

SH-4

Programming Manual

Preliminary

HITACHI

Version 1.2.1

2/17/98

Hitachi, Ltd.

Notice

When using this document, keep the following in mind:

1. This document may, wholly or partially, be subject to change without notice.
2. All rights are reserved: No one is permitted to reproduce or duplicate, in any form, the whole or part of this document without Hitachi's permission.
3. Hitachi will not be held responsible for any damage to the user that may result from accidents or any other reasons during operation of the user's unit according to this document.
4. Circuitry and other examples described herein are meant merely to indicate the characteristics and performance of Hitachi's semiconductor products. Hitachi assumes no responsibility for any intellectual property claims or other problems that may result from applications based on the examples described herein.
5. No license is granted by implication or otherwise under any patents or other rights of any third party or Hitachi, Ltd.
6. **MEDICAL APPLICATIONS:** Hitachi's products are not authorized for use in MEDICAL APPLICATIONS without the written consent of the appropriate officer of Hitachi's sales company. Such use includes, but is not limited to, use in life support systems. Buyers of Hitachi's products are requested to notify the relevant Hitachi sales offices when planning to use the products in MEDICAL APPLICATIONS.

Contents

Section 1 Overview	1
1.1 SH-4 Features	1
1.2 Block Diagram.....	5
Section 2 Data Formats and Registers.....	7
2.1 Data Formats.....	7
2.2 General Registers.....	7
2.3 Floating-Point Registers	9
2.4 Control Registers	11
2.5 System Registers.....	12
2.6 Memory-Mapped Registers	14
Section 3 Memory Management Unit (MMU).....	15
3.1 Overview	15
3.1.1 Features	15
3.1.2 The Role of the MMU.....	15
3.1.3 Register Configuration.....	18
3.1.4 Caution	18
3.2 Register Descriptions	19
3.3 Memory Space.....	23
3.3.1 Physical Memory Space.....	23
3.3.2 External Memory Space.....	25
3.3.3 Virtual Memory Space.....	26
3.3.4 On-Chip RAM Space	27
3.3.5 Address Translation	27
3.3.6 Single Virtual Memory Mode and Multiple Virtual Memory Mode.....	28
3.3.7 Address Space Identifier (ASID).....	28
3.4 TLB Functions	28
3.4.1 Unified TLB (UTLB) Configuration	28
3.4.2 Instruction TLB (ITLB) Configuration.....	31
3.4.3 Address Translation Method	32
3.5 MMU Functions.....	34
3.5.1 MMU Hardware Management.....	34
3.5.2 MMU Software Management.....	34
3.5.3 MMU Instruction (LDTLB)	34
3.5.4 Hardware ITLB Miss Handling.....	35
3.5.5 Avoiding Synonym Problems.....	35

3.6 MMU Exceptions.....	36
3.6.1 Instruction TLB Multiple Hit Exception	36
3.6.2 Instruction TLB Miss Exception	37
3.6.3 Instruction TLB Protection Violation Exception	38
3.6.4 Data TLB Multiple Hit Exception.....	39
3.6.5 Data TLB Miss Exception.....	39
3.6.6 Data TLB Protection Violation Exception.....	40
3.6.7 Initial Page Write Exception.....	41
3.7 Memory-Mapped TLB Configuration	42
3.7.1 ITLB Address Array	42
3.7.2 ITLB Data Array 1	43
3.7.3 ITLB Data Array 2	44
3.7.4 UTLB Address Array.....	45
3.7.5 UTLB Data Array 1	46
3.7.6 UTLB Data Array 2.....	47
 Section 4 Caches.....	 49
4.1 Overview	49
4.1.1 Features	49
4.1.2 Register Configuration.....	50
4.2 Register Descriptions	50
4.3 Operand Cache (OC).....	53
4.3.1 Configuration.....	53
4.3.2 Read Operation.....	54
4.3.3 Write Operation.....	55
4.3.4 Write-Back Buffer	56
4.3.5 Write-Through Buffer.....	57
4.3.6 RAM Mode.....	57
4.3.7 OC Index Mode	57
4.3.8 Coherency between Cache and External Memory	58
4.3.9 Prefetch Operation.....	58
4.4 Instruction Cache (IC).....	59
4.4.1 Configuration.....	59
4.4.2 Read Operation.....	60
4.4.3 IC Index Mode.....	61
4.5 Memory-Mapped Cache Configuration	61
4.5.1 IC Address Array	61
4.5.2 IC Data Array	62
4.5.3 OC Address Array	63
4.5.4 OC Data Array.....	65

4.6 Store Queue	66
4.6.1 SQ Configuration.....	66
4.6.2 SQ Write.....	66
4.6.3 Transfer to External Memory	66
4.6.4 SQ Protection.....	67
Section 5 Exceptions	69
5.1 Overview	69
5.1.1 Features	69
5.1.2 Control Registers	69
5.2 Exception Handling Functions	70
5.2.1 Exception Handling Flow	70
5.2.2 Exception Handling Vector Addresses	71
5.3 Exception Types and Priorities.....	72
5.4 Exception Flow	75
5.4.1 Exception Flow.....	75
5.4.2 Exception Requests and BL Bit.....	76
5.4.3 Return from Exception Handling	76
5.5 Description of Exceptions	77
5.5.1 Resets	77
5.5.2 General Exceptions.....	82
5.5.3 Interrupts	92
5.5.4 Priority Order with Multiple Exceptions	95
Section 6 Floating-Point Unit	97
6.1 Overview	97
6.2 Data Formats.....	97
6.2.1 Floating-Point Format.....	97
6.2.2 Non-Numbers (NaN).....	99
6.2.3 Denormalized Numbers	100
6.3 Registers	101
6.3.1 Floating-Point Registers.....	101
6.3.2 Floating-Point Unit Status/Control Register (FPSCR)	103
6.3.3 Floating-Point Communication Register (FPUL).....	104
6.4 Rounding	104
6.5 Floating-Point Exceptions	105
6.6 Graphic Support fiture.....	106
6.6.1 Geometric Instructions.....	106
6.6.2 two single precision data transfer	107

Section 7 Instruction Set	109
7.1 Execution Environment.....	109
7.2 Addressing Modes.....	110
7.3 Instruction Set.....	113
Section 8. Pipelining	125
8.1 Pipeline.....	125
8.2 Parallel-Executability.....	130
8.3 Execution Cycle and Pipeline Stall	133
Section 9 Power-Down Modes	149
9.1 Overview	149
9.1.1 Types of Power-Down Modes.....	149
9.1.2 Register Configuration.....	150
9.2 Register Descriptions	151
9.2.1 Standby Control Register (STBCR)	151
9.2.2 Supporting Module Pin High Impedance Control.....	153
9.2.3 Supporting Module Pin Pull-Up Control	153
9.3 Sleep Mode.....	154
9.3.1 Transition to Sleep Mode.....	154
9.3.2 Exit from Sleep Mode.....	154
9.4 Standby Mode.....	154
9.4.1 Transition to Standby Mode.....	154
9.4.2 Exit from Standby Mode.....	155
9.4.3 Clock Pause Function	156
9.5 Module Standby Function	156
9.5.1 Transition to Module Standby Function	156
9.5.2 Exit from Module Standby Function	157
Section 10 Instruction Description	159
ADD (Add Binary): Arithmetic Instruction.....	172
ADDC (Add with Carry): Arithmetic Instruction	173
ADDV (Add with V Flag Overflow Check): Arithmetic Instruction.....	174
AND (AND Logical): Logic Operation Instruction	175
BF (Branch if False): Branch Instruction.....	176
BF/S (Branch if False with Delay Slot): Branch Instruction Class: Delayed branch instruction.....	177
BRA (Branch): Branch Instruction Class: Delayed branch instruction.....	179
BRAF (Branch Far): Branch Instruction Class: Delayed branch instruction.....	180
BSR (Branch to Subroutine): Branch Instruction Class: Delayed branch instruction.....	181
BSRF (Branch to Subroutine Far): Branch Instruction Class: Delayed branch instruction	183
BT (Branch if True): Branch Instruction	184
BT/S (Branch if True with Delay Slot): Branch Instruction.....	185

CLRMAC (Clear MAC Register): System Control Instruction	186
CLRS (Clear S Bit): System Control Instruction	186
CLRT (Clear T Bit): System Control Instruction.....	187
CMP/cond (Compare Conditionally): Arithmetic Instruction	188
DIV0S (Divide Step 0 as Signed): Arithmetic Instruction	192
DIV0U (Divide Step 0 as Unsigned): Arithmetic Instruction.....	193
DIV1 (Divide Step 1): Arithmetic Instruction	193
DMULS.L (Double-Length Multiply as Signed): Arithmetic Instruction.....	198
DMULU.L (Double-Length Multiply as Unsigned): Arithmetic Instruction	199
DT (Decrement and Test): Arithmetic Instruction.....	200
EXTS (Extend as Signed): Data Transfer Instruction	201
EXTU (Extend as Unsigned): Data Transfer Instruction.....	202
FABS (Floating Point Absolute Value): Floating-Point Instruction	203
FADD (Floating Point Add): Floating-Point Instruction	203
FCMP (Floating Point Compare): Floating-Point Instruction.....	206
FCNVDS (Floating Point Convert Double to Single Precision): Floating-Point Instruction	209
FCNVSD (Floating Point Convert Single to Double Precision): Floating-Point Instruction	211
FDIV (Floating Point Divide): Floating-Point Instruction.....	213
FIPR (Floating Point Inner Product): Floating-Point Instruction.....	216
FLDI0 (Floating Point Load 0.0): Floating-Point Instruction.....	218
FLDI1 (Floating Point Load 1.0): Floating-Point Instruction.....	219
FLDS (Floating Point Load to System Register): Floating-Point Instruction.....	220
FLOAT (Floating Point Convert from Integer): Floating-Point Instruction.....	221
FMAC (Floating Point Multiply and Accumulate): Floating-Point Instruction	222
FMOV (Floating Point Move): Floating-Point Instruction	225
FMOV (Floating Point Move Extension): Floating-Point Instruction.....	228
FMUL (Floating Point Multiply): Floating-Point Instruction.....	230
FNEG (Floating Point Negate): Floating-Point Instruction	232
FRCHG (FR-Bit Change): Floating-Point Instruction.....	233
FSCHG (SZ-Bit Change): Floating-Point Instruction	234
FSQRT (Floating Point Square Root): Floating-Point Instruction.....	235
FSTS (Floating Point Store System Register): Floating-Point Instruction	237
FSUB (Floating Point Subtract): Floating-Point Instruction.....	238
FTRC (Floating Point Truncate and Convert to Integer): Floating-Point Instruction.....	240
FTRV (Floating Point Transform Vector): Floating-Point Instruction	243
JMP (Jump): Branch Instruction.....	245
JSR (Jump to Subroutine): Branch Instruction.....	246
LDC (Load to Control Register): System Control Instruction (Privileged Instruction).....	247
LDS (Load to FPU System Register): System Control Instruction.....	251
LDS (Load to System Register): System Control Instruction.....	254
LDTLB (Load PTEH/PTEL to TLB): System Control Instruction (Privileged Only).....	256
MAC.L (Multiply and Accumulate Long): Arithmetic Instruction	257

MOV (Move): Data Transfer Instruction	261
MOV (Move Constant Value): Data Transfer Instruction	265
MOV (Move Global Data): Data Transfer Instruction	267
MOV (Move Displacement Addressing): Data Transfer Instruction	270
MOVA (Move Effective Address): Data Transfer Instruction	273
MOVCA.L (Move with Cache Block Allocation): Data Transfer Instruction	274
MOVT (Move T Bit): Data Transfer Instruction	275
MUL.L (Multiply Long): Arithmetic Instruction.....	276
MULS.W (Multiply as Signed Word): Arithmetic Instruction.....	277
MULU.W (Multiply as Unsigned Word): Arithmetic Instruction	278
NEG (Negate): Arithmetic Instruction	279
NEGC (Negate with Carry): Arithmetic Instruction	280
NOP (No Operation): System Control Instruction	281
NOT (NOT—Logical Complement): Logic Operation Instruction	282
OCBI (Operand Cache Block Invalidate): Data Transfer Instruction	283
OCBP (Operand Cache Block Purge): Data Transfer Instruction.....	284
OCBWB (Operand Cache Block Write Back): Data Transfer Instruction.....	285
OR (OR Logical) Logic Operation Instruction	286
PREF (Prefetch Data to Cache).....	288
ROTCL (Rotate with Carry Left): Shift Instruction.....	289
ROTCR (Rotate with Carry Right): Shift Instruction	290
ROTL (Rotate Left): Shift Instruction.....	291
ROTR (Rotate Right): Shift Instruction.....	292
RTE (Return from Exception): System Control Instruction (Privileged Only)	293
RTS (Return from Subroutine): Branch Instruction.....	294
SETS (Set S Bit): System Control Instruction	295
SETT (Set T Bit): System Control Instruction.....	296
SHAD (Shift Arithmetic Dynamically): Shift Instruction	297
SHAL (Shift Arithmetic Left): Shift Instruction.....	299
SHAR (Shift Arithmetic Right): Shift Instruction.....	300
SHLD (Shift Logical Dynamically): Shift Instruction	301
SHLL (Shift Logical Left): Shift Instruction	303
SHLLn (Shift Logical Left n Bits): Shift Instruction	304
SHLR (Shift Logical Right): Shift Instruction.....	306
SHLRn (Shift Logical Right n Bits): Shift Instruction.....	307
SLEEP (Sleep): System Control Instruction (Privileged Only)	309
STC (Store Control Register): System Control Instruction	310
STS (Store from FPU System Register): System Control Instruction.....	315
STS (Store System Register): System Control Instruction	318
SUB (Subtract Binary): Arithmetic Instruction	320
SUBC (Subtract with Carry): Arithmetic Instruction.....	321
SUBV (Subtract with V Flag Underflow Check): Arithmetic Instruction	322
SWAP (Swap): Data Transfer Instruction.....	323

TAS (Test and Set): Logic Operation Instructions.....	324
TRAPA (Trap Always): System Control Instruction.....	325
TST (Test Logical): Logic Operation Instruction	326
XOR (Exclusive OR Logical): Logic Operation Instruction	328
XTRCT (Extract): Data Transfer Instruction	330

Section 1 Overview

1.1 SH-4 Features

The SH-4 is a 32-bit RISC (reduced instruction set computer) microcomputer, featuring object code upward-compatibility with SH-1, SH-2, SH-3, and SH-3E microcomputers. It includes an 8-kbyte instruction cache, a 16-kbyte operand cache with a choice of copy-back or write-through mode, and an MMU (memory management unit) with a 4-entry full-associative instruction TLB (translation lookaside buffer) and a 64-entry full-associative unified TLB.

The SH-4 has an on-chip bus state controller (BSC) that allows direct connection to DRAM, SDRAM, and SGRAM without external circuitry. 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
LSI	<ul style="list-style-type: none">• Frequency: 167Mhz• Performance:<ul style="list-style-type: none">— 300MIPS (167Mhz)— 1.17GFLOPS (167Mhz)• Superscalar: Paralell execution of two instructions• Voltage: 2.5V(internal), 3.3V(IO)• Package: 208-pin QFP• External bus:<ul style="list-style-type: none">— Separate 26-bit address+64-bit data— 1/2, 1/3, 1/4, 1/6, 1/8 external bus frequency (vs. internal)
CPU	<ul style="list-style-type: none">• Original Hitachi SH architecture• 32-bit internal data bus• General-register files:<ul style="list-style-type: none">— 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 compatibility with the SH series):<ul style="list-style-type: none">— Instruction length: 16-bit fixed length for improved code efficiency— Load-store architecture— Delayed branch instructions— Conditional execution— Instruction set based on C language• Two-way superscalar execution with FPU• Instruction execution time: two instructions/cycle at maximum• Logical address space: 4 Gbytes (448-Mbyte physical memory space)• Space identifier ASID: 8 bits, 256 logical address spaces• On-chip multiplier• Five-stage pipeline

Table 1.1 SH-4 Features (Cont.)

Item	Features
FPU	<ul style="list-style-type: none"> • Integrated floating-point co-processor • Single (32b) and double (64b) precision supported • IEEE754 compliant data type and exception supported • Round mode: round-to-nearest, round-to-zero • Denorm mode: flush-to-zero, conform IEEE754 • Floating-point registers: 32 bit x 16 word x 2 bank • Single x 16 word / Double x 8 word / x 2 bank • 32-bit CPU-FPU interface register(FPUL) • FMAC (multiply and accumulate) instruction supported • FDIV (divide) / FSQRT (square root) instructions supported • FLD0 / FLD1 (load constant 0/1) instructions supported • Instruction execution time <ul style="list-style-type: none"> — Latency (FMAC/FADD/FSUB/FMUL): 3 cycles (single), 8 cycles (double) — Pitch (FMAC/FADD/FSUB/FMUL): 1 cycle (single), 6 cycles (double) — Note: FMAC is supported for only single precision. • 3D graphic instructions: <ul style="list-style-type: none"> — 4-dimensional vector transformation and matrix operation (FTRV), 4 cycles (pitch), 7 cycles (latency) — Inner product of 4-dimensional vectors (FIPR), 1cycle (pitch), 4 cycles (latency) • Five-stage pipeline
Clock pulse generator (CPG)	<ul style="list-style-type: none"> • Clock mode: <ul style="list-style-type: none"> — CPU frequency: 1, 1/2, 1/4, 1/8 (vs 167MHz) — Bus frequency: 1/2, 1/3, 1/4, 1/6, 1/8 (vs 167MHz) — Peripheral frequency: 1/2, 1/3, 1/4, 1/6, 1/8 (vs 167Mhz) • Power-down modes: <ul style="list-style-type: none"> — Sleep mode — Standby mode — Module stop function • One watch-dog timer channel
Memory management unit (MMU)	<ul style="list-style-type: none"> • 4 Gbytes of address space, 256 address spaces (ASID 8 bits) • Single virtual address mode and multiple virtual address mode • Supports multiple page sizes: 1k, 4k, 64k, 1Mbytes • 4-entry full-associative TLB for instruction • 64-entry, full-associative TLB for instruction and operand • Supports software selection of replacement method and round-robin replacement algorithms • Contents of TLB are directly accessible by address mapping
Cache memory	<ul style="list-style-type: none"> • Instruction cache (IC) <ul style="list-style-type: none"> — 8k bytes, direct mapping — 256 entries, 32-byte block length — Normal mode (8-kbyte cache) — RAM mode (4-kbyte cache + 4-kbyte RAM) • Operand cache (OC) <ul style="list-style-type: none"> — 16k bytes, direct mapping — 512 entries, 32-byte block length — Normal mode (16-kbyte cache) — RAM mode (8-kbyte cache + 8-kbyte RAM) — Selectable write policy (copy-back/write-through) • 1-stage copy-back buffer, 1-stage write-through buffer • Contents of cache memories can be accessed directly by address mapping (can be used as on-chip memory)

Table 1.1 SH-4 Features (Cont.)

Item	Features
Interrupt controller (INTC)	<ul style="list-style-type: none"> • 5 independent external interrupts: NMI, IRL3 - IRL0 • 15-level encoded external interrupts: IRL3 - IRL0 • On-chip peripheral interrupts: set priority levels for each module
User break controller (UBC)	<ul style="list-style-type: none"> • Supports debugging by user break interrupts • 2 break channels • Addresses, data values, type of access, and data size can all be set as break conditions • Supports a sequential break function
Bus state controller (BSC)	<ul style="list-style-type: none"> • Supports external memory access <ul style="list-style-type: none"> — 64/32/16/8-bit external data bus • Physical address space divided into seven areas, each a maximum 64 Mbytes, with the following features settable for each area: <ul style="list-style-type: none"> — Bus size (8, 16, 32, or 64 bits) — Number of wait cycles (also supports a hardware wait function) — Setting the type of space enables direct connection to DRAM, synchronous DRAM, synchronous GRAM, and burst ROM — Supports fast page mode and EDO for DRAM — Supports PCMCIA interface — Outputs chip select signal (CS0–CS6) for corresponding area • DRAM/synchronous DRAM/synchronous GRAM refresh function <ul style="list-style-type: none"> — Programmable refresh interval — Supports CAS-before-RAS refresh and self-refresh modes • DRAM/synchronous DRAM/synchronous GRAM burst access function • Usable as either big or little endian machine
Direct memory access controller (DMAC)	<ul style="list-style-type: none"> • 4-channel physical address DMA controller • Transfer data size: 8, 16, 32, 64 bits, or 32 bytes • Address mode: <ul style="list-style-type: none"> — 1-bus-cycle single address mode — 2-bus-cycle dual address mode • Transfer request: external, on-chip module, or auto-request • Bus mode: cycle steal, or burst mode
Timer (TMU)	<ul style="list-style-type: none"> • 3-channel auto-reload type 32-bit timer • Input capture function • 7 types of counter input clocks can be selected
Real time clock (RTC)	<ul style="list-style-type: none"> • Built-in clock and calendar functions • On-chip 32-kHz crystal oscillator circuit with a maximum resolution (cycle interrupt) of 1/256 second
Serial communication interface (SCI1,2)	<ul style="list-style-type: none"> • 2 full-duplex communication channels (SCI1,2) • Channel 1 (SCI1) <ul style="list-style-type: none"> — Select start-stop sync mode or clock sync system — Supports smart card interface • Channel 2 (SCI2) <ul style="list-style-type: none"> — Supports clock sync system — Integrates 16-byte FIFO for transmitter and receiver each
Package	<ul style="list-style-type: none"> • 208-pin QFP

1.2 Block Diagram

Figure 1.1 is a functional block diagram of the SH-4.

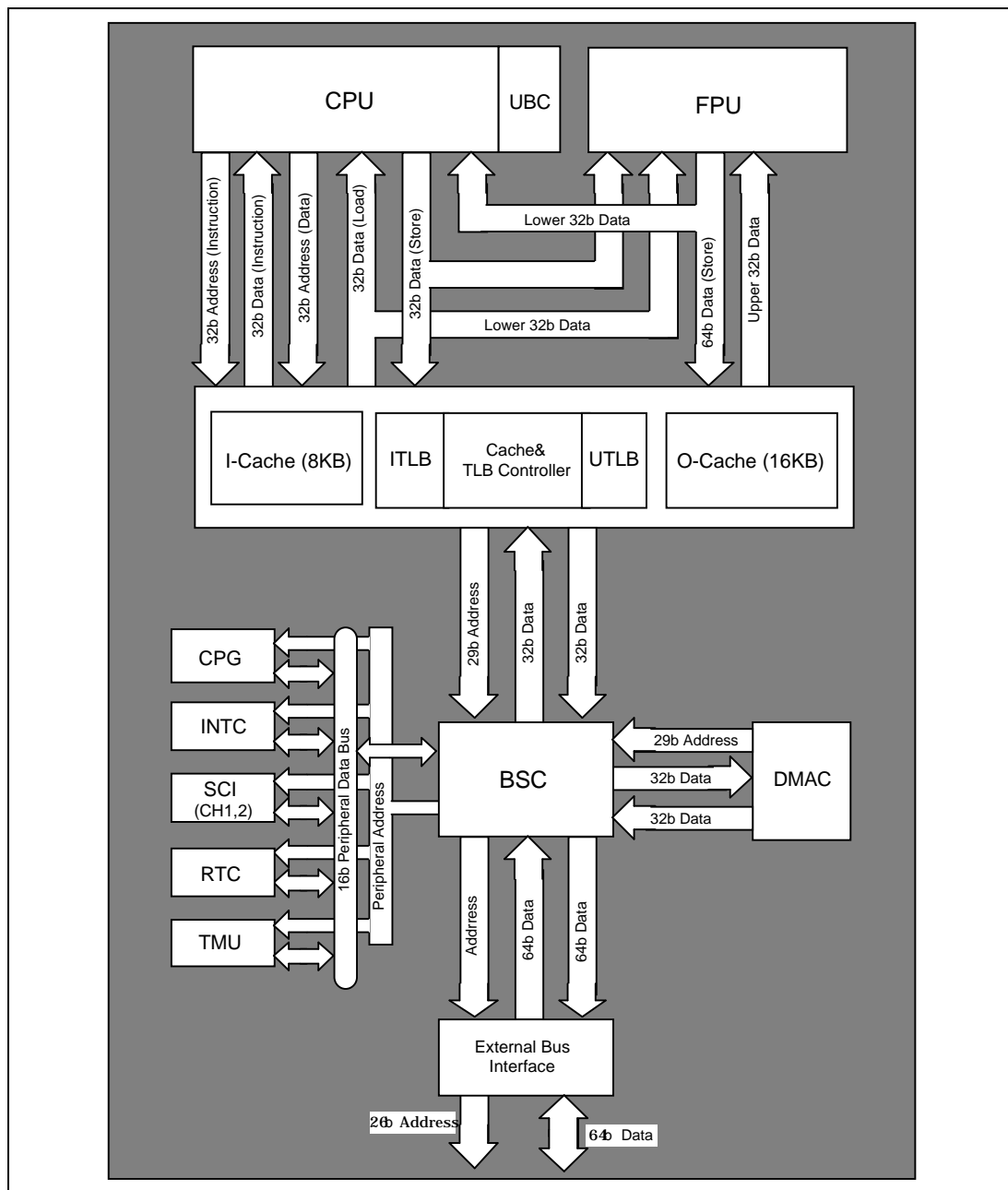


Figure 1.1 SH-4 Functional Block Diagram

Section 2 Data Formats and Registers

2.1 Data Formats

The data formats which are supported in SH4, are shown in figure 2.1.

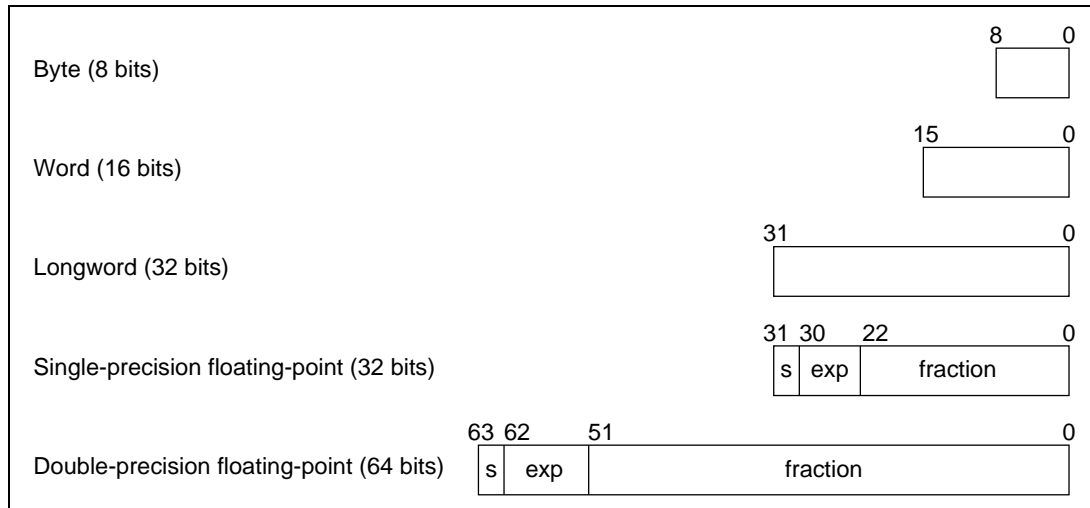


Figure 2.1 Data Formats

2.2 General Registers

Figure 2.2 shows the relationship between the processor modes and general registers. The SH4 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 SH4 has two processor modes: privileged mode and user mode. R0–R7 are assigned as follows.

- **R0_BANK0–R7_BANK0**
In user mode (SR.MD = 0), or in privileged mode (SR.MD = 1) with BANK0 referencing (SR.RB = 0), R0–R7 are assigned to R0_BANK0–R7_BANK0.
- **R0_BANK1–R7_BANK1**
In privileged mode (SR.MD = 1) with BANK1 referencing (SR.RB = 1), R0–R7 are assigned to R0_BANK1–R7_BANK1. The mode is always changed to privileged mode after a reset, exception, and interrupt.

SR.MD = 0 or (SR.MD = 1 and SR.RB = 0)			(SR.MD = 1 and SR.RB = 1)		
R0	R0 Bank 0		R0_Bank		
R1	R1 Bank 0		R1_Bank		
R2	R2 Bank 0		R2_Bank		
R3	R3 Bank 0		R3_Bank		
R4	R4 Bank 0		R4_Bank		
R5	R5 Bank 0		R5_Bank		
R6	R6 Bank 0		R6_Bank		
R7	R7 Bank 0		R7_Bank		
R0_Bank	R0 Bank 1	R0			
R1_Bank	R1 Bank 1	R1			
R2_Bank	R2 Bank 1	R2			
R3_Bank	R3 Bank 1	R3			
R4_Bank	R4 Bank 1	R4			
R5_Bank	R5 Bank 1	R5			
R6_Bank	R6 Bank 1	R6			
R7_Bank	R7 Bank 1	R7			
R8	R8	R8			
R9	R9	R9			
R10	R10	R10			
R11	R11	R11			
R12	R12	R12			
R13	R13	R13			
R14	R14	R14			
R15	R15	R15			

Figure 2.2 General Registers

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

2.3 Floating-Point Registers

Figure 2.3 shows the floating-point registers. There are thirty-two 32-bit floating-point registers, interleaved into two banks (FPR0_BANK0–FPR15_BANK0 and FPR0_BANK1–FPR15_BANK1). These 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 corresponding between FPRn_BANKi and the reference name is determined by the FR bit in FPSCR, see Figure 2.3.

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

Figure 2.3 Floating-Point Registers

- 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 pair, DRi (8 registers): a DR register consists of 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 consists of 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 extension 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 extension register pair, XD_i (8 registers): An XD register consists of 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 extension register matrix, XMTRX: XMTRX consists of all 16 XF registers

XMTRX =	XF0	XF4	XF8	XF12
	XF1	XF5	XF9	XF13
	XF2	XF6	XF10	XF14
	XF3	XF7	XF11	XF15

2.4 Control Registers

- Processor Status Register, SR (32-bit, privilege protection, initial value = 0111 0000 0000 0000 0000 00?? 1111 00??)

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

— MD: Processor Mode

MD = 0: User mode (some instructions can not be executed and some resources can not 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 are accessed as LDC/STC R0_BANK–R7_BANK.)

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

— BL: Exception/Interrupt Block Bit (set to 1 by a reset, exception, or interrupt)

BL = 1: Interrupt requests are masked. A reset occurs when a general exception is requested.

— 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: 0xF*** 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, whose level is less than IMASK, are masked.

— S: Specifies a saturation operation for MAC instructions.

— T: a true/false condition, or carry/borrow bit.

— Res and

Reserved: When READ, Zero is read from the reserved field. When WRITE, a value of the corresponding fields in the source operand should be zero.

- Saved Status Register, SSR (32-bit, privilege protection, initial value = undefined)
The contents of SR are saved to SSR when an exception or interrupt is taken.
- Saved Program Counter, SPC (32-bit, privilege protection, initial value = undefined)
The address of the interrupted instruction is saved to SPC.

- Global Base Register, GBR (32-bit, initial value = undefined)
GBR is referenced as the base address in a GBR-based MOV instructions.
- Vector Base Register, VBR (32-bit, privilege protection, initial value = H'0000 0000)
VBR is referenced as the base address of the transition target address on an exception and interrupt.
 $\text{Transition address} = \text{VBR} + \text{vector offset}$
- Saved General Register 15, SGR (32-bit, privilege protection, initial value = undefined)
The contents of R15 are saved to SGR when an exception or interrupt is taken.
- Debug Vector Base Register, DBR (32-bit, privilege protection, initial value = undefined)
When the user break debug function is enabled (BRCCR.UBDE = 1), DBR is referenced as the transition target address to a user break handler. In this case, it is not VBR.

2.5 System Registers

- Multiply and Accumulate Register High, MACH (32-bit, initial value = undefined)
- Multiply and Accumulate Register Low, MACL (32-bit, initial value = undefined)
MACH/MACL is referenced as the accumulator in a MAC instruction, and as the destination register in a MUL instruction.
- Procedure Register, PR (32-bit, initial value = undefined)
The return address is saved into PR on a subroutine calls (BSR, BSRF, and JSR). The saved return address in PR is referenced on a subroutine return (RTS).
- Program Counter, PC (32-bit, initial value = H'A000 0000)
PC indicates the instruction fetch address.
- Floating-Point Unit Status/Control Register, FPSCR (32-bit, initial value = H'00040001)

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

— 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: FMOV instruction transfers a single-precision floating-point number.

SZ = 1: FMOV transfers 64-bit data which may consist of two single-precision floating-point numbers.

— 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 operation of graphics-related instructions is undefined).

The mode 'SZ=1 and PR=1' is reserved: FPU operation is undefined on that mode.

— DN: Denormalization Mode

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

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

		FPU Error (E)	Invalid Op. (V)	Zero Div. (Z)	Overflow (O)	Underflow (U)	Indexact (I)
Cause	FPU exception cause field	Bit 17	Bit 16	Bit 15	Bit 14	Bit 13	Bit 12
Enable	FPU exception enable field	Non	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7
Flag	FPU exception flag field	Non	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2

When an FPU computation instruction is executed, the cause field is set to zero, at first. Then the corresponding bit in the cause and flag fields is set to 1, if an FPU exception is requested. The flag field retains the status of the requested exceptions since when the flag field had been cleared most recently.

— RM: Rounding Mode

RM = 00: Round to Nearest

RM = 01: Round to Zero

RM = 10: Reserved

RM = 11: Reserved

Programming Notes: When SZ=1 and Big Endian, FMOV can be used as a double-precision floating-point load or store. When Little Endian, single-precision floating-point FMOV must be executed twice to load or store a double-precision floating-point number.

2.6 Memory-Mapped Registers

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

- 0x1F00 0000 to 0x1FFF FFFF, and
- 0xFF00 0000 to 0xFFFF FFFF.

These two areas are used as follows.

0x1F00 0000–0x1FFF FFFF: This area should be referenced on a address translation mode. The physical page number field in TLB does not cover 32-bit address space, because the external memory is defined as a 29-bit address space in SH4 Architecture. When address translating, the memory-mapped registers can be accessed to set a page number in this area into the corresponding field of TLB. The length of in an access that involves address translation. Use the page number in this area as the actual page number set in the TLB. When address translation is not performed, the operation of an access to this area is undefined.

- 0xFF00 0000–0xFFFF FFFF: This area should be referenced in an access without an address translation.

Undefined locations in the two areas should not be accessed. The operation of an access to the undefined location is not defined. Additionally, the memory-mapped registers should be accessed with the defined data size. When accessing with an illegal size, the operation is also undefined.

Programming Note: An access to area 0xFF00 0000–0xFFFF FFFF in user mode causes an address error. In user mode, memory-mapped registers can be referenced by an access with address translation.

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 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 kbyte, and 1 Mbyte). It is possible to set the virtual address space access right and implement storage protection in privileged mode and user mode, respectively.

3.1.2 The 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 kbyte 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.

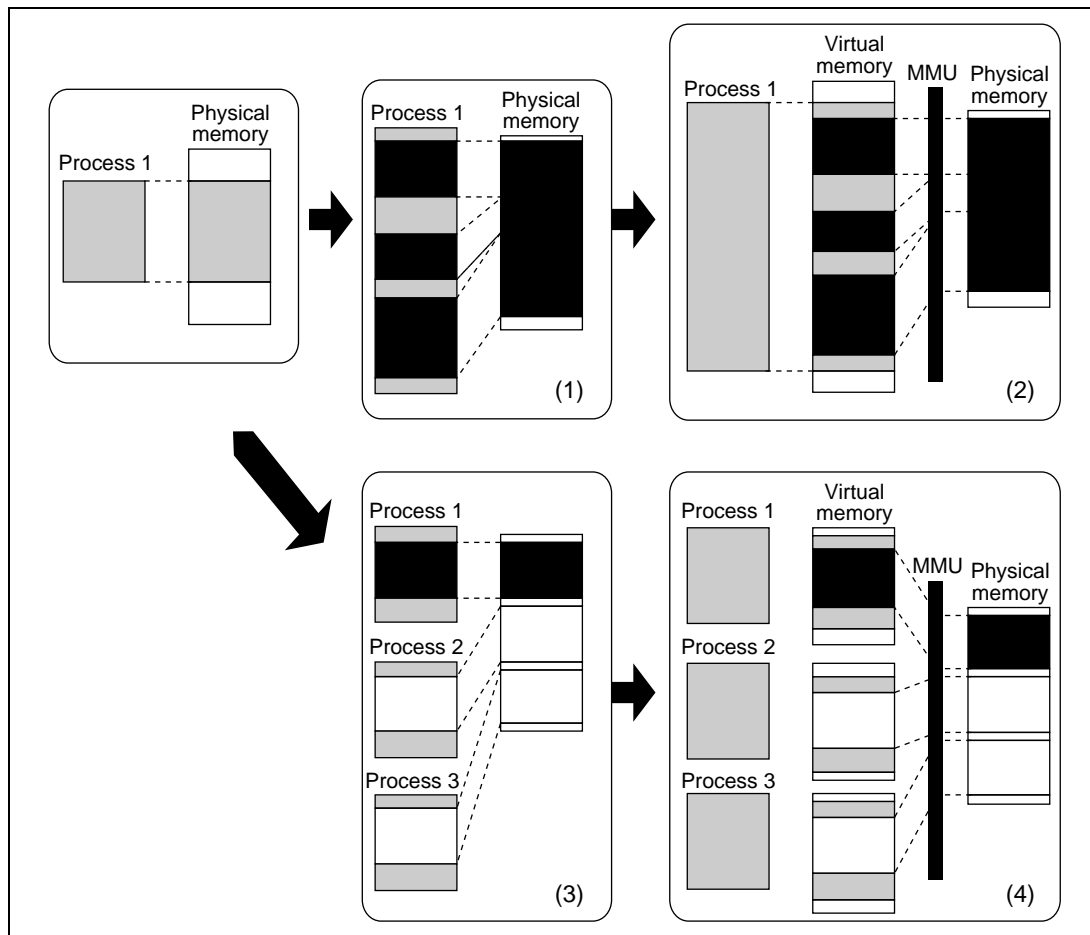


Figure 3.1 Role of the MMU

3.1.3 Register Configuration

The MMU registers are shown in table 3.1.

Table 3.1 MMU Registers

Name	Abbreviation	R/W	Size	Initial Value* ¹	Address* ²
Page table entry high register	PTEH	R/W	Longword	Undefined	0xFF00 0000
Page table entry low register	PTL	R/W	Longword	Undefined	0xFF00 0004
Page table entry assistance register	PTEA	R/W	Longword	Undefined	0xFF00 0034
Translation table base register	TTB	R/W	Longword	Undefined	0xFF00 0008
TLB exception address register	TEA	R/W	Longword	Undefined	0xFF00 000C
MMU control register	MMUCR	R/W	Longword	0x0000 0000	0xFF00 0010

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. When making an access from physical address space area 7 using the TLB, the upper 3 bits of the address are ignored.

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.

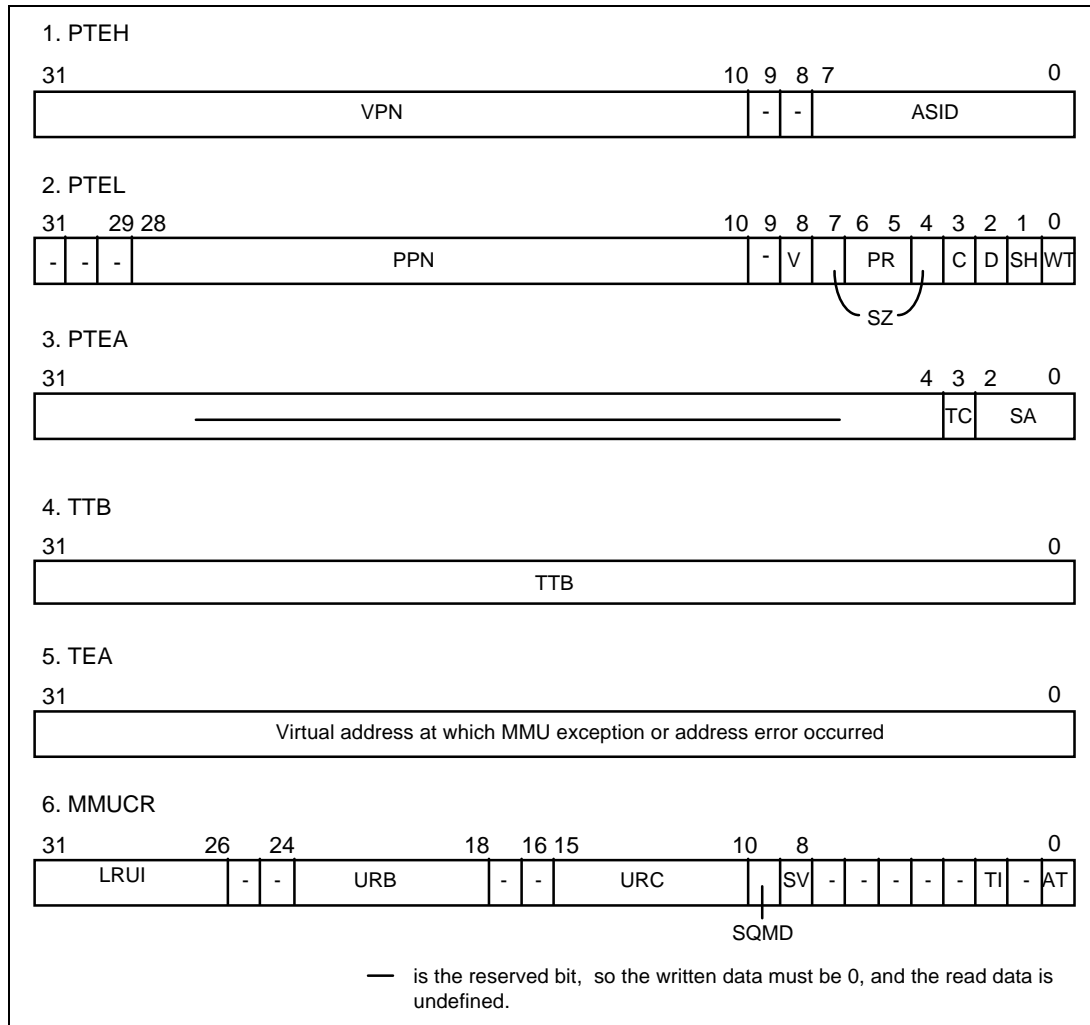


Figure 3.2 MMU-Related Registers

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 UTLB by means of the LDTLB 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 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'FF000034 in the P4 area and H'1F000034 in area 7. PTEA is used to store assistant bits for PCMCIA access to UTLB by means of the LDTLB instruction. 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)

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. 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 ITLB can be confirmed using the LRUI bits. LRUI is updated by means of the algorithm shown below.

	LRUI					
	[5]	[4]	[3]	[2]	[1]	[0]
When using ITLB entry 0	0	0	0	—	—	—
When using ITLB entry 1	1	—	—	0	0	—
When using ITLB entry 2	—	1	—	1	—	0
When using ITLB entry 3	—	—	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 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 P1~P4 area (except Store queue area) in user mode will cause an address error.

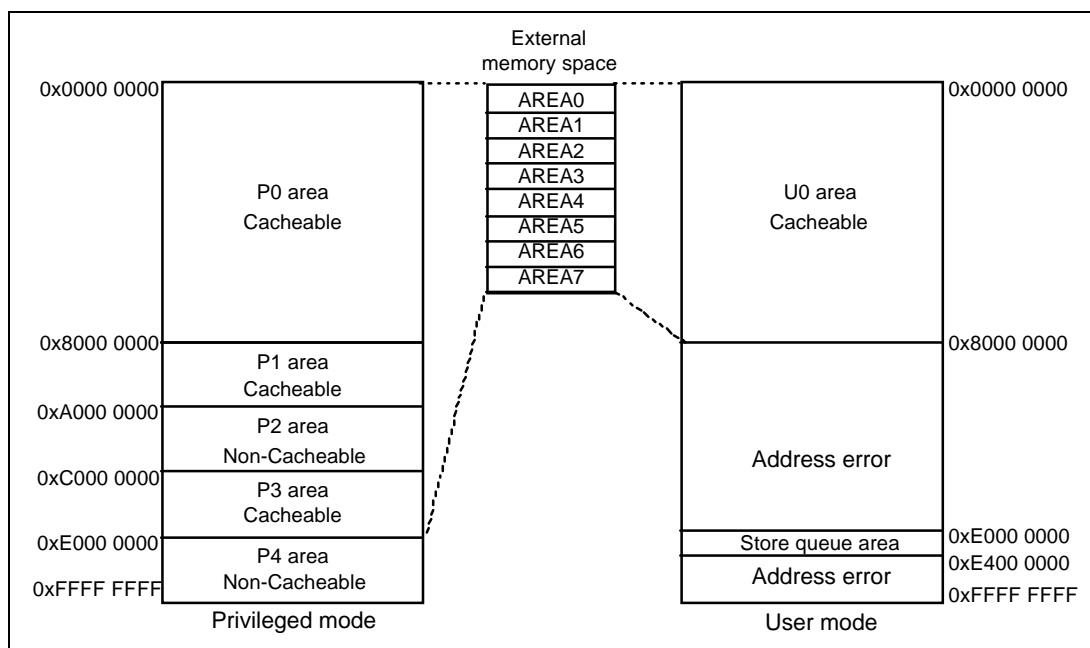


Figure 3.3 Physical Memory Space (MMUCR.AT == 0)

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, switching between the copy-back method and the write-through method for write accesses is outside the P1 area specified by the CCR.WT bit. P1 area switching follows the CCR.CB bit specification. 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 built-in I/O channels. This area cannot be accessed using the cache. The P4 area is shown in detail in figure 3.4.

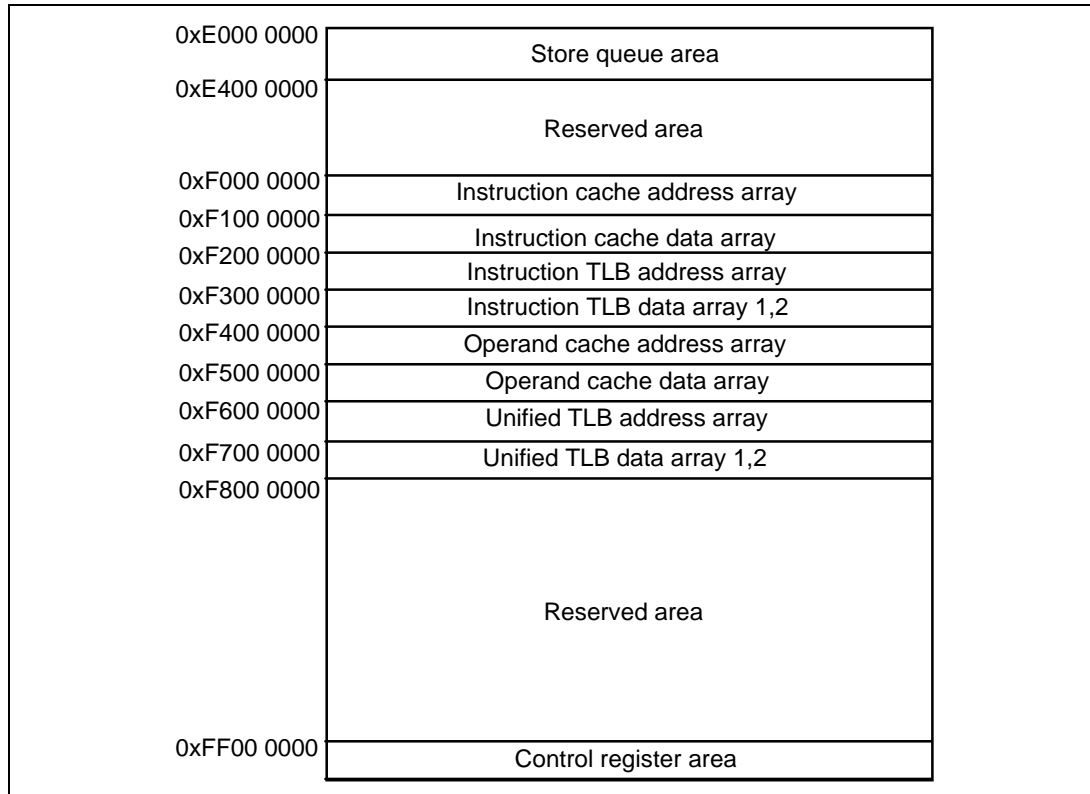


Figure 3.4 P4 Area

The area from 0xE0000000 to 0xE3FF FFFF comprises addresses for accessing the store queues (SQ). When the MMU is disabled (MMUCR.AT=0), the SQ access right is specified by the MMUCR.SQMD bit. (See section 4.6.)

The area from 0xF000 0000 to 0xF0FF FFFF is used for direct access to the instruction cache address array. (See section 4.5.1.)

The area from 0xF100 0000 to 0xF1FF FFFF is used for direct access to the instruction cache data array. (See section 4.5.2.)

The area from 0xF200 0000 to 0xF2FF FFFF is used for direct access to the instruction TLB address array. (See section 3.7.1.)

The area from 0xF300 0000 to 0xF3FF FFFF is used for direct access to the instruction TLB data array 1,2. (See section 3.7.2,3.7.3)

The area from 0xF400 0000 to 0xF4FF FFFF is used for direct access to the operand cache address array. (See section 4.5.3.)

The area from 0xF500 0000 to 0xF5FF FFFF is used for direct access to the operand cache data array. (See section 4.5.4.)

The area from 0xF600 0000 to 0xF6FF FFFF is used for direct access to the common TLB address array. (See section 3.7.4.)

The area from 0xF700 0000 to 0xF7FF FFFF is used for direct access to the common TLB data array1,2. (See section 3.7.5, 3.7.6.)

The area from 0xFF00 0000 to 0xFFFF FFFF is the on-chip supporting module 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 8 areas as shown in figure 3.5. Areas 0 to 6 relate to memory, such as SRAM, SDRAM, DRAM, and PCMCIA. Area 7 is a reserved area. See section 13, External Bus Control, for details.

0x0000 0000	Area 0
0x0400 0000	Area 1
0x0800 0000	Area 2
0x0C00 0000	Area 3
0x1000 0000	Area 4
0x1400 0000	Area 5
0x1800 0000	Area 6
0x1C00 0000	Area 7 (reserved area)
0x1FFF FFFF	

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 0x1F00 0000 to 0x1FFF 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.

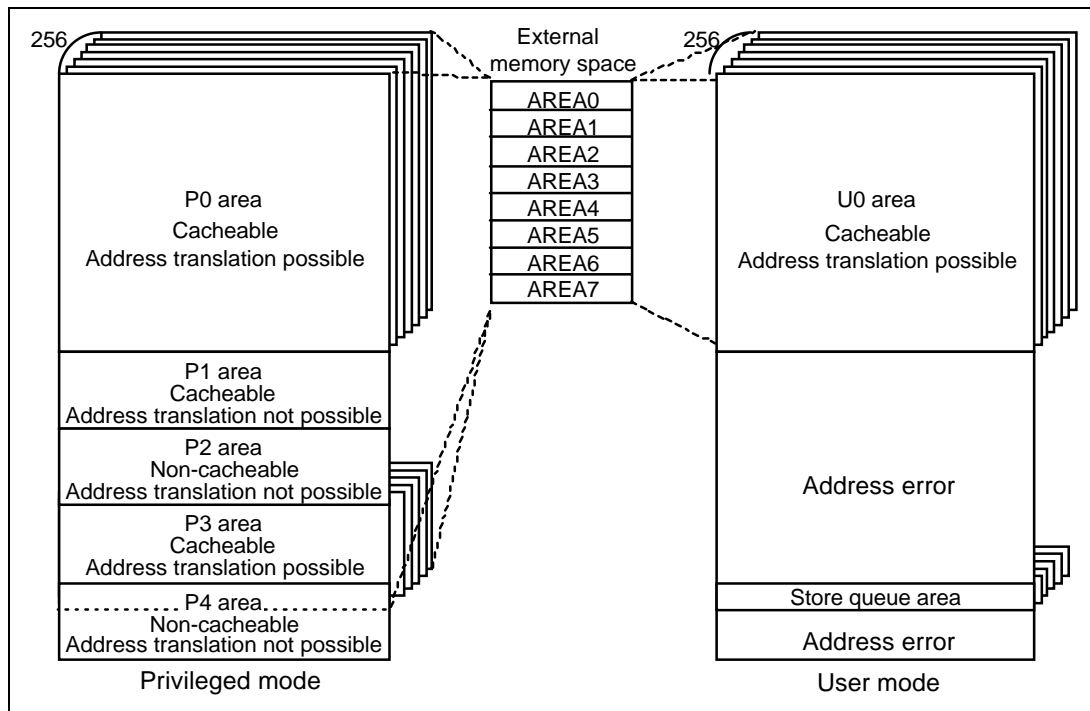


Figure 3.6 Virtual Memory Space (MMUCR.AT == 1)

P0, P3, U0 Areas: The P0 (except 0x7C00 0000-0x7FFF FFFF), P3, and U0 areas 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 the CCR is in the cache-enabled state and the TLB cacheability 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 0x1F00 0000 to 0x1FFF FFFF of area 7 in external memory space are allocated to the control register area. This enables on-chip supporting module 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 and P4 areas (except 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: see section 4.6, Store Queues, for details.

3.3.4 On-Chip RAM Space

In the SH-4, half (8 kB) of the operand cache (16 kB) 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 0x7C00 0000 to 0x7FFF FFFF are an on-chip RAM area. Only data accesses (byte/word/longword/quadword) can be used in this area. In this case, address translation does not apply to the area from 0x7C00 0000 to 0x7FFF FFFF even if MMUCR.AT is 1. This area can be used only in RAM-mode (CCR.ORA = 1).

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 handling routine. In the TLB miss exception handling routine, the address translation table in external memory is searched, and the corresponding physical address and page management information are recorded in the TLB. However, if a TLB miss occurs in the ITLB, a search is carried out by hardware to see if the address translation information is recorded in the UTLB. If the UTLB contains the corresponding address translation information, external memory is not searched. This procedure

is called hardware TLB miss handling. 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 memory 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).

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 full-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

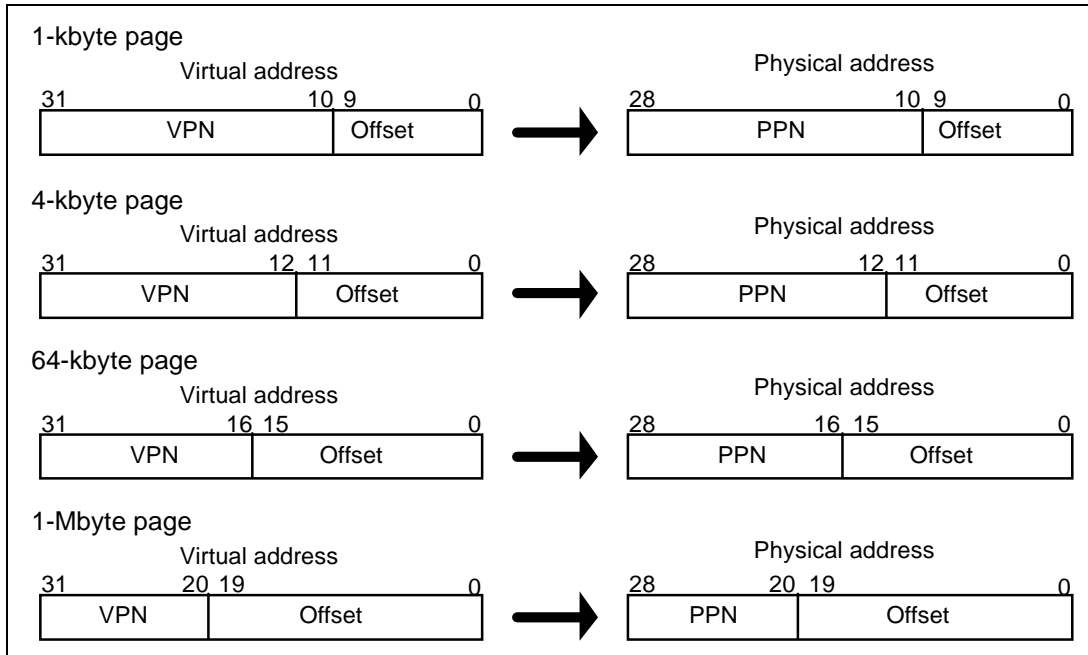


Figure 3.8 Relationship between Page Size and Address Format

- **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**
 - 0: Pages are not shared by processes
 - 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 (28-10) are valid.

With a 4-kbyte page, PPN (28-12) are valid.

With a 64-kbyte page, PPN (28-16) are valid.

With a 1-Mbyte page, PPN (28-20) are valid.

The synonym problem must be taken into account when setting the PPN. (See section 3.5.5.)
- **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.
- **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
- **SA: Space attribute bits**
Valid only when the page is mapped onto area 5 or 6 PCMCIA.
000: Undefined
001: Variable-size I/O space (base size according to IOIS#16 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 the wait control register provided in the bus control unit for areas 5 and 6.
0: Wait control register 0 is used
1: Wait control register 1 is 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 full-associative type entries. The address translation information is almost the same as that in the UTLB, but with the following differences:

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.

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

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.

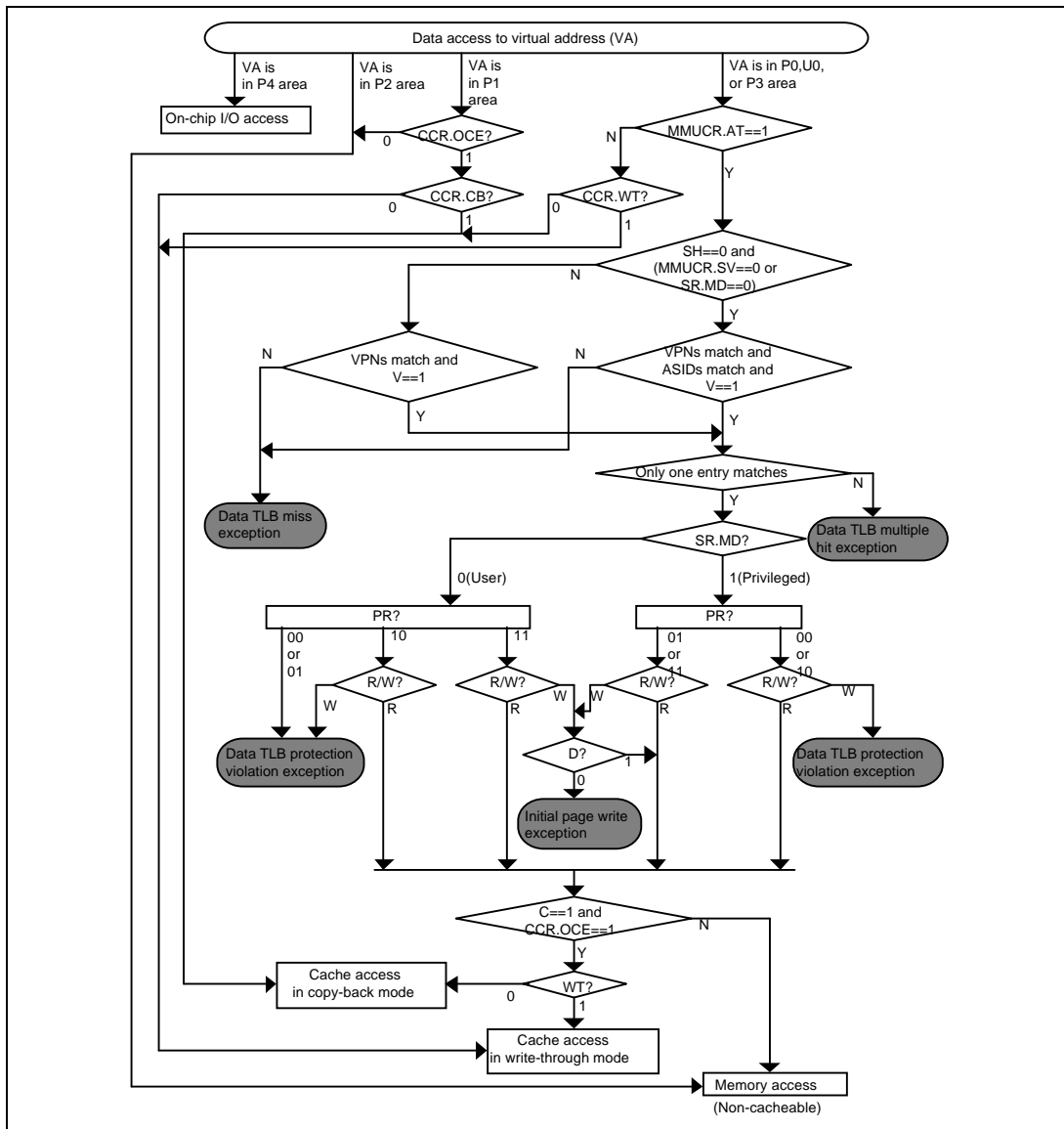


Figure 3.10 Flowchart of Memory Access Using UTLB

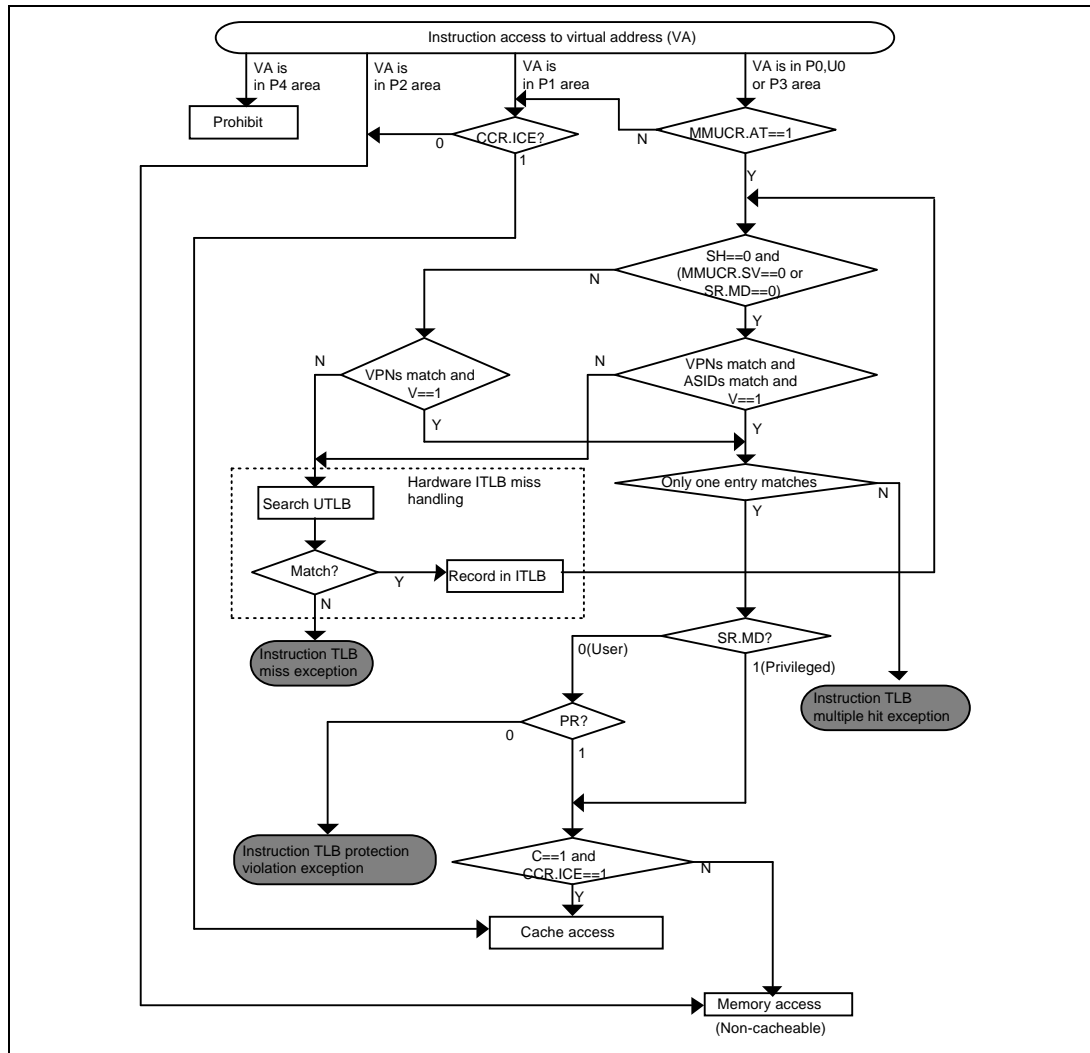


Figure 3.11 Flowchart of Memory Access Using ITLB

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 and external memory 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.
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.

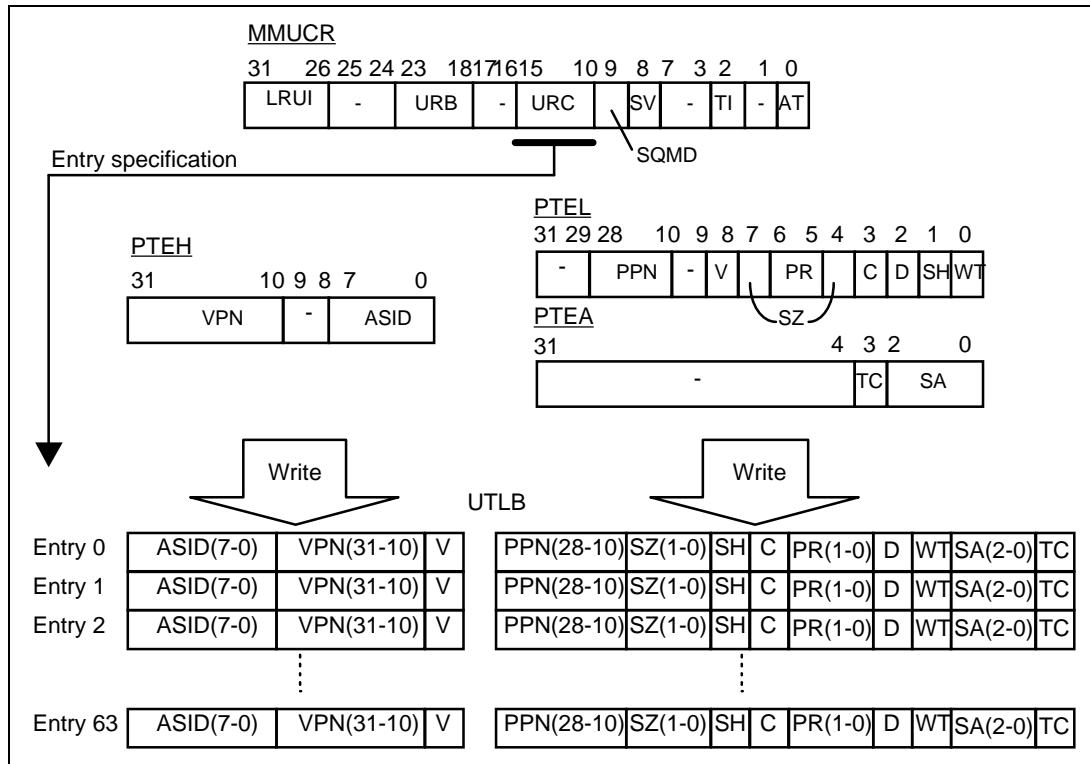


Figure 3.12 Operation of LDTLB Instruction

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 (13–5) of the virtual address in order to achieve fast operand cache operation. However, (13–10) of the virtual address in the case of a 1-kbyte page, and (13–12) of the virtual address in the case of a 4-kbyte

page, are subject to address translation. As a result, (13–10) of the physical address after translation may differ from (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 (CCR.OIX==1), VPN(25) is used instead of VPN(13) above, and therefore the restrictions described above apply to VPN(25).

Note: To provide for future SH Series expansion, when multiple items of address translation information use the same physical memory, 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.
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, and write to PTEA the values of the SA and TC if necessary.
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 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.
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 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 lost.

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.
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, and write to PTEA the values of the SA and TC if necessary.
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 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.

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 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.
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, SH, V, and WT bits in the page table entry recorded in external memory, and write to PTEA the values of the SA and TC if necessary.
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 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 using a MOV instruction in privileged mode. Operation is not guaranteed if access is made from a program in another area. [In this case, the branch instruction to P0/U0/P1/P3 area should be issued 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 a data array1, and SA, TC as a data array2. VPN, V, D, and ASID in the UTLB can be accessed as an address array, and PPN, V, SZ, PR, C, D, WT, and SH as a data array1, and SA, TC as a data array2. V and D can be accessed from both the address array side and the data array1 side. Only longword access is possible. Instruction fetches cannot be performed in these areas. Reserved bits should be written with 0, and treated as don't care bits in a read.

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, (31–24) have the value H'F2 indicating the ITLB address array, and the entry is selected by (9–8). As longword access is used, 0 should be specified for address field (1–0). ~~The other bits are don't care bits.~~

In the data field, VPN is indicated by (31–10), V by (8), and ASID by (7–0). ~~(9) is a don't care bit.~~

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.

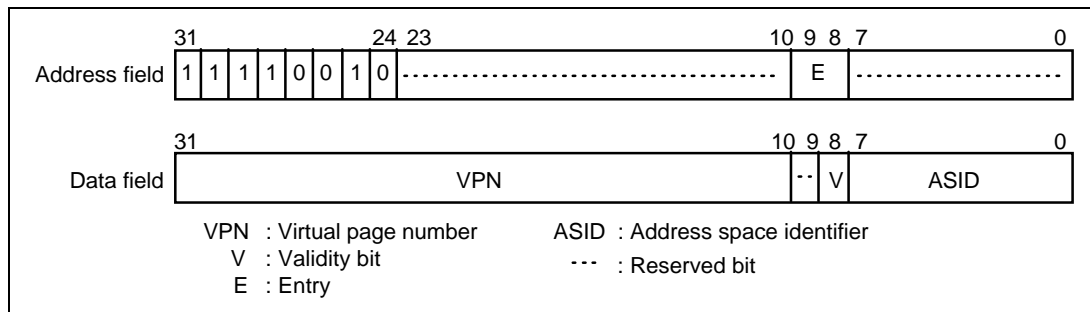


Figure 3.13 Memory-Mapped ITLB Address Array

3.7.2 ITLB Data Array 1

The 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 1 are specified in the data field.

In the address field, (31–23) have the value H'F30 indicating the ITLB data array 1, and the entry is selected by (9–8).

In the data field, PPN is indicated by (28–10), V by (8), SZ by (7) (4), PR by (6), C by (3), and SH by (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.

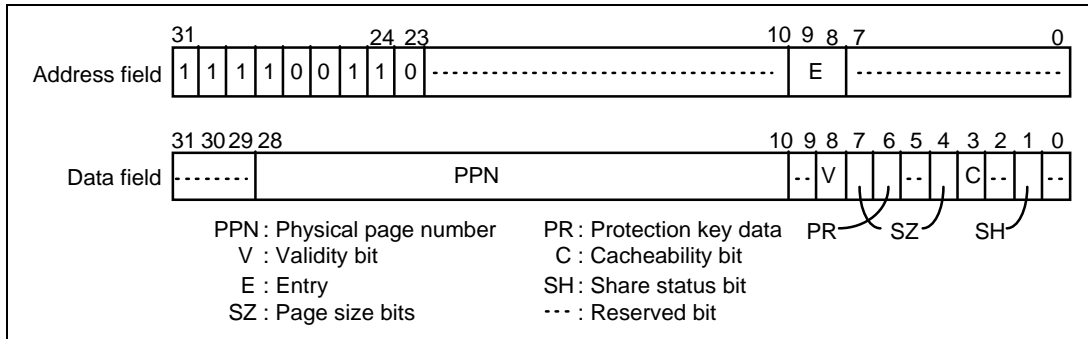


Figure 3.14 Memory-Mapped ITLB Data Array 1

3.7.3 ITLB Data Array 2

The 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 the data array 2 are specified in the data field.

In the address field, (31-23) have the value H'F38 indicating ITLB data array 2, and the entry is selected by (9-8).

In the data field, SA is indicated by (2-0) and TC by (3).

The following two kinds of operation can be used on ITLB data array_2:

1. ITLB data array [20](#) read
SA and TC are read into the data field from the ITLB entry corresponding to the entry set in the data 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.

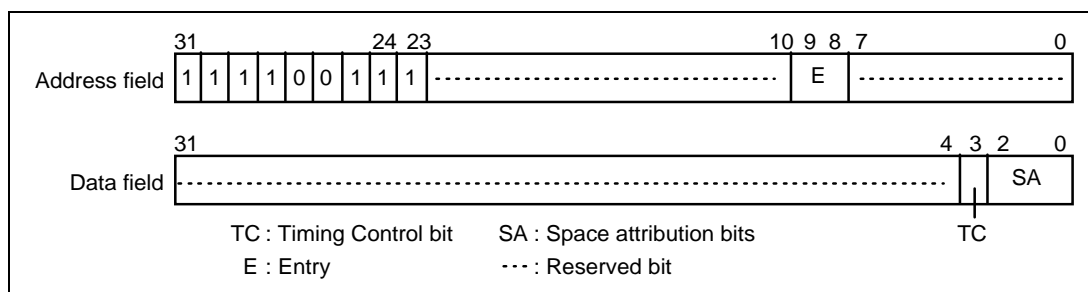


Figure 3.15 Memory-Mapped ITLB Data Array 2

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 address array are specified in the data field.

In the address field, (31–24) have the value H'F6 indicating the UTLB address array, and the entry is selected by (13–8). The address array (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 (31–10), D by (9), V by (8), and ASID by (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 the occurrence of a TLB miss exception results in no operation. 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.

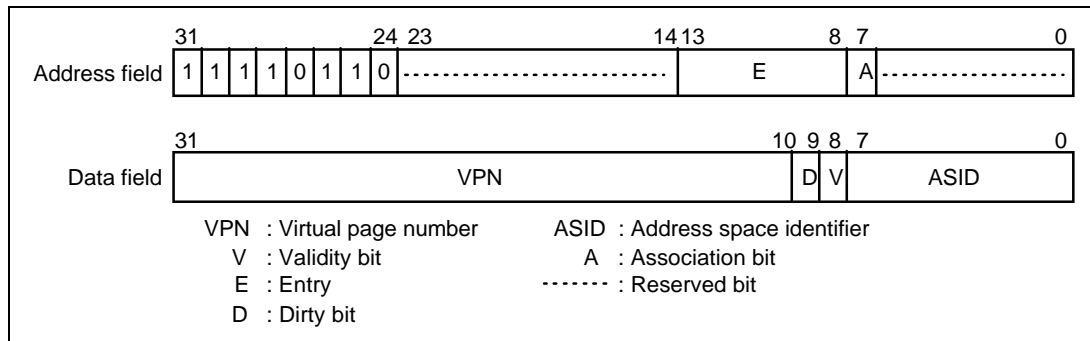


Figure 3.16 Memory-Mapped UTLB Address Array

3.7.5 UTLB Data Array 1

The 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 1 are specified in the data field.

In the address field, (31–23) have the value H'F70 indicating UTLB data array 1, and the entry is selected by (13–8).

In the data field, PPN is indicated by (28–10), V by (8), SZ by (7) (4), PR by (6–5), C by (3), D by (2), SH by (1), and WT by (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.

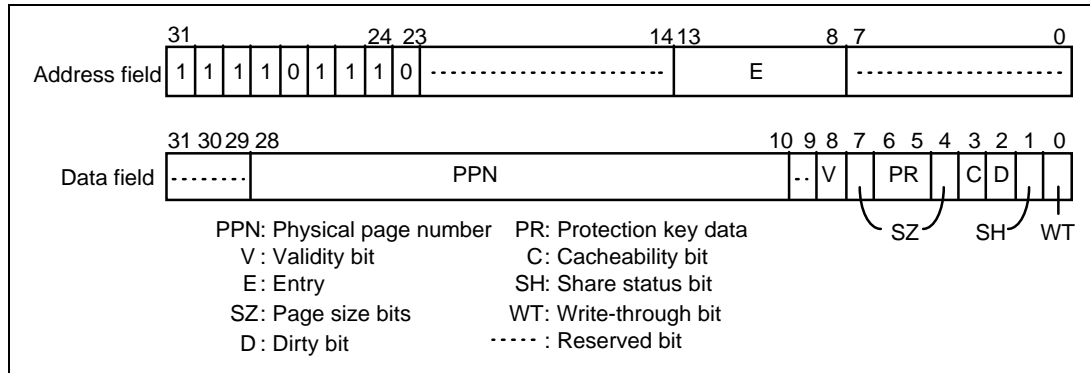


Figure 3.17 Memory-Mapped UTLB Data Array 1

3.7.6 UTLB Data Array 2

The 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 the data array 2 are specified in the data field.

In the address field, (31–23) have the value H'F78 indicating UTLB data array 2, and the entry is selected by (13–8).

In the data field, TC is indicated by (3), and SA by (2-0). ~~The other bits are don't care bits.~~

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.

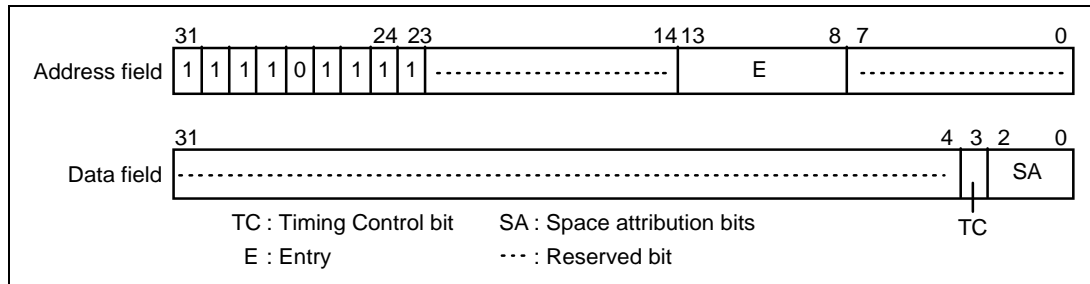


Figure 3.18 Memory Mapped UTLB Data Array 2

Section 4 Caches

4.1 Overview

4.1.1 Features

The SH-4 has an on-chip 8-kbyte instruction cache for instructions and 16-kbyte operand cache for data. Half of the memory of the operand cache (8 kbytes) can also be used as on-chip RAM. Two 32-byte store queues (SQs) are also provided for storing data in high-speed external memory. The features of the caches and store queues 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 x 32 bytes Addresses 0xE0000000-0xE3FFFFFF
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 summarizes the specifications of the cache control register.

Table 4.2 Cache Control Register

Name	Abbreviation	R/W	Size	Initial Value*1	Address*2
Cache control register	CCR	R/W	Longword	0x00000000	0xFF00001C
Queue address control register 0	QACR0	R/W	Longword	Undefined	0xFF000038
Queue address control register 1	QACR1	R/W	Longword	Undefined	0xFF00003C

Note: *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 registers.

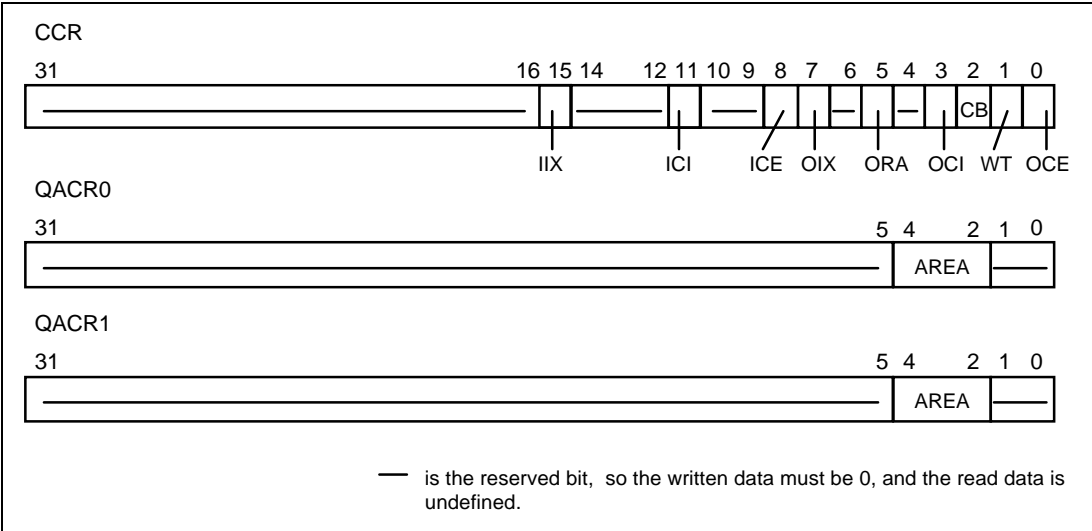


Figure 4.1 Cache Control Register

(1) Cache Control Register (CCR)

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'FF00001C in the P4 area and H'1F00001C 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.

- IIX : IC index mode bit
 - 0: Address(12:5) used as IC index
 - 1: Address (25)(11:5) used as IC index
- 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 mode bit
 - 0: Address (13:5) used as OC index
 - 1: Address (25)(12:5) used as OC index
- ORA: OC RAM enable bit. When the OC (operand cache) 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 out of the OC's 16 kbytes are to be used as RAM. When the OC is not enabled (OCE == 0), the ORA bit should also be 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 enable bit. Indicates the write mode for the P1 area cache.
 - 0: Write-through mode
 - 1: Copy-back mode

- **WT:** Write-through enable bit. Indicates the write mode for the P0, U0, and P3 area cache. 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'FF000038 in the P4 area and H'1F000038 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'FF00003C in the P4 area and H'1F00003C in area 7. QACR1 specifies the area onto which store queue 1 (SQ1) is mapped when the MMU is off.

- **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 memory-mapped cache (see section 4.5). 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 (31–10) of the address resulting from effective address translation by the MMU:
 - a. If the tag matches and the V bit is 1 Ø 3a
 - b. If the tag matches and the V bit is 0 Ø 3b
 - c. If the tag does not match and the V bit is 0 Ø 3b
 - d. If the tag does not match, the V bit is 1, and the U bit is 0 Ø 3b
 - e. If the tag does not match, the V bit is 1, and the U bit is 1 Ø 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. 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. 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 (31–10) of the address resulting from effective address translation by the MMU:

	Copy-back	Write-through
a. If the tag matches and the V bit is 1	Ø 3a	Ø 3b
b. If the tag matches and the V bit is 0	Ø 3c	Ø 3d
c. If the tag does not match and the V bit is 0	Ø 3c	Ø 3d
d. If the tag does not match, the V bit is 1, and the U bit is 0	Ø 3c	Ø 3d
e. If the tag does not match, the V bit is 1, and the U bit is 1	Ø 3e	Ø 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 (copy-back, no 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. 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 **not** performed to the external memory corresponding to the effective address. In this case, a write to cache is not performed.

3e. Cache miss (copy-back, with 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. 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.

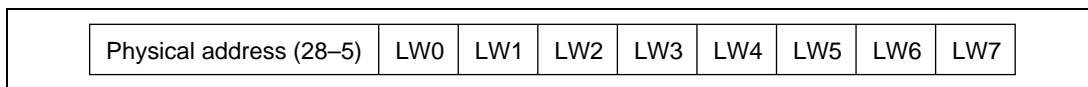


Figure 4.3 Configuration of Write-Back Buffer

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.

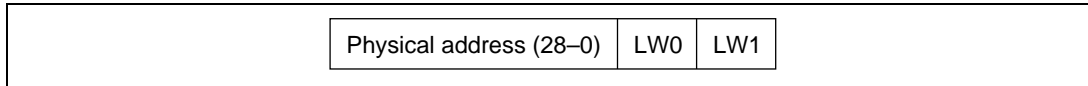


Figure 4.4 Configuration of Write-Through Buffer

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 [384284](#) to 511. Other entries can still be used as cache. RAM can be accessed using addresses 0x7C00 0000 to 0x7[FFF FC00](#)+FFF. ~~Addresses 0x7C00 2000 to 0x7FFF FFFF are a reserved area.~~ 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. When RAM mode is specified, address translation does not apply to this area.

4.3.7 OC Index Mode

Setting CCR.OIX to 1 enables OC indexing to be performed using (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 (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 (25), providing efficient use of the cache.

4.3.8 Coherency between Cache and External Memory

Coherency between cache and memory should be assured by software. In the SH-4, the following four new instructions are supported for cache operations. Details of these instructions are given in the Programming Manual.

- 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 an MMU exception occurs at the intended prefetch address, the result is no operation, and an exception is not generated. Details of the prefetch instruction are given in the Programming Manual.

- Prefetch instruction: PREF @Rn

4.4 Instruction Cache (IC)

4.4.1 Configuration

Figure 4.5 shows the configuration of the instruction cache.

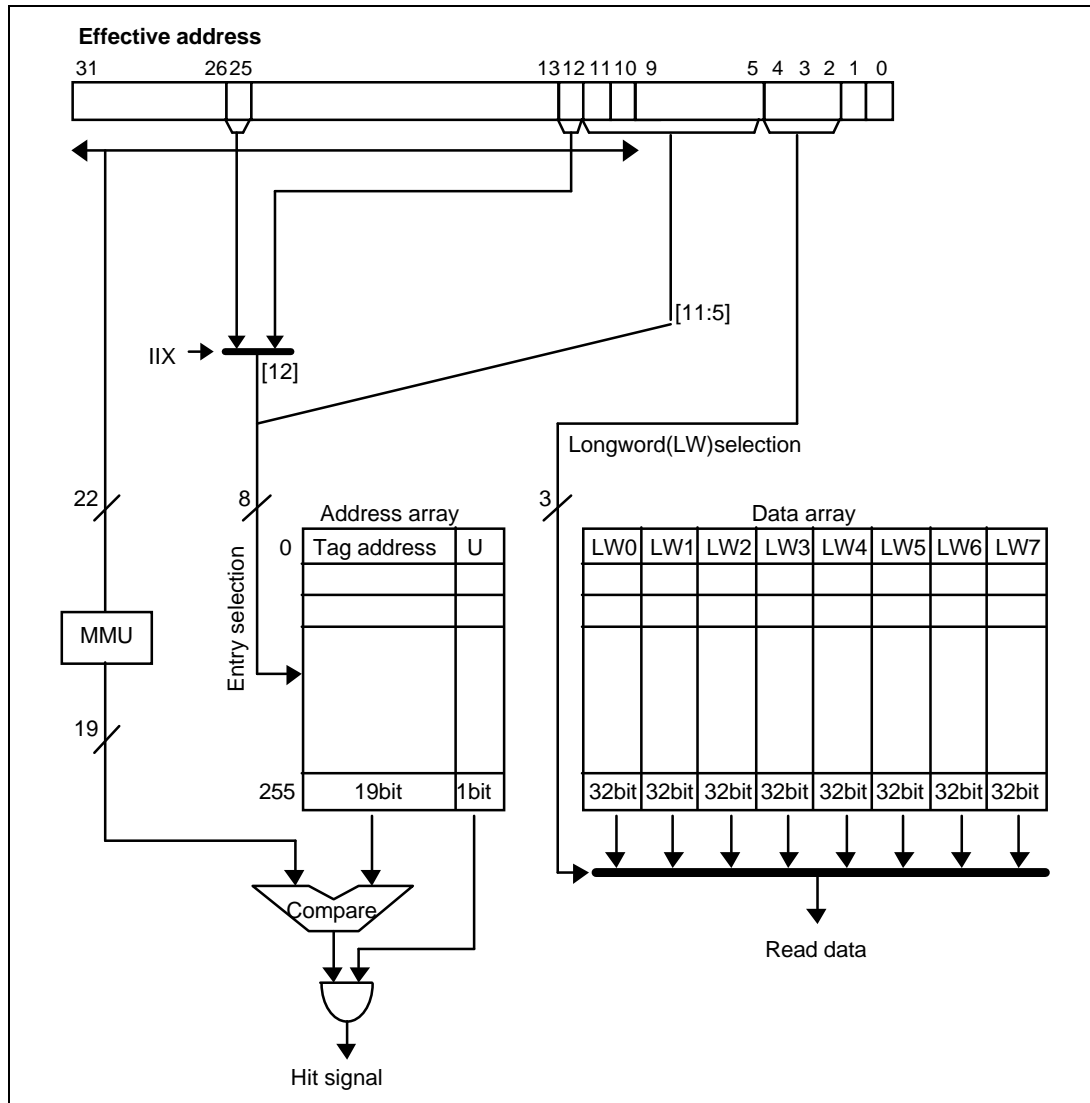


Figure 4.5 Configuration of Instruction cache

- The instruction cache consists of 256 cache lines, each composed of a 19-bit tag, V bit, U bit, and 32-byte data (16 instructions).
- 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.
- 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:
 - a. If the tag matches and the V bit is 1 Ø 3a
 - b. If the tag matches and the V bit is 0 Ø 3b
 - c. If the tag does not match and the V bit is 0 Ø 3b
 - d. If the tag does not match and the V bit is 1 Ø 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. While the remaining one cache line of data is being read, the CPU can execute the next processing. 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 (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 (12:5) of the effective address; therefore, when of 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 (25), providing efficient use of the cache.

4.5 Memory-Mapped Cache Configuration

To enable the IC and OC to be managed by software, their contents can be read and written by a P2 area program using a MOV instruction in privileged mode. Operation is not guaranteed if access is made from a program in another area. [In this case, the branch instruction to P0/U0/P1/P3 areas should be issued 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. Reserved bits should be written with 0, and treated as don't care bits in a read.

4.5.1 IC Address Array

The IC address array is allocated to addresses 0xF000 0000 to 0xF0FF 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, (31–24) have the value 0xF0 indicating the IC address array, and the entry is specified by (12–5). [CCR.IIX has no effect to the entry.](#) The address array (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 \(1–0\).](#)

In the data field, the tag is indicated by (31–10), and the V bit by (0). As the IC address array tag is 19 bits in length, data field (31–29) are not used in the case of a write in which association is not performed. Data field (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 (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 instruction TLB miss exception 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 TLB multiple hit exception handling routine.

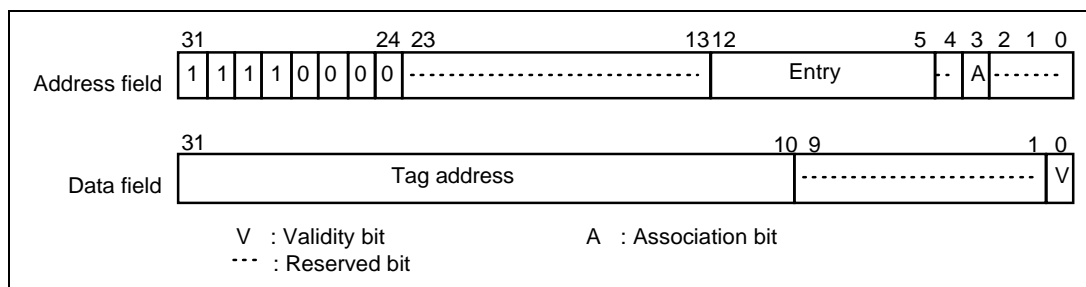


Figure 4.6 Memory-Mapped IC Address Array

4.5.2 IC Data Array

The IC data array is allocated to addresses 0xF100 0000 to 0xF1FF 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, (31–24) have the value 0xF1 indicating the IC data array, and the entry is specified by (12–5). [CCR.IIX has no effect to the entry.](#) Address field (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 \(1–0\).](#)

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 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 entry corresponding to the entry set in the address field.

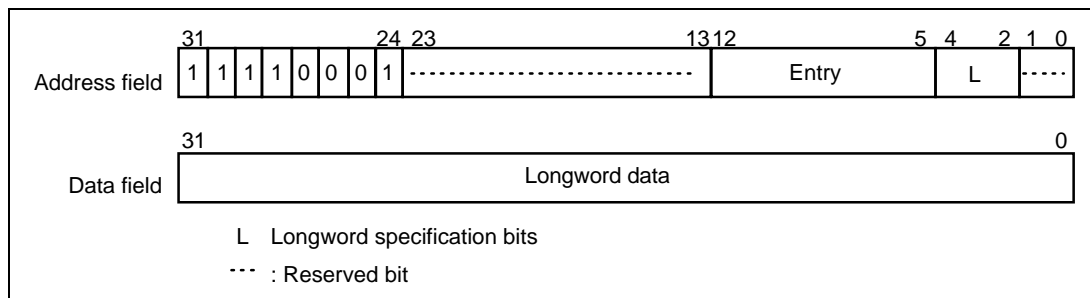


Figure 4.7 Memory-Mapped IC Data Array

4.5.3 OC Address Array

The OC address array is allocated to addresses 0xF400 0000 to 0xF4FF 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, (31–24) have the value 0xF4 indicating the OC address array, and the entry is specified by (13–5). [CCR.OIX and CCR.ORA has no effect to the entry.](#) The address array (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 \(1–0\).](#)

In the data field, the tag is indicated by (31–10), the U bit by (1), and the V bit by (0). As the OC address array tag is 19 bits in length, data field (31–29) are not used in the case of a write in which association is not performed. Data field (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 (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 a data TLB miss exception 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 TLB multiple hit exception handling routine.

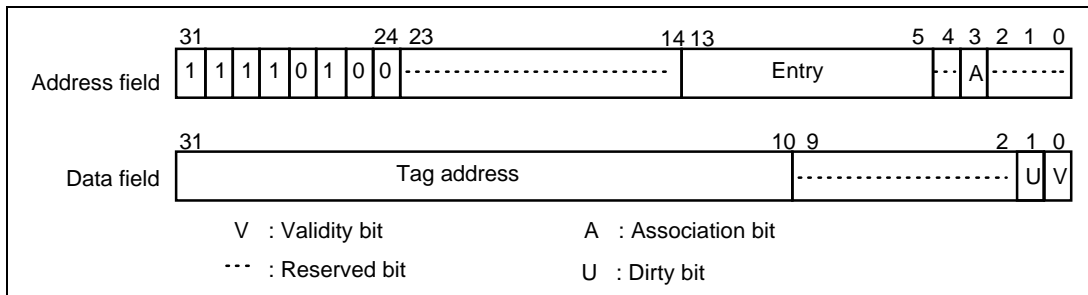


Figure 4.8 Memory-Mapped OC Address Array

4.5.4 OC Data Array

The OC data array is allocated to addresses 0xF500 0000 to 0xF5FF 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, (31–24) have the value 0xF5 indicating the OC data array, and the entry is specified by (13–5). [CCR.OIX and CCR.ORA has no effect to the entry.](#) Address field (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 (1–0). ~~The other bits are don't care bits.~~

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 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 entry corresponding to the entry set in the address field. This write does not set the U bit to 1 on the address array side.

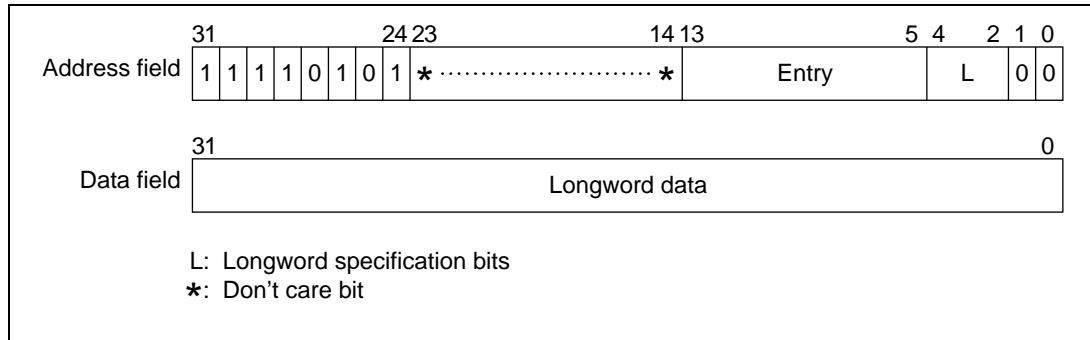


Figure 4.9 Memory-Mapped OC Data Array

4.6 Store Queue

Two 32-byte store queues (SQs) are supported to perform high-speed writes to external memory.

4.6.1 SQ Configuration

There are two 32-byte store queues, SQ0 and SQ 1, as shown in figure 4.10. These two store queues can be set independently.

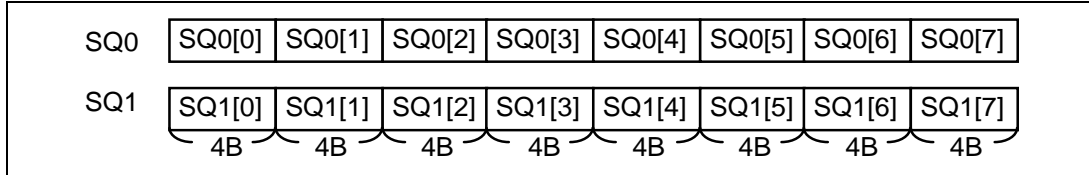


Figure 4.10 Store Queue Configuration

4.6.2 SQ Write

A write to the SQs can be performed using a store instruction (MOV) on P4 area 0xE0000000 to 0xE3FFFFFFC. 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 SQ0/1
(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 0xE0000000 to 0xE3FF FFFC starts a burst transfer from the SQs to external memory. The burst 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 (28:0) specification is as shown below, according to whether the MMU is on or off.

(1) MMU on

The SQ area (0xE0000000 to 0xE3FFFFFF) is set in VPN of the UTLB, and the transfer destination external memory address in PPN. [As area 7 is a reserved area, setting area 7 in PPN is prohibited.](#) 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. Since burst transfer is prohibited for PCMCIA areas, the SA and TC bits also have no meaning.

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 (9:5), the address prior to address translation is generated in the same way as 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.

(2) MMU off

The SQ area (0xE0000000 to 0xE3FFFFFF) is set 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 (25:6)
(5):	0/1	0: SQ0 specification 1: SQ1 specification and external memory address (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 (28:26) corresponding to SQ0

QACR1(4:2): External memory address (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.

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.

(1) 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.

(2) 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 the process 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 process. 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.

SH4 exception handling is of three kinds: for resets, general exceptions, and interrupts.

5.1.2 Control Registers

Table 5.1 shows memory-mapped control registers used in exception handling.

Table 5.1 Exception-Related Control Registers

Name	Address	Size	Initial Value after Reset		Value In Power-Down Mode	
			Power-On	Manual	Sleep	Standby
TRA	H'FF00 0020 H'1F00 0020	32	Undefined	Undefined	Retained	Retained
EXPEVT	H'FF00 0024 H'1F00 0024	32	H'0000 0000	H'0000 0020	Retained	Retained
INTEVT	H'FF00 0028 H'1F00 0028	32	Undefined	Undefined	Retained	Retained

- EXPEVT

31 12 11 0

Reserved	Exception code
----------	----------------

Reserved: Always read as zero. Zero should be specified when writing.

When a reset or general exception occurs, an exception code that identifies the exception is stored in EXPEVT.

- INTEVT

31	12 11	0
Reserved		Interrupt code

Reserved: Always read as zero. Zero should be specified when writing.

When an interrupt occurs, an exception code that identifies the interrupt is stored in INTEVT.

- TRA

31	10 9	2 0
Reserved		# Immediate data 00

Reserved: Always read as zero. Zero should be specified when writing.

When an unconditional trap (TRAPA instruction) is executed, the 8-bit immediate data in the TRAP instruction code is stored in TRA.

5.2 Exception Handling Functions

5.2.1 Exception Handling Flow

In exception handling, the contents of the program counter (PC) and status register (SR) are saved in the saved program counter (SPC) and saved status register (SSR), and the CPU starts execution of the appropriate exception handling routine according to the vector address. An exception handling routine is a program written 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 basic processing flow is as follows. See section 2 for the meaning of the individual SR bits.

1. The PC and SR contents are saved in the SPC and SSR.
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 case of 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) or interrupt event register (INTEVT).
7. The CPU branches to the determined exception handling vector address, and the exception handling routine begins.

5.2.2 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 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.

5.3 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 Exception Types and Priorities

Exception Category	Execution Mode	Exception	Priority Level	Priority Order	Vector	Base	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'A0000000	—	—	H'000
		Instruction TLB multiple-hit exception	1	3	H'A000 0000	—	—	H'140
		Data TLB multiple-hit exception	1	4	H'A000 0000	—	—	H'140
General exception	Re-execution type	User break before instruction execution	2	0	VBR/DBR	H'100/—	—	H'1E0
		Instruction address error	2	1	VBR	H'100	—	H'0E0
		Instruction TLB miss exception	2	2	VBR	H'400	—	H'040
		Instruction TLB protection violation exception	2	3	VBR	H'100	—	H'0A0
		General illegal instruction exception	2	4	VBR	H'100	—	H'180
		Slot illegal instruction exception	2	4	VBR	H'100	—	H'1A0
		General FPU disable exception	2	4	VBR	H'100	—	H'800
		Slot FPU disable exception	2	4	VBR	H'100	—	H'820
		Data address error (read)	2	5	VBR	H'100	—	H'0E0
		Data address error (write)	2	5	VBR	H'100	—	H'100
		Data TLB miss exception (read)	2	6	VBR	H'400	—	H'040
		Data TLB miss exception (write)	2	6	VBR	H'400	—	H'060
		Data TLB protection violation exception (read)	2	7	VBR	H'100	—	H'0A0
		Data TLB protection violation exception (write)	2	7	VBR	H'100	—	H'0C0
		FPU exception	2	8	VBR	H'100	—	H'120
		Initial page write exception	2	9	VBR	H'100	—	H'080
	Completion type	Unconditional trap (TRAPA)	2	4	VBR	H'100	—	H'160
		User break after instruction execution	2	10	VBR/DBR	H'100/—	—	H'1E0

Table 5.2 Exception Types and Priorities (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 interrupt	IRL3/2/1/0	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

Table 5.2 Exception Types and Priorities (cont)

Exception Category	Execution Mode	Exception	Priority Level	Priority Order	Vector Address	Offset	Exception Code
Interrupt	Completion type	Supporting module interrupt (module/source)	4	*2	(VBR)	H'600	H'400
		TMU0					
		TMU1					H'420
		TMU2					H'440
		TICPI2					H'460
		RTC					H'480
		PRI					H'4A0
		CUI					H'4C0
		SCI					H'4E0
		ERI					H'500
		RXI					H'520
		TXI					H'540
		TEI					H'560
		WDT					H'580
		ITi					H'5A0
		REF					H'600
		RCMI					
		ROVI					
		Hitachi-UDI					
		Hitachi-UDI					
		port					H'620
		GPIO					H'640
		DMAC					H'660
		DMTE0					H'680
		DMTE1					H'6A0
		DMTE2					H'6C0
		DMTE3					
		DAERR					
		SCIF					H'700
		ERI					H'720
		RXI					H'740
		SBRK					H'760
		TXI					

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 supporting modules.

Notes: 1. When BRCR.UBDE = 1, PC = DBR; otherwise, PC = VBR + H'100

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

5.4 Exception Flow

5.4.1 Exception Flow

Figure 5.1 shows an outline flowchart of the basic operations in instruction execution and exception handling. For the sake of clarity, the following description assumes that instructions are executed sequentially, one by one. Figure 5.1 shows the relative priority order of the different kinds of exceptions (reset/general exception/interrupt). Register settings in the event of an exception are shown only for SSR, SPC, EXPEVT/INTEVT, SR, and PC, but other registers may be set automatically by hardware, depending on the exception. See section 5.5 for details. Also, see section 5.5.4 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.

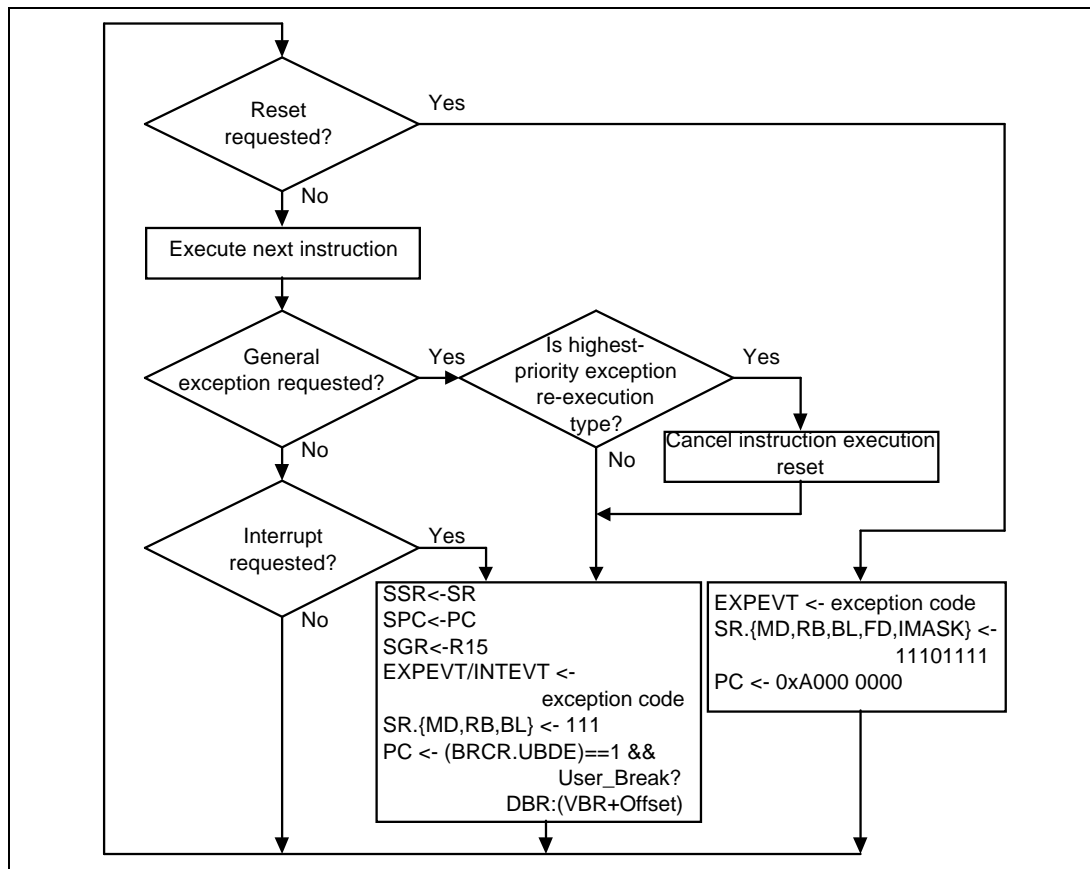


Figure 5.1 Instruction Execution and Exception Handling

5.4.2 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 are set to their post-reset state, the registers of the other modules retain their contents prior to the exception, 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 Hardware manual section 20, User Break Controller. 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, the SPC and SSR are saved and then the BL bit in SR is cleared to 0, to enable multiple exception state acceptance.

5.4.3 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 the PC and the SSR contents to SR, the CPU returns from the exception handling routine by branching to the SPC address. If the 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.

5.5 Description of Exceptions

The various exception handling operations are described here, covering exception sources, transition addresses, and the processor state after transition.

5.5.1 Resets

(1) Power-On Reset

- Source:
 - $\overline{\text{SCK2}}$ pin high level and $\overline{\text{RESET}}$ pin low level
 - When $\text{WTCSR.WT/IT} = 1$ && $\text{WTCSR.RTS} = 0$ and Watch Dog Timer is overflow, see Hardware manual section 10, Clock Oscillation Circuits.
- Transition address: H'A000 0000
- Transition operations:

Exception code H'000 is set in the lower 12 bits of EXPEVT, initialization of VBR and SR is performed, and a branch is made to PC = H'A000 0000.

In the initialization process, 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 supporting 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.

```
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

- Source:
 - $\overline{\text{SCK2}}$ pin low level and $\overline{\text{RESET}}$ pin low level
 - When the BL bit in SR is 1 and an exception other than a userbreak is generated.
 - When WTCSR.RSTS =1 and Watch Dog Timer is overflow, see Hardware manual section 10, Clock Oscillation Circuits.
- 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 process, 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 supporting 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

Type	Reset State Transition Conditions		Internal States	
	SCK2	RESET	CPU	On-Chip Supporting Modules
Power-on reset	High	Low	Initialized	See Register Configuration in each section
Manual reset	Low	Low	Initialized	

(3) Hitachi-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 process, 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 supporting module initialization is performed. For details, see the register descriptions in the relevant sections.

Hitachi-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 bit) at which this exception occurred is set in TEA, and the corresponding virtual page number (22 bit) is set in PTEH(31-10). The PTEH.ASID indicates the ASID where the 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 process, 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 supporting 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) Data TLB Multiple-Hit Exception

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

The virtual address (32 bit) at which this exception occurred is set in TEA, and the corresponding virtual page number (22 bit) is set in PTEH(31-10). The PTEH.ASID indicates the ASID where the 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 process, 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 supporting 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;  
}
```

5.5.2 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). The PTEH.ASID indicates the ASID where the exception occurred.

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

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 in SR are set to 1, 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;
    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). The PTEH.ASID indicates the ASID where the exception occurred.

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

Exception code H'040 is set in EXPEVT. The BL, MD, and RB bits in SR are set to 1, 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;  
    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). The PTEH.ASID indicates the ASID where the exception occurred.

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

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

```
Initial_page_write_exception()  
{  
    TEA = EXCEPTION_ADDRESS;  
    PTEH.VPN = PAGE_NUMBER;  
    SPC = PC;  
    SSR = SR;  
    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 enabled	Cannot be accessed
01	Read/write access enabled	Cannot be accessed
10	Only read access enabled	Only read access enabled
11	Read/write access enabled	Read/write access enabled

- 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). The PTEH.ASID indicates the ASID where the exception occurred.

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

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 in SR are set to 1, 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;  
    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 enabled	Cannot be accessed
1	Access enabled	Access enabled

- 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). The PTEH.ASID indicates the ASID where the exception occurred.

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

Exception code $H'0A0$ is set in EXPEVT. The BL, MD, and RB bits in SR are set to 1, 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;  
    EXPEVT = H'000000A0;  
    SR.MD = 1;  
    SR.RB = 1;  
    SR.BL = 1;  
    PC = VBR + H'00000100;  
}
```

(6) Data Address Error

- Source:
 - Word data access from a non-word boundary ($4n + 1$ or $4n + 3$)
 - Longword data access from a non-longword boundary ($4n + 1$, $4n + 2$, or $4n + 3$)
 - Quadword data access from a non-quadword boundary ($8n + 1$, $8n + 2$, or $8n + 3$, $8n + 4$, $8n + 5$, $8n + 6$, $8n + 7$)
 - 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). The PTEH.ASID indicates the ASID where the exception occurred.

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

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 in SR are set to 1, and a branch is made to PC = VBR + H'0100.

See section 3, MMU.

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


(7) Instruction Address Error

- Source:
 - Instruction fetch from an odd address ($4n + 1$ or $4n + 3$)
 - 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). The PTEH.ASID indicates the ASID where the exception occurred.

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

Exception code H'0E0 is set in EXPEVT. The BL, MD, and RB bits in SR are set to 1, and a branch is made to PC = VBR + H'0100. See section 3, MMU.

Instruction_address_error()

```
{
    TEA = EXCEPTION_ADDRESS;
    PTEH.VPN = PAGE_NUMBER;
    SPC = PC;
    SSR = SR;
    EXPEVT = H'000000E0;
    SR.MD = 1;
    SR.RB = 1;
    SR.BL = 1;
    PC = VBR + H'00000100;
}
```

(8) Unconditional Trap

- Condition: 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 the SPC. The value of SR when the TRAPA instruction is executed are saved in SSR. The 8-bit immediate value in the TRAPA instruction is multiplied by 4, and the result is set in TRA (9–2). Exception code H'160 is set in EXPEVT. The BL, MD, and RB bits in SR are set to 1, and a branch is made to PC = VBR + H'0100.

TRAPA_exception()

```
{
```

```

    SPC = PC + 2;
    SSR = SR;
    TRA = imm << 2;
    EXPEVT = H'00000160;
    SR.MD = 1;
    SR.RB = 1;
    SR.BL = 1;
    PC = VBR + H'00000100;
}

```

(9) General Illegal Instruction Exception

- Source:

- 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 the SPC and SSR.

Exception code H'180 is set in EXPEVT. The BL, MD, and RB bits in SR are set to 1, 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;
    EXPEVT = H'00000180;
    SR.MD = 1;
    SR.RB = 1;
    SR.BL = 1;
    PC = VBR + H'00000100;
}

```

(10) Slot Illegal Instruction Exception

- Source:
 - 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 the PC in a delay slot
Instructions that modify the 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 the SPC. The SR contents when this exception occurred are saved in SSR.

Exception code H'1A0 is set in EXPEVT. The BL, MD, and RB bits in SR are set to 1, 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;  
    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 the SPC and SSR.

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

```
General_fpu_disable_exception()  
{  
    SPC = PC;  
    SSR = SR;  
    EXPEVT = H'00000800;  
    SR.MD = 1;  
    SR.RB = 1;  
    SR.BL = 1;  
    PC = VBR + H'00000100;  
}
```

Note: FPU instructions are instructions in which the first 4 bits of the instruction code are F, and the LDS, STS, LDC.L, and STC.L instructions corresponding to FPUL and FPSCR.

(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 the SPC. The SR contents when this exception occurred are saved in SSR.

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

```
Slot_fpu_disable_exception()  
{  
    SPC = PC - 2;  
    SSR = SR;  
    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 breakpoint controller
- Transition address: VBR + H'0000 0100
- Transition operations:

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

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

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

For details of the PC, etc., when a data break is set, see Hardware manual section 20, User Break Controller.

```
user_break_exception()  
{  
    SPC = (pre-execution break ? PC : PC + 2);  
    SSR = SR;  
    EXPEVT = H'000001E0;  
    SR.MD = 1;  
    SR.RB = 1;  
    SR.BL = 1;  
    PC = (BRCCR.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 the SPC and SSR. Exception code H'120 is set in EXPEVT. The BL, MD, and RB bits in SR are set to 1, and a branch is made to PC = VBR + H'0100.

```
FPU_exception()  
{  
    SPC = PC;  
    SSR = SR;  
    EXPEVT = H'00000120;  
    SR.MD = 1;  
    SR.RB = 1;  
    SR.BL = 1;  
    PC = VBR + H'00000100;  
}
```

5.5.3 Interrupts

(1) NMI

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

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

Exception code H'1C0 is set in INTEVT. The BL, MD, and RB bits in SR are set to 1, 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. See Hardware manual section 19, Interrupt Controller, for details.

```

NMI ( )
{
    SPC = PC;
    SSR = SR;
    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 less 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 the SPC. The SR contents at the time of acceptance are set in SSR.

The code corresponding to the IRL (3–0) level is set in INTEVT. See Hardware manual table 19.5, Interrupt Exception Handling Sources and Priority Order, for the corresponding codes. The BL, MD, and RB bits in SR are set to 1, 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. See Hardware manual section 19, Interrupt Controller, for details.

```

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

```

(3) Supporting Module Interrupts

- Source: The interrupt mask bit setting in SR is less than the supporting module ([Hitachi-UDI](#), DMAC, TMU, RTC, SCI, SCIF, WDT, REF) interrupt 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 the SPC. The SR contents at the time of acceptance are set in SSR.

The code corresponding to the interrupt source is set in INTEVT. The BL, MD, and RB bits in SR are set to 1, 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 level setting registers (IRPA–IRPC) in the interrupt controller. See Hardware manual section 19, Interrupt Controller, for details.

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


5.5.4 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 levels 1 and 2.
- b. The delay slot instruction is checked for priority levels 1 and 2.
- c. 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.)
- d. 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 (BSR, BSRF, JSR PC \emptyset PR operation) is inhibited.

Section 6 Floating-Point Unit

6.1 Overview

The FPU has the following features:

- Conforms to IEEE754 standard
- 32 single-precision floating-point registers (can also be referenced as 16 double-precision registers)
- Two rounding modes: Round to Nearest and Round to Zero
- Two normalization 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 DF bit in SR is set to 1, the floating-point unit (FPU) is disabled, and an attempt to execute an FPU instruction will cause an illegal instruction 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 SH4 can handle single-precision and double-precision floating-point numbers, using the formats shown in figures 6.1 and 6.2.

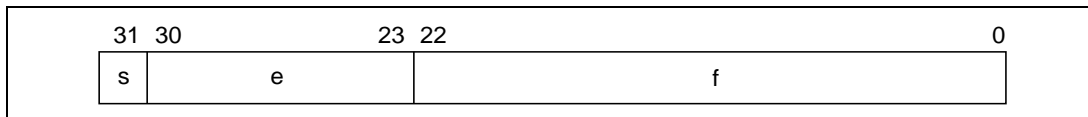


Figure 6.1 Format of Single-Precision Floating-Point Number

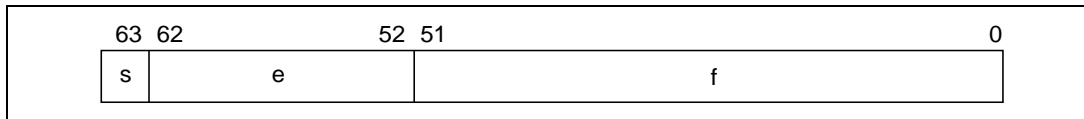


Figure 6.2 Format of Double-Precision Floating-Point Number

The exponent is expressed in biased form, as follows:

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

The range of unbiased exponent E is $E_{\min} - 1$ to $E_{\max} + 1$. The two values $E_{\min} - 1$ and $E_{\max} + 1$ are distinguished as follows. $E_{\min} - 1$ indicates zero (both positive and negative sign) and a denormalized number, and $E_{\max} + 1$ indicates positive or negative infinity or **not-a-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 $f! = 0$, v is **not-a-number** (NaN) irrespective of sign s
- If $E = E_{\max} + 1$ and $f = 0$, $v = (-1)^s$ (infinity) [positive or negative infinity]
- If $E_{\min} < E <= E_{\max}$, $v = (-1)^s 2^E (1.f)$ [normalized number]
- If $E = E_{\min} - 1$ and $f! = 0$, $v = (-1)^s 2^{E_{\min}} (0.f)$ [denormalized number]
- If $E = E_{\min} - 1$ and $f = 0$, $v = (-1)^s 0$ [positive or negative zero]

Table 6.2 shows the range of each number in hexadecimal.

Table 6.2 Floating-Point Number Ranges

Type	Single-Precision	Double-Precision
sNaN	7FFFFFFF to 7FC00000	7FFFFFFF FFFFFFFF to 7FF80000 00000000
qNaN	7FBFFFFFFF to 7F800001	7FF7FFFF FFFFFFFF to 7FF00000 00000001
+Infinity	7F800000	7FF00000 00000000
+Normalized	7F7FFFFFFF to 00800000	7FEFFFFFFF FFFFFFFF to 00100000 00000000
+Denormalized	007FFFFFFF to 00000001	000FFFFFFF FFFFFFFF to 00000000 00000001
+Zero	00000000	00000000 00000000
−Zero	80000000	80000000 00000000
−Denormalized	80000001 to 807FFFFFFF	80000000 00000001 to 800FFFFFFF FFFFFFFF
−Normalized	80800000 to FF7FFFFFFF	80100000 00000000 to FFEFFFFFFF FFFFFFFF
−Infinity	FF800000	FFF00000 00000000
qNaN	FF800001 to FFBFFFFFFF	FFF00000 00000001 to FFF7FFFF FFFFFFFF
sNaN	FFC00000 to FFFFFFFF	FFF80000 00000000 to FFFFFFFF FFFFFFFF

6.2.2 Non-Numbers (NaN)

Figure 6.3 shows the bit pattern of **not-a-number** (NaN). A value is NaN in the following case:

- Sign bit: don't care
- Exponent field: all 1 bits
- Fraction field: at least 1 of bits excluding MSB

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

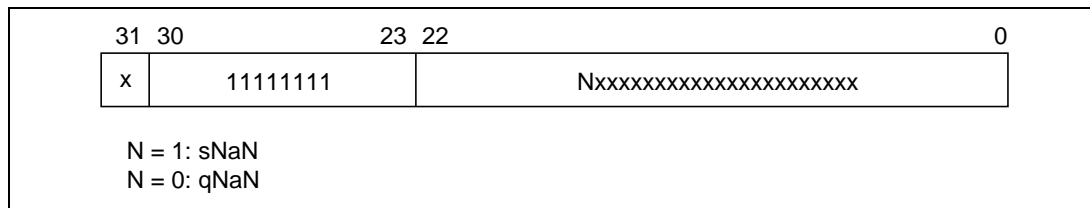


Figure 6.3 NaN Bit Pattern of Single-precision

If a **signaling NaN** (sNaN) is input in an operation that generates a floating-point value other than copy, FABS, or FNEG:

- When the EV bit in the FPSCR register is 0, the operation result (output) is a quiet NaN (qNaN).
- When the EV bit in the FPSCR register is 1, an invalid operation exception will be generated. In this case, the contents of the operation destination register do not change.

If a quiet NaN is input in an operation that generates a floating-point value, and a signaling NaN has not been input in that operation, the output will **almost** always be a quiet NaN irrespective of the setting of the EV bit in the FPSCR register. An exception will not be generated in this case.

The value of a qNaN generated by the SH4 as an operation result is always as follows.

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

See the individual instruction descriptions for details of floating-point operations when **not-a-number** (NaN) is input.

6.2.3 Denormalized Numbers

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

When the DN bit in the FPU's status register FPSCR is 1, a denormalized number (operand source 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 (operand source or operation result) is processed as it is. See the individual instruction descriptions for details of floating-point operations when a denormalized number is input.

6.3 Registers

6.3.1 Floating-Point Registers

Figure 2.2 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.

- Floating-point registers, FPRi_BANKj (32 registers)
FPR0_BANK0–FPR15_BANK0
FPR0_BANK1–FPR15_BANK1
- Single-precision floating-point registers, FRi (16 registers)
When FPSCR.FR = 0, FR0–FR15 indicate to FPR0_BANK0–FPR15_BANK0;
when FPSCR.FR = 1, FR0–FR15 indicate FPR0_BANK1–FPR15_BANK1.
- 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}
- 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 indicate to FPR0_BANK1–FPR15_BANK1;
when FPSCR.FR = 1, XF0–XF15 indicate FPR0_BANK0–FPR15_BANK0.
- Double-precision floating-point extended registers, XD_i (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 =	XF0	XF4	XF8	XF12
	XF1	XF5	XF9	XF13
	XF2	XF6	XF10	XF14
	XF3	XF7	XF11	XF15

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

Figure 6.4 Floating-Point Registers

6.3.2 Floating-Point Unit Status/Control Register (FPSCR)

- Floating-Point Unit Status/Control Register, FPSCR (32-bit, initial value = undefined)

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

— 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: An FMOV instruction comprises a single-precision floating-point FMOV.

SZ = 1: An FMOV instruction comprises pair single-precision floating-point FMOVs (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 operation of graphics-related instructions is undefined).

Do not set both SZ and PR to 1. This setting is reserved.

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

— DN: Denormalization Mode

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

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

		FPU Error (E)	Invalid Op. (V)	Zero Div. (Z)	Overflow (O)	Underflow (U)	Indexact (I)
Cause	FPU exception source field	Bit 17	Bit 16	Bit 15	Bit 14	Bit 13	Bit 12
Enable	FPU exception enable field	Non	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7
Flag	FPU exception request field	Non	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2

When an FPU exception is requested, the bit corresponding the cause/flag field is set to 1. Each time an FPU operation instruction is executed, the cause field is zeroized first. The flag field retains the set value of 1 until zeroized by software.

— RM: Rounding Mode

RM = 00: Round to Nearest

RM = 01: Round to Zero

RM = 10: Reserved

RM = 11: Reserved

- Notes: 1. The **SZ** and **PR** bits and cause, enable, and flag fields for exceptions O/U/I have been added.
2. The cause field for exception E has been added.

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 floating-point number, the processing flow is as follows:

R1 \emptyset (LDS instruction) \emptyset FPUL \emptyset (FLOAT instruction) \emptyset FR1

6.4 Rounding

In a floating-point instruction, rounding is performed when generating the final operation result from the intermediate result. So the results of combinational instructions like as FMAC/FTRV/FIPR is/are different from that of using only basic instructions as FADD/FSUB/FMUL. Because number of Rounding is different, one time at FMAC and 2times at 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.

However, if the value above the Round bit of the unrounded value is the maximum expressible absolute value, the value is rounded to the maximum expressible absolute value if the Round bit is 0, or to infinity if the Round bit is 1.

- Round to Zero

The digits below the Round bit of the unrounded value are discarded.

6.5 Floating-Point Exceptions

FPU-related exceptions are as follows:

- General illegal instruction/slot illegal instruction exception
Occurs if an FPU instruction is executed when $SR.DF = 1$.
- FPU exceptions
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 fields is set to 1 and that in the flag fields is accumulated to 1. When an exception source does not occurs, the corresponding bit in the cause fields is set to 0 and that in the flag fields is not changed.

- Enable/disable Exception processings
SH4 supports the enable exception processings and the disable exception processing.
Enable exception processings is occurred as follows:
 - FPU error (E): When $FPSCR.DN = 0$ and a denormalized number is input
 - Invalid operation (V): When $FPSCR.EN.V = 1$ and (instruction = FTRV or (instruction \neq FTRV and an invalid operation))
 - Division by zero (Z): When $FPSCR.EN.Z = 1$ and division with a zero divisor
 - Overflow (O): When $FPSCR.EN.O = 1$ and the possibility of the operation result overflows
 - Underflow (U): When $FPSCR.EN.U = 1$ and the possibility of the operation result underflows
 - Inexact exception (I): When $FPSCR.EN.I = 1$ and the possibility of the operation result inexact

Each possibility is shown at each instruction description. All enable exceptions raised by the FPU are mapped onto the same SH Exception Event. The semantics of the exception are determined by software by reading the system register $FPSCR$ and interpreting the information maintained there. That all bits of $FPSCR$ cause field are not set show the enable field O, U, I and V (FTRV case only) is/are set, but Actual exception does not occurred. And any enable exception operations does not change the destination register.

Not above cases, the FPU do disable exception processing. All processing set 1 to the corresponding bits to sources V, Z, O, U, and I and each exception has each disable exception processings.

- Invalid operation (V): produce qNaN as a result.
- Division by zero (Z): produce a correctly signed infinity.
- Overflow (O): When rounding mode = RZ, produce a correctly signed maximum normalized number. When rounding mode = RN, produce a correctly signed infinity.
- Underflow (U): When FPSCR.DN = 0, produce a correctly signed zero. When FPSCR.DN = 1, produce denormalized number.
- Inexact exception (I): produce the inexact result.

6.6 Graphic Support future

SH4 support 2 types graphic future. The first future is the new instruction for geometry operations. The other is two single-precision transfer instructions for fast data movement.

6.6.1 Geometric Instructions

Geometric Instructions are approximate. To keep minimum hardware and to get high-performance, SH4 neglect relative small value of the partial results of 4 multiplier. So, the result of each operation have an error shown below.

$$\begin{aligned} \text{MAX ERROR} &= \text{Maximum result of multiplier} * 2^{(-25)} \\ &+ \text{MAX}(\text{result value} * (2^{(-23)}, 2^{(-149)})) \end{aligned}$$

(1) FIPR FVm, FVn (m, n: 0, 4, 8, 12)

This instruction is basically used for

- inner product (m != n) : typically this operation is used by the judgment of front/back of polygon surface.
- sum of square of each element (m = n): typically this operation is used by getting the length of a vector.

FIPR always set 1 to the inexact bit in the cause/flag fields for high performance operations. So, if the corresponding bit in the enable field be set, enable exception processing is done.

(2) FTRV XMTRX, FVn (n: 0, 4, 8, 12)

This instruction is basically used for

- matrix (4*4) * vector(4) : typically this operation is used by vector transformation(4 dimension), such as view point changing, changing angle/moving, and so on. Basically, affine transaction needs 4 * 4 matrix for angle + parallel movement. So SH4 supports 4 dimension operation.

— matrix (4*4) * matrix (4*4): for this operation, FTRV have to be used 4 times. For high performance vector transformations, each vectors had better be changed one time than many times on transformation. So transation matrixs are generated on two kinds of transformation.

FTRV always set 1 to the inexact bit in the cause/flag fields for high performance operations. So, if the corresponding bit in the enable filed be set, enable exception processing is done. Also for high performance, FTRV cannot check all data types in contentt of register. If the V bit in the enable filed be set, enable exception processing is done.

(3) FRCHG

This instruction change the bank registers. At example, for FTRV instruction, programmers have to set the MATRIX on the back. But to make the matrix, it is easy to use front register. If users can use LDC to FPSCR instruction, this instruction cost 4~5 machine cycles because keeping the FPU status. So, this instruction is supported for 1 machine cycle changing the FPSCR.FR bit.

6.6.2 two single precision data transfer

After new powerful geometric instruction,s SH4 supports fast data movment instructions. When FPSCR.SZ = 1, SH4 can move data at two single-precision transfer.

(1) FMOV DRm/XDm, DRn/XDn (m, ;n: 0, 2, 4, 6, 8, 10, 12,14)

(2) FMOV DRm/XDm, @Rn (m: 0, 2, 4, 6, 8, 10, 12,14 n=0~15)

These instructions can permit programmers to move two single-precision(64bit) transfer. So they can get twice performance of data band width.

(3) FSCHG

This instruction change the size bit of FPSCR. the perpuse is the same as FRCHG.

Section 7 Instruction Set

7.1 Execution Environment

PC: At the beginning of an instruction execution, PC indicates the instruction address of the instruction.

Data Size and Data Types: The SH4 instruction set is implemented with fixed-length 16-bit width instructions. SH4 accesses memory with several data sizes: byte (8 bits), word (16 bits), longword (32 bits), and quadword (64 bits). Single precision floating point data (32 bits) can be accessed from / to memory using either longword or quadword data size. Double precision floating point data (64 bits) can be accessed from / to memory using longword data size. When double precision floating operation is designated (FPSCR.PR=1), the result of quadword access operation is undefined. When SH4 moves byte and word size data from memory to register, the data is sign-extended.

Load/Store Architecture: The SH4 features a load-store architecture in which basic operations are executed in registers. Operations requiring memory access are executed in registers following register loading, except for bit-manipulation operations such as logical AND functions, which are executed directly in memory.

Delayed Branch: SH4 branch instructions and RTE are delayed branches, except two branch instructions: BF and BT. On a delayed branch, the next instruction of the branch is executed before the branch target instruction. This execution slot after the delayed branch is called "delay slot". For example, BRA execution sequence is as follows:

<u>static sequence</u>		<u>dynamic sequence</u>		
BRA	TARGET	BRA	TARGET	
ADD	R1, R0	ADD	R1, R0	; ADD in the delay slot is executed prior to
next_2		target_instr		; branching to TARGET.

Delay Slot: Some instructions cause the slot illegal instruction exception when they are executed in a delay slot, see Section 5 "Exception Handling". The next instruction of not-taken BF/S and BT/S is also a delay slot instruction.

T bit: The T bit in the status register (SR) is used to indicate the result of compare operations, and is referred with a conditional branch instructions. For example, the following shows a conditional branch sequence.

ADD	#1, R0	;T bit not modified by ADD operation
CMP/EQ	R1, R0	;T bit set to 1 when R0 = R1
BT	TARGET	;branch taken to TARGET when T bit = 1 (R0 = R1)

SR bits on RTE delay slot: In RTE delay slot the status register (SR) bit are referred as following. Instruction access uses MD bit before change, data access uses MD bit after change. Other bits: S, T, M, Q, FD, BL and RB after change are used for instruction execution of the delay slot. STC and STC.L SR instructions access all SR bits after change.

Constant Values: An 8-bit constant value can be specified in an instruction code, immediate value. Also 16- and 32-bit constant values can be defined in memory, literal constant value, and they can be referred with PC-relative load instructions:

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

There is no PC-relative load instruction for a floating point number. However SH4 has FLDI0 and FLDI1 instructions to set 0.0 or 1.0 to a single precision floating point register.

7.2 Addressing Modes

Addressing modes and effective address calculations are shown in Table 7.1. When a location in the virtual memory space is accessed, MMUCR.AT = 1, the effective address is translated to a physical memory address. The lowest 8 bits in PTEH are also referred as ASID of the access if the multiple virtual memory spaces system were chosen, MMUCR.SV = 0, see Section 3 "Memory Management Unit".

Table 7.1 Addressing Modes and Effective Addresses

Addressing Mode	Instruction Format	Effective Address Calculation Method	Calculation Formula	
Indirect Addressing	@Rn	Effective address is the content of Rn.	$Rn \rightarrow EA$ (EA: effective address)	
Post-increment Addressing	@Rn+	Effective address is the content of Rn. Additionally, Rn is increased by a byte number of the access size: 1 on a byte access, 2 on a word access, 4 on a longword access, 8 on a quadword access.	$Rn \rightarrow EA$ Byte access: $Rn + 1 \rightarrow Rn$ Word access: $Rn + 2 \rightarrow Rn$ Longword access: $Rn + 4 \rightarrow Rn$ Quadword access: $Rn + 8 \rightarrow Rn$	
Pre-decrement Addressing	@-Rn	Effective address is the result of subtracting the byte number of the access size from the content of Rn. The result of the subtraction is also stored in Rn. The number is 1 on a byte access, 2 on a word access, 4 on a longword access, or 8 on a quadword access.	Byte access: $Rn - 1 \rightarrow Rn$ Word access: $Rn - 2 \rightarrow Rn$ Longword access: $Rn - 4 \rightarrow Rn$ Quadword access: $Rn - 8 \rightarrow Rn$ $Rn \rightarrow EA$	
Displacement Addressing	@(disp:4, Rn)	Effective address is the result of adding the content of Rn and the displacement which is the result of multiplying the zero-extended value of the 4-bit disp by the access size: 1 on a byte access, 2 on a word access, 4 on a longword access.	Byte: $Rn + disp \rightarrow EA$ Word: $Rn + disp \times 2 \rightarrow EA$ Longword: $Rn + disp \times 4 \rightarrow EA$ (disp is zero-extended.)	
Index Addressing	@(R0, Rn)	Effective address is the sum of the contents of Rn and R0.	$Rn + R0 \rightarrow EA$	
GBR-based Displacement Addressing	@(disp:8, GBR)	Effective address is the result of adding the content of GBR and a displacement which is the result of multiplying the zero-extended value of the 8-bit disp by the access size: 1 on a byte access, 2 on a word access, 4 on a longword access.	Byte: $GBR + disp \rightarrow EA$ Word: $GBR + disp \times 2 \rightarrow EA$ Longword: $GBR + disp \times 4 \rightarrow EA$ (disp is zero-extended.)	
GBR-based Index Addressing	@(R0, GBR)	Effective address is the sum of the contents of GBR and R0.	$GBR + R0 \rightarrow EA$	

Table 7.1 Addressing Modes and Effective Addresses (continue)

Addressing Mode	Instruction Format	Effective Address Calculation Method	Calculation Formula
PC-relative Addressing	@(disp:8, PC)	Effective address is the result of adding 4, PC(instruction address of the instruction), and the displacement which is the result of multiplying the zero-extended value of the 8-bit disp by the access size: 2 on a word access, or 4 on a longword access. On a longword access, the lowest 2 bits of PC are masked to zero.	Word: $PC+4 + \text{disp} \times 2 \rightarrow EA$ Longword: $(PC \& 0xFFFF\text{FFFC}) + 4 + \text{disp} \times 4 \rightarrow EA$ (PC addresses the instruction.) (disp is zero-extended.)
PC-relative Branch	disp:8	Target address is the result of adding 4, PC (instruction address of the instruction), and the displacement which is the result of multiplying the sign-extended value of the 8-bit disp by 2.	$PC+4 + \text{disp} \times 2 \rightarrow \text{Branch_Target}$ (PC addresses the instruction.) (disp is sign-extended.)
	disp:12	Target address is the result of adding 4, PC (instruction address of the instruction), and the displacement which is the result of multiplying the sign-extended value of the 12-bit disp by 2.	$PC+4 + \text{disp} \times 2 \rightarrow \text{Branch_Target}$ (PC addresses the instruction.) (disp is sign-extended.)
PC-relative Branch	Rn	Effective address is the sum of PC and Rn contents.	$PC+4 + Rn \rightarrow \text{Branch_Target}$
Immediate value	#imm:8	8-bit immediate value 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 displacement (disp), the assembler descriptions in this manual show the value before scaling (x1, x2, or x4) is performed according to the operand size. This is done to clarify the operation of the IC. Refer to the relevant assembler notation rules for the actual assembler descriptions.

@ (disp:4, Rn) ; displacement addressing
 @ (disp:8, GBR) ; GBR-based displacement addressing
 @ (disp:8, PC) ; PC-relative addressing
 disp:8, disp:12 ; PC-relative branch

7.3 Instruction Set

The notation, used in the following lists of the SH4 instructions, is described in Table 7.2.

Table 7.2 Instruction List Notation

Item	Format	Explanation
Instruction mnemonics	OP.Sz SRC,DEST	OP: Operation code Sz: Size SRC: Source operand DEST: Source and/or Destination operand
Operation summary	→, ← (xx) M/Q/T & ^ ~ <<n, >>n	Direction of transfer Memory operand Flag bits in SR Logical AND of each bit Logical OR of each bit Exclusive OR of each bit Logical NOT of each bit n-bit shift
Instruction code	MSB ↔ LSB	mmmm: Register number (Rm, FRm) nnnn: Register number (Rn, FRn) 0000: R0, FR0 0001: R1, FR1 : 1111: R15, FR15 mmm: Register number (DRm, XDm, Rm_BANK) nnn: Register number (DRm, XDm, Rn_BANK) 000: DR0, XD0, R0_BANK 001: DR2, XD2, R1_BANK : 111: DR14, XD14, R7_BANK mm: Register number (FVm) nn: Register number (FVn) 00: FV0 01: FV4 10: FV8 11: FV12 iiii: immediate value dddd: Displacement
Privileged mode		Indicates whether privileged mode applies
T bit		Value of T bit after instruction is executed —: No change

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

Table 7.3 Fixed Point Transfer Instructions

Instruction		Operation	Code	Privileged	T Bit
MOV	#imm,Rn	imm → Sign extension → Rn	1110nnnniiiiiii	—	—
MOV.W	@(disp,PC),Rn	(disp × 2 + PC + 4) → Sign extension → Rn	1001nnnnddddddd	—	—
MOV.L	@(disp,PC),Rn	(disp × 4 + PC & 0xffff fffc + 4) → Rn	1101nnnnddddddd	—	—
MOV	Rm,Rn	Rm → Rn	0110nnnnnnmm0011	—	—
MOV.B	Rm,@Rn	Rm → (Rn)	0010nnnnnnmm0000	—	—
MOV.W	Rm,@Rn	Rm → (Rn)	0010nnnnnnmm0001	—	—
MOV.L	Rm,@Rn	Rm → (Rn)	0010nnnnnnmm0010	—	—
MOV.B	@Rm,Rn	(Rm) → Sign extension → Rn	0110nnnnnnmm0000	—	—
MOV.W	@Rm,Rn	(Rm) → Sign extension → Rn	0110nnnnnnmm0001	—	—
MOV.L	@Rm,Rn	(Rm) → Rn	0110nnnnnnmm0010	—	—
MOV.B	Rm,@-Rn	Rn-1 → Rn, Rm → (Rn)	0010nnnnnnmm0100	—	—
MOV.W	Rm,@-Rn	Rn-2 → Rn, Rm → (Rn)	0010nnnnnnmm0101	—	—
MOV.L	Rm,@-Rn	Rn-4 → Rn, Rm → (Rn)	0010nnnnnnmm0110	—	—
MOV.B	@Rm+,Rn	(Rm) → Sign extension → Rn, Rm + 1 → Rm	0110nnnnnnmm0100	—	—
MOV.W	@Rm+,Rn	(Rm) → Sign extension → Rn, Rm + 2 → Rm	0110nnnnnnmm0101	—	—
MOV.L	@Rm+,Rn	(Rm) → Rn, Rm + 4 → Rm	0110nnnnnnmm0110	—	—
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)	0001nnnnnnmmdddd	—	—
MOV.B	@(disp,Rm),R0	(disp + Rm) → Sign extension → R0	10000100nnnnmmdd	—	—
MOV.W	@(disp,Rm),R0	(disp × 2 + Rm) → Sign extension → R0	10000101nnnnmmdd	—	—
MOV.L	@(disp,Rm),Rn	(disp × 4 + Rm) → Rn	0101nnnnnnmmdd	—	—
MOV.B	Rm,@(R0,Rn)	Rm → (R0 + Rn)	0000nnnnnnmm0100	—	—
MOV.W	Rm,@(R0,Rn)	Rm → (R0 + Rn)	0000nnnnnnmm0101	—	—
MOV.L	Rm,@(R0,Rn)	Rm → (R0 + Rn)	0000nnnnnnmm0110	—	—
MOV.B	@(R0,Rm),Rn	(R0 + Rm) → Sign extension → Rn	0000nnnnnnmm1100	—	—
MOV.W	@(R0,Rm),Rn	(R0 + Rm) → Sign extension → Rn	0000nnnnnnmm1101	—	—
MOV.L	@(R0,Rm),Rn	(R0 + Rm) → Rn	0000nnnnnnmm1110	—	—
MOV.B	R0,@(disp,GBR)	R0 → (disp + GBR)	11000000ddddddd	—	—
MOV.W	R0,@(disp,GBR)	R0 → (disp × 2 + GBR)	11000001ddddddd	—	—
MOV.L	R0,@(disp,GBR)	R0 → (disp × 4 + GBR)	11000010ddddddd	—	—
MOV.B	@(disp,GBR),R0	(disp + GBR) → Sign extension → R0	11000100ddddddd	—	—
MOV.W	@(disp,GBR),R0	(disp × 2 + GBR) → Sign extension → R0	11000101ddddddd	—	—
MOV.L	@(disp,GBR),R0	(disp × 4 + GBR) → R0	11000110ddddddd	—	—
MOVA	@(disp,PC),R0	disp × 4 + PC & 0xffff fffc + 4 → R0	11000111ddddddd	—	—
MOVT	Rn	T → Rn	0000nnnn00101001	—	—
SWAP.B	Rm,Rn	Rm → Swap the bottom two bytes → Rn	0110nnnnnnmm1000	—	—
SWAP.W	Rm,Rn	Rm → Swap the two words → Rn	0110nnnnnnmm1001	—	—
XTRCT	Rm,Rn	Rm<<16 Rn>>16 → Rn	0010nnnnnnmm1101	—	—

Table 7.4 Arithmetic Instructions

Instruction		Operation	Code	Privileged	T Bit
ADD	Rm, Rn	$Rn + Rm \rightarrow Rn$	0011nnnnnnmm1100	—	—
ADD	#imm, Rn	$Rn + imm \rightarrow Rn$	0111nnnniiiiiii	—	—
ADDC	Rm, Rn	$Rn + Rm + T \rightarrow Rn$, Carry $\rightarrow T$	0011nnnnnnmm1110	—	Carry
ADDV	Rm, Rn	$Rn + Rm \rightarrow Rn$, Overflow $\rightarrow T$	0011nnnnnnmm1111	—	Overflow
CMP/EQ	#imm, R0	If $R0 = imm$, $1 \rightarrow T$, else $0 \rightarrow T$	10001000iiiiiii	—	result
CMP/EQ	Rm, Rn	If $Rn = Rm$, $1 \rightarrow T$, else $0 \rightarrow T$	0011nnnnnnmm0000	—	result
CMP/HS	Rm, Rn	If $Rn \geq Rm$ unsigned, $1 \rightarrow T$, else $0 \rightarrow T$	0011nnnnnnmm0010	—	result
CMP/GE	Rm, Rn	If $Rn \geq Rm$ signed, $1 \rightarrow T$, else $0 \rightarrow T$	0011nnnnnnmm0011	—	result
CMP/HI	Rm, Rn	If $Rn > Rm$ unsigned, $1 \rightarrow T$, else $0 \rightarrow T$	0011nnnnnnmm0110	—	result
CMP/GT	Rm, Rn	If $Rn > Rm$ signed, $1 \rightarrow T$, else $0 \rightarrow T$	0011nnnnnnmm0111	—	result
CMP/PZ	Rn	If $Rn \neq 0$, $1 \rightarrow T$, else $0 \rightarrow T$	0100nnnn00010001	—	result
CMP/PL	Rn	If $Rn > 0$, $1 \rightarrow T$, else $0 \rightarrow T$	0100nnnn00010101	—	result
CMP/STR	Rm, Rn	If any byte in $(Rn \wedge Rm) = 0$, $1 \rightarrow T$, else $0 \rightarrow T$	0010nnnnnnmm1100	—	result
DIV1	Rm, Rn	Iteration for division (Rn/Rm)	0011nnnnnnmm0100	—	result
DIV0S	Rm, Rn	Initialization for signed division	0010nnnnnnmm0111	—	result
DIV0U		Initialization for unsigned division	0000000000011001	—	0
DMULS.L	Rm, Rn	Signed $Rn \times Rm \rightarrow 64$ bits \rightarrow MACH, MACL	0011nnnnnnmm1101	—	—
DMULU.L	Rm, Rn	Unsigned $Rn \times Rm \rightarrow 64$ bits \rightarrow MACH, MACL	0011nnnnnnmm0101	—	—
DT	Rn	$Rn - 1 \rightarrow Rn$, if $Rn = 0$, $1 \rightarrow T$, else $0 \rightarrow T$	0100nnnn00010000	—	result
EXTS.B	Rm, Rn	Lowest byte in $Rm \rightarrow$ sign-extended $\rightarrow Rn$	0110nnnnnnmm1110	—	—
EXTS.W	Rm, Rn	Lower word in $Rm \rightarrow$ sign-extended $\rightarrow Rn$	0110nnnnnnmm1111	—	—
EXTU.B	Rm, Rn	Lowest byte in $Rm \rightarrow$ zero-extended $\rightarrow Rn$	0110nnnnnnmm1100	—	—
EXTU.W	Rm, Rn	Lower word in $Rm \rightarrow$ zero-extended $\rightarrow Rn$	0110nnnnnnmm1101	—	—
MAC.L	@Rm+, @Rn+	Signed $(Rn) \times (Rm) + MAC \rightarrow MAC$, $Rn + 4 \rightarrow Rn$, $Rm + 4 \rightarrow Rm$ ($32 \times 32 + 64 \rightarrow 64$ bits)	0000nnnnnnmm1111	—	—
MAC.W	@Rm+, @Rn+	Signed $(Rn) \times (Rm) + MAC \rightarrow MAC$, $Rn + 2 \rightarrow Rn$, $Rm + 2 \rightarrow Rm$ ($16 \times 16 + 64 \rightarrow 64$ bits)	0100nnnnnnmm1111	—	—
MUL.L	Rm, Rn	$Rn \times Rm \rightarrow MACL$ ($32 \times 32 \rightarrow 32$ bits)	0000nnnnnnmm0111	—	—
MULS.W	Rm, Rn	Signed $Rn \times Rm \rightarrow MAC$ ($16 \times 16 \rightarrow 32$ bits)	0010nnnnnnmm1111	—	—
MULU.W	Rm, Rn	Unsigned $Rn \times Rm \rightarrow MAC$ ($16 \times 16 \rightarrow 32$ bits)	0010nnnnnnmm1110	—	—
NEG	Rm, Rn	$0 - Rm \rightarrow Rn$	0110nnnnnnmm1011	—	—
NEGC	Rm, Rn	$0 - Rm - T \rightarrow Rn$, Borrow $\rightarrow T$	0110nnnnnnmm1010	—	Borrow
SUB	Rm, Rn	$Rn - Rm \rightarrow Rn$	0011nnnnnnmm1000	—	—
SUBC	Rm, Rn	$Rn - Rm - T \rightarrow Rn$, Borrow $\rightarrow T$	0011nnnnnnmm1010	—	Borrow
SUBV	Rm, Rn	$Rn - Rm \rightarrow Rn$, Underflow $\rightarrow T$	0011nnnnnnmm1011	—	Underflow

Table 7.5 Logical Instructions

Instruction		Operation	Code	Privileged	T Bit
AND	Rm, Rn	$Rn \& Rm \rightarrow Rn$	0010nnnnmmmm1001	—	—
AND	#imm, R0	$R0 \& imm \rightarrow R0$	11001001iiiiiii	—	—
AND.B	#imm, @(R0, GBR)	$(R0 + GBR) \& imm \rightarrow (R0 + GBR)$	11001101iiiiiii	—	—
NOT	Rm, Rn	$\sim Rm \rightarrow Rn$	0110nnnnmmmm0111	—	—
OR	Rm, Rn	$Rn Rm \rightarrow Rn$	0010nnnnmmmm1011	—	—
OR	#imm, R0	$R0 imm \rightarrow R0$	11001011iiiiiii	—	—
OR.B	#imm, @(R0, GBR)	$(R0 + GBR) imm \rightarrow (R0 + GBR)$	11001111iiiiiii	—	—
TAS.B	@Rn	If (Rn) = 0, $1 \rightarrow T$, else $0 \rightarrow T$, $1 \rightarrow$ MSB of (Rn)	0100nnnn00011011	—	result
TST	Rm, Rn	$Rn \& Rm$, if result = 0, $1 \rightarrow T$, else $0 \rightarrow T$	0010nnnnmmmm1000	—	result
TST	#imm, R0	$R0 \& imm$, if result = 0, $1 \rightarrow T$, else $0 \rightarrow T$	11001000iiiiiii	—	result
TST.B	#imm, @(R0, GBR)	$(R0 + GBR) \& imm$, if result = 0, $1 \rightarrow T$, else $0 \rightarrow T$	11001100iiiiiii	—	result
XOR	Rm, Rn	$Rn \wedge Rm \rightarrow Rn$	0010nnnnmmmm1010	—	—
XOR	#imm, R0	$R0 \wedge imm \rightarrow R0$	11001010iiiiiii	—	—
XOR.B	#imm, @(R0, GBR)	$(R0 + GBR) \wedge imm \rightarrow (R0 + GBR)$	11001110iiiiiii	—	—

Table 7.6 Shift Instructions

Instruction		Operation	Code	Privileged	T Bit
ROTL	Rn	$T \leftarrow Rn \leftarrow \text{MSB}$	0100nnnn00000100	—	MSB
ROTR	Rn	$\text{LSB} \rightarrow Rn \rightarrow T$	0100nnnn00000101	—	LSB
ROTCL	Rn	$T \leftarrow Rn \leftarrow T$	0100nnnn00100100	—	MSB
ROTCR	Rn	$T \rightarrow Rn \rightarrow T$	0100nnnn00100101	—	LSB
SHAD	Rm, Rn	Rm ³ 0: $Rn \ll Rm \rightarrow Rn$ Rm < 0: $Rn \gg Rm \rightarrow Rn$ arithmetically	0100nnnnnnnnnn1100	—	—
SHAL	Rn	$T \leftarrow Rn \leftarrow 0$	0100nnnn00100000	—	MSB
SHAR	Rn	$\text{MSB} \rightarrow Rn \rightarrow T$	0100nnnn00100001	—	LSB
SHLD	Rm, Rn	Rm ³ 0: $Rn \ll Rm \rightarrow Rn$ Rm < 0: $Rn \gg Rm \rightarrow Rn$ logically	0100nnnnnnnnnn1101	—	—
SHLL	Rn	$T \leftarrow Rn \leftarrow 0$	0100nnnn00000000	—	MSB
SHLR	Rn	$0 \rightarrow Rn \rightarrow T$	0100nnnn00000001	—	LSB
SHLL2	Rn	$Rn \ll 2 \rightarrow Rn$	0100nnnn00001000	—	—
SHLR2	Rn	$Rn \gg 2 \rightarrow Rn$	0100nnnn00001001	—	—
SHLL8	Rn	$Rn \ll 8 \rightarrow Rn$	0100nnnn00011000	—	—
SHLR8	Rn	$Rn \gg 8 \rightarrow Rn$	0100nnnn00011001	—	—
SHLL16	Rn	$Rn \ll 16 \rightarrow Rn$	0100nnnn00101000	—	—
SHLR16	Rn	$Rn \gg 16 \rightarrow Rn$	0100nnnn00101001	—	—

Table 7.7 Branch Instructions

Instruction		Operation	Code	Privileged	T Bit
BF	label	If $T = 0$, $\text{disp} \times 2 + \text{PC} + 4 \rightarrow \text{PC}$	10001011dddddddd	—	—
BF/S	label	If $T = 0$, $\text{disp} \times 2 + \text{PC} + 4 \rightarrow \text{PC}$ (delayed branch)	10001111dddddddd	—	—
BT	label	If $T = 1$, $\text{disp} \times 2 + \text{PC} + 4 \rightarrow \text{PC}$	10001001dddddddd	—	—
BT/S	label	If $T = 1$, $\text{disp} \times 2 + \text{PC} + 4 \rightarrow \text{PC}$ (delayed branch)	10001101dddddddd	—	—
BRA	label	$\text{disp} \times 2 + \text{PC} + 4 \rightarrow \text{PC}$ (delayed branch)	1010dddddddddddd	—	—
BRAF	Rn	$\text{Rn} + \text{PC} + 4 \rightarrow \text{PC}$ (delayed branch)	0000nnnn00100011	—	—
BSR	label	$\text{PC} + 4 \rightarrow \text{PR}$, $\text{disp} \times 2 + \text{PC} + 4 \rightarrow \text{PC}$ (delayed branch)	1011dddddddddddd	—	—
BSRF	Rn	$\text{PC} + 4 \rightarrow \text{PR}$, $\text{Rn} + \text{PC} + 4 \rightarrow \text{PC}$ (delayed branch)	0000nnnn00000011	—	—
JMP	@Rn	$\text{Rn} \rightarrow \text{PC}$ (delayed branch)	0100nnnn00101011	—	—
JSR	@Rn	$\text{PC} + 4 \rightarrow \text{PR}$, $\text{Rn} \rightarrow \text{PC}$ (delayed branch)	0100nnnn00001011	—	—
RTS		$\text{PR} \rightarrow \text{PC}$ (delayed branch)	0000000000001011	—	—

Table 7.8 System Control Instructions

Instruction	Operation	Code	Privileged	T Bit
CLRMACH	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	0100mmmm11110110	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	000000000111000	Privileged	—
MOVCA.L R0, @Rn	store long not fetching block	0000nnnn11000011	—	—
NOP	no operation	0000000000001001	—	—
OCBI @Rn	invalidate data cache block	0000nnnn10010011	—	—
OCBP @Rn	writeback and invalidate data cache block	0000nnnn10100011	—	—
OCBWB @Rn	writeback data cache block	0000nnnn10110011	—	—
PREF @Rn	(Rn) → data cache	0000nnnn10000011	—	—
RTE	SSR → SR, SPC → PC (delayed branch)	000000000101011	Privileged	—
SETS	1 → S	0000000001011000	—	—
SETT	1 → T	0000000000011000	—	1
SLEEP	Sleep or Standby	0000000000011011	Privileged	—

Table 7.8 System Control Instructions (continue)

Instruction		Operation	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)	0000nnnn1mmmm0010	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)	0100nnnn1mmmm0011	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 → SPC, SR → SSR, 0x160 → EXPEVT imm × 4 → TRA, VBR + 0x100 → PC	11000011iiiiiii	—	—

Table 7.9 Floating Point Single-precision Instructions

Instruction		Operation	Code	Privileged	T Bit
FLDI0	FRn	0.0 → FRn	1111nnnn10001101	—	—
FLDI1	FRn	1.0 → FRn	1111nnnn10011101	—	—
FMOV	FRm, FRn	FRm → FRn	1111nnnnnnmm1100	—	—
FMOV.S	@Rm, FRn	(Rm) → FRn	1111nnnnnnmm1000	—	—
FMOV.S	@Rm+, FRn	(Rm) → FRn, Rm + 4 → Rm	1111nnnnnnmm1001	—	—
FMOV.S	@(R0, Rm), FRn	(R0 + Rm) → FRn	1111nnnnnnmm0110	—	—
FMOV.S	FRm, @Rn	FRm → (Rn)	1111nnnnnnmm1010	—	—
FMOV.S	FRm, @-Rn	Rn - 4 → Rn, FRm → (Rn)	1111nnnnnnmm1011	—	—
FMOV.S	FRm, @(R0, Rn)	FRm → (R0 + Rn)	1111nnnnnnmm0111	—	—
FMOV	DRm, DRn	DRm → DRn	1111nnnn0mmm01100	—	—
FMOV	@Rm, DRn	(Rm) → DRn	1111nnnn0mmm1000	—	—
FMOV	@Rm+, DRn	(Rm) → DRn, Rm + 8 → Rm	1111nnnn0mmm1001	—	—
FMOV	@(R0, Rm), DRn	(R0 + Rm) → DRn	1111nnnn0mmm0110	—	—
FMOV	DRm, @Rn	DRm → (Rn)	1111nnnnnnmm01010	—	—
FMOV	DRm, @-Rn	Rn - 8 → Rn, DRm → (Rn)	1111nnnnnnmm01011	—	—
FMOV	DRm, @(R0, Rn)	DRm → (R0 + Rn)	1111nnnnnnmm00111	—	—
FLDS	FRm, FPUL	FRm → FPUL	1111nnnn00011101	—	—
FSTS	FPUL, FRn	FPUL → FRn	1111nnnn00001101	—	—
FABS	FRn	FRn & 0x7fff ffff → FRn	1111nnnn01011101	—	—
FADD	FRm, FRn	FRn + FRm → FRn	1111nnnnnnmm0000	—	—
FCMP/EQ	FRm, FRn	if FRn = FRm, 1 → T, else 0 → T	1111nnnnnnmm0100	—	result
FCMP/GT	FRm, FRn	if FRn > FRm, 1 → T, else 0 → T	1111nnnnnnmm0101	—	result
FDIV	FRm, FRn	FRn / FRm → FRn	1111nnnnnnmm0011	—	—
FLOAT	FPUL, FRn	int_to_float[FPUL] → FRn	1111nnnn00101101	—	—
FMAC	FR0, FRm, FRn	FRm * FR0 + FRn → FRn	1111nnnnnnmm1110	—	—
FMUL	FRm, FRn	FRn * FRm → FRn	1111nnnnnnmm0010	—	—
FNEG	FRn	FRn ^ 0x8000 0000 → FRn	1111nnnn01001101	—	—
FSQRT	FRn	square_root[FRn] → FRn	1111nnnn01101101	—	—
FSUB	FRm, FRn	FRn - FRm → FRn	1111nnnnnnmm0001	—	—
FTRC	FRm, FPUL	float_to_int[FRm] → FPUL	1111nnnn00111101	—	—

Table 7.10 Floating Point Double-precision Instructions

Instruction		Operation	Code	Privileged	T Bit
FABS	DRn	DRn & 0x7fff ffff ffff → DRn	1111nnnn001011101	—	—
FADD	DRm, DRn	DRn + DRm → DRn	1111nnnn0mmm00000	—	—
FCMP/EQ	DRm, DRn	if DRn = DRm, 1 → T, else 0 → T	1111nnnn0mmm00100	—	result
FCMP/GT	DRm, DRn	if DRn > DRm, 1 → T, else 0 → T	1111nnnn0mmm00101	—	result
FCNVDS	DRm, FPUL	double_to_float[DRm] → FPUL	1111mmmm010111101	—	—
FCNVSD	FPUL, DRn	float_to_double[FPUL] → DRn	1111nnnn010101101	—	—
FDIV	DRm, DRn	DRn / DRm → DRn	1111nnnn0mmm00011	—	—
FLOAT	FPUL, DRn	int_to_double[FPUL] → DRn	1111nnnn000101101	—	—
FMUL	DRm, DRn	DRn * DRm → DRn	1111nnnn0mmm00010	—	—
FNEG	DRn	DRn ^ 0x8000 0000 0000 → DRn	1111nnnn001001101	—	—
FSQRT	DRn	square_root[DRn] → DRn	1111nnnn001101101	—	—
FSUB	DRm, DRn	DRn - DRm → DRn	1111nnnn0mmm00001	—	—
FTRC	DRm, FPUL	double_to_int[DRm] → FPUL	1111mmmm000111101	—	—

Table 7.11 Floating Point Control Instructions

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

Table 7.12 Floating Point Graphics Accelerating Instructions

Instruction		Operation	Code	Privileged	T Bit
FMOV	DRm, XDn	DRm → XDn	1111nnn1mm01100	—	—
FMOV	XDm, DRn	XDm → DRn	1111nnn0mm11100	—	—
FMOV	XDm, XDn	XDm → XDn	1111nnn1mm11100	—	—
FMOV	@Rm, XDn	(Rm) → XDn	1111nnn1mm1000	—	—
FMOV	@Rm+, XDn	(Rm) → XDn, Rm + 8 → Rm	1111nnn1mm1001	—	—
FMOV	@(R0, Rm), XDn	(R0+ Rm) → XDn	1111nnn1mm0110	—	—
FMOV	XDm, @Rn	XDm → (Rn)	1111nnnnmm11010	—	—
FMOV	XDm, @-Rn	Rn - 8 → Rn, XDm → (Rn)	1111nnnnmm11011	—	—
FMOV	XDm, @(R0+Rn)	XDm → (R0 + Rn)	1111nnnnmm10111	—	—
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 SH4 is a 2-ILP(Instruction-Level-Parallelism) superscalar pipelining microprocessor. The execution of an instruction is pipelined and two instructions can be executed in parallel. The execution cycles depend on the implementation of a processor. Definitions in this chapter may not be applicable to other models in SH4 family.

8.1 Pipeline

Figure 8.1 shows base 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), write-back(S/FS). An instruction is executed as a combination of the base pipelines. Figure 8.2 shows the instruction execution patterns.

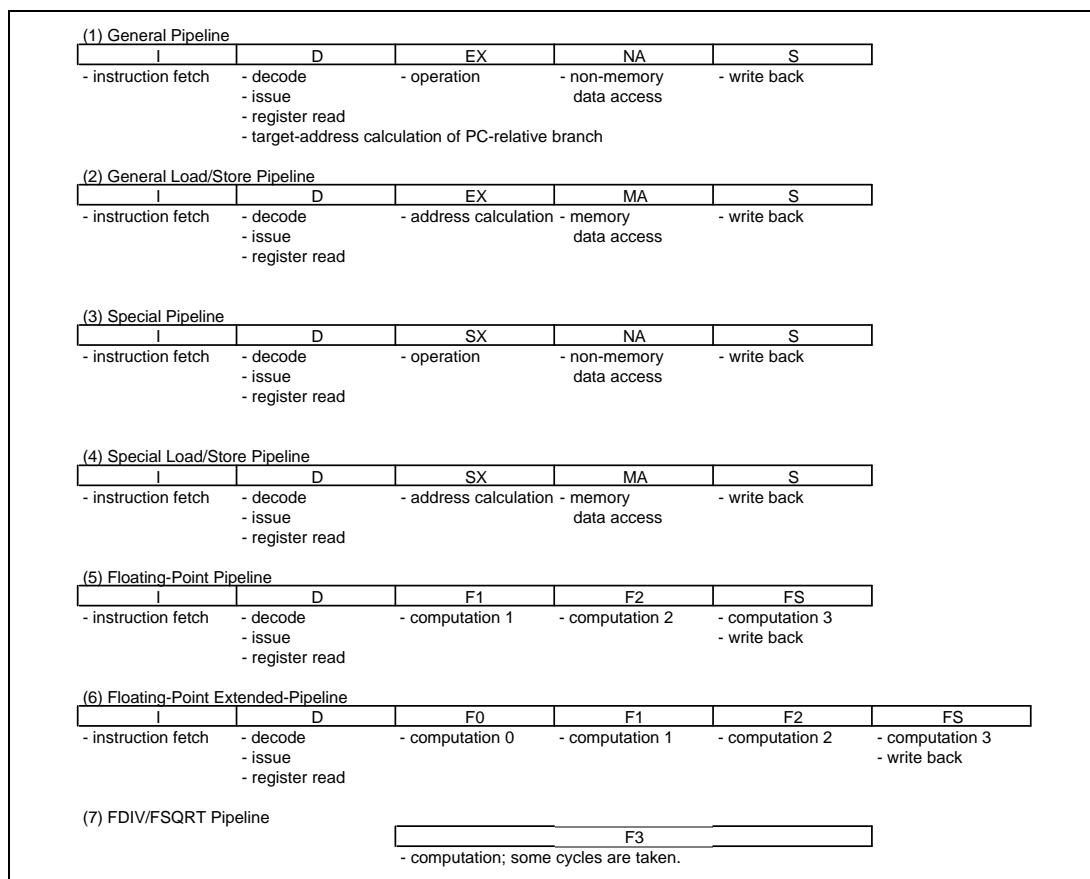


Figure 8.1 Base Pipelines

(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.[SD], LDS.L 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	
						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

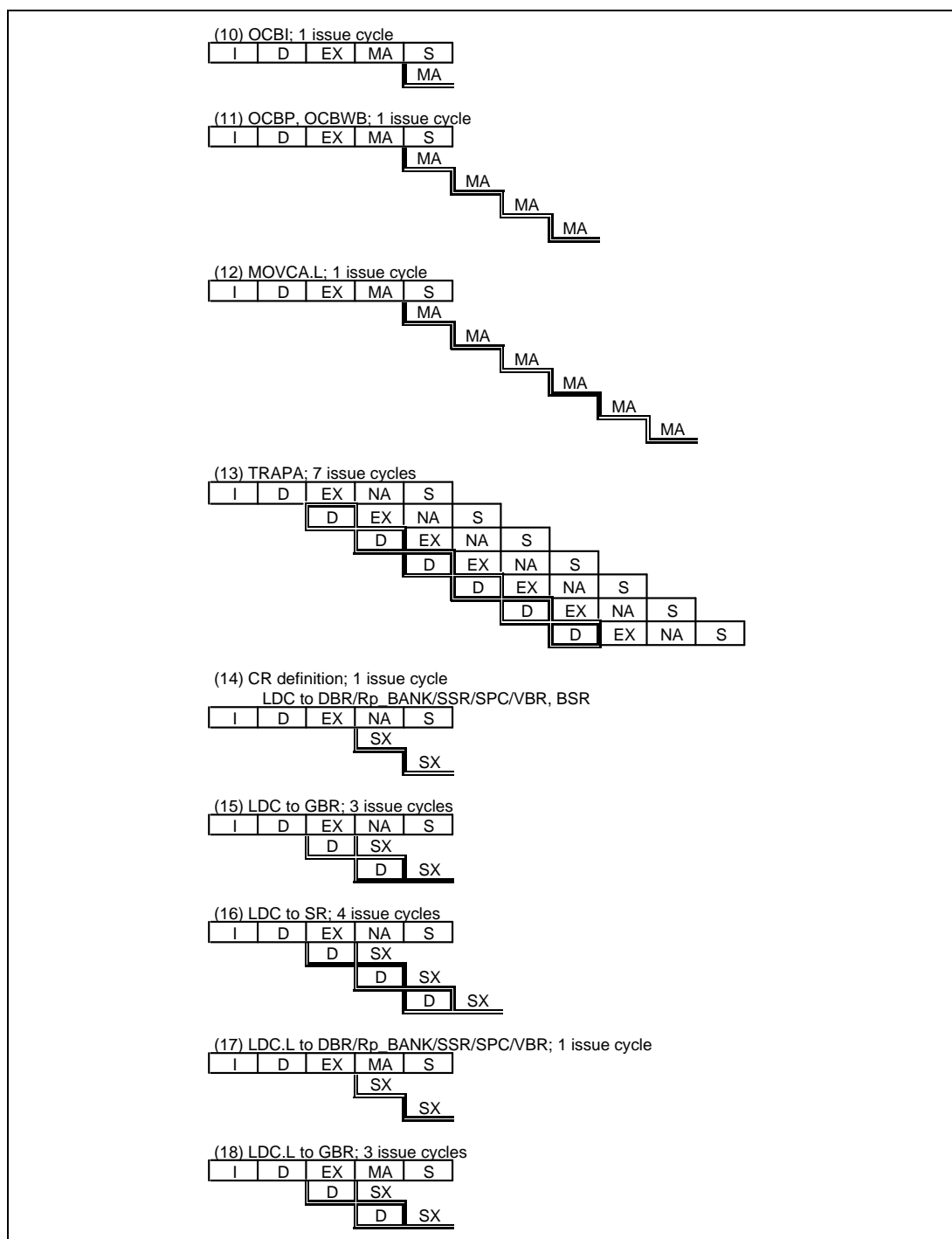


Figure 8.2 Instruction Execution Patterns (continued)

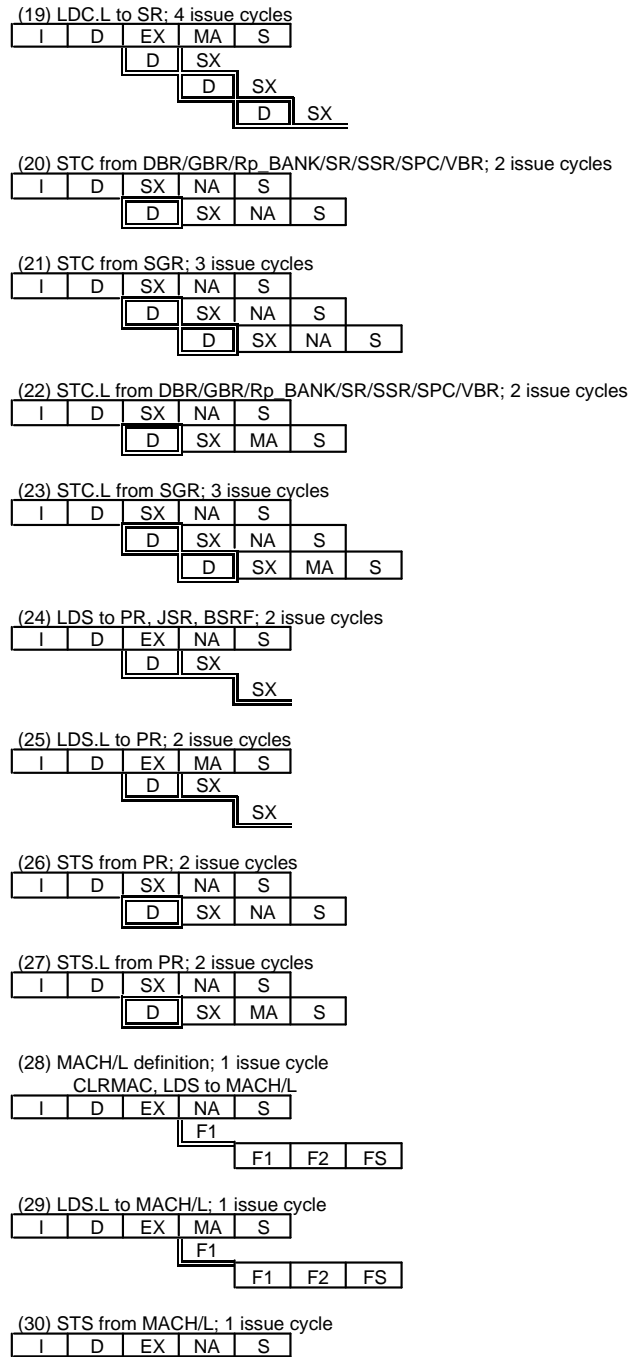


Figure 8.2 Instruction Execution Patterns (continued)

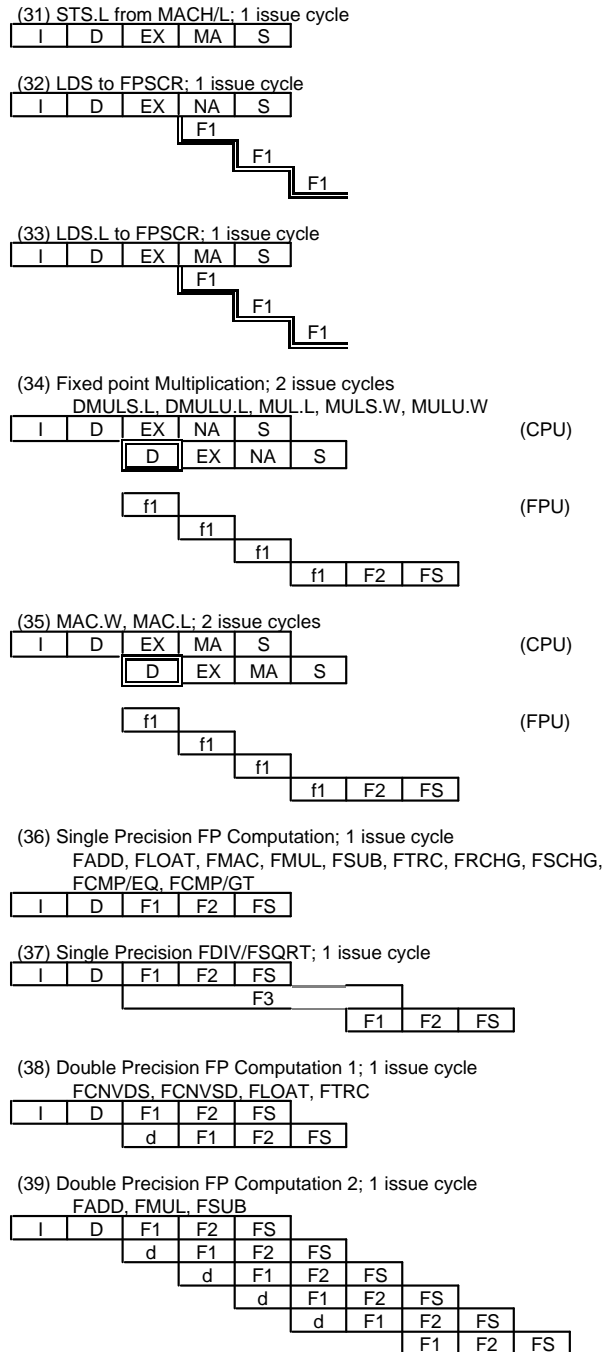


Figure 8.2 Instruction Execution Patterns (continued)

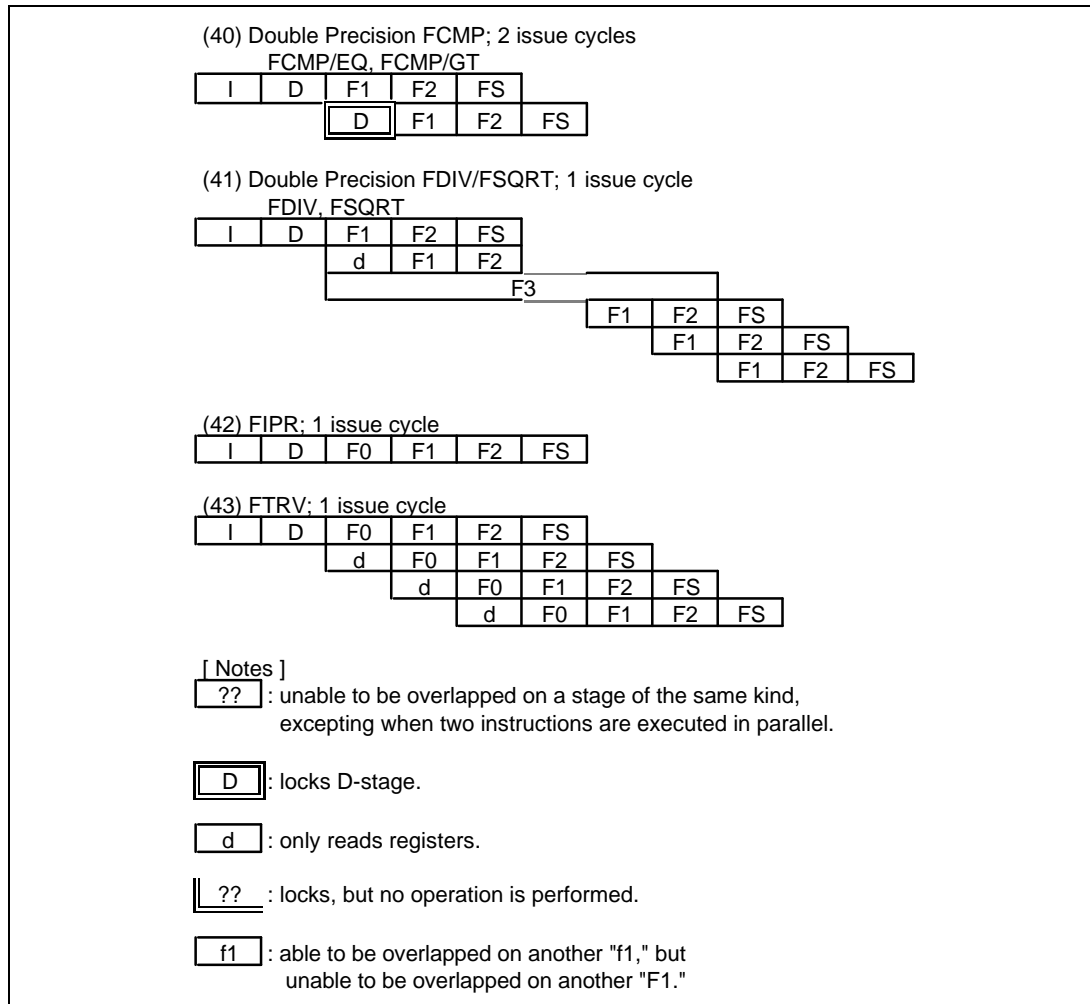


Figure 8.2 Instruction Execution Patterns (continued)

8.2 Parallel-Executability

Instructions are categorized in six groups, depending on their utilization of internal functional blocks, as shown in Table 8.1. Table 8.2 shows parallel-executability of two instructions in terms of groups. For example, ADD categorized into EX and BRA in 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

(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,@-Rm	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 (continued)

(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			

(6) CO group

AND.B	#Imm,@(R0,GBR)	LDS	Rm,FPSCR	STC	SR,Rn
BRAF	Rm	LDS	Rm,MACH	STC	SSR,Rn
BSRF	Rm	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: parallel-executable

X: not parallel-executable

8.3 Execution Cycle and Pipeline Stall

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

- I-clock: CPU, FPU, MMU, and Caches,
- B-clock: External Bus Controller, and
- P-clock: Peripheral Units.

The frequency ratios of the three clocks are determined with FRQCR(Frequency Control Register). In this section, the machine cycle is based on the I-clock otherwise noted. For details on FRQCR, refer to Section 10 “Clock Oscillation Circuits.”

The execution cycles of instructions are summarized in Table 8.3, where the penalty cycles due to pipeline stall is not considered:

- 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 beginning of locking, and
- lock time: period of locking in unit of machine cycle.

The execution sequence of instructions is expressed as a combination of the execution patterns shown in Figure 8.2. Each instruction keeps machine cycles of its issue rate separate from its next instruction. Normally, execution, data-access, and write-back stages cannot be overlapped on the same stages of other instructions. It is the only exception of overlapping when two instructions are executed parallel under parallel-executability condition. Refer to (a) through (d) of Figure 8.3 for some simple cases.

Latency is the interval between issue and completion of an instruction, and it is also the interval between executions of two instructions having any dependencies one another. When there is dependency between two instructions fetched simultaneously, the second of the two is to be stalled for the following cycles:

- (latency) cycles when a flow dependency(read-after-write) exists,
- (latency-2) cycles when an output dependency(write-after-write) exists, or
- 1 or 2 cycles when an anti-flow dependency(write-after-read) exists in case
(a) preceding (the first) instruction is FTRV(1 cycle), or
(b) double-precision FADD/FSUB/FMUL(2 cycles).

Under the flow dependency, there are some exceptional extension/reduction of latency, depending on the sequential combinations of instructions(Figure 8.3 (e)):

- when a floating-point(FP) computation is followed by a FP-register store, the latency of the computation might be reduced 1 cycle,
- if there is a load of shift-amount just before SHAD/SHLD, the latency of the load is extended 1 cycle,
- in case an instruction with less than 2-cycle latency, which involves write-back to FP-register, accompanies a double-precision FP instructions, FIPR, or FTRV, the latency of the first instruction is extended to 2 cycles.

As for the pipeline stall due to flow dependency, there are some variety of its period originated from the combinations of instructions with dependency, or timing of fetch. See also Figure 8.3 (e).

Concerning the stall cycles of an instruction with output dependency, the longest latency to the last write-back of all destination operands should be applied as the substitution to “latency-2” (see Figure 8.3 (f)). Stall due to an output dependency of FPSCR is double-sided. When two FP operations reflects the result of them in the cause-field of FPSCR, there is no stall, whereas a 3-cycle latency is seen between preceding FRCHG/FSCHG and an FP operation categorized LS-type, such as FMOV.S FRm,@-Rn.

The anti-flow dependency can be occurred only between preceding double-precision FADD/FMUL/FSUB or FTRV and following FMOV/FLDI0/FLDI1/FABS/FNEG. See Figure 8.3 (g).

In case an instruction in execution locks any resources, or functional blocks for primitive operations, the following instruction(s) which happened to meet the locking have to be stalled(Figure 8.3 (h)). This kind of stall can be compensated by inserting one or more instructions independent of locked resources and making distance between interfering instructions. For example, when a load and ADD referring the loaded value are in continuous position, inserting three instructions with no dependency to the load removes the 2-cycle stall of ADD. The software performance can be improved by such instruction scheduling.

Other penalties appear in occurrence of exceptions or external data accesses:

- instruction TLB miss, the penalty is 7 I-clocks,
- instruction access to external memory(instruction-cache miss, etc.),
- data access to external memory(operand-cache miss, etc.), the penalty is 2 I-clocks and 3 B-clocks, or
- data access to memory-mapped control registers, the penalty is 2 I-clocks and 3 P-clocks.

During penalty cycles of instruction TLB miss and external instruction access, no instruction is issued, but the executions of instructions having already issued are continued. The penalty for

data accesses is pipeline freeze, that is, the executions of some uncompleted instructions are interrupted till the reach of the requested data. The number of penalty cycles for instruction and data accesses strongly depend on user's memory subsystems. See Section 4 "Caches" and Section 13 "Bus State Controller" for details.

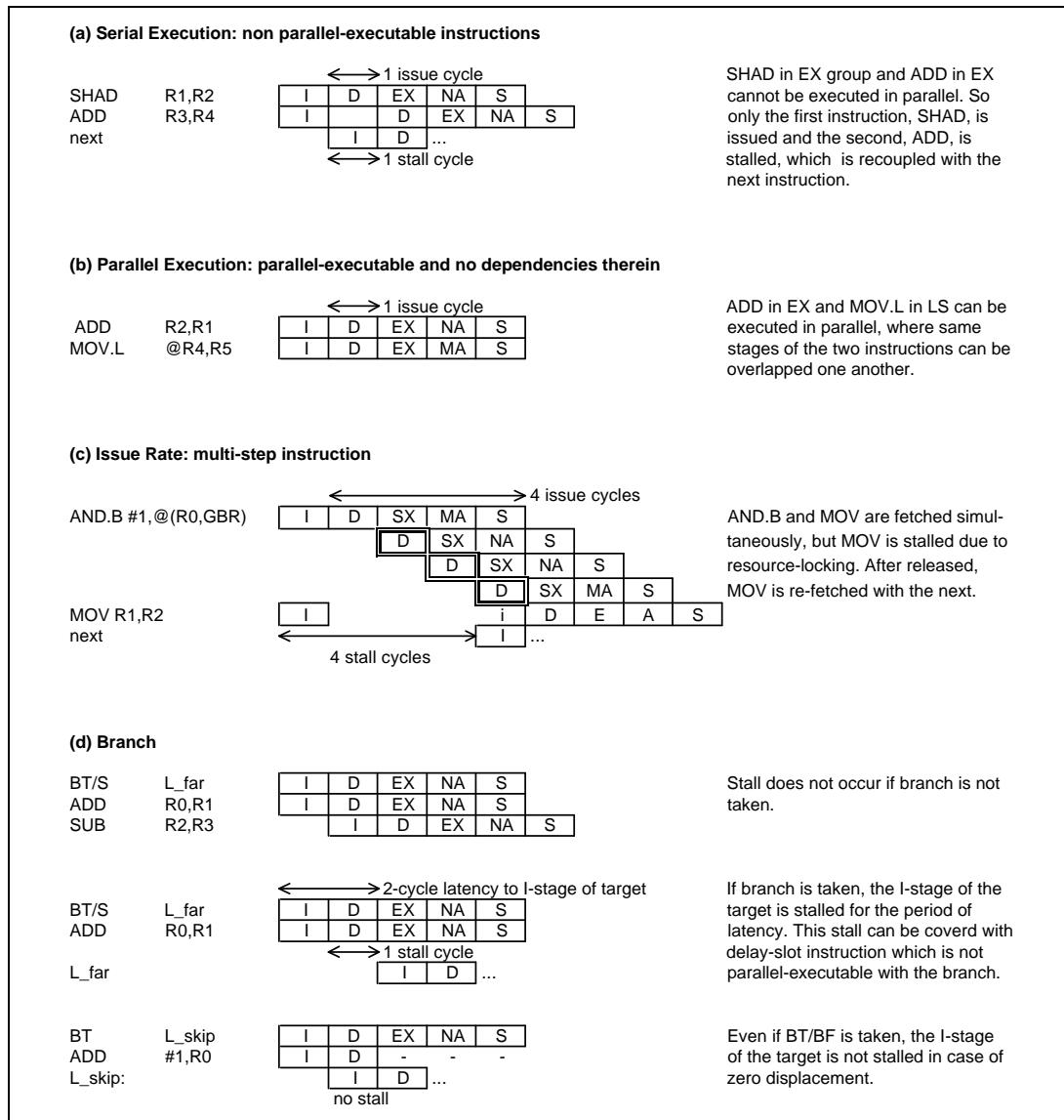


Figure 8.3 Examples of Pipeline Execution

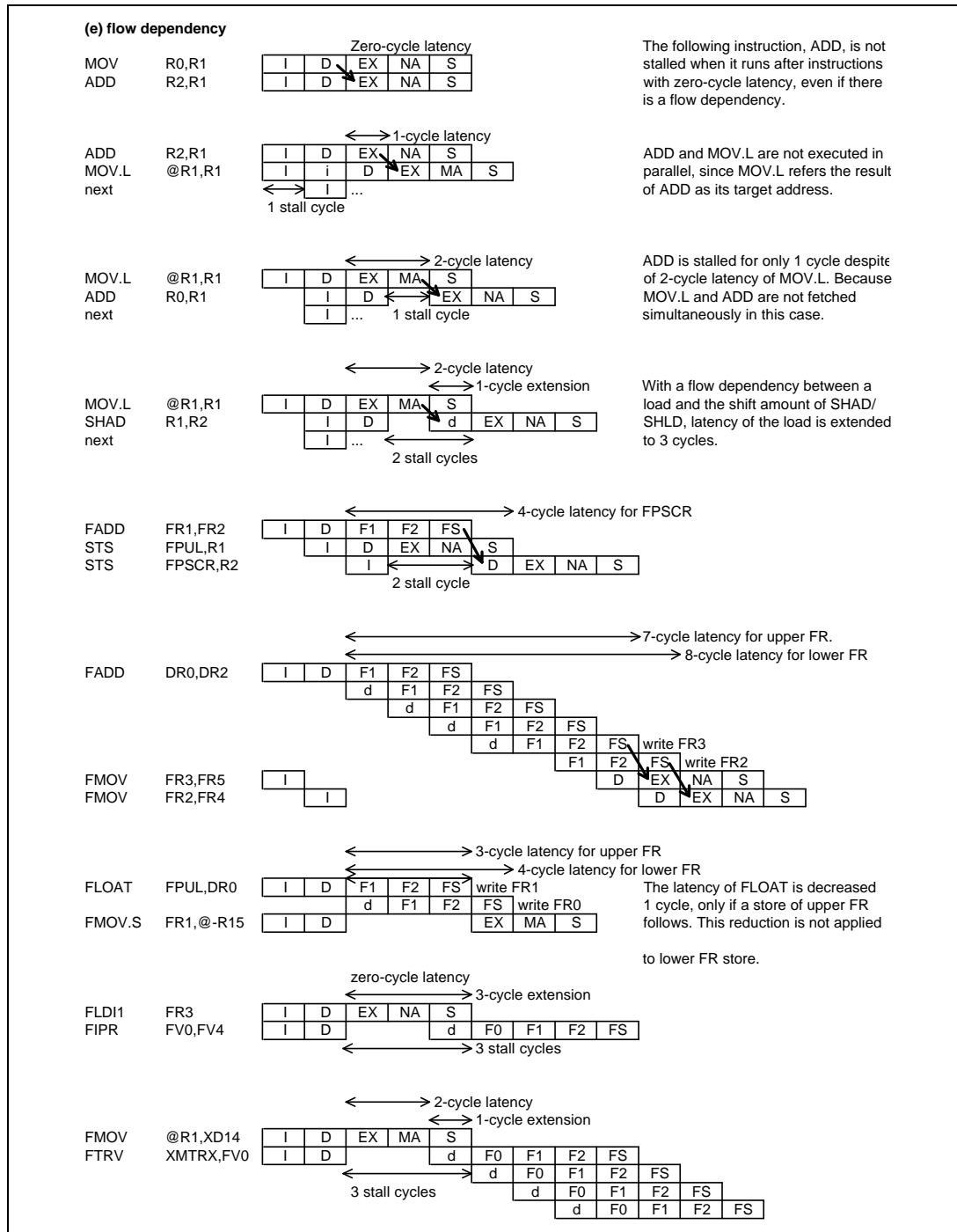
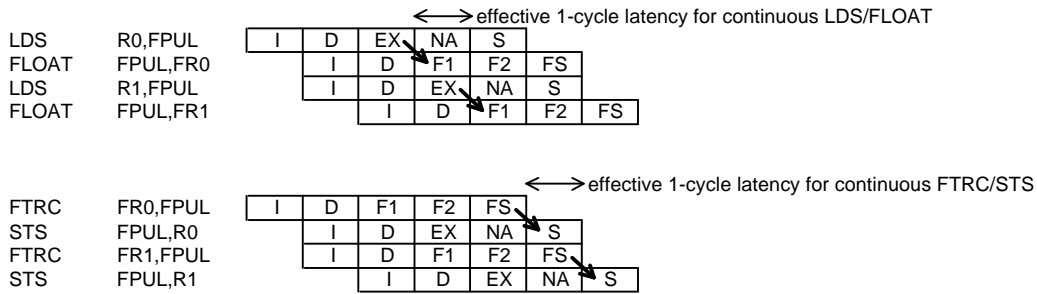
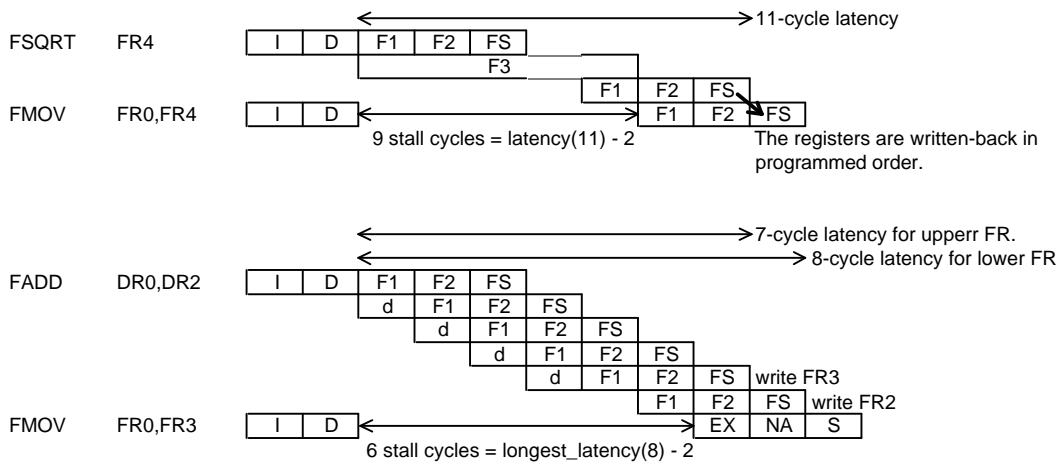


Figure 8.3 Examples of Pipeline Execution (continued)

(e) flow dependency (continued)



(f) output dependency



(g) anti-flow dependency

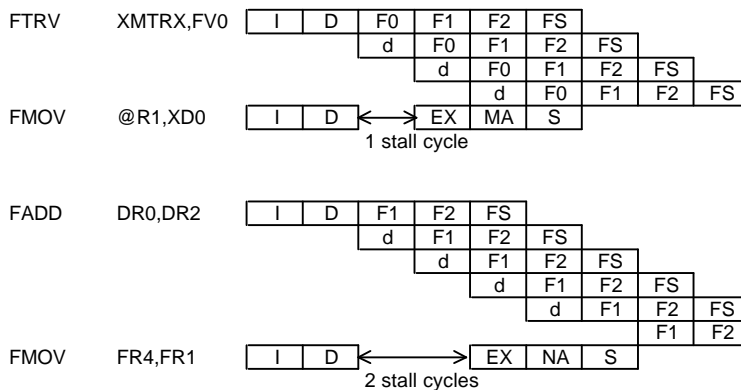


Figure 8.3 Examples of Pipeline Execution (continued)

(h) Resource Conflict

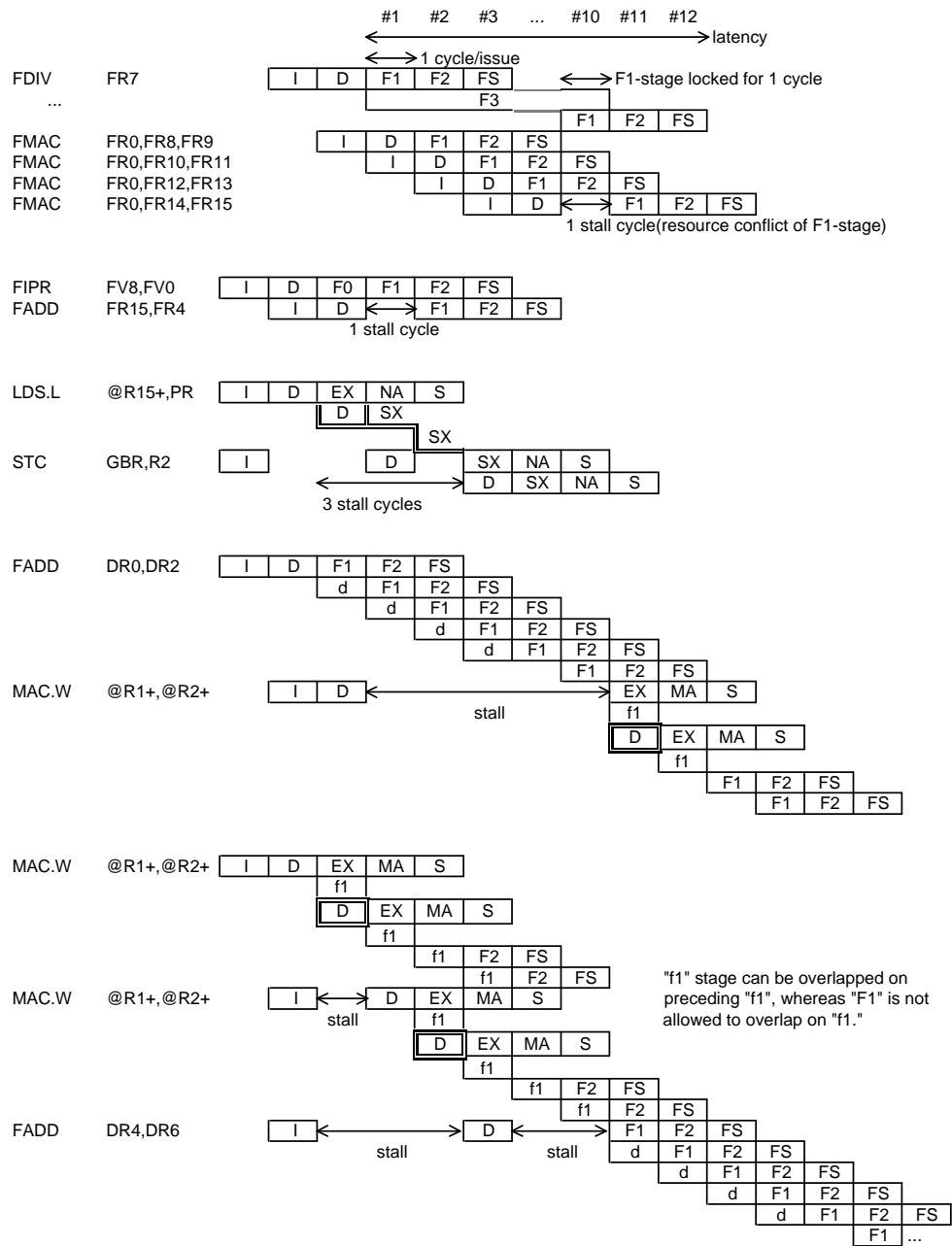


Figure 8.3 Examples of Pipeline Execution (continued)

Table 8.3 Execution Cycles

functional category	#	instruction	inst group	issue rate	latency	exec pattern	Lock			
							stage	begin	cycle	
Data Tranfer Instruction	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	-	-	-

Table 8.3 Execution Cycles (continued)

functional category	#	instruction	inst group	issue rate	latency	exec pattern	Lock			
							stage	begin	cycle	
Fixed Point Arithmetic Instruction	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	-	-	-
	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	-	-	-
	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	-	-	-

Table 8.3 Execution Cycles (continued)

functional category	#	instruction	inst group	issue rate	latency	exec pattern	lock		
							stage	begin	cycle
	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	-	-	-
	63	DIV0U	EX	1	1	#1	-	-	-
	64	DIV1 Rm,Rn	EX	1	1	#1	-	-	-
	65	DMULS.L Rm,Rn	CO	2	4	#34	F1	4	2
	66	DMULU.L Rm,Rn	CO	2	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 Instruction	78	AND Rm,Rn	EX	1	1	#1	-	-	-
	79	AND #Imm,R0	EX	1	1	#1	-	-	-
	80	AND.B #Imm,@(R0,GB R)	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,GB R)	CO	4	4	#6	-	-	-

Table 8.3 Execution Cycles (continued)

functional category	#	instruction	inst group	issue rate	latency	exec pattern	lock			
							stage	begin	cycle	
	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,GB R)	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,GB R)	CO	4	4	#6	-	-	-
Shift Instruction	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 Instruction	108	BF	disp	BR	1	2(or 1)	#1	-	-	-
	109	BF/S	disp	BR	1	2(or 1)	#1	-	-	-

Table 8.3 Execution Cycles (continued)

functional category	#	instruction	inst group	issue rate	latency	exec pattern	lock		
							stage	begin	cycle
	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 Rm	CO	2	3	#4	-	-	-
	114	BSR disp	BR	1	2	#14	SX	3	2
	115	BSRF Rm	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	-	-	-
System Control Instruction	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

Table 8.3 Execution Cycles (continued)

functional category	#	instruction	inst group	issue rate	latency	exec pattern	lock		
							stage	begin	cycle
	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	-	- -
	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	-	- -

Table 8.3 Execution Cycles (continued)

functional category	#	instruction	inst group	issue rate	latency	exec pattern	lock		
							stage	begin	cycle
Single- Precision Floating-Point Instruction	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	-	-	-
	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	-	-	-
	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	#38	F3	2	10
							F1	11	1
	187	FLOAT FPUL,FRn	FE	1	3/4	#36	F1	2	2
	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

Table 8.3 Execution Cycles (continued)

functional category	#	instruction	inst group	issue rate	latency	exec pattern	lock			
							stage	begin	cycle	
	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 Instruction	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	(3,4)/5	#38	F1	2	2
206		FCNVSD	FPUL,DRn	FE	1	(3,4)/5	#38	F1	2	2
207		FDIV	DRm,DRn	FE	23	(24,25)/26	#41	F3	2	21
								F1	20	3
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
210		FNEG	DRn	LS	1	0	#1	-	-	-
211		FSQRT	DRn	FE	22	(23,24)/25	#41	F3	2	20
								F1	19	3
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 Instruction	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	-	-	-

Table 8.3 Execution Cycles (continued)

functional category	#	instruction	inst group	issue rate	latency	exec pattern	lock		
							stage	begin	cycle
Graphics Acceleration Instruction	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	-	-	-
	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	#36	-	-	-
	233	FSCHG	FE	1	1	#36	-	-	-
	234	FTRV XMTRX,FVn	FE	1	(5,5,6,7)/8	#43	F0	2	4
							F1	3	4

Notes:

1. inst group: instruction group (see Table 8.2).
2. latency "L1/L2...": latencies corresponding each definitions, including FPSCR.
[Example] MOV.B @Rm+, Rn "1/2": the latency for Rm is 1 cycle, and the latency for Rn is 2 cycles.
3. latency of branch: interval until the target instruction is fetched.
4. latency of conditional branch "2(or 1)": the latency is 2 for nonzero displacement, while it is 1 for zero.
5. latency of double-precision floating-point instruction "(L1,L2)/L3": L1 is the latency for FR[n+1], L2 for FR[n], and L3 for FPSCR.
6. latency of FTRV "(L1,L2,L3,L4)/L5": L1 is for FR[n], L2 for FR[n+1], L3 for FR[n+2], L4 for FR[n+3], and L5 is for FPSCR.
7. exec pattern: the pattern number of the instruction execution (see Figure 8.2).
8. lock/stage: stage locked by the instruction.
9. lock/begin: beginning cycle of the locking; 1 is the first D-stage of the instruction.
10. lock/cycle: number of cycles locked.

Exceptions:

1. When a floating-point computation instruction is followed by a floating-point store, the latency of the computation is reduced 1 cycle.
2. When the preceding instruction loads the shift amount of the following SHAD/SHLD, the latency of the load is extended 1 cycle.
3. When any instructions with less than 2-cycle latency is followed by a double-precision floating-point instructions, FIPR, or FTRV, the latency of the first is extended to 2 cycles.
[Example] In case of "FMOV FR4,FR0" and "FIPR FV0,FV4," FIPR is stalled for 2 cycles.

Section 9 Power-Down Modes

9.1 Overview

In the power-down modes, some of the on-chip supporting modules and the CPU functions are halted, enabling power consumption to be reduced.

9.1.1 Types of Power-Down Modes

The following power-down modes and functions are provided:

- Sleep mode
- Standby mode
- Module standby function (DMAC, TMU, RTC, and SCI/SCIF on-chip supporting modules)

Table 9.1 shows the conditions for entering these modes from the program execution state, the status of the CPU and supporting modules in each mode, and the method of exiting each mode.

Table 9.1 Power-Down Modes

Power-Down Mode	Entering Conditions	Status						
		CPG	CPU	On-Chip Memory	On-Chip Supporting Modules	Pins	External Memory	Exiting Method
Sleep	SLEEP instruction executed while STBY bit is 0 in STBCR	Operating	Halted (registers held)	Held	Operating	Held	Refreshing	<ul style="list-style-type: none"> • Interrupt • Reset
Standby	SLEEP instruction executed while STBY bit is 1 in STBCR	Halted	Halted (registers held)	Held	Halted*	Held	Self-refreshing	<ul style="list-style-type: none"> • Interrupt • Reset
Module standby	Setting MSTP bit to 1 in STBCR	Operating	Operating	Held	Specified modules halted*	Held	Refreshing	<ul style="list-style-type: none"> • Clearing MSTP bit to 0 • Reset

Note: *The RTC operates when the START bit in RCR2 is 1 (see section 12).

The TMU performs count operations when the RTC output clock is selected as the counter input clock (see section 11).

9.1.2 Register Configuration

Table 9.2 shows the configuration of the register used for power-down mode control.

Table 9.2 Power-Down Mode Register

Name	Abbreviation	R/W	Initial Value	P4 Address	AREA7 Address	Access Size
Standby control register	STBCR	R/W	H'00	H'FFC00004	H'1FC00004	8

9.2 Register Descriptions

9.2.1 Standby Control Register (STBCR)

The standby control register (STBCR) is an 8-bit readable/writable register that specifies the power-down mode status. It is initialized to H'00 by a power-on reset via the RESET pin or due to watchdog timer overflow.

Bit:	7	6	5	4	3	2	1	0
	STBY	PHZ	PPU	MSTP4	MSTP3	MSTP2	MSTP1	MSTP0
Initial value:	0	0	0	0	0	0	0	0
R/W:	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

- Bit 7—Standby (STBY): Specifies a transition to standby mode.

Bit 7: STBY	Description
0	Transition to sleep mode on execution of SLEEP instruction (Initial value)
1	Transition to standby mode on execution of SLEEP instruction

- Bit 6—Supporting Module Pin High Impedance Control (PHZ): Controls the state of supporting module related pins in standby mode. When the PHZ bit is set to 1, supporting module related pins go to the high-impedance state in standby mode.

For the relevant pins, see section 9.2.2, Supporting Module Pin High Impedance Control.

Bit 6: PHZ	Description
0	Supporting module related pins are in normal state (Initial value)
1	Supporting module related pins go to high-impedance state

- Bit 5—Supporting Module Pin Pull-Up Control (PPU): Controls the state of supporting module related pins. When the PPU bit is cleared to 0 and supporting module related pins go to the input or high-impedance state, these pins are pulled up.

For the relevant pins, see section 9.2.3, Supporting Module Pin Pull-Up Control.

Bit 5: PPU	Description
0	Supporting module related pins are pulled up (Initial value)
1	Supporting module related pins are in normal state

- Bit 4—Module Stop 4 (MSTP4): Specifies stopping of the clock supply to the DMAC. The clock supply to the DMAC is stopped when the MSTP4 bit is set to 1. When DMA transfer operated, the MSTP4 bit is set to 1 after the transfer is stopped. For operating DMA transfer after setting MSTP4 bit to 1, second DMA setting is required.

Bit 4: MSTP4	Description
0	DMAC operates (Initial value)
1	DMAC clock supply is stopped

- Bit 3—Module Stop 3 (MSTP3): Specifies stopping of the clock supply to serial communication interface channel 2 (SCIF) among the on-chip supporting modules. The clock supply to SCIF is stopped when the MSTP3 bit is set to 1.

Bit 3: MSTP3	Description
0	SCIF operates (Initial value)
1	SCIF clock supply is stopped

- Bit 2—Module Stop 2 (MSTP2): Specifies stopping of the clock supply to the timer unit (TMU) among the on-chip supporting modules. The clock supply to the TMU is stopped when the MSTP2 bit is set to 1.

Bit 2: MSTP2	Description
0	TMU operates (Initial value)
1	TMU clock supply is stopped

- Bit 1—Module Stop 1 (MSTP1): Specifies stopping of the clock supply to the realtime clock (RTC) among the on-chip supporting modules. The clock supply to the RTC is stopped when the MSTP1 bit is set to 1. When the clock supply is stopped, RTC registers cannot be accessed but the counters continue to operate.

Bit 1: MSTP1	Description
0	RTC operates (Initial value)
1	RTC clock supply is stopped

- Bit 0—Module Stop 0 (MSTP0): Specifies stopping of the clock supply to serial communication interface channel 1 (SCI) among the on-chip supporting modules. The clock supply to SCI is stopped when the MSTP0 bit is set to 1.

Bit 0: MSTP0	Description
0	SCI operates (Initial value)
1	SCI clock supply is stopped

9.2.2 Supporting Module Pin High Impedance Control

When bit 6 in the standby control register (STBCR) is set to 1, supporting module related pins go to the high-impedance state in standby mode.

(1) Relevant pins

1. SCI/SCIF related pins

MD0/SCKMD1/TXD2MD2/RXD2MD7/TXDMD8/RTS2SCK2/ $\overline{\text{MRESET}}$
RXDCTS2

2. DMAC related pins

$\overline{\text{DREQ0}}$ $\overline{\text{DREQ1}}$ DACK0DACK1DRAK1DRAK0

(2) Other information

High impedance control is not performed when the above pins are in the output state.

9.2.3 Supporting Module Pin Pull-Up Control

When bit 5 in the standby control register (STBCR) is cleared to 0, supporting module related pins are pulled up.

(1) Relevant pins

1. SCI/SCIF related pins

MD0/SCKMD1/TXD2MD2/RXD2MD7/TXDMD8/RTS2SCK2/ $\overline{\text{MRESET}}$
RXDCTS2

2. DMAC related pins

$\overline{\text{DREQ0}}$ $\overline{\text{DREQ1}}$ DACK0DACK1DRAK1DRAK0

3. TMU related pins

TCLK

(2) Other information

The above pins are pulled up when in the input or high-impedance state.

9.3 Sleep Mode

9.3.1 Transition to Sleep Mode

If a SLEEP instruction is executed when the STBY bit in STBCR is cleared to 0, the chip switches from the program execution state to sleep mode. After execution of the SLEEP instruction, the CPU halts but its register contents are retained. The on-chip supporting modules continue to operate, and the clock continues to be output from the CKIO pin.

In sleep mode, a high-level signal is output at the STATUS1 pin, and a low-level signal at the STATUS0 pin.

9.3.2 Exit from Sleep Mode

Sleep mode is exited by means of an interrupt (NMI, IRL, or on-chip supporting module) or a reset. In sleep mode, interrupts are accepted even if the BL bit in the SR register is 1. If necessary, the SPC and SSR should be saved to the stack before executing the SLEEP instruction.

Exit by Interrupt: When an NMI, IRL, or on-chip supporting module interrupt is generated, sleep mode is exited and interrupt exception handling is executed. The code corresponding to the interrupt source is set in the INTEVT register.

Exit by Reset: Sleep mode is exited by means of a power-on reset via the $\overline{\text{RESET}}$ pin, a manual reset, or a power-on reset or manual reset executed when the watchdog timer overflows.

9.4 Standby Mode

9.4.1 Transition to Standby Mode

If a SLEEP instruction is executed when the STBY bit in STBCR is set to 1, the chip switches from the program execution state to standby mode. In standby mode, the on-chip supporting modules halt as well as the CPU. Clock output from the CKIO pin is also stopped.

The CPU and cache register contents are retained. Some on-chip supporting module registers are initialized. The state of the supporting module registers in standby mode is shown in table 9.4.

Table 9.4 State of Registers in Standby Mode

Module	Initialized Registers	Registers That Retain Their Contents
Interrupt controller	—	All registers
Break controller	—	All registers
Bus state controller	—	All registers
On-chip oscillation circuits	—	All registers
Timer unit	TSTR register*	All registers except TSTR
Realtime clock	—	All registers

Note: * Not initialized when the realtime clock (RTC) is in use (see section 11).

When a operating mode is transfered to standby mode, end the DMA transfer. There is no guarantee on the transfer result when a operating mode is transfered to standby mode while DMA transfer.

The procedure for a transition to standby mode is shown below.

1. Clear the TME bit in the WDT timer control register (WTCSR) to 0, and stop the WDT.
Set the initial value for the up-count in the WDT timer counter (WTCNT), and set the clock to be used for the up-count in bits CKS2–CKS0 in the WTCSR register.
2. Set the STBY bit in the STBCR register to 1, then execute a SLEEP instruction.
3. When standby mode is entered and the chip's internal clock stops, a low-level signal is output at the STATUS1 pin, and a high-level signal at the STATUS0 pin.

9.4.2 Exit from Standby Mode

Standby mode is exited by means of an interrupt (NMI, IRL, or on-chip supporting module) or a reset via the RESET pin.

Exit by Interrupt: A hot start can be performed by means of the on-chip WDT. When an NMI, IRL^{*1}, or on-chip supporting module (except interval timer)^{*2} interrupt is detected, the WDT starts counting. After the count overflows, clocks are supplied to the entire chip, standby mode is exited, and both the STATUS1 and the STATUS0 pin go low. Interrupt exception handling is then executed, and the code corresponding to the interrupt source is set in the INTEVT register. In standby mode, interrupts are accepted even if the BL bit in the SR register is 1, and so, if necessary, the SPC and SSR should be saved to the stack before executing the SLEEP instruction.

The phase of the CKIO pin clock output may be unstable immediately after an interrupt is detected, until standby mode is exited.

- Notes:
1. Only when the RTC is used, standby mode can be exited by means of IRL3–IRL0 (when the IRL3–IRL0 level is higher than the SR register I3–I0 mask level).
 2. Standby mode can be exited by means of an RTC interrupt or a TMU interrupt (when operating on the RTC clock).

Exit by Reset: Standby mode is exited by means of a reset (power-on or manual) via the $\overline{\text{RESET}}$ pin. The $\overline{\text{RESET}}$ pin should be held low until clock oscillation stabilizes. The internal clock continues to be output at the CKIO pin.

9.4.3 Clock Pause Function

In standby mode, it is possible to stop or change the frequency of the clock input from the EXTAL pin. This function is used as follows.

1. Enter standby mode following the transition procedure described above.
2. When standby mode is entered and the chip's internal clock stops, a low-level signal is output at the STATUS1 pin, and a high-level signal at the STATUS0 pin.
3. The input clock is stopped, or its frequency changed, after the STATUS1 pin goes low and the STATUS0 pin high.
4. When the frequency is changed, input an NMI or IRL interrupt after the change. When the clock is stopped, input an NMI or IRL interrupt after applying the clock.
5. After the time set in the WDT, clock supply begins inside the chip, the STATUS1 and STATUS0 pins both go low, and operation is resumed from interrupt exception handling.

9.5 Module Standby Function

9.5.1 Transition to Module Standby Function

Setting the MSTP4–MSTP0 bits in the standby control register to 1 enables the clock supply to the corresponding on-chip supporting modules to be halted. Use of this function allows power consumption in sleep mode to be further reduced.

In the module standby state, the on-chip supporting module external pins retain their states prior to halting of the modules. Most registers retain their states prior to halting of the modules.

Bit		Description
MSTP4	0	DMAC operates
	1	Clock supplied to DMAC is stopped
MSTP3	0	SCIF operates
	1	Clock supplied to SCIF is stopped
MSTP2	0	TMU operates
	1	Clock supplied to TMU is stopped, and register is initialized* ¹
MSTP1	0	RTC operates
	1	Clock supplied to RTC is stopped* ²
MSTP0	0	SCI operates
	1	Clock supplied to SCI is stopped

Notes: 1. The register initialized is the same as in standby mode (see table 9-4).
2. The counter operates when the START bit in RCR2 is 1 (see section 11).

9.5.2 Exit from Module Standby Function

The module standby function is exited by clearing the MSTP3–MSTP0 bits to 0, or by a power-on reset via the RESET pin or a power-on reset caused by watchdog timer overflow.

Section 10 Instruction Description

The notation of the instruction descriptions is as follows.

Instruction Mnemonics (Descriptive Name): Classification

Class: is indicated if the instruction is a delayed branch instruction.

Format	Abstract	Code	T Bit
Notation in Assembly Language	A brief description of operation	Opcode; displayed in order MSB × LSB	Effect to T bit

Description: Description of operation

Notes: Notes on using the instruction

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

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

```
unsigned char Read_Byte(unsigned int Addr);
unsigned short Read_Word(unsigned int Addr);
unsigned int Read_Long(unsigned int Addr);
```
- Writes data of each length to address Addr. An address error will occur if word data is written to an address other than 2n or if longword data is written to an address other than 4n:

```
unsigned char Write_Byte(unsigned int Addr, unsigned int Data);
unsigned short Write_Word(unsigned int Addr, unsigned int Data);
unsigned int Write_Long(unsigned int Addr, unsigned int Data);
```
- Execution of an instruction in delay slot at an address Addr:

```
Delay_Slot(unsigned int Addr);
```

- List registers:

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

The content of PC is equal to an instruction address of an instruction at the beginning of the instruction execution.

- Definition of SR structures:

```
struct SR0 {
    unsigned int dummy0:22;
    unsigned int M0:1;
    unsigned int Q0:1;
    unsigned int I0:4;
    unsigned int dummy1:2;
    unsigned int S0:1;
    unsigned int T0:1;
};
```

- Definition of bits in SR:

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

- Error display function:

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

- Floating point definition:

```
#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 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,FPUL;

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) return(PZERO);
            else 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)){

```

```

        if(sign_of(n) == 0) return(PZERO);
        else                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 {
        double d;
        int l[2];
    } dstd,srcd;
    union {
        /* "int double" format is as follows: */
        int double x; /* 1-bit sign */
        int l[4]; /* 15-bit exporment */
    } dstx; /* 112-bit fraction */
    if(FPSCR_PR == 0) {
        if(type == FADD) srcf.f = FR[m];
        else             srcf.f = -FR[m];
    }
}

```

```

dstd.d = FR[n]; /* convert float to double */
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 toward nearest or even */
if(FPSCR_RM == 1) {
    dstd.l[1] &= 0xe0000000; /* round toward 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]; /* convert double to extended double */
    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 toward nearest or even */
if(FPSCR_RM == 1) {
    dstx.l[2] &= 0xf0000000; /* round toward 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 {
        int double x;
        int l[4];
    } tmpx;

    if(FPSCR_PR == 0) {
        tmpd.d = FR[n]; /* convert single to double */
        tmpd.d *= FR[m]; /* exact product */
        tmpf.f = FR[m]; /* round toward nearest or even */
        if(tmpf.f != tmpd.d) set_I();
        if((tmpf.f > tmpd.d) && (SPSCR_RM == 1)) {
            tmpf.l -= 1; /* round toward zero */
        }
        check_single_exception(&FR[n],tmpf.f);
    } else {
        tmpx.x = DR[n>>1]; /* convert double to int double */
        tmpx.x *= DR[m>>1]; /* exact product */
        tmpd.d = DR[m>>1]; /* round toward nearest or even */
        if(tmpd.d != tmpx.x) set_I();
        if(tmpd.d > tmpx.x) && (SPSCR_RM == 1)) {
            tmpd.l[1] -= 1; /* round toward 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++) {
            /* flush denormalized values if FPSCR_DN == 1) */
            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];

/* The multiplication array emulation is necessary for obtaining the

```



```

    same result as that of the FIPR hardware, because the hardware cut
    lower 18 bits of the array output before carry propagate addition.
    The following flow is different from the hardware algorism but simple. */
    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();
    if(FPSCR_RM == 1) {
        tmp.l -= 1; /* largest magunitude finite number */
        result = tmp.f;
    }
}

if(result < 0.0) abs = -result;
else abs = result;
tmp.l = 0x00800000; /* minimum magunitude normalized number */
if(abs < tmp.f) {
    if((FPSCR_DN == 1) && (abs != 0.0)) {
        set_I();
        if(result < 0.0) result = -0.0; /* flush demornalized value */
        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();
    if(FPSCR_RM == 1) {
        tmp.l[0] -= 1;
        tmp.l[1] = 0xffffffff;
        result = tmp.d; /* largest magunitude finite number */
    }
}

if(result < 0.0) abs = -result;
else
    abs = result;
tmp.l[0] = 0x00100000; /* minimum magunitude 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; /* flush demornalized value */
        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;}
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;
    }
}

```

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

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

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

ADD (Add Binary): Arithmetic Instruction

Format	Abstract	Code	T Bit
ADD Rm,Rn	$Rm + Rn \oslash Rn$	0011nnnnmmmm1100	—
ADD #imm,Rn	$Rn + \#imm \oslash Rn$	0111nnnniiiiiii	—

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

Operation:

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

ADDI(int i,int n) /* ADD #imm,Rn */
{
    if ((i&0x80)==0)
        R[n]+=(0x000000FF & i);
    else    R[n]+=(0xFFFFF00 | i);
    PC+=2;
}
```

Examples:

ADD	R0,R1	Before execution	R0 = H'7FFFFFFF, R1 = H'00000001
		After execution	R1 = H'80000000
ADD	#H'01,R2	Before execution	R2 = H'00000000
		After execution	R2 = H'00000001
ADD	#H'FE,R3	Before execution	R3 = H'00000001
		After execution	R3 = H'FFFFFFF

ADDC (Add with Carry): Arithmetic Instruction

Format	Abstract	Code	T Bit
ADDC Rm,Rn	$Rn + Rm + T \oslash Rn, \text{carry} \oslash T$	0011nnnnnnmmmm1110	Carry

Description: adds contents of general register Rm, T bit, and Rn, stores the result in Rn. and stores the carry in T bit. This instruction may be used to calculate data whose size is longer than 32 bits.

Operation:

```
ADDC (int m,int n)      /* ADDC Rm,Rn  */
{
    unsigned long result;

    result=(unsigned long)R[m]+ (unsigned long)R[n]+ (unsigned long)T;
    R[n]=(unsigned int)(result & 0x00000000FFFFFFFF)
    T=(result & 0x0000000100000000)>>32;
    PC+=2;
}
```

Examples:

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

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

Format	Abstract	Code	T Bit
ADDV Rm,Rn	$Rn + Rm \oslash Rn$, overflow \oslash T	0011nnnnmmmm1111	Overflow

Description: adds signed 32-bit contents of general register Rm and Rn, and stores the result in Rn. If an overflow or an underflow occurs, T bit is set to 1. Otherwise, T bit is set to zero.

Operation:

```

ADDV(int m,int n)      /* ADDV Rm,Rn  */
{
    int src_Rn=(int)R[n];

    R[n]+=R[m];
    if( ((int)R[m]>=0 && Rn>=0 && (int)R[n]<0)      /* Overflow */
        || ((int)R[m]<0 && Rn<0 && (int)R[n]>0)    /* Underflow */
        )      T=1;
    else      T=0;
    PC+=2;
}

```

Examples:

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

AND (AND Logical): Logic Operation Instruction

Format	Abstract	Code	T Bit
AND Rm,Rn	$Rn \& Rm \rightarrow Rn$	0010nnnnnnmmmm1001	—
AND #imm,R0	$R0 \& imm \rightarrow R0$	11001001iiiiiii	—
AND.B #imm,@(R0,GBR)	$(R0 + GBR) \& imm \rightarrow (R0 + GBR)$	11001101iiiiiii	—

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

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

Operation:

```

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

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

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

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

```

Examples:

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

BF (Branch if False): Branch Instruction

Format	Abstract	Code	T Bit
BF label	When T = 0, PC + 4 + disp*2 \neq PC; When T = 1, nop	10001011ddddddd	—

Description: conditionally branches to an address (PC + displacement * 2), depending on T bit. If T = zero, the branch is taken. If T = 1, it does not branch. The PC source value is an instruction address of the BF. The 8-bit displacement is sign-extended and doubled. Consequently, the branch range is -256 to +254 bytes.

Note: If the displacement is too short to reach a branch target, a combination of BF and BRA or JMP makes this branch possible.

Operation:

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

    if ((d&0x80)==0)
        disp=(0x000000FF & d);
    else    disp=(0xFFFFF00 | d);
    if (T==0)
        PC=PC+4+(disp*2);
    else    PC+=2;
}
```

Example:

```
CLRT          T is always cleared to 0
BT    TRGET_T  Does not branch, because T = 0
BF    TRGET_F  Branches to TRGET_F, because T = 0
NOP
NOP
TRGET_F:      ♦ Branch destination of the BF instruction
```

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

Class: Delayed branch instruction

Format	Abstract	Code	T Bit
BF/S label	When T = 0, PC + 4 + disp*2 \neq PC; When T = 1, nop	10001111ddddddd	—

Description: conditionally branches to an address (PC + 4 + displacement * 2), depending on T bit. If T = zero, the branch is taken. If T = 1, it does not branch. The PC source value is an instruction address of the BF/S. The 8-bit displacement is sign-extended and doubled. Consequently, the branch range is -256 to +254 bytes.

Note: Since this is a delayed branch instruction, the instruction immediately following is executed before the branch. Between the time this instruction and the instruction immediately following are executed, no interrupts are accepted. When the instruction immediately following is a branch instruction, it is recognized as an illegal slot instruction. If the displacement is too short to reach a branch target, a combination of BF and BRA or JMP makes this branch possible.

Operation:

```
BFS(int d) /* BFS disp */
{
    int disp;

    if ((d&0x80)==0)
        disp=(0x000000FF & d);
    else    disp=(0xFFFFFFFF00 | d);
    if (T==0) {
        Delay_Slot(PC+2);
        PC=PC+4+(disp*2);
    }
    else    PC+=2;
}
```

Examples:

```
CLRT                T is always 0
BT/S  TARGET_T      Does not branch, because T = 0
NOP
BF/S  TARGET_F      Branches to TARGET, because T = 0
ADD   R0 , R1        Executed before branch.
NOP
TARGET_F:           ♦ Branch destination of the BF/S instruction
```

BRA (Branch): Branch Instruction

Class: Delayed branch instruction

Format	Abstract	Code	T Bit
BRA label	$PC + 4 + \text{disp} * 2 \nrightarrow PC$	1010ddddddddddd	—

Description: branches unconditionally to an address ($PC + 4 + \text{displacement} * 2$). BRA is a delayed branch. The PC source value is an instruction address of the BRA. The 12-bit displacement is sign-extended and doubled. Consequently, the branch range is -4096 to $+4094$ bytes. If the displacement is too short to reach a branch target, JMP instruction makes this branch possible.

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

Operation:

```
BRA(int d) /* BRA disp */
{
    int disp;

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

Examples:

```

BRA    TRGET    Branches to TRGET
ADD    R0,R1    Executes ADD before branching
NOP
TRGET:    ♦ Branch destination of the BRA instruction
```

BRAF (Branch Far): Branch Instruction

Class: Delayed branch instruction

Format	Abstract	Code	T Bit
BRAF Rn	$PC + 4 + Rn \nrightarrow PC$	0000nnnn00100011	—

Description: branches unconditionally to an address ($PC + 4 + Rn$). The target address is a result of adding PC, 4 and the 32-bit contents of the general register Rn. the PC source value is an instruction address of the BRAF.

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

Operation:

```
BRAF(int n) /* BRAF Rn */
{
    Delay_Slot(PC+2);
    PC=PC+4+R[n];
}
```

Examples:

MOV.L	\$(TARGET-BRAF_PC),R0	Sets displacement.
BRAF	R0	Branches to TARGET
ADD	R0,R1	Executes ADD before branching
BRAF_PC:		
NOP		
TARGET:		◆ Branch destination of the BRAF instruction

BSR (Branch to Subroutine): Branch Instruction

Class: Delayed branch instruction

Format	Abstract	Code	T Bit
BSR label	PC + 4 \nrightarrow PR, PC + 4 + disp*2 \nrightarrow PC	1011ddddddddddd	—

Description: branches to an address (PC + 4 + displacement * 2), and stores an address (PC + 4) in PR. The PC source value is an instruction address of the BSR. The 12-bit displacement is sign-extended and doubled. Consequently, the branch range is -4096 to +4094 bytes. If the displacement is too short to reach the branch destination, JSR instruction makes this branch possible.

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

Operation:

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

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

Examples:

BSR	TRGET	Branches to TRGET
MOV	R3,R4	Executes the MOV instruction before branching
ADD	R0,R1	◆ Return address from the subroutine procedure (PR data)
.....		
.....		
TRGET:	◆ Procedure entrance	
MOV	R2,R3	
RTS		Returns to the above ADD instruction
MOV	#1,R0	Executes MOV before branching

BSRF (Branch to Subroutine Far): Branch Instruction

Class: Delayed branch instruction

Format	Abstract	Code	T Bit
BSRF Rn	PC + 4 \nrightarrow PR, PC + 4 + Rn \nrightarrow PC	0000nnnn00000011	—

Description: branches to an address (PC + 4 + Rn), and stores an address (PC + 4) in PR. The PC source value is an instruction address of the BSRF. The branch target is a result of adding PC, 4 and the 32-bit content of the general register Rn.

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

Operation:

```
BSRF(int n) /* BSRF Rn */
{
    PR=PC+4;
    Delay_Slot(PC+2);
    PC=PC+4+R[n];
}
```

Examples:

```
MOV.L    #(TARGET-BSRF_PC), R0 Sets displacement.
BSRF     R0                      Branches to TARGET
MOV      R3, R4                  Executes the MOV instruction before branching

BSRF_PC:
ADD      R0, R1
. . . . .

TARGET:  ♦ Procedure entrance
MOV      R2, R3
RTS                                Returns to the above ADD instruction
MOV      #1, R0                  Executes MOV before branching
```

BT (Branch if True): Branch Instruction

Format	Abstract	Code	T Bit
BT label	When T = 1, PC + 4 + disp*2 \neq PC; When T = 0, nop	10001001ddddddd	—

Description: conditionally branches to an address (PC + 4 + displacement * 2), depending on T bit. If T = 1, the branch is taken. If T = zero, it does not branch. The PC source value is an instruction address of the BT. The 8-bit displacement is sign-extended and doubled. Consequently, the branch range is -256 to +254 bytes.

Note: If the displacement is too short to reach a branch target, a combination of BT and BRA or JMP makes this branch possible.

Operation:

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

    if ((d&0x80)==0)
        disp=(0x000000FF & d);
    else    disp=(0xFFFFF00 | d);
    if (T==1)
        PC=PC+4+(disp*2);
    else    PC+=2;
}
```

Examples:

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

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

Format	Abstract	Code	T Bit
BT/S label	When T = 1, PC + 4 + disp*2 \neq PC; When T = 0, nop	10001101ddddddd	—

Description: conditionally branches to an address (PC + 4 + displacement * 2), depending on T bit. If T = 1, the branch is taken. If T = zero, it does not branch. The PC source value is an instruction address of the BT/S. The 8-bit displacement is sign-extended and doubled. Consequently, the branch range is -256 to +254 bytes. If the displacement is too short to reach a branch target, a combination of BT/S and BRA or JMP makes this branch possible.

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

Operation:

```
BTS(int d) /* BTS disp */
{
    int disp;

    if ((d&0x80)==0)
        disp=(0x000000FF & d);
    else    disp=(0xFFFFF00 | d);
    if (T==1) {
        Delay_Slot(PC+2);
        PC=PC+4+(disp*2);
    }
    else PC+=2;
}
```

Examples:

SETT		T is always 1
BF/S	TARGET_F	Does not branch, because T = 1
NOP		
BT/S	TARGET_T	Branches to TARGET, because T = 1
ADD	R0,R1	Executes before branching.
NOP		
TARGET_T:		◆ Branch target of the BT/S instruction

CLRMAC (Clear MAC Register): System Control Instruction

Format	Abstract	Code	T Bit
CLRMAC	0 \emptyset MACH, MACL	0000000000101000	—

Description: Clears the MACH and MACL registers.

Operation:

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

Examples:

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

CLRS (Clear S Bit): System Control Instruction

Format	Abstract	Code	T Bit
CLRS	0 \emptyset S	0000000001001000	—

Description: Clears the S bit in SR.

Operation:

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

Examples:

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

CLRT (Clear T Bit): System Control Instruction

Format	Abstract	Code	T Bit
CLRT	$0 \nabla T$	0000000000001000	0

Description: Clears the T bit in SR.

Operation:

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

Examples:

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

CMP/cond (Compare Conditionally): Arithmetic Instruction

Format	Abstract	Code	T Bit
CMP/EQ Rm, Rn	When Rn = Rm, 1 \neq T	0011nnnnnnmm0000	Comparison result
CMP/GE Rm, Rn	When signed and Rn \leq Rm, 1 \neq T	0011nnnnnnmm0011	Comparison result
CMP/GT Rm, Rn	When signed and Rn > Rm, 1 \neq T	0011nnnnnnmm0111	Comparison result
CMP/HI Rm, Rn	When unsigned and Rn > Rm, 1 \neq T	0011nnnnnnmm0110	Comparison result
CMP/HS Rm, Rn	When unsigned and Rn \leq Rm, 1 \neq T	0011nnnnnnmm0010	Comparison result
CMP/PL Rn	When Rn > 0, 1 \neq T	0100nnnn00010101	Comparison result
CMP/PZ Rn	When Rn \leq 0, 1 \neq T	0100nnnn00010001	Comparison result
CMP/STR Rm, Rn	When any byte in Rn is equal to the corresponding byte in Rm, 1 \neq T	0010nnnnnnmm1100	Comparison result
CMP/EQ #imm, R0	When R0 = imm, 1 \neq T	10001000iiiiiii	Comparison result

Description: Compares general register Rn data with Rm data, and sets the T bit to 1 if a specified condition (cond) is satisfied. The T bit is cleared to 0 if the condition is not satisfied, and the Rn data does not change. The nine conditions in table 6.1 can be specified. Conditions PZ and PL are the results of comparisons between Rn and 0. Sign-extended 8-bit immediate data can also be compared with R0 by using condition EQ. Here, R0 data does not change. Table 6.1 shows the mnemonics for the conditions.

Table 6.1 CMP Mnemonics

Mnemonics	Condition
CMP/EQ Rm, Rn	If Rn = Rm, T = 1
CMP/GE Rm, Rn	If Rn \leq Rm with signed data, T = 1
CMP/GT Rm, Rn	If Rn > Rm with signed data, T = 1
CMP/HI Rm, Rn	If Rn > Rm with unsigned data, T = 1
CMP/HS Rm, Rn	If Rn \leq Rm with unsigned data, T = 1
CMP/PL Rn	If Rn > 0, T = 1
CMP/PZ Rn	If Rn \leq 0, T = 1
CMP/STR Rm, Rn	If any byte in Rn is equal to the corresponding byte in Rm, T = 1
CMP/EQ #imm, R0	If R0 = imm, T = 1

Operation:

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

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

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

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

```

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

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

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

CMPSTR(int m,int n)     /* CMP_STR Rm,Rn */
{
    if(      (R[m]&0xFF000000)==( R[n]&0xFF000000)
        ||  (R[m]&0x00FF0000)==( R[n]&0x00FF0000)
        ||  (R[m]&0x0000FF00)==( R[n]&0x0000FF00)
        ||  (R[m]&0x000000FF)==( R[n]&0x000000FF)
    )        T=1;
    else    T=0;
    PC+=2;
}

```



```

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

    if ((i&0x80)==0)
        imm=(0x000000FF & i);
    else    imm=(0xFFFFFFFF00 | i);
    if (R[0]==imm)
        T=1;
    else    T=0;
    PC+=2;
}

```

Examples:

```

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

```

DIV0S (Divide Step 0 as Signed): Arithmetic Instruction

Format	Abstract	Code	T Bit
DIV0S Rm,Rn	MSB of Rn \oslash Q, MSB of Rm \oslash M, M [^] Q \oslash T	0010nnnnmmmm0111	Calculation result

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

Operation:

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

Examples: See DIV1.

DIV0U (Divide Step 0 as Unsigned): Arithmetic Instruction

Format	Abstract	Code	T Bit
DIV0U	$0 \nabla M/Q/T$	0000000000011001	0

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

Operation:

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

Example: See DIV1.

DIV1 (Divide Step 1): Arithmetic Instruction

Format	Abstract	Code	T Bit
DIV1 R_m, R_n	1 step division ($R_n \div R_m$)	0011nnnnmmmm0100	Calculation result

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

In one-step division, the dividend is shifted one bit left, the divisor is subtracted and the quotient bit reflected in the Q bit according to the status (positive or negative). Zero division, overflow detection, and remainder operation are not supported. Check for zero division and overflow division before dividing.

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

Operation:

```
DIVl(int m,int n) /* DIVl Rm,Rn */
{
    unsigned int 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  Upper 16 bits = divisor, lower 16 bits = 0
TSTR0,R0  Zero division check
BT  ZERO_DIV
CMP/HS R0,R1  Overflow check
BT  OVER_DIV
DIV0U      Flag initialization
.arepeat 16
DIV1  R0,R1  Repeat 16 times
.aendr
ROTCL R1
EXTU.W  R1,R2      R1 = Quotient

```

Example 2:

```

        R1:R2 (64 bits)/R0 (32 bits) = R2 (32 bits): Unsigned
TSTR0,R0  Zero division check
BT  ZERO_DIV
CMP/HS R0,R1  Overflow check
BT  OVER_DIV
DIV0U      Flag initialization
.arepeat 32
ROTCL R2  Repeat 32 times
DIV1  R0,R1
.aendr
ROTCL  R2      R2 = Quotient

```

Example 3:

$R1\ (16\ \text{bits})/R0\ (16\ \text{bits}) = R1\ (16\ \text{bits})$: Signed
 SHLL16 R0 Upper 16 bits = divisor, lower 16 bits = 0
 EXTS.W R1,R1 Sign-extends the dividend to 32 bits
 XOR R2,R2 R2 = 0
 MOVR1,R3
 ROTCL R3
 SUBC R2,R1 Decrements if the dividend is negative
 DIV0S R0,R1 Flag initialization
.arepeat 16
 DIV1 R0,R1 Repeat 16 times
.aendr
 EXTS.W R1,R1
 ROTCL R1 R1 = quotient (one's complement)
 ADDC R2,R1 Increments and takes the two's complement if the MSB of the quotient is 1
 EXTS.W R1,R1 R1 = quotient (two's complement)

Example 4:

$R2\ (32\ \text{bits}) / R0\ (32\ \text{bits}) = R2\ (32\ \text{bits})$: Signed
 MOVR2,R3
 ROTCL R3
 SUBC R1,R1 Sign-extends the dividend to 64 bits (R1:R2)
 XOR R3,R3 R3 = 0
 SUBC R3,R2 Decrements and takes the one's complement if the dividend is negative
 DIV0S R0,R1 Flag initialization
.arepeat 32
 ROTCL R2 Repeat 32 times
 DIV1 R0,R1
.aendr
 ROTCL R2 R2 = Quotient (one's complement)
 ADDC R3,R2 Increments and takes the two's complement if the MSB of the quotient is 1. R2 = Quotient (two's complement)

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

Format	Abstract	Code	T Bit
DMULS.L Rm, Rn	With sign, $Rn \in Rm \cap MACH$, MACL	0011nnnnmmmm110 1	—

Description: multiply a 32-bit content of general register Rm by a 32-bit content of Rn, stores higher 32 bits of the 64-bit result in MACH, and stores lower 32 bits of the 64-bit result in MACL. The operation is a signed arithmetic operation.

Operation:

```
DMULS(int m,int n) /* DMULS.L Rm,Rn */
{
    long  result;

    result=(long)((int)R[m]))*(long)((int)R[n]));
    MACH=(unsigned int)((unsigned long)result>>32);
    MACL=(unsigned int)(result&0x00000000FFFFFFFF);
    PC+=2;
}
```

Examples:

```
DMULS  R0,R1  Before execution      R0 = H'FFFFFFFE, R1 = H'00005555
                After execution  MACH = H'FFFFFFF, MACL = H'FFFF5556
STS MACH,R2 Operation result (higher)
STS  MACL,R3  Operation result (lower)
```


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

Format	Abstract	Code	T Bit
DMULU.L Rm,Rn	Without sign, $Rn \in Rm \emptyset$ MACH, MACL	0011nnnnmmmm0101	—

Description: multiply a 32-bit content of general register Rm by a 32-bit content of Rn, stores higher 32 bits of the 64-bit result in MACH, and stores lower 32 bits of the 64-bit result in MACL. The operation is an unsigned arithmetic operation.

Operation:

```
DMULU(int m,int n) /* DMULU.L Rm,Rn */
{
    unsigned long    result;

    result=(unsigned long)R[m]* (unsigned long)R[n];
    MACH=(unsigned int)((unsigned long)result>>32);
    MACL=(unsigned int)(result&0x00000000FFFFFFFF);
    PC+=2;
}
```

Examples:

```
DMULU  R0,R1    Before execution    R0 = H'FFFFFFFE, R1 = H'00005555
                After execution    MACH = H'00005554, MACL = H'FFFF5556
STS MACH,R2    Operation result (higher)
STS  MACL,R3    Operation result (lower)
```

DT (Decrement and Test): Arithmetic Instruction

Format	Abstract	Code	T Bit
DT Rn	Rn - 1 \nexists Rn; When Rn is 0, 1 \nexists T, when Rn is nonzero, 0 \nexists T	0100nnnn0001000 0	Comparison result

Description: decreases the content of general register Rn by 1, stores the result in Rn, and compares the result with zero. If the result is zero, T bit is set to 1. Otherwise, the T bit is set to zero.

Operation:

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

Example:

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

EXTS (Extend as Signed): Data Transfer Instruction

Format	Abstract	Code	T Bit
EXTS.B Rm,Rn	Sign-extend Rm from byte \varnothing Rn	0110nnnnnnmmml1110	—
EXTS.W Rm,Rn	Sign-extend Rm from word \varnothing Rn	0110nnnnnnmmml1111	—

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

Operation:

```
EXTSB(int m,int n)      /* EXTS.B Rm,Rn */
{
    if ((R[m]&0x00000080)==0)
        R[n]=R[m] & 0x000000FF;
    else  R[n]=R[m] | 0xFFFFF000;
    PC+=2;
}

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

Examples:

```
EXTS.B R0,R1      Before execution  R0 = H'00000080
                   After execution  R1 = H'FFFFFF80
EXTS.W R0,R1      Before execution  R0 = H'00008000
                   After execution  R1 = H'FFF8000
```

EXTU (Extend as Unsigned): Data Transfer Instruction

Format	Abstract	Code	T Bit
EXTU.B Rm,Rn	Zero-extend Rm from byte \emptyset Rn	0110nnnnnnmmmm1100	—
EXTU.W Rm,Rn	Zero-extend Rm from word \emptyset Rn	0110nnnnnnmmmm1101	—

Description: zero-extends the content of general register Rm, and stores the result in Rn. If byte length is specified, bit 8 to bit 31 in Rn are set to zero. If word length is specified, bit 16 to bit 31 in Rn are set to zero.

Operation:

```

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

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

```

Examples:

```

EXTU.B R0,R1  Before execution  R0 = H'FFFFFF80
                After execution  R1 = H'00000080

EXTU.W R0,R1  Before execution  R0 = H'FFFF8000
                After execution  R1 = H'00008000

```

FABS (Floating Point Absolute Value): Floating-Point Instruction

PR	Format	Operation	Instruction Code	T Bit
0	FABS FRn	FRn \emptyset FRn	1111nnnn01011101	—
1	FABS DRn	DRn \emptyset DRn	1111nnn001011101	—

Description

Clears the most significant bit (sign bit) of the floating-point number in FRn/DRn to 0, and writes the result to FRn/DRn. FPSCR.cause field and FPSCR.flag field do not change.

Operation

```
void FABS (int n){
    FR[n] = FR[n] & 0x7fffffff;
    pc += 2;
}
/* The same operation is performed regardless of the precision. */
```

FADD (Floating Point Add): Floating-Point Instruction

PR	Format	Operation	Instruction Code	T Bit
0	FADD FRm,FRn	FRn+FRm \emptyset FRn	1111nnnnmmmm0000	—
1	FADD DRm,DRn	DRn+DRm \emptyset DRn	1111nnn0mmm00000	—

Description

When the PR bit in FPSCR is 0: Arithmetically adds the two single-precision floating-point numbers contained in FRm and FRn, and writes the result to FRn.

When the PR bit in FPSCR is 1: Arithmetically adds the two double-precision floating-point numbers contained in DRm and DRn, and writes the result to DRn.

If the O, U, or I bit in the FPSCR.enable field is enabled, an FPU exception trap will be raised regardless of the exception occurrence, FPSCR.cause and FPSCR.flag fields reflect the actual FPU exception status, and FRn or DRn does not change. Appropriate actions must be taken 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

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

Note: When DN = 1, a denormalized number is treated as zero.

Exceptions

FPU error
Invalid operation
Overflow
Underflow
Inexact

FCMP (Floating Point Compare): Floating-Point Instruction

PR	Format	Operation	Instruction Code	T Bit
0	FCMP/EQ FRm,FRn	(FRn == FRm) ? 1 : 0 ∅ T	1111nnnnnnmmmm0100	1/0
1	FCMP/EQ DRm,DRn	(DRn == DRm) ? 1 : 0 ∅ T	1111nnn0mmmm00100	1/0
0	FCMP/GT FRm,FRn	(FRn > FRm) ? 1 : 0 ∅ T	1111nnnnnnmmmm0101	1/0
1	FCMP/GT DRm,DRn	(DRn > DRm) ? 1 : 0 ∅ T	1111nnn0mmmm00101	1/0

Description

1. When the PR bit in FPSCR is 0: Arithmetically compares the two single-precision floating-point numbers contained in FRm and FRn, and writes 1 to the T bit if the values are equal, or 0 if not equal.
2. When the PR bit in FPSCR is 1: Arithmetically compares the two double-precision floating-point numbers contained in DRm and DRn, and writes 1 to the T bit if the values are equal, or 0 if not equal.
3. When the PR bit in FPSCR is 0: Arithmetically compares the two single-precision floating-point numbers contained in FRm and FRn, and writes 1 to the T bit if FRn > FRm, or 0 otherwise.
4. When the PR bit in FPSCR is 1: Arithmetically compares the two double-precision floating-point numbers contained in DRm and DRn, and writes 1 to 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 case(data_type_of(m)){
NORM:      case(data_type_of(n)){
            PINF   :return(GT);          break;
            NINF   :return(LT);          break;
            default:                    break;
        }
PZERO:
NZERO:      case(data_type_of(n)){
            PZERO  :
            NZERO  :return(EQ);          break;
            default:                    break;
        }
PINF :      case(data_type_of(n)){
            PINF   :return(EQ);          break;
            default:return(LT);          break;
        }
NINF :      case(data_type_of(n)){
            NINF   :return(EQ);          break;
            default:return(GT);          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;
}
```

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, a denormalized number is treated as zero.

FCMP/GT	FRn,DRn								
FRm,DRm	NORM	DENORM	+0	−0	+INF	−INF	qNaN	sNaN	
NORM	CMP				GT	!GT	UO	Invalid	
DENORM									
+0									
−0									
+INF	!GT				!GT				
−INF	GT				!GT				
qNaN									
sNaN									

Note: When DN = 1, a denormalized number is treated as zero.

Exceptions

Invalid operation

FCNVDS (Floating Point Convert Double to Single Precision): Floating-Point Instruction

PR	Format	Operation	Instruction Code	T Bit
0	—	reserved	1111mmmm10111101	—
1	FCNVDS DRm,FPUL	(float)DRm \rightarrow FPUL	1111mmmm010111101	—

Description

Converts the double-precision floating-point number in DRn to a single-precision floating-point number, and writes the result to FPUL.

If the O, U, or I bit in the FPSCR.enable field is enabled, an FPU exception trap will be raised regardless of the exception occurrence, FPSCR.cause and FPSCR.flag fields reflect the actual FPU exception status, and FPUL does not change. Appropriate actions must be taken by software.

Operation

```
void FCNVDS(int m){
    case((FPSCR.PR){
        0: undefined_operation(); /* reserved */
        1: fcnvds(m); break; /* FCNVDS */
    }
}
void fcnvds(int m)
{
    pc += 2;
    clear_cause();
    case(data_type_of(m)){
        NORM :
        PZERO :
        NZERO : normal_fcnvds(m); 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)
{
    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, a denormalized number is treated as zero.

Exceptions

FPU error
Invalid operation
Overflow
Underflow
Inexact

FCNVSD (Floating Point Convert Single to Double Precision): Floating-Point Instruction

PR	Format	Operation	Instruction Code	T Bit
0	—	reserved	1111nnnn10101101	—
1	FCNVSD FPUL,DRn	(double)FPUL \rightarrow DRn	1111nnn010101101	—

Description

Interprets the contents of FPUL as a single-precision floating-point number, converts that number to a double-precision floating-point number, and writes the result to DRn.

Operation

```
void FCNVSD(int n){
    pc += 2;
    clear_cause();
    case((FPSCR_PR){
        0: undefined_operation(); /* reserved */
        1: fcnvdsd (n,&FPUL); break; /* FCNVSD */
    }
}
void fcnvdsd(int n, float *FPUL)
{
    case(fpul_typ()){
        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 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, a denormalized number is treated as zero.

Exceptions

FPU error

Invalid operation

FDIV (Floating Point Divide): Floating-Point Instruction

PR	Format	Operation	Instruction Code	T Bit
0	FDIV FRm,FRn	FRn/FRm \div FRn	1111nnnnnnmmmm0011	—
1	FDIV DRm,DRn	DRn/DRm \div DRn	1111nnn0mmmm00011	—

Description

When the PR bit in FPSCR is 0: Arithmetically divides the single-precision floating-point number in FRn by the single-precision floating-point number in FRm, and writes the result to FRn.

When the PR bit in FPSCR is 1: Arithmetically divides the double-precision floating-point number in DRn by the double-precision floating-point number in DRm, and writes the result to DRn.

If the O, U, or I bit in the FPSCR.enable field is enabled, an FPU exception trap will be raised regardless of the exception occurrence, FPSCR.cause and FPSCR.flag fields reflect the actual FPU exception status, and FRn or DRn does not change. Appropriate actions must be taken 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 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	NaN	sNaN
NORM	DIV	0		INF		Error	qNaN	Invalid
+0	DZ	Invalid		+INF	-INF	DZ		
−0				-INF	+INF			
+INF	0	+0	−0	Invalid				
−INF		−0	+0					
DENORM						Error		
qNaN							qNaN	
sNaN								

Note: When DN = 1, a denormalized number is treated as zero.

Exceptions

FPU error
 Invalid operation
 Divide by zero
 Overflow
 Underflow
 Inexact

FIPR (Floating Point Inner Product): Floating-Point Instruction

PR	Format	Operation	Instruction Code	T Bit
0	FIPR FV _m , FV _n	FV _m • FV _n Ø FR[n+3]	1111nnmm11101101	—
1	—	reserved	1111nnmm11101101	—

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

Description

Writes the inner products of the single-precision floating-point vector in FV_m and the single-precision floating-point vector in FV_n to single-precision floating-point register FR[n+3].

The operation is sometimes more or less accurate than the calculation using FMUL and FMAC because of the algorism difference. The order of the operations is as follows:

1. Multiplication of terms (result of each term is 30 digits)
2. Digit alignment (truncation below 30 digits)
3. Addition of all terms
4. Normalization, rounding

Special cases are as follows:

1. If the input operands include sNaN: invalid operation
2. If the input operands for a term are infinity and zero: invalid operation
3. If the input operands include qNaN, other than as above, the result is qNaN.
4. If the input operands include infinity, other than as above:
 - a. If the multiplication results for two terms or more are infinity and not of the same sign: invalid operation
 - b. Other than the above, the result is infinity with the proper sign.
5. If the input operands do not include sNaN, qNaN, or infinity: the same as for a normal instruction

If the O, U, or I bit in the FPSCR.enable field is enabled, an FPU exception trap will be raised regardless of the exception occurrence, FPSCR.cause and FPSCR.flag fields reflect the actual FPU exception status, and FR_n or DR_n does not change. Appropriate actions must be taken 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();
}
```

Exceptions

Invalid operation

Overflow

Underflow

Inexact

FLDI0 (Floating Point Load 0.0): Floating-Point Instruction

PR	Format	Operation	Instruction Code	T Bit
0	FLDI0 FRn	0x00000000 \emptyset FRn	1111nnnn10001101	—
1	—	reserved	1111nnnn10001101	—

Description

When the PR bit in FPSCR is 0: Loads floating point zero(0x00000000) to the floating point register FRn.

Operation

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

Exceptions

None

FLDI1 (Floating Point Load 1.0): Floating-Point Instruction

PR	Format	Operation	Instruction Code	T Bit
0	FLDI1 FRn	0x3F800000 \rightarrow FRn	1111nnnn10011101	—
1	—	reserved	1111nnnn10011101	—

Description

When the PR bit in FPSCR is 0: Loads floating point one (0x3F800000) to the floating point register FRn.

Operation

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

Exceptions

None

FLDS (Floating Point Load to System Register): Floating-Point Instruction

Format	Operation	Instruction Code	T Bit
FLDS FRm,FPUL	FRm \rightarrow FPUL	1111mmmm00011101	—

Description

Copies the content of floating point register FRm to the system register FPUL.

Operation

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

Exceptions

None

FLOAT (Floating Point Convert from Integer): Floating-Point Instruction

PR	Format	Operation	Instruction Code	T Bit
0	FLOAT FPUL,FRn	(float)FPUL \rightarrow FRn	1111nnnn00101101	—
1	FLOAT FPUL,DRn	(double)FPUL \rightarrow DRn	1111nnn000101101	—

Description

When the PR bit in FPSCR is 0: Converts the 32-bit integer value in FPUL to a single-precision floating-point number, and writes the result to FRn.

When the PR bit in FPSCR is 1: Converts the 32-bit integer value in FPUL to a double-precision floating-point number, and writes the result to DRn.

If the I bit in the FPSCR.enable field is enabled and FPSCR.PR is 0, an FPU exception trap will be raised regardless of the exception occurrence, FPSCR.cause and FPSCR.flag fields reflect the actual FPU exception status, and FRn does not change. Appropriate actions must be taken by software.

Operation

```
void FLOAT(int n)
{
    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 */
    }
}
```

Exceptions

Inexact (not generated when FPSCR.PR = 1)

FMAC (Floating Point Multiply and Accumulate): Floating-Point Instruction

PR	Format	Operation	Instruction Code	T Bit
0	FMAC FR0,FRm,FRn	$FR0 * FRm + FRn \oslash FRn$	1111nnnnnnmmmm1110	—
1	—	reserved	1111nnnnnnmmmm1110	—

Description

When the PR bit in FPSCR is 0: Floating-point-arithmetically multiplies the contents of floating point registers FR0 and FRm. And the result of this operation is accumulated to floating point register FRn. The operation is the fusion type, and the intermediate product is not rounded.

If the O, U, or I bit in the FPSCR.enable field is enabled, an FPU exception trap will be raised regardless of the exception occurrence, FPSCR.cause and FPSCR.flag fields reflect the actual FPU exception status, and FRn or DRn does not change. Appropriate actions must be taken 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		Error	qNaN	sNaN		
	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					Invalid					+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								
!NaN	Denorm	Error										
qNaN	0					Invalid						
	INF			Invalid								
	Norm											
!NaN	qNaN	qNaN										
all types	sNaN											
sNaN	all types	Invalid										

Note: When DN = 1, a denormalized number is treated as zero.

Exceptions

FPU error
 Invalid operation
 Overflow
 Underflow
 Inexact

FMOV (Floating Point Move): Floating-Point Instruction

SZ	Format	Operation	Instruction Code	T Bit
0	FMOV FRm,FRn	FRm \varnothing FRn	1111nnnnnnmmmm1100	—
1	FMOV DRm,DRn	DRm \varnothing DRn	1111nnn0mmmm01100	—
0	FMOV.S FRm,@Rn	FRm \varnothing (Rn)	1111nnnnnnmmmm1010	—
1	FMOV DRm,@Rn	DRm \varnothing (Rn)	1111nnnnnnmm01010	—
0	FMOV.S @Rm,FRn	(Rm) \varnothing FRn	1111nnnnnnmmmm1000	—
1	FMOV @Rm,DRn	(Rm) \varnothing DRn	1111nnn0mmmm1000	—
0	FMOV.S @Rm+,FRn	(Rm) \varnothing FRn, Rm+=4	1111nnnnnnmmmm1001	—
1	FMOV @Rm+,DRn	(Rm) \varnothing DRn, Rm+=8	1111nnn0mmmm1001	—
0	FMOV.S FRm,@-Rn	Rn-=4, FRm \varnothing (Rn)	1111nnnnnnmmmm1011	—
1	FMOV DRm,@-Rn	Rn-=8, DRm \varnothing (Rn)	1111nnnnnnmm01011	—
0	FMOV.S @(R0,Rm),FRn	(R0+Rm) \varnothing FRn	1111nnnnnnmmmm0110	—
1	FMOV @(R0,Rm),DRn	(R0+Rm) \varnothing DRn	1111nnn0mmmm0110	—
0	FMOV.S FRm,@(R0,Rn)	FRm \varnothing (R0+Rn)	1111nnnnnnmmmm0111	—
1	FMOV DRm,@(R0,Rn)	DRm \varnothing (R0+Rn)	1111nnnnnnmm00111	—

Description

1. Transfers the contents of FRm to FRn.
2. Transfers the contents of DRm to DRn.
3. Stores the contents of FRm into memory, using the contents of Rn as the effective address.
4. Stores the contents of DRm into memory, using the contents of Rn as the effective address.
5. Loads data from memory into FRn, using the contents of Rm as the effective address.
6. Loads data from memory into DRn, using the contents of Rm as the effective address.
7. Loads data from memory into FRn, using the contents of Rm as the effective address, and then increments the value in Rm by 4.
8. Loads data from memory into DRn, using the contents of Rm as the effective address, and then increments the value in Rm by 8.
9. First decrements the value in Rn by 4, and then stores the contents of FRm into memory, using the contents of Rn as the effective address.
10. First decrements the value in Rn by 8, and then stores the contents of DRm into memory, using the contents of Rn as the effective address.
11. Loads data from memory into FRn, using the sum of the contents of R0 and Rm as the effective address.
12. Loads data from memory into DRn, using the sum of the contents of R0 and Rm as the effective address.

13. Stores the contents of FRm into memory, using the sum of the contents of R0 and Rn as the effective address.
14. Stores the contents of DRm into memory, using the sum of the contents of R0 and Rn as the effective address.

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] -= 4;
    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;
}

```

Exceptions

Data TLB miss exception

Data protection violation exception

Initial write exception

Address error

FMOV (Floating Point Move Extension): Floating-Point Instruction

SZ	Format	Operation	Instruction Code	T Bit
1	FMOV XDm,@Rn	$\text{XDm} \oslash (\text{Rn})$	1111nnnnmmml1010	—
1	FMOV @Rm,XDn	$(\text{Rm}) \oslash \text{XDn}$	1111nnn1mmmm1000	—
1	FMOV @Rm+,XDn	$(\text{Rm}) \oslash \text{XDn}, \text{Rm} += 8$	1111nnn1mmmm1001	—
1	FMOV XDm,@-Rn	$\text{Rn} -= 8, \text{XDm} \oslash (\text{Rn})$	1111nnnnmmml1011	—
1	FMOV @(R0,Rm),XDn	$(\text{R0} + \text{Rm}) \oslash \text{XDn}$	1111nnn1mmmm0110	—
1	FMOV XDm,@(R0,Rn)	$\text{XDm} \oslash (\text{R0} + \text{Rn})$	1111nnnnmmml1011	—
1	FMOV XDm,XDn	$\text{XDm} \oslash \text{XDn}$	1111nnn1mmml1100	—
1	FMOV XDm,DRn	$\text{XDm} \oslash \text{DRn}$	1111nnn0mmml1100	—
1	FMOV DRm,XDn	$\text{DRm} \oslash \text{XDn}$	1111nnn1mmml0100	—

Description

1. Stores the contents of XDm into memory, using the contents of Rn as the effective address.
2. Loads data from memory into XDn, using the contents of Rm as the effective address.
3. Loads data from memory into XDn, using the contents of Rm as the effective address, and then increments the value in Rm by 8.
4. First decrements the value in Rn by 8, and then stores the contents of XDm in memory, using the contents of Rn as the effective address.
5. Loads data from memory into XDn, using the sum of the contents of R0 and Rm as the effective address.
6. Stores the contents of XDm into memory, using the sum of the contents of R0 and Rn as the effective address.
7. Transfers the contents of XDm to XDn.
8. Transfers the contents of XDm to DRn.
9. Transfers the contents of DRm 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;
}
```

Exceptions

Data TLB miss exception

Data protection violation exception

Initial write exception

Address error

FMUL (Floating Point Multiply): Floating-Point Instruction

PR	Format	Operation	Instruction Code	T Bit
0	FMUL FRm,FRn	FRn*FRm \rightarrow FRn	1111nnnnnnmmmm0010	—
1	FMUL DRm,DRn	DRn*DRm \rightarrow DRn	1111nnn0mmmm00010	—

Description

When the PR bit in FPSCR is 0: Arithmetically multiplies together the two single-precision floating-point numbers in FRm and FRn, and writes the result to FRn.

When the PR bit in FPSCR is 1: Arithmetically multiplies together the two double-precision floating-point numbers in DRm and DRn, and writes the result to DRn.

If the O, U, or I bit in the FPSCR.enable field is enabled, an FPU exception trap will be raised regardless of the exception occurrence, FPSCR.cause and FPSCR.flag fields reflect the actual FPU exception status, and FRn or DRn does not change. Appropriate actions must be taken 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_fmula(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		Error	qNaN	Invalid				
+0	0	+0	−0	Invalid								
−0		−0	+0									
+INF	INF	Invalid		+INF	−INF							
−INF				−INF	+INF							
DENORM	Error						qNaN	Invalid				
qNaN												
sNaN												

Note: When DN = 1, a denormalized number is treated as zero.

Exceptions

FPU error
 Invalid operation
 Overflow
 Underflow
 Inexact

FNEG (Floating Point Negate): Floating-Point Instruction

PR	Format	Operation	Instruction Code	T Bit
0	FNEG FRn	$-\text{FRn} \oslash \text{FRn}$	1111nnnn01001101	—
1	FNEG DRn	$-\text{DRn} \oslash \text{DRn}$	1111nnn001001101	—

Description

Inverts the most significant bit (sign bit) of the floating-point number in FRn/DRn, and writes the result to FRn/DRn. FPSCR.cause field and FPSCR.flag field do not change.

Operation

```
void FNEG (int n){  
    FR[n] = -FR[n];  
    pc += 2;  
}  
/* The same operation is performed regardless of the precision. */
```

FRCHG (FR-Bit Change): Floating-Point Instruction

PR	Format	Operation	Instruction Code	T Bit
0	FRCHG	\sim FPSCR.FR \oslash FPSCR.FR	11111011111111101	—
1	—	reserved	11111011111111101	—

Description

Inverts the FR bit in floating-point status register FPSCR. When the FR bit in FPSCR is changed, the mapping of FR0 to FR15 and XR0 to XR15 corresponding to FPPR0 to FPPR31 is changed. When FPSCR.FR = 0, FR0 to FR15 correspond to FPPR0 to FPPR15, and XR0 to XR15 correspond to FPPR6 to FPPR31; when FPSCR.FR = 1, FR0 to FR15 correspond to FPPR6 to FPPR31, and XR0 to XR15 correspond to FPPR0 to FPPR15.

Operation

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

FSCHG (SZ-Bit Change): Floating-Point Instruction

PR	Format	Operation	Instruction Code	T Bit
0	FSCHG	\sim FPSCR.SZ \oslash FPSCR.SZ	1111001111111101	—
1	—	reserved	1111001111111101	—

Description

Inverts the SZ bit in floating-point status register FPSCR. When the SZ bit in FPSCR is changed, the transfer size of FMOV instructions is changed. When FPSCR.SZ = 0, the transfer size is 32 bit; when FPSCR.SZ = 1, the transfer size is 64 bit.

Operation

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

FSQRT (Floating Point Square Root): Floating-Point Instruction

PR	Format	Operation	Instruction Code	T Bit
0	FSQRT FRn	FRn \oslash FRn	1111nnnn01101101	—
1	FSQRT DRn	DRn \oslash DRn	1111nnnn001101101	—

Description

When the PR bit in FPSCR is 0: Writes the square root of the single-precision floating-point number in FRn to FRn.

When the PR bit in FPSCR is 1: Writes the square root of the double-precision floating-point number in DRn to DRn.

If the I bit in the FPSCR.enable field is enabled, an FPU exception trap will be raised regardless of the exception occurrence, FPSCR.cause and FPSCR.flag fields reflect the actual FPU exception status, and FRn or DRn does not change. Appropriate actions must be taken 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, a denormalized number is treated as zero.

Exceptions

FPU error
Invalid operation
Inexact

FSTS (Floating Point Store System Register): Floating-Point Instruction

Format	Operation	Instruction Code	T Bit
FSTS FPUL,FRn	FPUL \varnothing FRn	1111nnnn00001101	—

Description

Copies the content of the system register FPUL to floating point register FRn.

Operation

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

Exceptions

None

FSUB (Floating Point Subtract): Floating-Point Instruction

PR	Format	Operation	Instruction Code	T Bit
0	FSUB FRm,FRn	FRn-FRm \oslash FRn	1111nnnnmmmm0001	—
1	FSUB DRm,DRn	DRn-DRm \oslash DRn	1111nnn0mmm00001	—

Description

When the PR bit in FPSCR is 0: Arithmetically subtracts the single-precision floating-point number in FRm from the single-precision floating-point number in FRn, and writes the result to FRn.

When the PR bit in FPSCR is 1: Arithmetically subtracts the double-precision floating-point number in DRm from the double-precision floating-point number in DRn, and writes the result to DRn.

If the O, U, or I bit in the FPSCR.enable field is enabled, an FPU exception trap will be raised regardless of the exception occurrence, FPSCR.cause and FPSCR.flag fields reflect the actual FPU exception status, and FRn or DRn does not change. Appropriate actions must be taken 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;
        }
        case PZERO:      break;
        case NZERO: switch (data_type_of(n)){
            case NZERO: zero(n,0); break;
            default:      break;
        }
        case PINF: switch (data_type_of(n)){
            case PINF:    invalid(n);    break;
            default:      inf(n,1);      break;
        }
        case NINF: switch (data_type_of(n)){
            case NINF:    invalid(n);    break;
            default:      inf(n,0);      break;
        }
    }
}
```


FSUB Special Cases

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

Note: When DN = 1, a denormalized number is treated as zero.

Exceptions

FPU error
Invalid operation
Overflow
Underflow
Inexact

FTRC (Floating Point Truncate and Convert to Integer): Floating-Point Instruction

PR	Format	Operation	Instruction Code	T Bit
0	FTRC FRm,FPUL	(long)FRm \varnothing FPUL	1111mmmm00111101	—
1	FTRC DRm,FPUL	(long)DRm \varnothing FPUL	1111mmmm000111101	—

Description

When the PR bit in FPSCR is 0: Converts the single-precision floating-point number in FRm to an integer, and writes this number to FPUL.

When the PR bit in FPSCR is 1: Converts the double-precision floating-point number in DRm to an integer, and writes this number to FPUL.

The rounding method used in conversion is always to discard the fractional part.

If the I bit in the FPSCR.enable field is enabled, an FPU exception will be generated before execution, and therefore appropriate action must be taken by software.

Operation

```
#define N_INT_SINGLE_RANGE 0xcf000000 /* -1.000000 * 2^31 */
#define P_INT_SINGLE_RANGE 0x4effffff /* 1.fffffe * 2^30 */
#define N_INT_DOUBLE_RANGE 0xc1e00000 /* higher of -1.0000000000000 * 2^31 */
#define P_INT_DOUBLE_RANGE 0x41dfffff /* higher of 1.fffffffffffff * 2^30 */

void FTRC(int m)
{
    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_single_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] < 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(FR_HEX[m] > P_INT_DOUBLE_RANGE)
            return(PINF); /* out of range,+INF */
        else return(NORM); /* +0,+NORM */
    } else {
        if(FR_HEX[m] < N_INT_DOUBLE_RANGE)
            return(NINF); /* out of range ,+INF,NaN*/
        else return(NORM); /* -0,-NORM */
    }
}

void ftrc_invalid(int sign)
{
    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, a denormalized number is treated as zero.

Exceptions

FPU error
Invalid operation
Inexact

FTRV (Floating Point Transform Vector): Floating-Point Instruction

PR	Format	Operation	Instruction Code	T Bit
0	FTRV XMTRX,FVn	$\text{XMTRX} * \text{FVn} \oslash \text{FVn}$	1111nn0111111101	—
1	—	reserved	1111nn0111111101	—

Description

When the PR bit in the FPSCR register is 0: Finds the matrix products of the 4-row/4-column conversion matrix XMTRX comprising floating-point registers XF0 to XF15, and vector FVn comprising floating-point registers FRn to FR[n+3], and writes the result of the operation in FVn.

XMTRX					FVn		FVn
XF[0]	XF[4]	XF[8]	XF[12]		FR[n]		FR[n]
XF[1]	XF[5]	XF[9]	XF[13]	∞	FR[n+1]	\oslash	FR[n+1]
XF[2]	XF[6]	XF[10]	XF[14]		FR[n+2]		FR[n+2]
XF[3]	XF[7]	XF[11]	XF[15]		FR[n+3]		FR[n+3]

The operation is sometimes more or less accurate than the calculation using FMUL and FMAC because of the algorism difference. The order of the operations is as follows:

1. Multiplication of terms (result of each term is 30 digits)
2. Digit alignment, truncation below 30 digits
3. Addition of all terms
4. Normalization, rounding

Special cases are as follows:

1. If the input operands include sNaN: invalid operation
2. If the input operands for a term are infinity and zero: invalid operation
3. If the input operands include qNaN, other than as above, the result is qNaN.
4. If the input operands include infinity, other than as above:
 - a. If the multiplication results for two terms or more are infinity and not of the same sign: invalid operation
 - b. Other than the above, the result is infinity with the proper sign.
5. If the input operands do not include sNaN, qNaN, or infinity: the same as for a normal instruction

If the V, O, U, or I bit in the FPSCR.enable field is enabled, an FPU exception trap will be raised regardless of the exception occurrence, FPSCR.cause and FPSCR.flag fields reflect the actual FPU exception status, and FRn or DRn does not change. Appropriate actions must be taken 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);
            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();
}
```

Exceptions

Invalid operation
 Overflow
 Underflow
 Inexact

JMP (Jump): Branch Instruction

Class: Delayed branch instruction

Format	Abstract	Code	T Bit
JMP @Rn	Rn \varnothing PC	0100nnnn00101011	—

Description: branches unconditionally to an address specified with Rn.

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

Operation:

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

Examples:

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

JSR (Jump to Subroutine): Branch Instruction

Class: Delayed branch instruction

Format	Abstract	Code	T Bit
JSR @Rn	PC + 4 \nrightarrow PR, Rn \nrightarrow PC	0100nnnn00001011	—

Description: Branches to the subroutine at a specified address after executing the instruction following this JSR instruction. The return address (PC+4) is stored in PR. The jump target is an address specified with general register Rn. The JSR instruction and RTS instruction are used for subroutine calls.

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

Operation:

```
JSR(int n) /* JSR @Rn */
{
    PR=PC+4;
    Delay_Slot(PC+2);
    PC=R[n];
}
```

Examples:

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


LDC (Load to Control Register): System Control Instruction

(Privileged Instruction)

Format	Operation	Instruction Code	T Bit
LDC Rm,SR	Rm \oslash SR	0100mmmm00001110	LSB
LDC Rm,GBR	Rm \oslash GBR	0100mmmm00011110	—
LDC Rm,VBR	Rm \oslash VBR	0100mmmm00101110	—
LDC Rm,SSR	Rm \oslash SSR	0100mmmm00111110	—
LDC Rm,SPC	Rm \oslash SPC	0100mmmm01001110	—
LDC Rm,DBR	Rm \oslash DBR	0100mmmm11110101	—
LDC Rm,R0_BANK	Rm \oslash R0_BANK	0100mmmm10001110	—
LDC Rm,R1_BANK	Rm \oslash R1_BANK	0100mmmm10011110	—
LDC Rm,R2_BANK	Rm \oslash R2_BANK	0100mmmm10101110	—
LDC Rm,R3_BANK	Rm \oslash R3_BANK	0100mmmm10111110	—
LDC Rm,R4_BANK	Rm \oslash R4_BANK	0100mmmm11001110	—
LDC Rm,R5_BANK	Rm \oslash R5_BANK	0100mmmm11011110	—
LDC Rm,R6_BANK	Rm \oslash R6_BANK	0100mmmm11101110	—
LDC Rm,R7_BANK	Rm \oslash R7_BANK	0100mmmm11111110	—
LDC.L @Rm+,SR	(Rm) \oslash SR, Rm+4 \oslash Rm	0100mmmm00000111	LSB
LDC.L @Rm+,GBR	(Rm) \oslash GBR, Rm+4 \oslash Rm	0100mmmm00010111	—
LDC.L @Rm+,VBR	(Rm) \oslash VBR, Rm+4 \oslash Rm	0100mmmm00100111	—
LDC.L @Rm+,SSR	(Rm) \oslash SSR, Rm+4 \oslash Rm	0100mmmm00110111	—
LDC.L @Rm+,SPC	(Rm) \oslash SPC, Rm+4 \oslash Rm	0100mmmm01000111	—
LDC.L @Rm+,DBR	(Rm) \oslash DBR, Rm+4 \oslash Rm	0100mmmm11110110	—
LDC.L @Rm+,R0_BANK	(Rm) \oslash R0_BANK, Rm+4 \oslash Rm	0100mmmm10000111	—
LDC.L @Rm+,R1_BANK	(Rm) \oslash R1_BANK, Rm+4 \oslash Rm	0100mmmm10010111	—
LDC.L @Rm+,R2_BANK	(Rm) \oslash R2_BANK, Rm+4 \oslash Rm	0100mmmm10100111	—
LDC.L @Rm+,R3_BANK	(Rm) \oslash R3_BANK, Rm+4 \oslash Rm	0100mmmm10110111	—
LDC.L @Rm+,R4_BANK	(Rm) \oslash R4_BANK, Rm+4 \oslash Rm	0100mmmm11000111	—
LDC.L @Rm+,R5_BANK	(Rm) \oslash R5_BANK, Rm+4 \oslash Rm	0100mmmm11010111	—
LDC.L @Rm+,R6_BANK	(Rm) \oslash R6_BANK, Rm+4 \oslash Rm	0100mmmm11100111	—
LDC.L @Rm+,R7_BANK	(Rm) \oslash R7_BANK, Rm+4 \oslash Rm	0100mmmm11110111	—

Description

These instructions store the source operand in the control register SR, GBR, VBR, SSR, SPC, DBR, or R0_BANK to R7_BANK. With the exception of LDC Rm,GBR and LDC.L @-Rn,GBR, the LDC and 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 LDC/LDC.L instructions accessing Rm_BANK, Rm_BANK0 is accessed when the RB bit in the SR register is 1, and Rm_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;
}

```

Exceptions

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

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

No	Format	Abstract	Code	T Bit
.				
1	LDS Rm ,FPUL	Rm->FPUL	0100mmmm01011010	—
2	LDS.L @Rm+,FPUL	(Rm)->FPUL, Rm+=4	0100mmmm01010110	—
3	LDS Rm ,FPSCR	Rm->FPSCR	0100mmmm01101010	—
4	LDS.L @Rm+,FPSCR	(Rm)->FPSCR, Rm+=4	0100mmmm01100110	—

Description:

1. Copies the content of general purpose register Rm to system register FPUL.
2. Loads the content of the memory location addressed by general register Rm. The result of this operation is written to system register FPUL. Upon completion, the value in Rm is incremented by 4.
3. Copies the content of general purpose register Rm to system register FPSCR. The predetermined bits of FPSCR remain unchanged.
4. Loads the content of the memory location addressed by general register Rm. The result of this operation is written to system register FPSCR. Upon completion, the value in Rm is incremented by 4. The predetermined bits of FPSCR remain unchanged.

Operation:

```
#define FPSCR_MASK 0x003FFFFFFF

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

LDSMFPUL(int m)     /* 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;
}
```

Exceptions:

Data TLB miss exception
Data Access Protection exception
Address Error.

Examples:

- LDS

Example 1:

```
MOV.L  #h'12345678, R2 ; Before executing the LDS and FSTS instructions:
                               ;      R2 = H'12345678
FLDI0  FR3              ;      FR3 = 0
LDS     R2, FPUL         ; After executing the LDS and FSTS instructions:
                               ;      FPUL = H'12345678
FSTS    FPUL, FR3        ;      FR3 = H'12345678
```

Example 2:

```
MOV.L  #h'00040801, R4 ; After executing the LDS instruction:
LDS     R4, FPSCR       ; FPSCR = H'00040801
```

- LDS.L

Example 1:

```
LDI0    FR0              ; Before executing the LDS.L and FSTS instructions:
MOV.L    #h'87654321, R4 ; FR0 = 0
MOV.L    #h'0C700128, R8 ; R8 = H'0C700128
MOV.L    R4, @R8          ; After executing the LDS.L and FSTS instructions:
LDS.L    @R8+, FPUL        ; FR0 = H'87654321
FSTS     FPUL, FR0         ; R8 = H'0C70012C
```

Example 2:

```
MOV.L    #h'00040C01, R4 ; Before executing the LDS.L instruction:
MOV.L    #h'0C700134, R8 ; R8 = H'0C700134
MOV.L    R4, @R8          ; After executing the LDS.L instruction:
                               ; R8 = H'0C700138
LDS.L    @R8+, FPSCR       ; FPSCR = H'00040C01
```

LDS (Load to System Register): System Control Instruction

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

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;
}
LDSR(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;
}

```


Examples:

LDS R0,PR Before execution R0 = H'12345678, PR = H'00000000

 After execution PR = H'12345678

LDS.L @R15+,MACL Before execution R15 = H'10000000

 After execution R15 = H'10000004, MACL = (H'10000000)

LDTLB (Load PTEH/PTEL to TLB): System Control Instruction (Privileged Only)

Format	Abstract	Code	T Bit
LDTLB	PTEH/PTEL \emptyset TLB	0000000000111000	—

Description: Loads contents of PTEH/PTEL registers to a translation lookaside buffer (TLB) specified with MMUCR.URC(Random Counter field in MMU Control Register). LDTLB is a privileged instruction and can be used in privileged mode only. If used in user mode, it causes an illegal instruction exception.

Note: As LDTLB is for loading PTEH and PTEL to the TLB, the instruction should be issued when MMU is off (MMUCR.AT = 0) or should be placed in the P1 or P2 space with MMU enabled (see section 3, MMU, of the *SH-4 Hardware Manual*). After an LDTLB instruction is issued, there should be at least one instruction between the LDTLB instruction and issuance of an instruction involving an access to the P0, U0, or P3 area: BRAF, BSRF, JMP, JSR, RTS, and RTE.

Operation:

```
LDTLB() /* LDTLB */
{
    TLB[MMUCR.URC].ASID=PTEH&0x000000FF;
    TLB[MMUCR.URC].VPN=(PTEH&0xFFFFFC00)>>10;
    TLB[MMUCR.URC].PPN=(PTEL&0x1FFFFC00)>>10;
    TLB[MMUCR.URC].SZ=(PTEL&0x00000080)>>6 | (PTEL&0x00000010)>>4;
    TLB[MMUCR.URC].SH=(PTEL&0x00000002)>>1;
    TLB[MMUCR.URC].PR=(PTEL&0x00000060)>>5;
    TLB[MMUCR.URC].WT=(PTEL&0x00000001);
    TLB[MMUCR.URC].C=(PTEL&0x00000008)>>3;
    TLB[MMUCR.URC].D=(PTEL&0x00000004)>>2;
    TLB[MMUCR.URC].V=(PTEL&0x00000100)>>8;
    PC+=2;
}
```

Examples:

MOV @R0, R1	Load page table entry to R1
MOV R1, @R2	Load R1 to PTEL, R2 = H'FFFFFFF4
LDTLB	Load PTEH/PTEL to TLB

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

Format	Abstract	Code	T Bit
MAC.L @Rm+,@Rn+	Signed operation, $(Rn) \times (Rm) + MAC \oslash MAC, Rm + 4 \oslash Rm, Rn + 4 \oslash Rn$	0000nnnnmmmm1111	—

Description: Does signed multiplication of 32-bit operands obtained using the contents of general registers Rm and Rn as addresses. The 64-bit result is added to contents of the MAC register, and the final result is stored in the MAC register. Every time an operand is read, Rm and Rn are incremented by four.

When the S bit is equal to zero, the 64-bit result is stored in the coupled MACH and MACL registers. When bit S is set to 1, addition to the MAC register is a saturation operation of 48 bits starting from the LSB. For the saturation operation, only the lower 48 bits of the MACL register are enabled and the result is limited to between H'FFFF800000000000 (minimum) and H'00007FFFFFFFFFFFFF (maximum).

Operation:

```
MACL(int m,int n) /* MAC.L @Rm+,@Rn+ */
{
    int src_m,src_n;
    long mul,mac;

    src_n=(int)Read_Long(R[n]);
    R[n]+=4;
    src_m=(int)Read_Long(R[m]);
    R[m]+=4;

    mul=(long)src_m*(long)src_n;
    mac=(long)MACH<<32 | (long)MACL;
    mac+=mul;

    if(S==1){
        if(mac>0x00007FFFFFFFFFFFFF)
            mac=0x00007FFFFFFFFFFFFF;
        else if(mac<0xFFFF800000000000)
            mac=0xFFFF800000000000;
    }
    MACH=(unsigned int)((unsigned long)mac)>>32;
    MACL=(unsigned int)(mac&0x00000000FFFFFFFF);
    PC+=2;
}
```

Examples:

```
MOVA    TBLM,R0    Table address
MOVR0,R1
MOVA    TBLN,R0    Table address
CLRMAC    MAC register initialization
MAC.L   @R0+,@R1+
MAC.L   @R0+,@R1+
STSMACL,R0    Store result into R0
.....
.align 2
TBLM    .data.l    H'1234ABCD
        .data.l    H'5678EF01
TBLN    .data.l    H'0123ABCD
        .data.l    H'4567DEF0
```

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

Format	Abstract	Code	T Bit
MAC.W @Rm+,@Rn+	With sign, $(Rn) \times (Rm) + MAC \oslash MAC$, Rm + 2 \oslash Rm, Rn + 2 \oslash Rn	0100nnnnmmmm1111	—

Description: Multiplies 16-bit operands obtained using the contents of general registers Rm and Rn as addresses. The 32-bit result is added to contents of the MAC register, and the final result is stored in the MAC register.

When the S bit is equal to 0, the 64-bit result is stored in the coupled MACH and MACL registers. Rm and Rn are incremented by 2.

When the bit S is equal to 1, the addition to the MAC register is a saturation operation. For the saturation operation, only the MACL register is enabled and the result is limited to between H'80000000 (minimum) and H'7FFFFFFF (maximum). If an overflow or an underflow occurs, the LSB of the MACH register is set to 1. The other bits of MACH are undefined. The result is stored in the MACL register, and the result is saturated to a value between H'80000000 (minimum) and H'7FFFFFFF (maximum).

Note: The content of higher 31bits of MACH is undefined value if S bit is equal to 1.

Operation:

```
MACW(int m,int n) /* MAC.W @Rm+,@Rn+ */
{
    int src_m,src_n;
    long mul,mac;

    src_n=(int)Read_Word(R[n]);
    R[n]+=2;
    src_m=(int)Read_Word(R[m]);
    R[m]+=2;

    mul=(long)src_m*(long)src_n;
    if(S==0){
        mac=(long)MACH<<32 | ((long)MACL&0x00000000FFFFFFFF);
        mac+=mul;
        MACH=(unsigned int)((unsigned long)mac>>32);
        MACL=(unsigned int)(mac&0x00000000FFFFFFFF);
    }else{
        if(MACL&0x80000000)
            mac=(long)MACL | 0xFFFFFFFF00000000;
        else    mac=(long)MACL & 0x00000000FFFFFFFF;
        mac+=mul;
        if(mac>0x000000007FFFFFFF){
            MACH|=0x00000001; MACL=0x7FFFFFFF;
        }else if(mac<0xFFFFFFFF80000000){
            MACH|=0x00000001; MACL=0x80000000;
        }else{
            MACH&=0xFFFFFFF0; MACL=(unsigned int)(mac&0x00000000FFFFFFFF);
        }
    }
    PC+=2;
}
```

Examples:

```
MOVW    TBLM,R0      Table address
MOV     R0,R1
MOVW    TBLN,R0      Table address
CLRMAC                     MAC register initialization
MAC.W   @R0+,@R1+
MAC.W   @R0+,@R1+
STS     MACL,R0       Store result into R0
.....
.align  2
TBLM    .data.w        H'1234
        .data.w        H'5678
TBLN    .data.w        H'0123
        .data.w        H'4567
```

MOV (Move): Data Transfer Instruction

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

Description: Transfers the source operand to the destination. When the operand is stored in memory, the transferred data can be a byte, word, or longword. Loaded data from memory is stored in a register after it is sign-extended to a longword.

Operation:

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

```

MOVBS(int m,int n)      /* MOV.B Rm,@Rn */
{
    Write_Byte(R[n],R[m]);
    PC+=2;
}
MOVWS(int m,int n)      /* MOV.W Rm,@Rn */
{
    Write_Word(R[n],R[m]);
    PC+=2;
}
MOVLS(int m,int n)      /* MOV.L Rm,@Rn */
{
    Write_Long(R[n],R[m]);
    PC+=2;
}
MOVBL(int m,int n)      /* MOV.B @Rm,Rn */
{
    R[n]=(int)Read_Byte(R[m]);
    if ((R[n]&0x80)==0)
        R[n]&=0x000000FF;
    else    R[n]|=0xFFFFF00;
    PC+=2;
}
MOVWL(int m,int n)      /* MOV.W @Rm,Rn */
{
    R[n]=(int)Read_Word(R[m]);
    if ((R[n]&0x8000)==0)
        R[n]&=0x0000FFFF;
    else    R[n]|=0xFFFF0000;
    PC+=2;
}
MOVLl(int m,int n)      /* MOV.L @Rm,Rn */
{
    R[n]=Read_Long(R[m]);
    PC+=2;
}
MOVBM(int m,int n)      /* MOV.B Rm,@-Rn */
{
    Write_Byte(R[n]-1,R[m]);
    R[n]-=1;
    PC+=2;
}
MOVWM(int m,int n)      /* MOV.W Rm,@-Rn */
{
    Write_Word(R[n]-2,R[m]);
    R[n]-=2;
    PC+=2;
}
MOVLm(int m,int n)      /* MOV.L Rm,@-Rn */
{
    Write_Long(R[n]-4,R[m]);
    R[n]-=4;
    PC+=2;
}
MOVBP(int m,int n) /* MOV.B @Rm+,Rn */
{
    R[n]=(int)Read_Byte(R[m]);

```



```

        if ((R[n]&0x80)==0)
            R[n]&=0x000000FF;
        else    R[n]|=0xFFFFFFFF00;
        if (n!=m) R[m]+=1;
        PC+=2;
    }
MOVWP(int m,int n)    /* MOV.W @Rm+,Rn */
{
    R[n]=(int)Read_Word(R[m]);
    if ((R[n]&0x8000)==0)
        R[n]&=0x0000FFFF;
    else    R[n]|=0xFFFF0000;
    if (n!=m) R[m]+=2;
    PC+=2;
}
MOVLPL(int m,int n)    /* MOV.L @Rm+,Rn */
{
    R[n]=Read_Long(R[m]);
    if (n!=m) R[m]+=4;
    PC+=2;
}
MOVBS0(int m,int n)    /* MOV.B Rm,@(R0,Rn) */
{
    Write_Byte(R[n]+R[0],R[m]);
    PC+=2;
}
MOVWS0(int m,int n)    /* MOV.W Rm,@(R0,Rn) */
{
    Write_Word(R[n]+R[0],R[m]);
    PC+=2;
}
MOVLS0(int m,int n)    /* MOV.L Rm,@(R0,Rn) */
{
    Write_Long(R[n]+R[0],R[m]);
    PC+=2;
}
MOVBLO(int m,int n)    /* MOV.B @(R0,Rm),Rn */
{
    R[n]=(int)Read_Byte(R[m]+R[0]);
    if ((R[n]&0x80)==0)
        R[n]&=0x000000FF;
    else    R[n]|=0xFFFFFFFF00;
    PC+=2;
}
MOVWLO(int m,int n)    /* MOV.W @(R0,Rm),Rn */
{
    R[n]=(int)Read_Word(R[m]+R[0]);
    if ((R[n]&0x8000)==0)
        R[n]&=0x0000FFFF;
    else    R[n]|=0xFFFF0000;
    PC+=2;
}
MOVLLO(int m,int n)    /* MOV.L @(R0,Rm),Rn */
{
    R[n]=Read_Long(R[m]+R[0]);
    PC+=2;
}

```

Examples:

MOV	R0, R1	Before execution	R0 = H'FFFFFFFF, R1 = H'00000000
		After execution	R1 = H'FFFFFFFF
MOV.W	R0, @R1	Before execution	R0 = H'FFFF7F80
		After execution	(R1) = H'7F80
MOV.B	@R0, R1	Before execution	(R0) = H'80, R1 = H'00000000
		After execution	R1 = H'FFFFF80
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)

MOV (Move Constant Value): Data Transfer Instruction

Format	Abstract	Code	T Bit
MOV #imm,Rn	#imm \oslash sign extension \oslash Rn	1110nnnniiiiiii	—
MOV.W @(disp,PC),Rn	(disp + PC + 4) \oslash sign extension \oslash Rn	1001nnnnddddddd	—
MOV.L @(disp,PC),Rn	(disp + PC + 4) \oslash Rn	1101nnnnddddddd	—

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

If the data is a word or longword, the data are loaded from a memory location at the address (PC + 4 + displacement). If the data is a word, the 8-bit displacement is zero-extended and doubled. Consequently, the range is up to PC + 4 + 510 bytes. The PC value is an instruction address of the MOV instruction. If the data is a longword, the 8-bit displacement is zero-extended and quadrupled. Consequently, the range is up to PC + 4 + 1020 bytes, but the lowest two bits of the PC are corrected to B'00.

Note: When the PC-relative load instructions are executed in a delay slot, the slot illegal instruction exception may be caused.

Operation:

```
MOVI(int i,int n)      /* MOV #imm,Rn */
{
    if ((i&0x80)==0)
        R[n]=(0x000000FF & i);
    else  R[n]=(0xFFFFFFFF00 | i);
    PC+=2;
}
```

```

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

    disp=(unsigned int)(0x000000FF & d);
    R[n]=(int)Read_Word(PC+4+(disp*2));
    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&0xFFFFFFF0)+4+(disp*4));
    PC+=2;
}

```

Examples:

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      TSTR0,R0
1008      MOV.L @(3,PC),R3   R3 = H'12345678
100A      BRANEXT          Delayed branch instruction
100C      NOP
100E      IMM.data.w       H'9ABC
1010      .data.w          H'1234
1012      NEXT      JMP@R3 Branch destination of the BRA instruction
1014      CMP/EQ #0,R0
          .align 4
1018      .data.l         H'12345678
101C      .data.l         H'9ABCDEF0

```

MOV (Move Global Data): Data Transfer Instruction

Format	Abstract	Code	T Bit
MOV.B @(disp,GBR),R0	(disp + GBR) \oslash sign extension \oslash R0	11000100dddddddd	—
MOV.W @(disp,GBR),R0	(disp + GBR) \oslash sign extension \oslash R0	11000101dddddddd	—
MOV.L @(disp,GBR),R0	(disp + GBR) \oslash R0	11000110dddddddd	—
MOV.B R0,@(disp,GBR)	R0 \oslash (disp + GBR)	11000000dddddddd	—
MOV.W R0,@(disp,GBR)	R0 \oslash (disp + GBR)	11000001dddddddd	—
MOV.L R0,@(disp,GBR)	R0 \oslash (disp + GBR)	11000010dddddddd	—

Description: Transfers the source operand to the destination. The data can be a byte, word, or longword, but only R0 register is available as the target register.

When the target data is a byte, the only change made is to zero-extend the 8-bit displacement. Consequently, an address within +255 bytes can be specified. When the target data is a word, the 8-bit displacement is zero-extended and doubled. Consequently, an address within +510 bytes can be specified. When the target data is a longword, the 8-bit displacement is zero-extended and is quadrupled. Consequently, an address within +1020 bytes can be specified. When the source operand is in memory, the loaded data is stored in the register after it is sign-extended to a longword.

Note: The destination register of a data load is always R0.

Operation:

```

MOVB LG(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]|=0xFFFFFFF0;
    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*2));
    if ((R[0]&0x8000)==0)
        R[0]&=0x0000FFFF;
    else R[0]|=0xFFFF0000;
    PC+=2;
}

MOVLG(int d) /* MOV.L @(disp,GBR),R0 */
{
    unsigned int disp;

    disp=(unsigned int)(0x000000FF & d);
    R[0]=Read_Long(GBR+(disp*4));
    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*2),R[0]);
    PC+=2;
}

```

```

MOVLSG(int d) /* MOV.L R0,@(disp,GBR) */
{
    unsigned int disp;

    disp=(unsigned int)(0x000000FF & d);
    Write_Long(GBR+(disp*4),R[0]);
    PC+=2;
}

```

Examples:

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

MOV (Move Displacement Addressing): Data Transfer Instruction

Format	Abstract	Code	T Bit
MOV.B R0,@(disp,Rn)	R0 \oslash (disp + Rn)	10000000nnnndddd	—
MOV.W R0,@(disp,Rn)	R0 \oslash (disp + Rn)	10000001nnnndddd	—
MOV.L Rm,@(disp,Rn)	Rm \oslash (disp + Rn)	0001nnnnmmmmdddd	—
MOV.B @(disp,Rm),R0	(disp + Rm) \oslash sign extension \oslash R0	10000100mmmmdddd	—
MOV.W @(disp,Rm),R0	(disp + Rm) \oslash sign extension \oslash R0	10000101mmmmdddd	—
MOV.L @(disp,Rm),Rn	(disp + Rm) \oslash Rn	0101nnnnmmmmdddd	—

Description: Transfers the source operand to the destination. The data can be a byte, word, or longword, but when a byte or word is selected, only the R0 register can be used. When the data is a byte, the only change made is to zero-extend the 4-bit displacement. Consequently, an address within +15 bytes can be specified. When the data is a word, the 4-bit displacement is zero-extended and doubled. Consequently, an address within +30 bytes can be specified. When the data is a longword, the 4-bit displacement is zero-extended and quadrupled. Consequently, an address within +60 bytes can be specified. If the displacement is too short to reach the memory operand, the aforementioned @(R0,Rn) mode must be used. When the source operand is in memory, the loaded data is stored in the register after it is sign-extended to a longword.

Note: When byte or word data is loaded, the destination register is always R0.

Operation:

```
MOVBS4(int d,int n)    /* MOV.B R0,@(disp,Rn) */
{
    unsigned int disp;

    disp=(unsigned int)(0x0000000F & d);
    Write_Byte(R[n]+disp,R[0]);
    PC+=2;
}
```



```

MOVWS4(int d,int n)    /* MOV.W R0,@(disp,Rn)  */
{
    unsigned int disp;

    disp=(unsigned int)(0x0000000F & d);
    Write_Word(R[n]+(disp*2),R[0]);
    PC+=2;
}

MOVLS4(int m,int d,int n) /* MOV.L Rm,@(disp,Rn)  */
{
    unsigned int disp;

    disp=(unsigned int)(0x0000000F & d);
    Write_Long(R[n]+(disp*4),R[m]);
    PC+=2;
}

MOVBL4(int m,int d)    /* MOV.B @(disp,Rm),R0  */
{
    unsigned int disp;

    disp=(unsigned int)(0x0000000F & d);
    R[0]=Read_Byte(R[m]+disp);
    if ((R[0]&0x80)==0)
        R[0]&=0x000000FF;
    else    R[0]|=0xFFFFFFF0;
    PC+=2;
}

```

```

MOVWL4(int m,int d)    /* MOV.W @(disp,Rm),R0 */
{
    unsigned int disp;

    disp=(unsigned int)(0x0000000F & d);
    R[0]=Read_Word(R[m]+(disp*2));
    if ((R[0]&0x8000)==0)
        R[0]&=0x0000FFFF;
    else    R[0]|=0xFFFF0000;
    PC+=2;
}

MOVL4(int m,int d,int n)    /* MOV.L @(disp,Rm),Rn */
{
    unsigned int disp;

    disp=(unsigned int)(0x0000000F & d);
    R[n]=Read_Long(R[m]+(disp*4));
    PC+=2;
}

```

Examples:

```

MOV.L    @(2,R0),R1    Before execution (R0 + 8) = H'12345670
                        After execution R1 = (H'12345670)

MOV.L    R0,@(H'F,R1)    Before execution R0 = H'FFFF7F80
                        After execution (R1 + 60) = H'FFFF7F80

```

MOVA (Move Effective Address): Data Transfer Instruction

Format	Abstract	Code	T Bit
MOVA @ (disp,PC),R0	disp + PC + 4 \varnothing R0	11000111ddddddd	—

Description: Stores the effective address of the source operand into general register R0. The 8-bit displacement is zero-extended and quadrupled. The PC value is an instruction address of the MOVA instruction, but the lowest two bits of the PC are corrected to B'00.

Note: When MOVA is executed in a delay slot, the slot illegal instruction exception may be caused.

Operation:

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

    disp=(unsigned int)(0x000000FF & d);
    R[0]=(PC&0xFFFFF0)+4+(disp*4);
    PC+=2;
}
```

Examples:

```
Address .org H'1006
1006      MOVA    STR,R0 Address of STR  $\varnothing$  R0
1008      MOV.B   @R0,R1 R1 = "X" ♦ PC location after correcting the lowest two bits
100A      ADDR4,R5 ♦ Original PC location for address calculation for the MOVA
instruction
          .align 4
100C      STR:    .sdata "XYZP12"
```

MOVCA.L (Move with Cache Block Allocation): Data Transfer Instruction

Format	Operation	Instruction Code	T Bit
MOVCA.L R0,@Rn	R0 \varnothing (Rn)	0000nnnn11000011	—

Description

This instruction stores the longword in R0 into memory, using the contents of general register Rn as the effective address. The operation of this instruction differs from that of other store instructions in the following case.

When the memory location to be accessed has a write-back attribute, and a cache miss occurs, the corresponding cache block is reserved, a block read from external memory is not performed, and the longword in R0 is written to that cache location. The contents of other locations in the cache block are undefined.

Operation

```
MOVCA.L(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;
}
```

Exceptions

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

Exceptions are examined taking a data access by means of this instruction as a longword store.

MOVT (Move T Bit): Data Transfer Instruction

Format	Abstract	Code	T Bit
MOVT Rn	$T \rightarrow Rn$	0000nnnn00101001	—

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

Operation:

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

Examples:

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

MUL.L (Multiply Long): Arithmetic Instruction

Format	Abstract	Code	T Bit
MUL.L Rm,Rn	$R_n \times R_m \rightarrow \text{MACL}$	0000nnnnmmmm0111	—

Description: Performs 32-bit multiplication of the contents of general registers Rn and Rm, and stores the bottom 32 bits of the result in the MACL register. The MACH register is not changed.

Operation:

```

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

```

Examples:

```

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

```

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

Format	Abstract	Code	T Bit
MULS.W Rm,Rn	Signed operation, $Rn \in Rm \nsubseteq MACL$	0010nnnnmmmm1111	—
MULS Rm,Rn			

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

Operation:

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

Examples:

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

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

Format	Abstract	Code	T Bit
MULU.W Rm,Rn	Unsigned, $R_n \times R_m \rightarrow \text{MACL}$	0010nnnnmmmm1110	—
MULU Rm,Rn			

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

Operation:

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

Examples:

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


NEG (Negate): Arithmetic Instruction

Format	Abstract	Code	T Bit
NEG Rm,Rn	$0 - Rm \oslash Rn$	0110nnnnmmmm1011	—

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

Operation:

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

Examples:

```
NEG R0,R1    Before execution R0 = H'00000001
               After execution R1 = H'FFFFFFF
```

NEGC (Negate with Carry): Arithmetic Instruction

Format	Abstract	Code	T Bit
NEGC Rm,Rn	$0 - Rm - T \oslash Rn$, Borrow $\oslash T$	0110nnnnmmmm1010	Borrow

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

Operation:

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

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

Examples:

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

NOP (No Operation): System Control Instruction

Format	Abstract	Code	T Bit
NOP	No operation	0000000000001001	—

Description: Increments the PC to execute the next instruction.

Operation:

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

Examples:

NOP Executes in one cycle

NOT (NOT—Logical Complement): Logic Operation Instruction

Format	Abstract	Code	T Bit
NOT Rm,Rn	$\sim Rm \rightarrow Rn$	0110nnnnmmmm0111	—

Description: Takes the one's complement of general register Rm data, and stores the result in Rn. This effectively inverts each bit of Rm data and stores the result in Rn.

Operation:

```
NOT(int m,int n)    /* NOT Rm,Rn */
{
    R[n]=~R[m];
    PC+=2;
}
```

Examples:

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

OCBI (Operand Cache Block Invalidate): Data Transfer Instruction

Format	Operation	Instruction Code	T Bit
OCBI @Rn	invalidate operand cache block	0000nnnn10010011	—

Description

This instruction accesses data using the contents of general register Rn as the effective address. In the event of a cache hit, the corresponding cache block is invalidated (V bit = 0). In this case, the cache block is not written back to external memory even if it is dirty (U bit = 1). In the event of a cache miss, or if the memory location to be accessed is non-cacheable, the data access results in no operation.

Operation

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

Exceptions

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

TLB miss, data TLB protection violation, and initial page write exceptions are not suppressed even if an OCBI data access results in no operation.

Exceptions are examined taking a data access by means of this instruction as a byte store.

OCBP (Operand Cache Block Purge): Data Transfer Instruction

Format	Operation	Instruction Code	T Bit
OCBP @Rn	purge operand cache block	0000nnnn10100011	—

Description

This instruction accesses data 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 then the cache block is invalidated (V bit = 0). If there is a cache hit and the corresponding cache block is clean (U bit = 0), the cache block is simply invalidated (V bit = 0). In the event of a cache miss, or if the memory location to be accessed is non-cacheable, the data access results in no operation.

Operation

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

Exceptions

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

TLB miss, data TLB protection violation, and address error exceptions are not suppressed even if an OCBP data access results in no operation.

Exceptions are examined taking a data access by means of this instruction as a byte load.

OCBWB (Operand Cache Block Write Back): Data Transfer Instruction

Format	Operation	Instruction Code	T Bit
OCBWB @Rn	Write back operand cache block	0000nnnn10110011	—

Description

This instruction accesses data 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 becomes clean (U bit = 0). In other cases—that is, in the event of a cache miss, or a cache hit when the cache block is clean (U bit = 0), or an access to a non-cacheable area—the data access results in no operation.

Operation

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

Exceptions

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

TLB miss, data TLB protection violation, and address error exceptions are not suppressed even if an OCBWB data access results in no operation.

Exceptions are examined taking a data access by means of this instruction as a byte load.

OR (OR Logical) Logic Operation Instruction

Format	Abstract	Code	T Bit
OR Rm,Rn	$Rn \mid Rm \oslash Rn$	0010nnnnmmmm1011	—
OR #imm,R0	$R0 \mid imm \oslash R0$	11001011iiiiiii	—
OR.B #imm,@(R0,GBR)	$(R0 + GBR) \mid imm \oslash (R0 + GBR)$	11001111iiiiiii	—

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

Operation:

```

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

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

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

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

```


Examples:

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

PREF (Prefetch Data to Cache)

Format	Abstract	Code	T Bit
PREF @Rn	prefetch cache lock	0000nnnn10000011	—

Description: loads a 32-byte data block, which begins at a 32-byte boundary, to Operand Cache. Lowest 5 bits of the address specified with Rn are masked to zero.

No address related error is detected in this instruction. In case of an error, the instruction operates as NOP.

Operation:

```
PREF(int n) /* PREF */
{
    prefetch_data_block(R[n]&H'FFFFFFE0);
    PC+=2;
}
```

Examples:

```
MOV.L SOFT_PF,R1    Address of R1 is SOFT_PF
PREF @R1             Load data from SOFT_PF to on-chip cache

.align 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
```

ROTCL (Rotate with Carry Left): Shift Instruction

Format	Abstract	Code	T Bit
ROTCL Rn	T \blacklozenge Rn \blacklozenge T	0100nnnn00100100	MSB

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

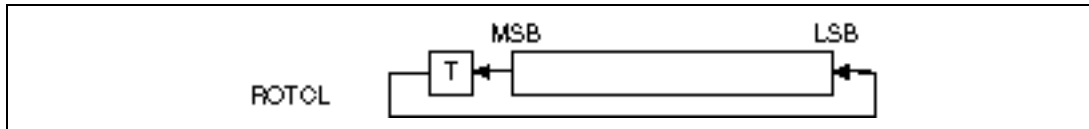


Figure 10.3 Rotate with Carry Left

Operation:

```

ROTCL(int n)    /* ROTCL Rn */
{
    int temp;

    if ((R[n]&0x80000000)==0)
        temp=0;
    else    temp=1;
    R[n]<<=1;
    if(T==1) R[n]|=0x00000001;
    else    R[n]&=0xFFFFFFFE;
    T=temp;
    PC+=2;
}

```

Examples:

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

ROTCR (Rotate with Carry Right): Shift Instruction

Format	Abstract	Code	T Bit
ROTCR Rn	T \oslash Rn \oslash T	0100nnnn00100101	LSB

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

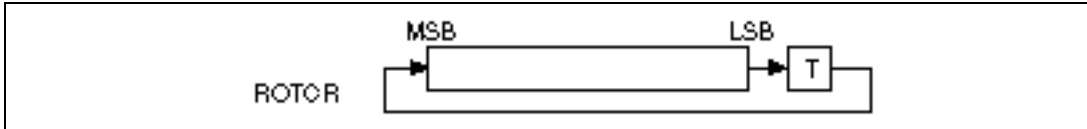


Figure 10.4 Rotate with Carry Right

Operation:

```
ROTCR(int n)    /* ROTCR Rn */
{
    int temp;

    if((R[n]&0x00000001)==0)
        temp=0;
    else    temp=1;
    R[n]>>=1;
    if(T==1) R[n]|=0x80000000;
    else    R[n]&=0x7FFFFFFF;
    T=temp;
    PC+=2;
}
```

Examples:

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

ROTL (Rotate Left): Shift Instruction

Format		Abstract	Code	T Bit
ROTL	Rn	T \blacklozenge Rn \blacklozenge MSB	0100nnnn00000100	MSB

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

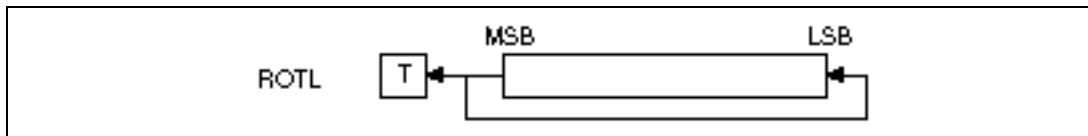


Figure 10.5 Rotate Left

Operation:

```

ROTL(int 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]&=0xFFFFFFFF;
    PC+=2;
}

```

Examples:

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

ROTR (Rotate Right): Shift Instruction

Format	Abstract	Code	T Bit
ROTR Rn	LSB \leftrightarrow Rn \leftrightarrow T	0100nnnn00000101	LSB

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

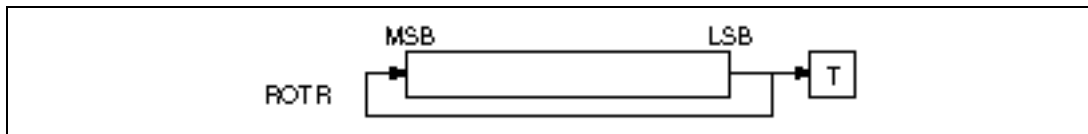


Figure 10.6 Rotate Right

Operation:

```
ROTR(int n) /* ROTR Rn */
{
    if ((R[n]&0x00000001)==0)
        T=0;
    else    T=1;
    R[n]>>=1;
    if(T==1) R[n]|=0x80000000;
    else    R[n]&=0x7FFFFFFF;
    PC+=2;
}
```

Examples:

```
ROTR    R0    Before execution R0 = H'00000001, T = 0
          After execution  R0 = H'80000000, T = 1
```

RTE (Return from Exception): System Control Instruction (Privileged Only)

Class: Delayed branch instruction

Format	Abstract	Code	T Bit
RTE	SSR \emptyset SR, SPC \emptyset PC	0000000000101011	—

Description: Returns from an exception routine. The PC and SR values are loaded from SPC and SSR. The program continues from the address specified by the loaded PC value. RTE is a privileged instruction and can be used in privileged mode only. If used in user mode, it causes an illegal instruction exception.

Note: Since this is a delayed branch instruction, the instruction after this RTE is executed before branching. No interrupts are accepted between this instruction and the next instruction. An instruction in delay slot of RTE should not cause any exception. If the next instruction is a branch instruction, it is acknowledged as an illegal slot instruction. The content of SR accessed by an instruction in the delay slot of an RTE is a value which is restored from the SSR by the RTE. The SR.MD value, which had been defined before the RTE execution, is however used to fetch a instruction in delay slot of the RTE.

Operation:

```
RTE() /* RTE */
{
    SR=SSR;
    Delay_Slot(PC+2);
    PC=SPC;
}
```

Examples:

```
RTE      Returns to the original routine
ADD #8,R15  Executes ADD before branching
```

RTS (Return from Subroutine): Branch Instruction

Class: Delayed branch instruction

Format	Abstract	Code	T Bit
RTS	PR \emptyset PC	0000000000001011	—

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

Note: Since this is a delayed branch instruction, the instruction after this RTS is executed before branching. No interrupts are accepted between this instruction and the next instruction. If the next instruction is a branch instruction, it is acknowledged as an illegal slot instruction. An instruction restoring the PR should be prior to an RTS instruction. That restoring instruction should not be the delay slot of the RTS.

Operation:

```
RTS() /* RTS */
{
    Delay_Slot(PC+2);
    PC=PR;
}
```

Examples:

```
MOV.L    TABLE,R3    R3 = Address of TRGET
JSR@R3 Branches to TRGET
NOP      Executes NOP before branching
ADDR0,R1 ♦ Return address for when the subroutine procedure is completed (PR data)
.....
TABLE: .data.l    TRGET    Jump table
.....
TRGET: MOV R1,R0      ♦ Procedure entrance
      RTS    PR data  $\emptyset$  PC
      MOV    #12,R0      Executes MOV before branching
```


SETS (Set S Bit): System Control Instruction

Format	Abstract	Code	T Bit
SETS	$1 \nabla S$	0000000001011000	—

Description: Sets the S bit to 1.

Operation:

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

Examples:

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

SETT (Set T Bit): System Control Instruction

Format	Abstract	Code	T Bit
SETT	$1 \nabla T$	0000000000011000	1

Description: Sets the T bit to 1.

Operation:

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

Examples:

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

SHAD (Shift Arithmetic Dynamically): Shift Instruction

Format	Abstract	Code	T Bit
SHAD Rm,Rn	Rn << Rm \oslash Rn (Rm \geq 0) Rn >> Rm \oslash Rn (Rm < 0)	0100nnnnmmmm1100	—

Description: Arithmetically shifts the contents of general register Rn. General register Rm indicates the shift direction and shift count (figure 6.7).

- Shift direction: Rm \geq 0, left
Rm < 0, right
- Shift count: Rm (4–0) are used; if negative, two's complement is set to Rm.
The maximum magnitude of the left shift count is 31 (0–31).
The maximum magnitude of the right shift count is 32 (1–32).

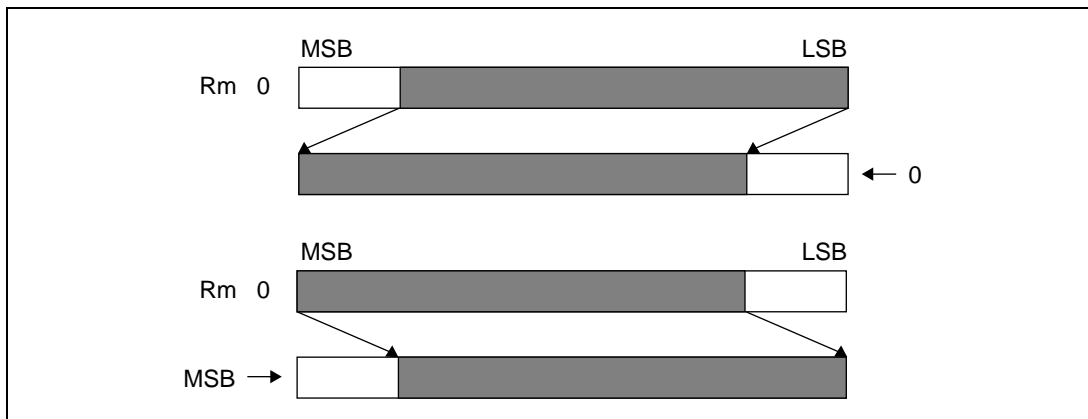


Figure 10.7 Shift Arithmetic Dynamically

Operation:

```
SHAD(int m,n) /* SHAD Rm,Rn */
{
    int sgn;
    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;
}
```

Examples:

SHAD	R1,R2	Before execution	R1 = H'FFFFFFEC, R2 = H'80180000
		After execution	R1 = H'FFFFFFEC, R2 = H'FFFFF801

SHAL (Shift Arithmetic Left): Shift Instruction

Format	Abstract	Code	T Bit
SHAL Rn	T \leftarrow Rn \leftarrow 0	0100nnnn00100000	MSB

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

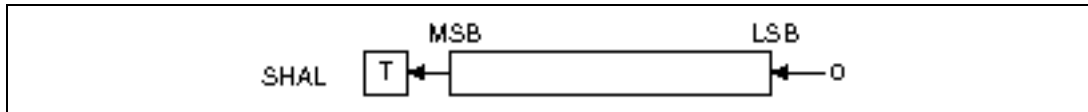


Figure 10.8 Shift Arithmetic Left

Operation:

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

Examples:

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

SHAR (Shift Arithmetic Right): Shift Instruction

Format	Abstract	Code	T Bit
SHAR Rn	MSB Ø Rn Ø T	0100nnnn00100001	LSB

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

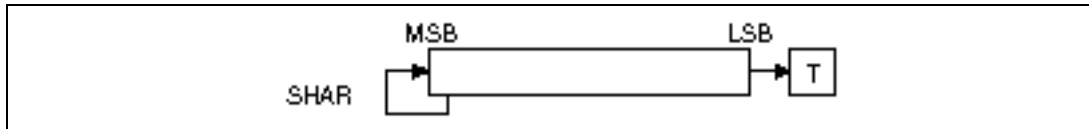


Figure 10.9 Shift Arithmetic Right

Operation:

```
SHAR(int n) /* SHAR Rn */
{
    int temp;

    if ((R[n]&0x00000001)==0) T=0;
    else T=1;
    if ((R[n]&0x80000000)==0) temp=0;
    else temp=1;
    R[n]>>=1;
    if (temp==1) R[n]|=0x80000000;
    else R[n]&=0x7FFFFFFF;
    PC+=2;
}
```

Examples:

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

SHLD (Shift Logical Dynamically): Shift Instruction

Format	Abstract	Code	T Bit
SHLD Rm,Rn	$Rn \ll Rm \oslash Rn (Rm \geq 0)$ $Rn \gg Rm \oslash Rn (Rm < 0)$	0100nnnnmmmm1101	—

Description: Arithmetically shifts the contents of general register Rn. General register Rm indicates the shift direction and shift count (figure 6.10). T bit is the last shifted bit of Rn.

- Shift direction: Rm ≥ 0 , left
Rm < 0 , right
- Shift count: Rm (4–0) are used; if negative, two's complement is set to Rm.
The maximum magnitude of the left shift count is 31 (0–31).
The maximum magnitude of the right shift count is 32 (1–32).

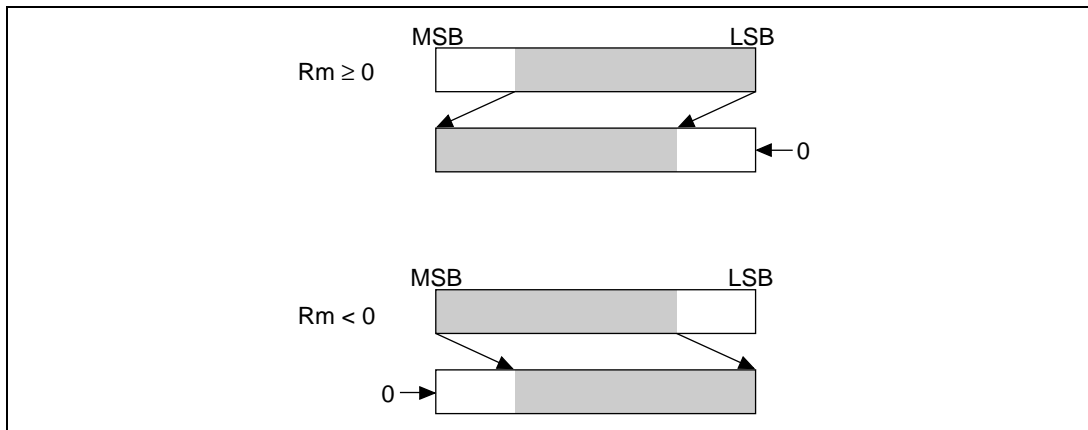


Figure 10.10 Shift Logical Dynamically

Operation:

```
SHLD(int m,n) /* SHLD Rm,Rn */
{
    int sgn;

    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;
}
```

Examples:

SHLD	R1,R2	Before execution	R1 = H'FFFFFFEC, R2 = H'80180000
		After execution	R1 = H'FFFFFFEC, R2 = H'00000801

SHLL (Shift Logical Left): Shift Instruction

Format	Abstract	Code	T Bit
SHLL Rn	T \leftarrow Rn \leftarrow 0	0100nnnn00000000	MSB

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

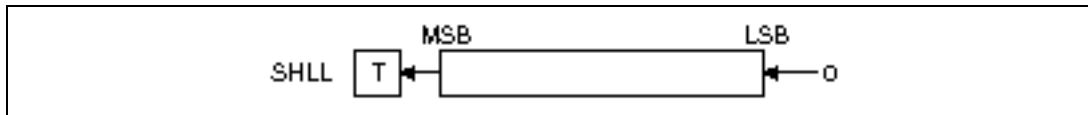


Figure 10.11 Shift Logical Left

Operation:

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

Examples:

```
SHLL    R0  Before execution R0 = H'80000001, T = 0
          After execution      R0 = H'00000002, T = 1
```

SHLLn (Shift Logical Left n Bits): Shift Instruction

Format	Abstract	Code	T Bit
SHLL2 Rn	$Rn \ll 2 \nexists Rn$	0100nnnn00001000	—
SHLL8 Rn	$Rn \ll 8 \nexists Rn$	0100nnnn00011000	—
SHLL16 Rn	$Rn \ll 16 \nexists Rn$	0100nnnn00101000	—

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

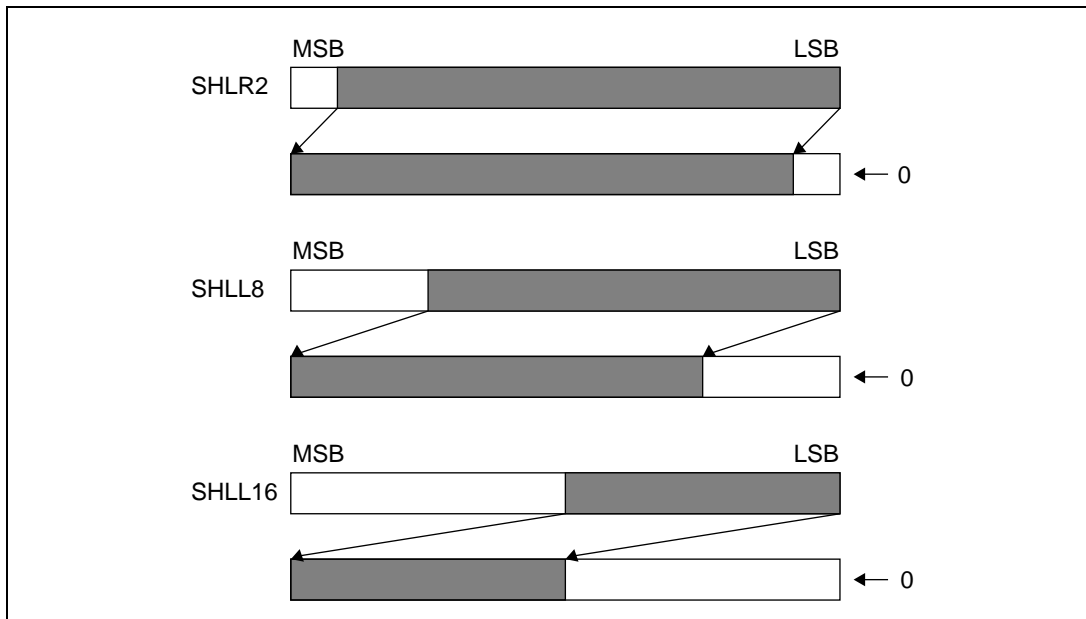


Figure 10.12 Shift Logical Left n Bits

Operation:

```
SHLL2(int n)    /* SHLL2 Rn */
{
    R[n]<<=2;
    PC+=2;
}

SHLL8(int n)    /* SHLL8 Rn */
{
    R[n]<<=8;
    PC+=2;
}

SHLL16(int n)   /* SHLL16 Rn */
{
    R[n]<<=16;
    PC+=2;
}
```

Examples:

```
SHLL2  R0  Before execution R0 = H'12345678
          After execution R0 = H'48D159E0

SHLL8  R0  Before execution R0 = H'12345678
          After execution R0 = H'34567800

SHLL16 R0  Before execution R0 = H'12345678
          After execution R0 = H'56780000
```

SHLR (Shift Logical Right): Shift Instruction

Format	Abstract	Code	T Bit
SHLR Rn	$0 \nabla Rn \nabla T$	0100nnnn00000001	LSB

Description: Logically shifts the contents of general register Rn to the right by one bit, and stores the result in Rn. The bit that is shifted out of the operand is transferred to the T bit (figure 6.13).

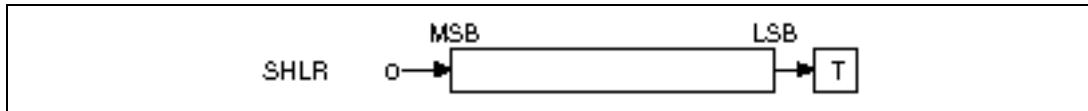


Figure 10.13 Shift Logical Right

Operation:

```
SHLR(int n) /* SHLR Rn */
{
    if ((R[n]&0x00000001)==0) T=0;
    else T=1;
    R[n]>>=1;
    R[n]&=0x7FFFFFFF;
    PC+=2;
}
```

Examples:

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

SHLRn (Shift Logical Right n Bits): Shift Instruction

Format	Abstract	Code	T Bit
SHLR2 Rn	$Rn \gg 2 \emptyset Rn$	0100nnnn00001001	—
SHLR8 Rn	$Rn \gg 8 \emptyset Rn$	0100nnnn00011001	—
SHLR16 Rn	$Rn \gg 16 \emptyset Rn$	0100nnnn00101001	—

Description: Logically shifts the contents of general register Rn to the right by 2, 8, or 16 bits, and stores the result in Rn. Bits that are shifted out of the operand are not stored (figure 6.14).

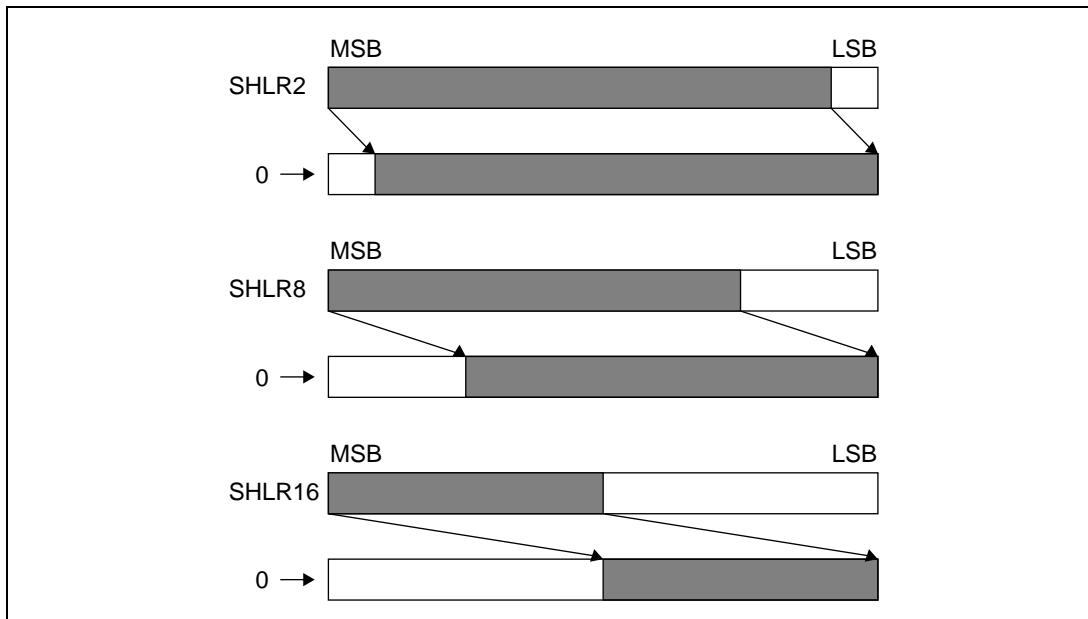


Figure 10.14 Shift Logical Right n Bits

Operation:

```
SHLR2(int n) /* SHLR2 Rn */
{
    R[n]>>=2;
    R[n]&=0x3FFFFFFF;
    PC+=2;
}

SHLR8(int n) /* SHLR8 Rn */
{
    R[n]>>=8;
    R[n]&=0x0FFFFFFF;
    PC+=2;
}

SHLR16(int n) /* SHLR16 Rn */
{
    R[n]>>=16;
    R[n]&=0x0000FFFF;
    PC+=2;
}
```

Examples:

```
SHLR2  R0  Before execution R0 = H'12345678
          After execution      R0 = H'048D159E

SHLR8  R0  Before execution R0 = H'12345678
          After execution      R0 = H'00123456

SHLR16 R0  Before execution R0 = H'12345678
          After execution      R0 = H'00001234
```

SLEEP (Sleep): System Control Instruction (Privileged Only)

Format	Abstract	Code	T Bit
SLEEP	Sleep	0000000000011011	—

Description: Sets the CPU into power-down mode. In power-down mode, instruction execution stops, but the CPU module status is maintained, and the CPU waits for an interrupt request. If an interrupt is requested, the CPU exits the power-down mode and begins exception processing.

SLEEP is a privileged instruction and can be used in privileged mode only. If used in user mode, it causes an illegal instruction exception.

Note: The performance of SLEEP depends on STBCR(Standby Control Register) , see Section 9 “Power-Down Modes”, SH-4 Hardware Manual.

Operation:

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

Examples:

SLEEP Enters power-down mode

STC (Store Control Register): System Control Instruction

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

Description

These instructions store the contents of control register SR, GBR, VBR, SSR, SPC, SGR, DBR, or R0_BANK to R7_BANK into a destination register or memory. With the exception of STC GBR,Rn and STC.L GBR,@-Rn, the STC and STC.L instructions are privileged instructions and can only be used in privileged mode. Use in user mode will cause an illegal instruction exception. STC GBR,Rn and STC.L GBR,@-Rn are non-privileged instructions.

With STC/STC.L instructions accessing Rn_BANK, Rn_BANK0 is accessed when the SR.RB bit is 1, and Rn_BANK1 is accessed when this bit is 0.

Operation

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

STCGBR(int n)          /* STC GBR,Rn */
{
    R[n]=GBR;
    PC+=2;
}

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

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

```

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

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

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

STCRn_BANK(int m)  /* STC Rn_BANK,Rm : Privileged */
                    /* n=0-7 */
{
    R[m]=Rn_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;
}

STCMDDBR(int n)     /* STC.L DBR,@-Rn : Privileged */
{
    R[n]--=4;
    Write_Long(R[n],DBR);
    PC+=2;
}

```

```

STCMRn_BANK(int m) /* STC.L Rn_BANK,@-Rm : Privileged */
                    /* n=0-7 */
{
    R[m]-=4;
    Write_Long(R[m],Rn_BANK);
    PC+=2;
}

```

Exceptions

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

STS (Store from FPU System Register): System Control Instruction

No.	Format	Abstract	Code	T Bit
1	STS FPUL, Rn	FPUL \varnothing Rn	0000nnnn01011010	—
2	STS.L FPUL, @-Rn	Rn $- 4 \varnothing$ Rn, FPUL \varnothing (Rn)	0100nnnn01010010	—
3	STS FPSCR, Rn	FPSCR \rightarrow Rn	0000nnnn01101010	—
4	STS.L FPSCR, @-Rn	Rn $- 4 \varnothing$ Rn, FPSCR \varnothing (Rn)	0100nnnn01100010	—

Description:

1. Copies the content of system register FPUL to general purpose register Rm.
2. Stores the content of system register FPUL in the memory location addressed by general register Rn, decremented by 4. Upon completion, the decremented value becomes the value of Rn.
3. Copies the content of system register FPSCR to general purpose register Rm.
4. Stores the content of system register FPSCR in the memory location addressed by general register Rn, decremented by 4. Upon completion, the decremented value becomes the value of Rn.

Operation:

```

STS(int n)          /* STS FPUL,Rn */
{
    R[n]= FPUL;
    PC+=2;
}
STS_SAVE(int n)     /* STS.L FPUL,@-Rn */
{
    R[n]-=4;
    Write_Long(R[n],FPUL) ;
    PC+=2;
}
STS(int n)          /* STS FPSCR,Rn */
{
    R[n]=FPSCR&0x003FFFFFF;
    PC+=2;
}
STS_RESTORE(int n)  /* STS.L FPSCR,@-Rn */
{
    R[n]-=4;
    Write_Long(R[n],FPSCR&0x003FFFFFF)
    PC+=2;
}

```

Exceptions: Address Error.**Examples:**

• STS

Example 1:

```

MOV.L  #H'12ABCDEF, R12
LDS.L  @R12, FPUL
STS    FPUL, R13

```

```

; After executing the STS instruction:
; R13 = 12ABCDEF

```

Example 2:

```
STS    FPSCR, R2
```

- ; After executing the STS instruction:
- ; The current content of FPSCR is stored in register R2

- STS.L

Example 1:

```
MOV.L  #H'0C700148, R7
```

```
STS.L  FPUL, @-R7
```

- ; Before executing the STS.L instruction:
- ; R7 = 0C700148
- ; After executing the STS.L instruction:
- ; R7 = 0C700144, and the content of FPUL is saved at memory
- ; location 0C700144.

Example 2:

```
MOV.L  #H'0C700154, R8
```

```
STS.L  FPSCR, @-R8
```

- ; After executing the STS.L instruction:
- ; The content of FPSCR is saved at memory location 0C700150.

STS (Store System Register): System Control Instruction

Format		Abstract	Code	T Bit
STS	MACH,Rn	MACH \oslash Rn	0000nnnn00001010	—
STS	MACL,Rn	MACL \oslash Rn	0000nnnn00011010	—
STS	PR,Rn	PR \oslash Rn	0000nnnn00101010	—
STS.L	MACH,@-Rn	Rn $- 4 \oslash$ Rn, MACH \oslash (Rn)	0100nnnn00000010	—
STS.L	MACL,@-Rn	Rn $- 4 \oslash$ Rn, MACL \oslash (Rn)	0100nnnn00010010	—
STS.L	PR,@-Rn	Rn $- 4 \oslash$ Rn, PR \oslash (Rn)	0100nnnn00100010	—

Description: Stores system registers MACH, MACL and PR data into a specified destination.

Operation:

```

STSMACH(int n) /* STS MACH,Rn */
{
    R[n]=MACH;
    PC+=2;
}

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

STSPPR(int n) /* STS PR,Rn */
{
    R[n]=PR;
    PC+=2;
}

STSMACH(int n) /* STS.L MACH,@-Rn */
{
    R[n]-=4;
    Write_Long(R[n],MACH);
    PC+=2;
}

```



```

STSMACL(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;
}

```

Examples:

STS MACH,R0 Before execution R0 = H'FFFFFFFF, MACH = H'00000000

After execution R0 = H'00000000

STS.L PR,@-R15 Before execution R15 = H'10000004

After execution R15 = H'10000000, @R15 = PR

SUB (Subtract Binary): Arithmetic Instruction

Format	Abstract	Code	T Bit
SUB Rm,Rn	$Rn - Rm \rightarrow Rn$	0011nnnnmmmm1000	—

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

Operation:

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

Examples:

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

SUBC (Subtract with Carry): Arithmetic Instruction

Format	Abstract	Code	T Bit
SUBC Rm,Rn	$R_n - R_m - T \oslash R_n$, Borrow $\oslash T$	0011nnnnmmmm1010	Borrow

Description: Subtracts Rm data and the T bit value from general register Rn data, and stores the result in Rn. The T bit changes according to the result. This instruction is used for subtraction of data that has more than 32 bits.

Operation:

```

SUBC(int m,int n) /* SUBC Rm,Rn */
{
    unsigned int tmp0,tmp1;

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

```

Examples:

```

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

```

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

Format	Abstract	Code	T Bit
SUBV Rm,Rn	$R_n - R_m \oslash R_n$, Underflow \oslash T	0011nnnnmmmm1011	Underflow

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

Operation:

```

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

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

```

Examples:

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

SWAP (Swap): Data Transfer Instruction

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

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

Operation:

```

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

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

SWAPW(int m,int n) /* SWAP.W Rm,Rn */
{
    unsigned int temp;

    R[n]=(R[m]<<16)|((R[m]>>16)&0x0000FFFF);
    PC+=2;
}

```

Examples:

```

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

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

```

TAS (Test and Set): Logic Operation Instructions

Format	Operation	Instruction Code	T Bit
TAS.B @Rn	If (Rn) is 0, then $1 \oplus T$ else $0 \oplus T$; $1 \oplus \text{MSB of (Rn)}$.	0100nnnn00011011	Test Result

Description

To the memory area indicated by the contents of general register Rn, this instruction purges the corresponding cache block, reads the byte data, and if the data is 0 then Tbit = 1, else Tbit = 0. And set 1 to the bit 7 of the data and write to the same address. Between these operations, the bus is not released. In this case, the purge operation is executed like the following.

Purge operation accesses data 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 then the cache block is invalidated (V bit = 0). If there is a cache hit and the corresponding cache block is clean (U bit = 0), the cache block is simply invalidated (V bit = 0). In the event of a cache miss, or if the memory location to be accessed is non-cacheable, the purge results in no operation.

The two memory accesses of TAS.B are executed atomically. Any other memory access is not performed between the two access of TAS.B.

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;
}
```

Exceptions

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

Exceptions are examined taking a data access by means of this instruction as a byte store.

TRAPA (Trap Always): System Control Instruction

Format	Abstract	Code	T Bit
TRAPA #imm	imm \oslash TRA, PC + 2 \oslash SPC, SR \oslash SSR, 1 \oslash SR.MD/BL/RB 0x160 \oslash EXPEVT VBR + H'00000100 \oslash PC	11000011iiiiiii	—

Description: Starts the trap exception processing. The (PC + 2) and SR values are saved in SPC and SSR. Eight-bit immediate data is stored in the TRA registers (bits 9 to 2). The processor goes into privileged mode (SR.MD = 1) with SR.BL = 1 and SR.RB = 1, that is, blocking exceptions and masking interrupts, and selecting BANK1 registers (R0_BANK1 to R7_BANK1). Exception code 0x160 is stored in the EXPEVT register (bits 11 to 0). The program branches to an address (VBR+H'00000100).

Operation:

```
TRAPA(int i)    /* TRAPA #imm */
{
    int imm;
    imm=(0x000000FF & i);
    TRA=imm<<2;
    SSR=SR;
    SPC=PC+2;
    SR.MD=1
    SR.BL=1
    SR.RB=1
    EXPEVT=0x00000160;
    PC=VBR+H'00000100;
}
```

TST (Test Logical): Logic Operation Instruction

Format	Abstract	Code	T Bit
TST Rm,Rn	Rn & Rm, if result is 0, 1 \emptyset T, else 0 \emptyset T	0010nnnnnnmmml000	Test results
TST #imm,R0	R0 & imm, if result is 0, 1 \emptyset T, else 0 \emptyset T	11001000iiiiiii	Test results
TST.B #imm,@(R0,GBR)	(R0 + GBR) & imm, if result is 0, 1 \emptyset T, else 0 \emptyset T	11001100iiiiiii	Test results

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

Operation:

```

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

TSTI(int i) /* TEST #imm,R0   */
{
    if((R[0]&(0x000000FF & i))==0)
        T=1;
    else   T=0;
    PC+=2;
}

```



```

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

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

```

Examples:

TST R0,R0 Before execution R0 = H'00000000

After execution T = 1

TST #H'80,R0 Before execution R0 = H'FFFFFF7F

After execution T = 1

TST.B #H'A5,@(R0,GBR) Before execution @(R0,GBR) = H'A5

After execution T = 0

XOR (Exclusive OR Logical): Logic Operation Instruction

Format	Abstract	Code	T Bit
XOR Rm,Rn	$R_n \wedge R_m \oslash R_n$	0010nnnnmmmm1010	—
XOR #imm,R0	$R_0 \wedge \text{imm} \oslash R_0$	11001010iiiiiii	—
XOR.B #imm,@(R0,GBR)	$(R_0 + \text{GBR}) \wedge \text{imm} \oslash (R_0 + \text{GBR})$	11001110iiiiiii	—

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

Operation:

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

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

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

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

Examples:

`XOR R0,R1` Before execution `R0 = H'AAAAAAAA, R1 = H'55555555`
After execution `R1 = H'FFFFFFFF`

`XOR #H'F0,R0` Before execution `R0 = H'FFFFFFFF`
After execution `R0 = H'FFFFFF0F`

`XOR.B #H'A5,@(R0,GBR)` Before execution `@(R0,GBR) = H'A5`
After execution `@(R0,GBR) = H'00`

XTRCT (Extract): Data Transfer Instruction

Format	Abstract	Code	T Bit
XTRCT Rm,Rn	middle 32 bits of 64-bit data \varnothing Rn	0010nnnnmmmm1101	—

Description: Extracts the middle 32 bits of the 64-bit data of general registers Rm and Rn, and stores the 32 bits in Rn (figure 6.15).

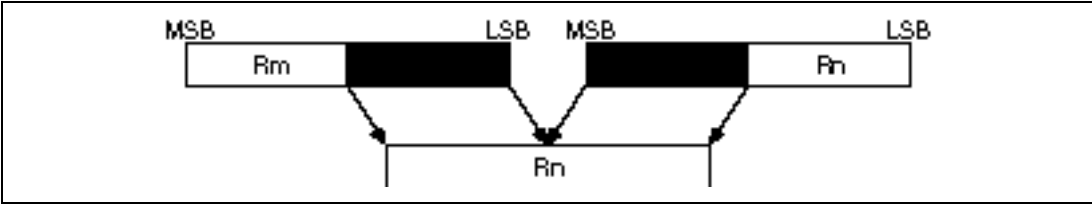


Figure 10.15 Extract

Operation:

```

XTRCT(int m,int n) /* XTRCT Rm,Rn */
{
    R[n]=((R[m]<<16)&0xFFFF0000 )|((R[n]>>16)&0x0000FFFF);
    PC+=2;
}

```

Example:

XTRCT	R0,R1	Before execution	R0 = H'01234567, R1 = H'89ABCDEF
		After execution	R1 = H'456789AB