compare

Format		Summary of Operation	Instruction Code	Execution States	T Bit
CMP/EQ	Rm,Rn	If Rn = Rm, 1 → T	`0011nnnnmmmm0000`	1	Result of comparison
CMP/GE	Rm,Rn	If Rn ≥ Rm, signed, 1 → T	`0011nnnnmmmm0011`	1	Result of comparison
CMP/GT	Rm,Rn	If Rn > Rm, signed, 1 → T	`0011nnnnmmmm0111`	1	Result of comparison
CMP/HI	Rm,Rn	If Rn > Rm, unsigned, 1 → T	`0011nnnnmmmm0110`	1	Result of comparison
CMP/HS	Rm,Rn	If Rn ≥ Rm, unsigned, 1 → T	`0011nnnnmmmm0010`	1	Result of comparison
CMP/PL	Rn	If Rn > 0, 1 → T	`0100nnnn00010101`	1	Result of comparison
CMP/PZ	Rn	If Rn ≥ 0, 1 → T	`0100nnnn00010001`	1	Result of comparison
CMP/STR	Rm,Rn	If any bytes are equal, 1 → T	`0010nnnnmmmm1100`	1	Result of comparison
CMP/EQ	#imm,R0	If R0 = imm, 1 → 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

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.
```

9.17 DIV0S DIVide (step 0) as Signed Arithmetic Instruction

Initialization for Signed Division

Format	Summary of Operation	Instruction Code	Execution States	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.

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.

9.18 DIV0U DIVide (step 0) as Unsigned Arithmetic Instruction

Initialization for Unsigned Division

Format	Summary of Operation	Instruction Code	Execution States	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.

Operation

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

Example

See the examples for the DIV1 instruction.

Format	Summary of Operation	Instruction Code	Execution States	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:

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

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.

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)
```

9.20 DMULS.L Double-length MULtiply as Signed — Arithmetic Instruction

Signed Double-Length Multiplication

Format	Summary of Operation	Instruction Code	Execution States	T Bit
DMULS.L Rm,Rn	Signed, Rn × Rm → MACH, MACL	0011nnnnmmmm1101	2–5	—

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.

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    ;et operation result (lower)
```

9.21 DMULU.L Double-length MULtiply as Unsigned — Arithmetic Instruction

Unsigned Double-Length Multiplication

Format	Summary of Operation	Instruction Code	Execution States	T Bit
DMULU.L Rm,Rn	Unsigned, Rn × Rm → MACH, MACL	0011nnnnmmmm0101	2–5	—

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.

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)
```

Format	Summary of Operation	Instruction Code	Execution States	T Bit
DT Rn	Rn – 1 → Rn; if Rn = 0, 1 → T if Rn ≠ 0, 0 → T	0100nnnn00010000	1	Test result

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.

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).
```

Format	Summary of Operation	Instruction Code	Execution States	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.

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
```

Format	Summary of Operation	Instruction Code	Execution States	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.

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
```

9.25 FABS Floating-point ABSolute value Floating-Point Instruction

Floating-Point
Absolute Value

PR	Format	Summary of Operation	Instruction Code	Execution States	T Bit
0	FABS FRn	|FRn| → FRn	1111nnnn01011101	1	—
1	FABS DRn	|DRn| → 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.

Operation

```
void FABS (int n){
     FR[n] = FR[n] & 0x7fffffff;
     pc += 2;
}
```

/* Same operation is performed regardless of precision. */

Possible Exceptions:

None

PR	Format	Summary of Operation	Instruction Code	Execution States	T Bit
0	FADD FRm,FRn	FRn+FRm → FRn	1111nnnnmmmm0000	1	—
1	FADD DRm,DRn	DRn+DRm → DRn	1111nnn0mmm00000	6	—

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.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 FRn or DRn is not updated. Appropriate processing should therefore be performed by software.

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

FRm,DRm	FRn,DRn							
	NORM	+0	–0	+INF	–INF	DENORM	qNaN	sNaN
NORM	ADD				–INF			
+0		+0						
–0			–0					
+INF				+INF	Invalid			
–INF	–INF			Invalid	–INF			
DENORM						Error		
qNaN							qNaN	
sNaN								Invalid

Note: When DN = 1, the value of a denormalized number is treated as 0.

Possible Exceptions:

- FPU error
- Invalid operation
- Overflow
- Underflow
- Inexact

No.	PR	Format	Summary of Operation	Instruction Code	Execution States	T Bit
1.	0	FCMP/EQ FRm,FRn	(FRn==FRm)?1:0 → T	1111nnnnmmmm0100	1	1/0
2.	1	FCMP/EQ DRm,DRn	(DRn==DRm)?1:0 → T	1111nnn0mmm00100	1	1/0
3.	0	FCMP/GT FRm,FRn	(FRn>FRm)?1:0 → T	1111nnnnmmmm0101	2	1/0
4.	1	FCMP/GT DRm,DRn	(DRn>DRm)?1:0 → T	1111nnn0mmm00101	2	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.

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

9.28 FCNVDS Floating-point CoNVert Double to Single precision Floating-Point Instruction

Double-Precision to Single-Precision Conversion

PR	Format	Summary of Operation	Instruction Code	Execution States	T Bit
0	—	—	—	—	—
1	FCNVDS DRm,FPUL	(float)DRm → FPUL	1111mmm010111101	2	—

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.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 FPUL is not updated. Appropriate processing should therefore be performed by software.

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, *FPUL)){
          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

FRn	+NORM	–NORM	+0	–0	+INF	–INF	qNaN	sNaN
FCNVDS(FRn 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:

- FPU error
- Invalid operation
- Overflow
- Underflow
- Inexact

9.29 FCNVSD Floating-point CoNVert Single to Double precision Floating-Point Instruction

Single-Precision to Double-Precision Conversion

PR	Format	Summary of Operation	Instruction Code	Execution States	T Bit
0	—	—	—	—	—
1	FCNVSD FPUL, DRn	(double) FPUL → DRn	1111nnn010101101	2	—

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.

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

PR	Format	Summary of Operation	Instruction Code	Execution States	T Bit
0	FDIV FRm,FRn	FRn/FRm → FRn	1111nnnnmmmm0011	10	—
1	FDIV DRm,DRn	DRn/DRm → DRn	1111nnn0mmm00011	23	—

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.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 FRn or DRn is not updated. Appropriate processing should therefore be performed by software.

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) && (SPSCR_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) && (SPSCR_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

FRm,DRm	FRn,DRn							
	NORM	+0	–0	+INF	–INF	DENORM	qNaN	sNaN
NORM	DIV	0		INF		Error		
+0	DZ	Invalid		+INF	–INF	DZ		
–0				–INF	+INF			
+INF	0	+0	–0	Invalid				
–INF		–0	+0					
DENORM						Error		
qNaN							qNaN	
sNaN								Invalid

Note: When DN = 1, the value of a denormalized number is treated as 0.

Possible Exceptions:

- FPU error
- Invalid operation
- Divide by zero
- Overflow
- Underflow
- Inexact

9.31 FIPR Floating-point Inner PRoduct Floating-Point Instruction

Floating-Point
Inner Product

PR	Format	Summary of Operation	Instruction Code	Execution States	T Bit
0	FIPR FVm,FVn	FVn · FVm → 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.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 FRn or DRn is not updated. Appropriate processing should therefore be performed by software.

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:

- Invalid operation
- Overflow
- Underflow
- Inexact

9.32 FLDI0 Floating-point LoaD Immediate 0.0 Floating-Point Instruction

0.0 Load

PR	Format	Summary of Operation	Instruction Code	Execution States	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.

Operation

```
void FLDI0(int n)
{
     FR[n] = 0x00000000;
     pc += 2;
}
```

Possible Exceptions:

None

Format	Summary of Operation	Instruction Code	Execution States	T Bit
FLDI1 FRn	0x3F800000 → FRn	1111nnnn10011101	1	—
—	—	—	—	—

Description

When FPSCR.PR = 0, this instruction loads floating-point 1.0 (0x3F800000) into FRn.

Operation

```
void FLDI1(int n)
{
     FR[n] = 0x3F800000;
     pc += 2;
}
```

Possible Exceptions:

None

9.34 FLDS Floating-point LoaD to System register — Floating-Point Instruction

Transfer to System Register

Format	Summary of Operation	Instruction Code	Execution States	T Bit
FLDS FRm,FPUL	FRm → FPUL	1111mmmm00011101	1	—

Description

This instruction loads the contents of floating-point register FRm into system register FPUL.

Operation

```
void FLDS(int m, float *FPUL)
{
     *FPUL = FR[m];
     pc += 2;
}
```

Possible Exceptions:

None

9.35 FLOAT Floating-point convert from integer

Integer to Floating-Point Conversion

PR	Format	Summary of Operation	Instruction Code	Execution States	T Bit
0	FLOAT FPUL,FRn	(float)FPUL → FRn	1111nnnn00101101	1	—
1	FLOAT FPUL,DRn	(double)FPUL → DRn	1111nnn000101101	2	—

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, 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.

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.

PR	Format	Summary of Operation	Instruction Code	Execution States	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.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 FRn or DRn is not updated. Appropriate processing should therefore be performed by software.

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

FRn	FR0	FRm								
		+Norm	–Norm	+0	–0	+INF	–INF	Denorm	qNaN	sNaN
Norm	Norm	MAC				INF				
	0					Invalid				
	INF	INF		Invalid		INF				
+0	Norm	MAC								
	0				+0	Invalid				
	INF	INF		Invalid		INF				
–0	+Norm	MAC		+0	–0	+INF	–INF			
	–Norm			–0	+0	–INF	+INF			
	+0	+0	–0	+0	–0	Invalid				
	–0	–0	+0	–0	+0					
	INF	INF		Invalid		INF				
+INF	+Norm	+INF					Invalid			
	–Norm						+INF			
	0					Invalid				
	+INF			Invalid		+INF				
	–INF	Invalid	+INF				+INF			
–INF	+Norm	–INF					–INF			
	–Norm									
	0									
	+INF	Invalid		Invalid			–INF			
	–INF	–INF				–INF	Invalid			
Denorm	Norm									
	0					Invalid				
	INF			Invalid						
!sNaN	Denorm							Error		
qNaN	0					Invalid				
	INF			Invalid						
	Norm									
!sNaN	qNaN								qNaN	
All types	sNaN									
SNaN	all types									Invalid

Note: When DN = 1, the value of a denormalized number is treated as 0.

Possible Exceptions:

- FPU error
- Invalid operation
- Overflow
- Underflow
- Inexact

No.	SZ	Format		Summary of Operation	Instruction Code	Execution States	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	1111nnnnmmmm1001	1	—
8.	1	FMOV	@Rm+,DRn	(Rm) → DRn,Rm+=8	1111nnn0mmmm1001	1	—
9.	0	FMOV.S	FRm,@-Rn	Rn-=4,FRm → (Rn)	1111nnnnmmmm1011	1	—
10.	1	FMOV	DRm,@-Rn	Rn-=8,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).

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 write exception
- Address error

No.	PR	Format	Summary of Operation	Instruction Code	Execution States	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	1111nnn1mmmm1001	1	—
4.	1	FMOV XDm,@-Rn	Rn-=8,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+,DBn */
{
   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 write exception
- Address error

9.39 FMUL Floating-point MULtiply Floating-Point Instruction
Floating-Point Multiplication

PR	Format	Summary of Operation	Instruction Code	Execution States	T Bit
0	FMUL FRm,FRn	FRn*FRm → FRn	1111nnnnmmmm0010	1	—
1	FMUL DRm,DRn	DRn*DRm → DRn	1111nnn0mmm00010	6	—

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.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 FRn or DRn is not updated. Appropriate processing should therefore be performed by software.

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

FRm,DRm	FRn,DRn							
	NORM	+0	–0	+INF	–INF	DENORM	qNaN	sNaN
NORM	MUL	0		INF				
+0	0	+0	–0	Invalid				
–0		–0	+0					
+INF	INF	Invalid		+INF	–INF			
–INF				–INF	+INF			
DENORM						Error		
qNaN							qNaN	
sNaN								Invalid

Note: When DN = 1, the value of a denormalized number is treated as 0.

Possible Exceptions:

- FPU error
- Invalid operation
- Overflow
- Underflow
- Inexact

9.40 FNEG Floating-point NEGate value Floating-Point Instruction

Floating-Point
Sign Inversion

PR	Format	Summary of Operation	Instruction Code	Execution States	T Bit
0	FNEG FRn	-FRn → FRn	1111nnnn01001101	1	—
1	FNEG DRn	-DRn → 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.

Operation

```
void FNEG (int n){
     FR[n] = -FR[n];
     pc += 2;
}

/* Same operation is performed regardless of precision. */
```

Possible Exceptions:

None

PR	Format	Summary of Operation	Instruction Code	Execution States	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.

Operation

```
void FRCHG()   /* FRCHG */
{
     if(FPSCR_PR == 0){
         FPSCR ^= 0x00200000; /* bit 21 */
         PC += 2;
     }
     else undefined_operation();
}
```

Possible Exceptions:
None

9.42 FSCHG Sz-bit CHanGe Floating-Point Instruction

SZ Bit
Inversion

PR	Format	Summary of Operation	Instruction Code	Execution States	T Bit
0	FSCHG	FPSCR.SZ=~FPSCR.SZ	1111001111111101	1	—
1	—	—	—	—	—

Description

This instruction inverts the SZ bit in floating-point register FPSCR. Changing the SZ bit in FPSCR switches FMOV instruction data transfer between one single-precision data unit and a data pair. When FPSCR.SZ = 0, the FMOV instruction transfers one single-precision data unit. When FPSCR.SZ = 1, the FMOV instruction transfers two single-precision data units as a pair.

Operation

```
void FSCHG()   /* FSCHG */
{
     if(FPSCR_PR == 0){
         FPSCR ^= 0x00100000; /* bit 20 */
         PC += 2;
     }
     else undefined_operation();
}
```

Possible Exceptions:

None

9.43 FSQRT Floating-point SQuare RooT Floating-Point Instruction

Floating-Point
Square Root

PR	Format	Summary of Operation	Instruction Code	Execution States	T Bit
0	FSQRT FRn	√FRn → FRn	1111nnnn01101101	9	—
1	FSQRT DRn	√DRn → DRn	1111nnnn01101101	22	—

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.

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) && (SPSCR_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) && (SPSCR_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	+0	–0	+INF	–INF	qNaN	sNaN
FSQRT(FRn)	SQRT	Invalid	+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

Format	Summary of Operation	Instruction Code	Execution States	T Bit
FSTS FPUL,FRn	FPUL → FRn	1111nnnn00001101	1	—

Description

This instruction transfers the contents of system register FPUL to floating-point register FRn.

Operation

```
void FSTS(int n, float *FPUL)
{
     FR[n] = *FPUL;
     pc += 2;
}
```

Possible Exceptions:

None

9.45 FSUB Floating-point SUBtract — Floating-Point Instruction

Floating-Point Subtraction

PR	Format	Summary of Operation	Instruction Code	Execution States	T Bit
0	FSUB FRm,FRn	FRn-FRm → FRn	1111nnnnmmmm0001	1	—
1	FSUB DRm,DRn	DRn-DRm → DRn	1111nnn0mmm00001	6	

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.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 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 (data_type_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

FRm,DRm	FRn,DRn							
	NORM	+0	–0	+INF	–INF	DENORM	qNaN	sNaN
NORM	SUB	SUB	SUB	+INF	–INF	Error	qNaN	Invalid
+0	SUB	+0	–0	+INF	–INF	Error	qNaN	Invalid
–0	SUB	+0	+0	+INF	–INF	Error	qNaN	Invalid
+INF	–INF	–INF	–INF	Invalid	–INF	Error	qNaN	Invalid
–INF	+INF	+INF	+INF	+INF	Invalid	Error	qNaN	Invalid
DENORM	Error	Error	Error	Error	Error	Error	qNaN	Invalid
qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	Invalid
sNaN	Invalid	Invalid	Invalid	Invalid	Invalid	Invalid	Invalid	Invalid

Note: When DN = 1, the value of a denormalized number is treated as 0.

Possible Exceptions:

- FPU error
- Invalid operation
- Overflow
- Underflow
- Inexact

9.46 FTRC Floating-point TRuncate and Convert to integer — Floating-Point Instruction

Conversion to Integer

PR	Format	Summary of Operation	Instruction Code	Execution States	T Bit
0	FTRC FRm,FPUL	(long)FRm → FPUL	1111mmmm00111101	1	—
1	FTRC DRm,FPUL	(long)DRm → FPUL	1111mmm000111101	2	—

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.

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);  break;
          NINF:     ftrc_invalid(1);  break;
          }
     }
     else{                  /* case FPSCR.PR=1 */
          case(ftrc_double_type_of(m)){
```

```
            NORM:     *FPUL = DR[m>>1]; break;
            PINF:     ftrc_invalid(0);  break;
            NINF:     ftrc_invalid(1);  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

9.47 FTRV Floating-point TRansform Vector — Floating-Point Instruction

Vector Transformation

PR	Format	Summary of Operation	Instruction Code	Execution States	T Bit
0	FTRV XMTRX,FVn	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].

$$\underbrace{\begin{bmatrix} XF[0] & XF[4] & XF[8] & XF[12] \\ XF[1] & XF[5] & XF[9] & XF[13] \\ XF[2] & XF[6] & XF[10] & XF[14] \\ XF[3] & XF[7] & XF[11] & XF[15] \end{bmatrix}}_{XMTRX} \times \underbrace{\begin{bmatrix} FR[n] \\ FR[n+1] \\ FR[n+2] \\ FR[n+3] \end{bmatrix}}_{FVn} \rightarrow \underbrace{\begin{bmatrix} FR[n] \\ FR[n+1] \\ FR[n+2] \\ FR[n+3] \end{bmatrix}}_{FVn}$$

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 FRn or DRn is not updated. Appropriate processing should therefore be performed by software.

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

Format	Summary of Operation	Instruction Code	Execution States	T Bit
JMP @Rn	Rn → PC	0100nnnn00101011	2	—

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
```

Format	Summary of Operation	Instruction Code	Execution States	T Bit
JSR @Rn	PC+4 → PR, Rn → PC	0100nnnn00001011	2	—

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.
```

9.50 LDC — LoaD to Control register — System Control Instruction

Load to Control Register — (Privileged Instruction)

Format	Summary of Operation	Instruction Code	Execution States	T Bit
LDC Rm, SR	Rm → SR	0100mmmm00001110	4	LSB
LDC Rm, GBR	Rm → GBR	0100mmmm00011110	3	—
LDC Rm, VBR	Rm → VBR	0100mmmm00101110	1	—
LDC Rm, SSR	Rm → SSR	0100mmmm00111110	1	—
LDC Rm, SPC	Rm → SPC	0100mmmm01001110	1	
LDC Rm, DBR	Rm → DBR	0100mmmm11111010	1	—
LDC Rm, R0_BANK	Rm → R0_BANK	0100mmmm10001110	1	—
LDC Rm, R1_BANK	Rm → R1_BANK	0100mmmm10011110	1	—
LDC Rm, R2_BANK	Rm → R2_BANK	0100mmmm10101110	1	—
LDC Rm, R3_BANK	Rm → R3_BANK	0100mmmm10111110	1	—
LDC Rm, R4_BANK	Rm → R4_BANK	0100mmmm11001110	1	—
LDC Rm, R5_BANK	Rm → R5_BANK	0100mmmm11011110	1	—
LDC Rm, R6_BANK	Rm → R6_BANK	0100mmmm11101110	1	—
LDC Rm, R7_BANK	Rm → R7_BANK	0100mmmm11111110	1	—
LDC.L @Rm+, SR	(Rm) → SR, Rm+4 → Rm	0100mmmm00000111	4	LSB
LDC.L @Rm+, GBR	(Rm) → GBR, Rm+4 → Rm	0100mmmm00010111	3	—
LDC.L @Rm+, VBR	(Rm) → VBR, Rm+4 → Rm	0100mmmm00100111	1	—
LDC.L @Rm+, SSR	(Rm) → SSR, Rm+4 → Rm	0100mmmm00110111	1	—
LDC.L @Rm+, SPC	(Rm) → SPC, Rm+4 → Rm	0100mmmm01000111	1	—
LDC.L @Rm+, DBR	(Rm) → DBR, Rm+4 → Rm	0100mmmm11110110	1	—
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 SR, GBR, VBR, SSR, SPC, DBR, 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

```
LDCSR(int m)      /* LDC Rm,SR : Privileged */
{
  SR=R[m]&0x700083F3;
  PC+=2;
}

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

LDCVBR(int m)      /* LDC Rm,VBR : Privileged */
{
  VBR=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;
}

LDCMSR(int m)     /* LDC.L @Rm+,SR : Privileged */
{
  SR=Read_Long(R[m])&0x700083F3;
  R[m]+=4;
  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;
}
```

```
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:

- General illegal instruction exception
- Illegal slot instruction exception
- Data TLB miss exception
- Data TLB protection violation exception
- Address error

9.51 LDS LoaD to FPU System register — System Control Instruction

Load to FPU System Register

Format		Summary of Operation	Instruction Code	Execution States	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.

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 miss exception
- Data access protection exception
- Address error

Format		Summary of Operation	Instruction Code	Execution States	T Bit
LDS	Rm,MACH	Rm → MACH	0100mmmm00001010	1	—
LDS	Rm,MACL	Rm → MACL	0100mmmm00011010	1	—
LDS	Rm,PR	Rm→ PR	0100mmmm00101010	2	—
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	2	—

Description

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

Operation

```
LDSMACH(int m)     /* LDS Rm,MACH */
{
    MACH=R[m];
    PC+=2;
}

LDSMACL(int m)     /* LDS Rm,MACL */
{
    MACL=R[m];
    PC+=2;
}

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

LDSMMACH(int m)     /* LDS.L @Rm+,MACH */
{
```

```
    MACH=Read_Long(R[m]);
    R[m]+=4;
    PC+=2;
}

LDSMMACL(int m)     /* LDS.L @Rm+,MACL */
{
    MACL=Read_Long(R[m]);
    R[m]+=4;
    PC+=2;
}

LDSMPR(int 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)

9.53 LDTLB LoaD PTEH/PTEL/PTEA to TLB — System Control Instruction

Load to TLB (Privileged Instruction)

Format	Summary of Operation	Instruction Code	Execution States	T Bit
LDTLB	PTEH/PTEL/PTEA → TLB	0000000000111000	1	—

Description

This instruction loads the contents of the PTEH/PTEL/PTEA 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/PTEA 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 3, Memory Management Unit, 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 areas P0, U0, and P3 (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;
    TLB[MMUCR. URC] .SA=(PTEA & 0x00000007);
    TLB[MMUCR. URC] .TC=(PTEA & 0x00000008)>>3;

    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, PTEA registers into TLB
```

9.54 MAC.L Multiply and ACcumulate Long — Arithmetic Instruction

Double-Precision Multiply-and-Accumulate Operation

Format	Summary of Operation	Instruction Code	Execution States	T Bit
MAC.L @Rm+,@Rn+	Signed, (Rn) × (Rm) + MAC → MAC Rn + 4 → Rn, Rm + 4 → Rm	0000nnnnmmmm1111	2–5	—

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).

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      ;
```

9.55 MAC.W Multiply and ACcumulate Word — Arithmetic Instruction

Single-Precision Multiply-and-Accumulate Operation

Format		Summary of Operation	Instruction Code	Execution States	T Bit
MAC.W	@Rm+,@Rn+	Signed,	`0100nnnnmmmm1111`	2–5	—
MAC	@Rm+,@Rn+	(Rn) × (Rm) + MAC → MAC			
		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       ;
```

Format		Summary of Operation	Instruction Code	Execution States	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.

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)

9.57 MOV — MOVe constant value — Data Transfer Instruction

Immediate Data Transfer

Format		Summary of Operation	Instruction Code	Execution States	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+4) → Rn	1101nnnndddddddd	1	—

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, an illegal slot 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            ;BRA branch instruction
1014        CMP/EQ    #0,R0          ;
            .align    4              ;
1018        .data.l   H'12345678     ;
101C        .data.l   H'9ABCDEF0     ;
```

9.58 MOV Global Data Transfer — MOVe global data — Data Transfer Instruction

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

Description

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

9.59 MOV (MOVe structure data): Data Transfer Instruction

Structure Data Transfer

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

Description

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
```

9.60 MOVA Effective Address Transfer — MOVe effective Address — Data Transfer Instruction

Format	Summary of Operation	Instruction Code	Execution States	T Bit
MOVA @(disp,PC),R0	disp×4+PC+4 → R0	11000111dddddddd	1	—

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, an illegal slot 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     ;← Original PC position in MOVA instruction address calculation
        .align   4
100C    STR:.sdata  "XYZP12"
```

9.61 MOVCA.L MOVe with Cache block Allocation — Data Transfer Instruction

Cache Block Allocation

Format	Summary of Operation	Instruction Code	Execution States	T Bit
MOVCA.L R0,@Rn	R0 → (Rn)	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.

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 miss exception
- Data TLB protection violation exception
- Initial page write exception
- Address error

Format	Summary of Operation	Instruction Code	Execution States	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.

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
```

9.63 MUL.L MULtiply Long Double-Precision Multiplication — Arithmetic Instruction

Format	Summary of Operation	Instruction Code	Execution States	T Bit
MUL.L Rm,Rn	Rn×Rm → MACL	0000nnnnmmmm0111	2–5	—

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.

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
```

9.64 MULS.W Signed Multiplication — MULtiply as Signed Word — Arithmetic Instruction

Format		Summary of Operation	Instruction Code	Execution States	T Bit
MULS.W	Rm,Rn	Signed, Rn × Rm → MACL	`0010nnnnmmmm1111`	2–5	—
MULS	Rm,Rn				

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.

Operation

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

Example

```
MULS.W     R0,R 1     ;Before execution  R0 = H'FFFFFFFE, R1 = H'00005555
                      ;After execution   MACL = H'FFFF5556
STS        MACL,R0    ;Get operation result
```

9.65 MULU.W MULtiply as Unsigned Word Arithmetic Instruction

Unsigned Multiplication

Format	Summary of Operation	Instruction Code	Execution States	T Bit
MULU.W Rm,Rn	Unsigned, Rn × Rm → MACL	0010nnnnmmmm1110	2–5	—
MULU Rm,Rn				

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.

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
```

Format	Summary of Operation	Instruction Code	Execution States	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.

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
```

Format	Summary of Operation	Instruction Code	Execution States	T Bit
NEGC Rm,Rn	0 – Rm – T → Rn, borrow → T	0110nnnnmmmm1010	1	Borrow

Description

This instruction subtracts the contents of general register 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.

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
```

Format	Summary of Operation	Instruction Code	Execution States	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.

Operation

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

Example

```
NOP                ;Time equivalent to one execution state elapses.
```

9.69 NOT NOT-logical complement Logical Instruction

Bit Inversion

Format	Summary of Operation	Instruction Code	Execution States	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.

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
```

9.70 OCBI Operand Cache Block Invalidate — Data Transfer Instruction

Cache Block Invalidation

Format	Summary of Operation	Instruction Code	Execution States	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.

Operation

```
OCBI(int n)        /* OCBI @Rn */
{
  invalidate_operand_cache_block(R[n]);
  PC+=2;
}
```

Possible Exceptions:

- Data TLB miss exception
- Data TLB protection violation exception
- Initial page write exception
- Address error

Note that the above exceptions are generated even if OCBI does not operate.

9.71 OCBP Operand Cache Block Purge Data Transfer Instruction

Cache Block Purge

Format	Summary of Operation	Instruction Code	Execution States	T Bit
OCBP @Rn	Operand cache block purge	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.

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 miss exception
- Data TLB protection violation exception
- Address error

Note that the above exceptions are generated even if OCBP does not operate.

9.72 OCBWB Operand Cache Block Write Back — Data Transfer Instruction

Cache Block Write-Back

Format	Summary of Operation	Instruction Code	Execution States	T Bit
OCBWB @Rn	Operand cache block write-back	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.

Operation

```
OCBWB(int n)       /* OCBWB @Rn */
     {
       if(is_dirty_block(R[n]))  write_back(R[n]);
       PC+=2;
     }
```

Possible Exceptions:

- Data TLB miss exception
- Data TLB protection violation exception
- Address error

Note that the above exceptions are generated even if OCBWB does not operate.

Format	Summary of Operation	Instruction Code	Execution States	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	4	—

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.

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

9.74 PREF PREFetch data to cache Data Transfer Instruction

Prefetch to Data Cache

Format	Summary of Operation	nstruction Code	Execution States	T Bit
PREF @Rn	Prefetch cache block	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 address-related errors. In the event of an error, the PREF instruction is treated as an NOP (no operation) instruction.

Operation

```
PREF(int n) /* PREF */
{
         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
```

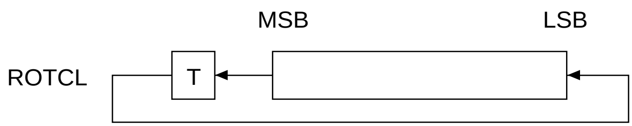
9.75 ROTCL ROTate with Carry Left Shift Instruction

One-Bit Left Rotation through T Bit

Format	Summary of Operation	Instruction Code	Execution States	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.



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
```

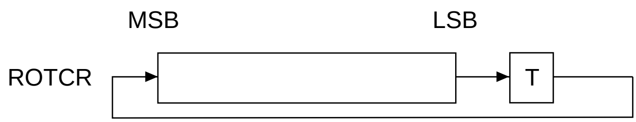
9.76 ROTCR ROTate with Carry Right Shift Instruction

One-Bit Right Rotation through T Bit

Format	Summary of Operation	Instruction Code	Execution States	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.



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;
}
```

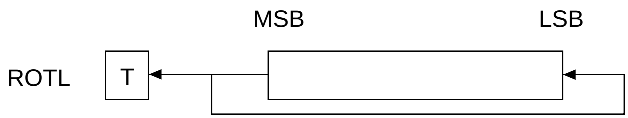
Example

ROTCR R0	;Before execution	R0 = H'00000001, T = 1
	;After execution	R0 = H'80000000, T = 1

Format	Summary of Operation	Instruction Code	Execution States	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.



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;
}
```

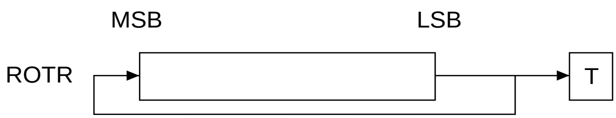
Example

ROTL R0	;Before execution	R0 = H'80000000, T = 0
	;After execution	R0 = H'00000001, T = 1

Format	Summary of Operation	Instruction Code	Execution States	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.



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
```

9.79 RTE — ReTurn from Exception — System Control Instruction

Return from Exception Handling — (Privileged Instruction)

Delayed Branch Instruction

Format	Summary of Operation	Instruction Code	Execution States	T Bit
RTE	SSR → SR, SPC→ PC	0000000000101011	5	—

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.

Format	Summary of Operation	Instruction Code	Execution States	T Bit
RTS	PR → PC	0000000000001011	2	—

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.
```

Format	Summary of Operation	Instruction Code	Execution States	T Bit
SETS	1 → S	0000000001011000	1	—

Description

This instruction sets the S bit to 1.

Operation

```
SETS( ) /* SETS */
{
        S=1;
        PC+=2;
}
```

Example

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

Format	Summary of Operation	Instruction Code	Execution States	T Bit
SETT	1 → T	0000000000011000	1	1

Description

This instruction sets the T bit to 1.

Operation

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

Example

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

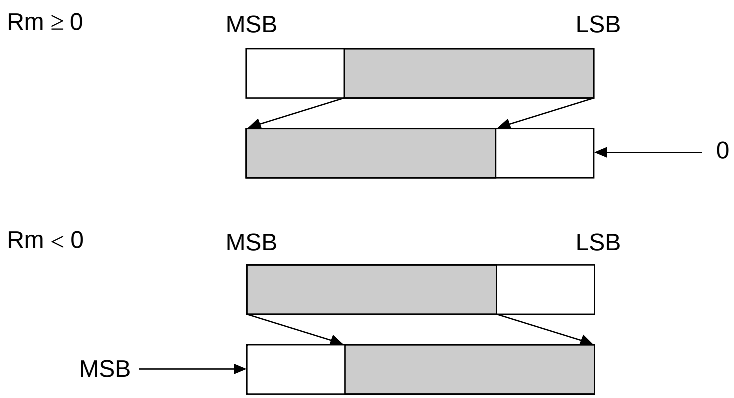
Format	Summary of Operation	Instruction Code	Execution States	T Bit
SHAD Rm, Rn	When Rm ≥ 0, Rn << Rm → Rn When Rm < 0, Rn >> Rm → [MSB → Rn]	0100nnnnmmmm1100	1	—

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.



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

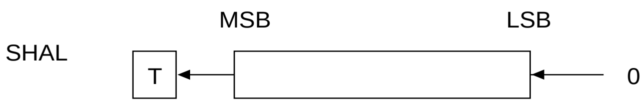
9.84 SHAL SHift Arithmetic Left Shift Instruction

One-Bit Left
Arithmetic Shift

Format	Summary of Operation	Instruction Code	Execution States	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.



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

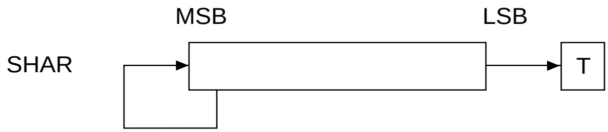
9.85 SHAR SHift Arithmetic Right Shift Instruction

One-Bit Right
Arithmetic Shift

Format	Summary of Operation	Instruction Code	Execution States	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.



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

9.86 SHLD — SHift Logical Dynamically — Shift Instruction
Dynamic Logical Shift

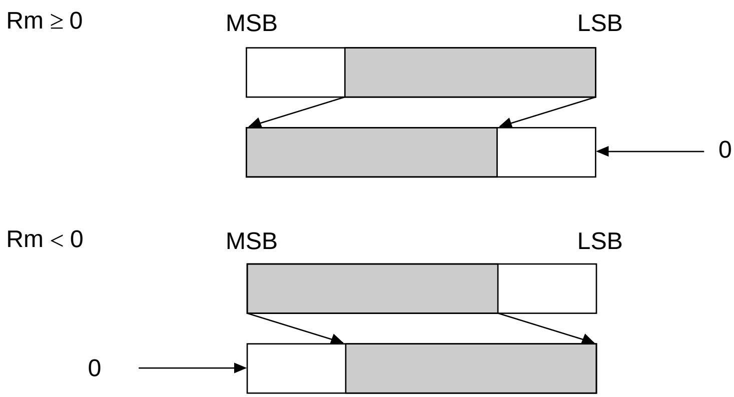
Format	Summary of Operation	Instruction Code	Execution States	T Bit
SHLD Rm, Rn	When Rm ≥ 0, Rn << Rm → Rn When Rm < 0, Rn >> Rm → [0 → Rn]	`0100nnnnmmmm1101`	1	—

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.



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
```

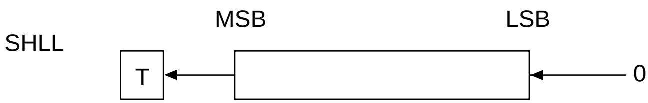
9.87 SHLL SHift Logical Left Shift Instruction

One-Bit Left
Logical Shift

Format	Summary of Operation	Instruction Code	Execution States	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.



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

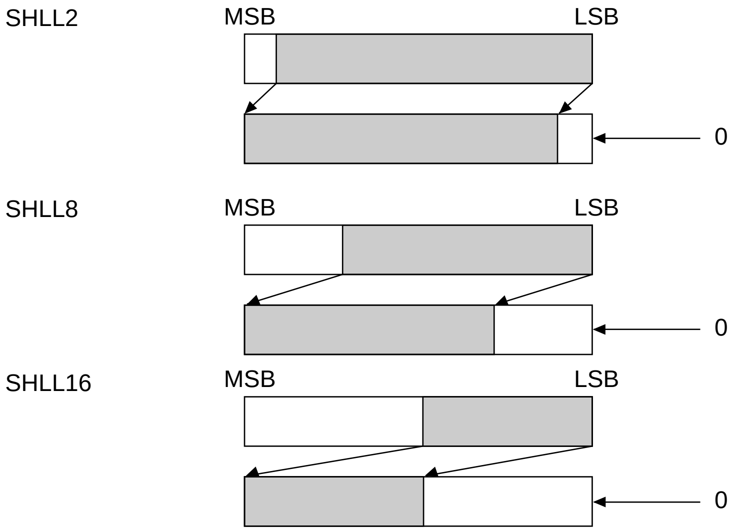
RENESAS

9.88 SHLLn n-Bit Left Logical Shift

Format		Summary of Operation	Instruction Code	Execution States	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.



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

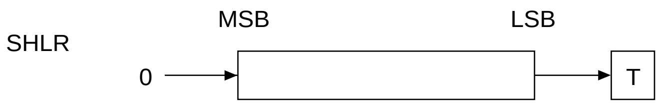
```
SHLL2   R0          ;Before execution   R0 = H'12345678
                    ;After execution    R0 = H'48D159E0
SHLL8   R0          ;Before execution   R0 = H'12345678
                    ;After execution    R0 = H'34567800
SHLL16  R0          ;Before execution   R0 = H'12345678
                    ;After execution    R0 = H'56780000
```

9.89 SHLR — SHift Logical Right (One-Bit Right Logical Shift) — Shift Instruction

Format	Summary of Operation	Instruction Code	Execution States	T Bit
SHLR Rn	0 → Rn → 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.



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

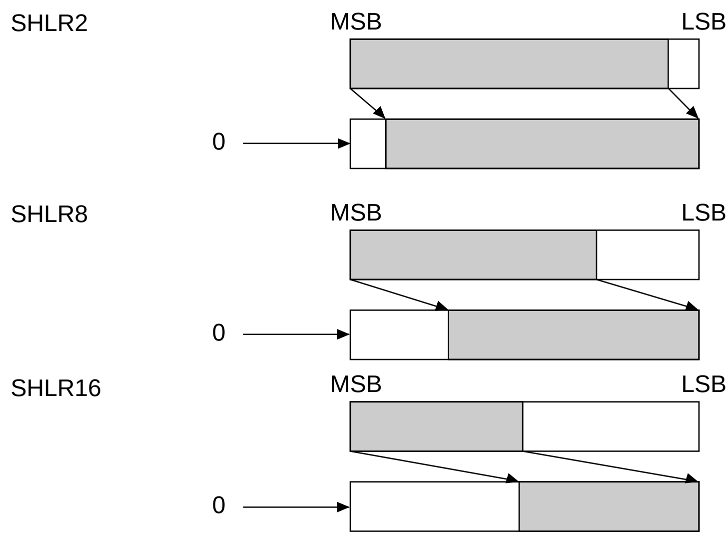
```
SHLR   R0          ;Before execution   R0 = H'80000001, T = 0
                   ;After execution    R0 = H'40000000, T = 1
```

9.90 SHLRn n bits SHift Logical Right Shift Instruction
n-Bit Right
Logical Shift

Format		Summary of Operation	Instruction Code	Execution States	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.



RENESAS

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
```

Format	Summary of Operation	Instruction Code	Execution States	T Bit
SLEEP	Sleep	0000000000011011	4	—

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 hardware manual, for details.

Operation

```
SLEEP( )    /* SLEEP */
{
    Sleep_standby();
}
```

Example

```
SLEEP                         ;Transition to power-down mode
```

Format		Summary of Operation	Instruction Code	Execution States	T Bit
STC	SR, Rn	SR → Rn	0000nnnn00000010	2	—
STC	GBR, Rn	GBR → Rn	0000nnnn00010010	2	—
STC	VBR, Rn	VBR → Rn	0000nnnn00100010	2	—
STC	SSR , Rn	SSR → Rn	0000nnnn00110010	2	—
STC	SPC, Rn	SPC → Rn	0000nnnn01000010	2	—
STC	SGR, Rn	SGR → Rn	0000nnnn00111010	3	—
STC	DBR , Rn	DBR → Rn	0000nnnn11111010	2	—
STC	R0_BANK, Rn	R0_BANK → Rn	0000nnnn10000010	2	—
STC	R1_BANK, Rn	R1_BANK → Rn	0000nnnn10010010	2	—
STC	R2_BANK, Rn	R2_BANK → Rn	0000nnnn10100010	2	—
STC	R3_BANK, Rn	R3_BANK → Rn	0000nnnn10110010	2	—
STC	R4_BANK, Rn	R4_BANK → Rn	0000nnnn11000010	2	—
STC	R5_BANK, Rn	R5_BANK → Rn	0000nnnn11010010	2	—
STC	R6_BANK, Rn	R6_BANK → Rn	0000nnnn11100010	2	—
STC	R7_BANK, Rn	R7_BANK → Rn	0000nnnn11110010	2	—
STC.L	SR, @-Rn	Rn-4 → Rn, SR → (Rn)	0100nnnn00000011	2	—
STC.L	GBR, @-Rn	Rn-4 → Rn, GBR → (Rn)	0100nnnn00010011	2	—
STC.L	VBR, @-Rn	Rn-4 → Rn, VBR → (Rn)	0100nnnn00100011	2	—
STC.L	SSR, @-Rn	Rn-4 → Rn, SSR → (Rn)	0100nnnn00110011	2	—
STC.L	SPC, @-Rn	Rn-4 → Rn, SPC → (Rn)	0100nnnn01000011	2	—
STC.L	SGR, @-Rn	Rn-4 → Rn, SGR → (Rn)	0100nnnn00110010	3	—
STC.L	DBR, @-Rn	Rn-4 → Rn, DBR → (Rn)	0100nnnn11110010	2	—
STC.L	R0_BANK, @-Rn	Rn-4 → Rn, R0_BANK → (Rn)	0100nnnn10000011	2	—
STC.L	R1_BANK, @-Rn	Rn-4 → Rn, R1_BANK → (Rn)	0100nnnn10010011	2	—
STC.L	R2_BANK, @-Rn	Rn-4 → Rn, R2_BANK → (Rn)	0100nnnn10100011	2	—
STC.L	R3_BANK, @-Rn	Rn-4 → Rn, R3_BANK → (Rn)	0100nnnn10110011	2	—
STC.L	R4_BANK, @-Rn	Rn-4 → Rn, R4_BANK → (Rn)	0100nnnn11000011	2	—
STC.L	R5_BANK, @-Rn	Rn-4 → Rn, R5_BANK → (Rn)	0100nnnn11010011	2	—
STC.L	R6_BANK, @-Rn	Rn-4 → Rn, R6_BANK → (Rn)	0100nnnn11100011	2	—
STC.L	R7_BANK, @-Rn	Rn-4 → Rn, R7_BANK → (Rn)	0100nnnn11110011	2	—

Description

This instruction stores control register SR, 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

```
STCSR(int n)      /* STC SR,Rn : Privileged */
     {
         R[n]=SR;
         PC+=2;
     }

STCGBR(int n)     /* STC GBR,Rn */
     {
         R[n]=SGR;
         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;
     }

STCMSR(int n)     /* STC.L SR,@-Rn : Privileged */
     {
         R[n]-=4;
         Write_Long(R[n],SR);
         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:

- General illegal instruction exception
- Slot illegal instruction exception
- Data TLB miss exception
- Data TLB protection violation exception
- Address error

Format		Summary of Operation	Instruction Code	Execution States	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.

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;
}
```

Possible Exceptions:

- Data TLB miss exception
- Data TLB protection violation exception
- Address error

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

Store from FPU
System Register

Format	Summary of Operation	Instruction Code	Execution States	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.

Operation

```
STS(int n, int *FPUL)           /* STS FPUL,Rn  */
{
     R[n]= *FPUL;
     PC+=2;
}
STS_SAVE(int n, int *FPUL)     /* STS.L FPUL,@-Rn  */
{
     R[n]-=4;
     Write_Long(R[n],*FPUL) ;
     PC+=2;
}
STS(int  n)          /* STS FPSCR,Rn  */
{
     R[n]=FPSCR&0x003FFFFF;
     PC+=2;
}
STS_RESTORE(int  n)  /* STS.L FPSCR,@-Rn  */
{
     R[n]-=4;
     Write_Long(R[n],FPSCR&0x003FFFFF)
```

```
        PC+=2;
    }
```

Possible Exceptions:

- Data TLB miss exception
- Data TLB protection violation exception
- Address error

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

; locatio\n 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.

Format	Summary of Operation	Instruction Code	Execution States	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.

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
```

Format	Summary of Operation	Instruction Code	Execution States	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.

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
```

9.97 SUBV SUBtract with (V flag) underflow check — Arithmetic Instruction

Binary Subtraction
with Underflow Check

Format	Summary of Operation	Instruction Code	Execution States	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.

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
```

Format	Summary of Operation	Instruction Code	Execution States	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.

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
```

Format	Summary of Operation	Instruction Code	Execution States	T Bit
TAS.B @Rn	If (Rn) = 0, 1 → T, else 0 → T 1 → MSB of (Rn)	0100nnnn00011011	5	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.

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 miss exception
- Data TLB protection violation exception
- Initial page write exception
- Address error

Exceptions are checked taking a data access by this instruction as a byte store.

Format	Summary of Operation	Instruction Code	Execution States	T Bit
TRAPA #imm	imm → TRA, PC+2 → SPC, SR → SSR, R15 → SGR, 1 → SR.MD/BL/RB, 0x160 → EXPEVT, VBR+H'00000100 → PC	11000011iiiiiiii	7	—

Description

This instruction starts trap exception handling. The values of (PC + 2), SR, and R15 are saved to SPC and SSR, 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 0x160 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.

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+H'00000100;
}
```

Format		Summary of Operation	Instruction Code	Execution States	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.

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

9.102 XOR eXclusive OR logical Logical Instruction

Exclusive

Logical OR

Format		Summary of Operation	Instruction Code	Execution States	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	4	—

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.

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

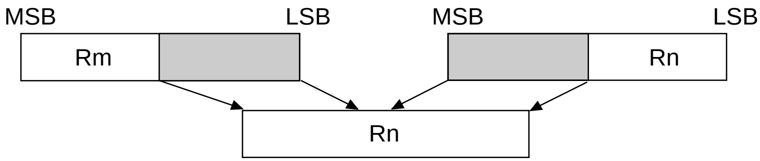
`XOR`	`R0,R1`	;Before execution	R0 = H'AAAAAAAA, R1 = H'55555555
		;After execution	R1 = H'FFFFFFFF
`XOR`	`#H'F0,R0`	;Before execution	R0 = H'FFFFFFFF
		;After execution	R0 = H'FFFFFF0F
`XOR.B`	`#H'A5,@(R0,GBR)`	;Before execution	(R0,GBR) = H'A5
		;After execution	(R0,GBR) = H'00

9.103 XTRCT eXTRaCT Middle Extraction from Linked Registers — Data Transfer Instruction

Format	Summary of Operation	Instruction Code	Execution States	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.



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

Appendix A Instruction Codes

A.1 Instruction Set by Addressing Mode

Table A.1 Instruction Set by Addressing Mode

Addressing Mode	Category	Sample Instruction		Type
No operand	—	NOP		13
Register direct	Destination operand only	MOVT	Rn	24
	Source and destination operands	ADD	Rm,Rn	56
	Transfer to control register or system register	LDC	Rm,SR	16
	Transfer from control register or system register	STS	MACH,Rn	17
Register indirect	Destination operand only	JMP	@Rn	7
	Register direct data transfer	MOV.L	Rm,@Rn	13
Register indirect with post-increment	Multiply-and-accumulate operation	MAC.W	@Rm+,@Rn+	2
	Direct data transfer from register	MOV.L	@Rm+,Rn	6
	Load to control register or system register	LDC.L	@Rm+SR	12
Register indirect with pre-decrement	Direct data transfer from register	MOV.L	Rm,@-Rn	6
	Store from control register or system register	STC.L	SR,@-Rn	13
Register indirect with displacement	Register direct data transfer	MOV.L	Rm,@(disp,Rn)	6
Indexed register indirect	Register direct data transfer	MOV.L	Rm,@(R0,Rn)	12
GBR indirect with displacement	Register direct data transfer	MOV.L	R0,@(disp,GBR)	6
Indexed GBR indirect	Immediate data transfer	AND.B	#imm,@(R0,GBR)	4
PC relative with displacement	Direct data transfer to register	MOV.L	@(disp,PC),Rn	3
PC relative using Rn	Branch instruction	BRAF	Rn	2
PC relative	Branch instruction	BRA	label	6

Table A.1 Instruction Set by Addressing Mode (cont)

Addressing Mode	Category	Sample Instruction		Type
Immediate	Load to register	FLDI0	FRn	2
	Register direct arithmetic/logic operation	ADD	#imm,Rn	7
	Exception vector specification	TRAPA	#imm	1
				Total 234

(1) No Operand

Table A.2 No Operand

Instruction	Operation	Instruction Code	Privileged	T Bit
DIV0U	0 → M/Q/T	0000000000011001	—	0
RTS	Delayed branch, PR → PC	0000000000001011	—	—
CLRMAC	0 → MACH, MACL	0000000000101000	—	—
CLRS	0 → S	0000000001001000	—	—
CLRT	0 → T	0000000000001000	—	0
LDTLB	PTEH/PTEL → TLB	0000000000111000	Privileged	—
NOP	No operation	0000000000001001	—	—
RTE	Delayed branch, SSR/SPC → SR/PC	0000000000101011	Privileged	—
SETS	1 → S	0000000001011000	—	—
SETT	1 → T	0000000000011000	—	1
SLEEP	Sleep or standby	0000000000011011	Privileged	—
FRCHG	~FPSCR.FR → FPSCR.FR	1111101111111101	—	—
FSCHG	~FPSCR.SZ → FPSCR.SZ	1111001111111101	—	—

(2) Register Direct

Table A.3 Destination Operand Only

Instruction		Operation	Instruction Code	Privileged	T Bit
MOVT	Rn	T → Rn	0000nnnn00101001	—	—
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
DT	Rn	Rn – 1 → Rn; when Rn = 0, 1 → T When Rn ≠ 0, 0 → T	0100nnnn00010000	—	Comparison result
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
SHAL	Rn	T ← Rn ← 0	0100nnnn00100000	—	MSB
SHAR	Rn	MSB → Rn → T	0100nnnn00100001	—	LSB
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	—	—
FABS	FRn	FRn & H'7FFF FFFF → FRn	1111nnnn01011101	—	—
FNEG	FRn	FRn ∧ H'80000000 → FRn	1111nnnn01001101	—	—
FSQRT	FRn	√FRn → FRn	1111nnnn01101101	—	—
FABS	DRn	DRn & H'7FFF FFFF FFFF FFFF → DRn	1111nnn001011101	—	—
FNEG	DRn	DRn ^ H'8000 0000 0000 0000 → DRn	1111nnn001001101	—	—
FSQRT	DRn	√DRn → DRn	1111nnn001101101	—	—

Table A.4 Source and Destination Operands

Instruction		Operation	Instruction Code	Privileged	T Bit
MOV	Rm,Rn	Rm → Rn	`0110nnnnmmmm0011`	—	—
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`	—	—
ADD	Rm,Rn	Rn + Rm → Rn	`0011nnnnmmmm1100`	—	—
ADDC	Rm,Rn	Rn + Rm + T → Rn, carry → T	`0011nnnnmmmm1110`	—	Carry
ADDV	Rm,Rn	Rn + Rm → Rn, overflow → T	`0011nnnnmmmm1111`	—	Overflow
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/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
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`	—	—
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`	—	—

Table A.4 Source and Destination Operands (cont)

Instruction		Operation	Instruction Code	Privileged	T Bit
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
AND	Rm,Rn	Rn & Rm → Rn	0010nnnnmmmm1001	—	—
NOT	Rm,Rn	~Rm → Rn	0110nnnnmmmm0111	—	—
OR	Rm,Rn	Rn | Rm → Rn	0010nnnnmmmm1011	—	—
TST	Rm,Rn	Rn & Rm; when result = 0, 1 → T Otherwise, 0 → T	0010nnnnmmmm1000	—	Test result
XOR	Rm,Rn	Rn ∧ Rm → Rn	0010nnnnmmmm1010	—	—
SHAD	Rm,Rn	When Rn ≥ 0, Rn << Rm → Rn When Rn < 0, Rn >> Rm → [MSB → Rn]	0100nnnnmmmm1100	—	—
SHLD	Rm,Rn	When Rn ≥ 0, Rn << Rm → Rn When Rn < 0, Rn >> Rm → [0 → Rn]	0100nnnnmmmm1101	—	—
FMOV	FRm,FRn	FRm → FRn	1111nnnnmmmm1100	—	—
FMOV	DRm,DRn	DRm → DRn	1111nnn0mmm01100	—	—
FADD	FRm,FRn	FRn + FRm → FRn	1111nnnnmmmm0000	—	—
FCMP/EQ	FRm,FRn	When FRn = FRm, 1 → T Otherwise, 0 → T	1111nnnnmmmm0100	—	Comparison result
FCMP/GT	FRm,FRn	When FRn > FRm, 1 → T Otherwise, 0 → T	1111nnnnmmmm0101	—	Comparison result
FDIV	FRm,FRn	FRn/FRm → FRn	1111nnnnmmmm0011	—	—
FMAC	FR0,FRm,FRn	FR0*FRm + FRn → FRn	1111nnnnmmmm1110	—	—
FMUL	FRm,FRn	FRn*FRm → FRn	1111nnnnmmmm0010	—	—
FSUB	FRm,FRn	FRn – FRm → FRn	1111nnnnmmmm0001	—	—
FADD	DRm,DRn	DRn + DRm → DRn	1111nnn0mmm00000	—	—
FCMP/EQ	DRm,DRn	When DRn = DRm, 1 → T Otherwise, 0 → T	1111nnn0mmm00100	—	Comparison result

Table A.4 Source and Destination Operands (cont)

Instruction		Operation	Instruction Code	Privileged	T Bit
FCMP/GT	DRm,DRn	When DRn > DRm, 1 → T Otherwise, 0 → T	1111nnn0mmm00101	—	Comparison result
FDIV	DRm,DRn	DRn /DRm → DRn	1111nnn0mmm00011	—	—
FMUL	DRm,DRn	DRn *DRm → DRn	1111nnn0mmm00010	—	—
FSUB	DRm,DRn	DRn – DRm → DRn	1111nnn0mmm00001	—	—
FMOV	DRm,XDn	DRm → XDn	1111nnn1mmm01100	—	—
FMOV	XDm,DRn	XDm → DRn	1111nnn0mmm11100	—	—
FMOV	XDm,XDn	XDm → XDn	1111nnn1mmm11100	—	—
FIPR	FVm,FVn	inner_product [FVm, FVn] → FR[n+3]	1111nnmm11101101	—	—
FTRV	XMTRX,FVn	transform_vector [XMTRX, FVn] → FVn	1111nn0111111101	—	—

Table A.5 Transfer to Control Register or System Register

Instruction		Operation	Instruction Code	Privileged	T Bit
LDC	Rm,SR	Rm → SR	0100mmmm00001110	Privileged	LSB
LDC	Rm,GBR	Rm → GBR	0100mmmm00011110	—	—
LDC	Rm,VBR	Rm → VBR	0100mmmm00101110	Privileged	—
LDC	Rm,SSR	Rm → SSR	0100mmmm00111110	Privileged	—
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	—
LDS	Rm,MACH	Rm → MACH	0100mmmm00001010	—	—
LDS	Rm,MACL	Rm → MACL	0100mmmm00011010	—	—
LDS	Rm,PR	Rm → PR	0100mmmm00101010	—	—
FLDS	FRm,FPUL	FRm → FPUL	1111mmmm00011101	—	—
FTRC	FRm,FPUL	(long) FRm → FPUL	1111mmmm00111101	—	—
FCNVDS	DRm,FPUL	double_to_ float[DRm] → FPUL	1111mmm010111101	—	—
FTRC	DRm,FPUL	(long) DRm → FPUL	1111mmm000111101	—	—
LDS	Rm,FPSCR	Rm → FPSCR	0100mmmm01101010	—	—
LDS	Rm,FPUL	Rm → FPUL	0100mmmm01011010	—	—

Table A.6 Transfer from Control Register or System Register

Instruction		Operation	Instruction Code	Privileged	T Bit
STC	SR,Rn	SR → Rn	0000nnnn00000010	Privileged	—
STC	GBR,Rn	GBR → Rn	0000nnnn00010010	—	—
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	—
STS	MACH,Rn	MACH → Rn	0000nnnn00001010	—	—
STS	MACL,Rn	MACL → Rn	0000nnnn00011010	—	—
STS	PR,Rn	PR → Rn	0000nnnn00101010	—	—
FSTS	FPUL,FRn	FPUL → FRn	1111nnnn00001101	—	—
FLOAT	FPUL,FRn	(float) FPUL → FRn	1111nnnn00101101	—	—
FCNVSD	FPUL,DRn	float_to_ double [FPUL] → DRn	1111nnn010101101	—	—
FLOAT	FPUL,DRn	(float)FPUL → DRn	1111nnn000101101	—	—
STS	FPSCR,Rn	FPSCR → Rn	0000nnnn01101010	—	—
STS	FPUL,Rn	FPUL → Rn	0000nnnn01011010	—	—

(3) Register Indirect

Table A.7 Destination Operand Only

Instruction		Operation	Instruction Code	Privileged	T Bit
TAS.B	@Rn	When (Rn) = 0, 1 → T Otherwise, 0 → T In both cases, 1 → MSB of (Rn)	0100nnnn00011011	—	Test result
JMP	@Rn	Delayed branch, Rn → PC	0100nnnn00101011	—	—
JSR	@Rn	Delayed branch, PC + 4 → PR, Rn → PC	0100nnnn00001011	—	—
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	—	—

Table A.8 Register Direct Data Transfer

Instruction		Operation	Instruction Code	Privileged	T Bit
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	—	—
MOVCA.L	R0,@Rn	R0 → (Rn) (without fetching cache block)	0000nnnn11000011	—	—
FMOV.S	@Rm,FRn	(Rm) → FRn	1111nnnnmmmm1000	—	—
FMOV.S	FRm,@Rn	FRm → (Rn)	1111nnnnmmmm1010	—	—
FMOV	@Rm,DRn	(Rm) → DRn	1111nnn0mmmm1000	—	—
FMOV	DRm,@Rn	DRm → (Rn)	1111nnnnmmm01010	—	—
FMOV	@Rm,XDn	(Rm) → XDn	1111nnn1mmmm1000	—	—
FMOV	XDm,@Rn	XDm → (Rn)	1111nnnnmmm11010	—	—

(4) Register Indirect with Post-Increment

Table A.9 Multiply-and-Accumulate Operation

Instruction		Operation	Instruction Code	Privileged	T Bit
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	—	—

Table A.10 Direct Data Transfer from Register

Instruction		Operation	Instruction Code	Privileged	T Bit
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	—	—
FMOV.S	@Rm+,FRn	(Rm) → FRn, Rm + 4 → Rm	1111nnnnmmmm1001	—	—
FMOV	@Rm+,DRn	(Rm) → DRn, Rm + 8 → Rm	1111nnn0mmmm1001	—	—
FMOV	@Rm+,XDn	(Rm) → XDn, Rm + 8 → Rm	1111nnn1mmmm1001	—	—

Table A.11 Load to Control Register or System Register

Instruction		Operation	Instruction Code	Privileged	T Bit
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+,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.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	—	—
LDS.L	@Rm+,FPSCR	(Rm) → FPSCR, Rm+4 → Rm	0100mmmm01100110	—	—
LDS.L	@Rm+,FPUL	(Rm) → FPUL, Rm+4 → Rm	0100mmmm01010110	—	—

(5) Register Indirect with Pre-Decrement

Table A.12 Direct Data Transfer from Register

Instruction		Operation	Instruction Code	Privileged	T Bit
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	—	—
FMOV.S	FRm,@-Rn	Rn-4 → Rn, FRm → (Rn)	1111nnnnmmmm1011	—	—
FMOV	DRm,@-Rn	Rn-8 → Rn, DRm → (Rn)	1111nnnnmmm01011	—	—
FMOV	XDm,@-Rn	Rn – 8 → Rn, XDm → (Rn)	1111nnnnmmm11011	—	—

Table A.13 Store from Control Register or System Register

Instruction		Operation	Instruction Code	Privileged	T Bit
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.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	—	—
STS.L	FPSCR,@-Rn	Rn – 4 → Rn, FPSCR → (Rn)	0100nnnn01100010	—	—
STS.L	FPUL,@-Rn	Rn – 4 → Rn, FPUL → (Rn)	0100nnnn01010010	—	—

(6) Register Indirect with Displacement

Table A.14 Register Direct Data Transfer

Instruction		Operation	Instruction Code	Privileged	T Bit
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	—	—

(7) Indexed Register Indirect

Table A.15 Register Direct Data Transfer

Instruction		Operation	Instruction Code	Privileged	T Bit
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	—	—
FMOV.S	@(R0,Rm),FRn	(R0 + Rm) → FRn	1111nnnnmmmm0110	—	—
FMOV.S	FRm,@(R0,Rn)	FRm → (R0 + Rn)	1111nnnnmmmm0111	—	—
FMOV	@(R0,Rm),DRn	(R0 + Rm) → DRn	1111nnn0mmmm0110	—	—
FMOV	DRm,@(R0,Rn)	DRm → (R0 + Rn)	1111nnnnmmm00111	—	—
FMOV	@(R0,Rm),DRn	(R0 + Rm) → DRn	1111nnn1mmmm0110	—	—
FMOV	XDm,@(R0,Rn)	XDm → (R0+Rn)	1111nnnnmmm10111	—	—

(8) GBR Indirect with Displacement

Table A.16 Register Direct Data Transfer

Instruction		Operation	Instruction Code	Privileged	T Bit
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	—	—

(9) Indexed GBR Indirect

Table A.17 Immediate Data Transfer

Instruction		Operation	Instruction Code	Privileged	T Bit
AND.B	#imm,@(R0,GBR)	(R0 + GBR) & imm → (R0 + GBR)	`11001101iiiiiiii`	—	—
OR.B	#imm,@(R0,GBR)	(R0 + GBR) | imm → (R0 + GBR)	`11001111iiiiiiii`	—	
TST.B	#imm,@(R0,GBR)	(R0 + GBR) & imm; when result = 0, 1 → T Otherwise, 0 → T	`11001100iiiiiiii`	—	Test result
XOR.B	#imm,@(R0,GBR)	(R0 + GBR) ∧ imm → (R0 + GBR)	`11001110iiiiiiii`	—	—

(10) PC Relative with Displacement

Table A.18 Direct Data Transfer to Register

Instruction		Operation	Instruction Code	Privileged	T Bit
MOV.W	@(disp,PC),Rn	(disp × 2 + PC + 4) → sign extension → Rn	`1001nnnndddddddd`	—	—
MOV.L	@(disp,PC),Rn	(disp × 4 + PC & H'FFFFFFFC + 4) → Rn	`1101nnnndddddddd`	—	—
MOVA	@(disp,PC),R0	disp × 4 + PC & H'FFFFFFFC + 4 → R0	`11000111dddddddd`	—	—

(11) PC Relative Using Rn

Table A.19 Branch Instructions

Instruction		Operation	Instruction Code	Privileged	T Bit
BRAF	Rn	Rn + PC + 4 → PC	`0000nnnn00100011`	—	—
BSRF	Rn	Delayed branch, PC + 4 → PR, Rn + PC + 4 → PC	`0000nnnn00000011`	—	—

(12) PC Relative

Table A.20 Branch Instructions

Instruction		Operation	Instruction Code	Privileged	T Bit
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	—	—
BSR	label	Delayed branch, PC + 4 → PR, disp × 2 + PC + 4 → PC	1011dddddddddddd	—	—

(13) Immediate

Table A.21 Load to Register

Instruction		Operation	Instruction Code	Privileged	T Bit
FLDI0	FRn	H'00000000 → FRn	1111nnnn10001101	—	—
FLDI1	FRn	H'3F800000 → FRn	1111nnnn10011101	—	—

Table A.22 Register Direct Arithmetic/Logic Operation

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

Table A.23 Exception Vector Specification

Instruction	Operation	Instruction Code	Privileged	T Bit
TRAPA #imm	PC + 2 → SPC, SR → SSR, #imm << 2 → TRA, H'160 → EXPEVT, VBR + H'0100 → PC	`11000011iiiiiiii`	—	—

Appendix B Instruction Prefetch Side Effects

The SH-4 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 20-byte area of any memory space. If program code is located in these areas, the memory area will be exceeded and a bus access for instruction pre-reading may be initiated. A case in which this is a problem is shown below.

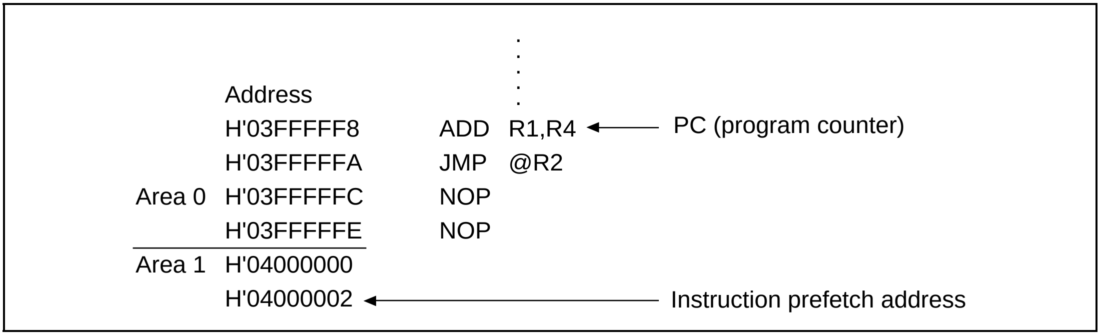


Figure B.1 Instruction Prefetch

Figure B.1 presupposes a case in which the instruction (ADD) indicated by the program counter (PC) and the address H'0400002 instruction prefetch are executed simultaneously. It is also assumed that the program branches to an area outside area 1 after executing the following JMP instruction and delay slot instruction.

In this case, the program flow is unpredictable, and a bus access (instruction prefetch) to area 1 may be initiated.

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, hangup 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 20 bytes of any area.

SH-4 Programming Manual

Publication Date: 1st Edition, August 1998
5th Edition, April 2001
Published by: Customer Service Division
Semiconductor & Integrated Circuits
Hitachi, Ltd.
Edited by: Technical Documentation Group
Hitachi Kodaira Semiconductor Co., Ltd.