



PRELIMINARY DATA

**HITACHI**

# SH-4 CPU Core Architecture

Last updated 12 September 2002 2:29

—HITACHI—



*STMicroelectronics and Hitachi, Ltd.*

ADCS 7182230F

SH-4 CPU Core Architecture

Issued by the MCDT Documentation Group on behalf of STMicroelectronics

Information furnished is believed to be accurate and reliable. However, STMicroelectronics assumes no responsibility for the consequences of use of such information nor for any infringement of patents or other rights of third parties which may result from its use. No license is granted by implication or otherwise under any patent or patent rights of STMicroelectronics. Specifications mentioned in this publication are subject to change without notice. This publication supersedes and replaces all information previously supplied. STMicroelectronics products are not authorized for use as critical components in life support devices or systems without the express written approval of STMicroelectronics.

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.

The ST logo is a registered trademark of STMicroelectronics.

SuperH is a registered trademark for products originally developed by Hitachi, Ltd. and is owned by Hitachi Ltd.

© 2000, 2001, 2002 STMicroelectronics and Hitachi, Ltd. All Rights Reserved.

STMicroelectronics Group of Companies

Australia - Brazil - Canada - China - Finland - France - Germany - Hong Kong - India - Israel - Italy - Japan  
Malaysia - Malta - Morocco - Singapore - Spain - Sweden - Switzerland - United Kingdom - U.S.A.

<http://www.st.com>



**HITACHI**

*STMicroelectronics and Hitachi, Ltd.*



# Contents

<b>Preface</b>	<b>xi</b>
<b>1 Overview</b>	<b>15</b>
1.1 SH-4 CPU core features	15
1.2 Block diagram	19
<b>2 Programming model</b>	<b>21</b>
2.1 General registers	22
2.2 System registers	25
2.3 Control registers	31
2.4 Floating-point registers	34
2.5 Memory-mapped registers	36
2.6 Data format in registers	37
2.7 Data formats in memory	37
2.8 Processor states	38
2.8.1 Reset state	38
2.8.2 Exception-handling state	38
2.8.3 Program execution state	38
2.8.4 Power-down state	39
2.9 Processor modes	40

<b>3</b>	<b>Memory management unit (MMU)</b>	<b>41</b>
3.1	Overview	41
3.2	Role of the MMU	41
3.3	Register descriptions	42
3.3.1	Page table entry high register (PTEH)	43
3.3.2	Page table entry low register (PTEL)	44
3.3.3	Translation table base register (TTB)	47
3.3.4	TLB exception address register (TEA)	47
3.3.5	MMU control register (MMUCR)	47
3.4	Address space	51
3.4.1	Physical address space	51
3.4.2	External memory space	52
3.4.3	Virtual address space	55
3.4.4	On-chip RAM space	56
3.4.5	Address translation	57
3.4.6	Single virtual memory mode and multiple virtual memory mode	57
3.4.7	Address space identifier (ASID)	58
3.5	TLB functions	58
3.5.1	Unified TLB (UTLB) configuration	58
3.5.2	Instruction TLB (ITLB) configuration	59
3.5.3	Address translation method	59
3.6	MMU functions	62
3.6.1	MMU hardware management	62
3.6.2	MMU software management	62
3.6.3	MMU instruction (LDTLB)	63
3.6.4	Hardware ITLB miss handling	64
3.6.5	Avoiding synonym problems	64
3.7	Handling MMU exceptions	65
3.7.1	ITLBMULTIHIT	65
3.7.2	ITLBMISS	65
3.7.3	EXECPROT	66

3.7.4	OTLBMULTIHIT	67
3.7.5	TLBMISS	67
3.7.6	READPROT	68
3.7.7	FIRSTWRITE	68
3.8	Memory-mapped TLB configuration	69
3.8.1	ITLB address array	70
3.8.2	ITLB data array 1	71
3.8.3	UTLB address array	72
3.8.4	UTLB data array 1	74
<b>4</b>	<b>Caches</b>	<b>75</b>
4.1	Overview	75
4.1.1	Features	75
4.2	Register descriptions	77
4.2.1	Cache control register (CCR)	77
4.2.2	Queue address control register 0 (QACR0)	80
4.2.3	Queue address control register 1 (QACR1)	81
4.3	Operand cache (OC)	82
4.3.1	Configuration	82
4.3.2	Read operation	84
4.3.3	Write operation	86
4.3.4	Write-back buffer	88
4.3.5	Write-through buffer	88
4.3.6	RAM mode	88
4.3.7	OC index mode	91
4.3.8	Coherency between cache and external memory	91
4.3.9	Prefetch operation	91
4.4	Instruction cache (IC)	92
4.4.1	Configuration	92
4.4.2	Read operation	94
4.4.3	IC index mode	94

4.5	Memory-mapped cache configuration	95
4.5.1	IC address array	95
4.5.4	IC data array	97
4.5.5	OC address array	98
4.5.6	OC data array	99
4.6	Store queues	101
4.6.1	SQ configuration	101
4.6.2	SQ writes	102
4.6.3	SQ reads (implementation dependant)	102
4.6.4	Transfer to external memory	102
<b>5</b>	<b>Exceptions</b>	<b>105</b>
5.1	Overview	105
5.2	Register descriptions	105
5.2.1	Exception event register (EXPEVT)	106
5.2.2	Interrupt event register (INTEVT)	106
5.2.3	TRAPA exception register (TRA)	107
5.3	Exception handling functions	108
5.3.1	Exception handling flow	108
5.3.2	Exception handling vector addresses	108
5.4	Exception types and priorities	109
5.5	Exception flow	110
5.5.1	Exception flow	110
5.5.2	Exception source acceptance	112
5.5.3	Exception requests and BL bit	114
5.5.4	Return from exception handling	114
5.6	Description of exceptions	115
5.6.1	Resets	115
5.6.2	General exceptions	120
5.6.3	Interrupts	138
5.6.4	Priority order with multiple exceptions	141
5.7	Usage notes	142

<b>6</b>	<b>Floating-point unit</b>	<b>145</b>
6.1	Overview	145
6.2	Floating-point format	146
6.2.1	Non-numbers (NaN)	148
6.2.2	Denormalized numbers	149
6.3	Rounding	149
6.4	Floating-point exceptions	150
6.5	Graphics support functions	152
6.5.1	Geometric operation instructions	152
6.5.2	Pair single-precision data transfer	154
<b>7</b>	<b>Instruction set</b>	<b>155</b>
7.1	Execution environment	155
7.2	Addressing modes	158
7.3	Instruction set summary	163
<b>8</b>	<b>Instruction specification</b>	<b>179</b>
8.1	Overview	179
8.2	Variables and types	180
8.2.1	Integer	180
8.2.2	Boolean	181
8.2.3	Bit-fields	181
8.2.4	Arrays	181
8.2.5	Floating point values	182
8.3	Expressions	182
8.3.1	Integer arithmetic operators	182
8.3.2	Integer shift operators	184
8.3.3	Integer bitwise operators	184
8.3.4	Relational operators	186
8.3.5	Boolean operators	186
8.3.6	Single-value functions	187

<b>8.4</b>	<b>Statements</b>	<b>190</b>
8.4.1	Undefined behavior	190
8.4.2	Assignment	190
8.4.3	Conditional	192
8.4.4	Repetition	192
8.4.5	Exceptions	193
8.4.6	Procedures	193
<b>8.5</b>	<b>Architectural state</b>	<b>194</b>
<b>8.6</b>	<b>Memory model</b>	<b>196</b>
8.6.1	Support functions	197
8.6.2	Reading memory	198
8.6.3	Prefetching memory	200
8.6.4	Writing memory	200
<b>8.7</b>	<b>Cache model</b>	<b>202</b>
<b>8.8</b>	<b>Floating-point model</b>	<b>202</b>
8.8.1	Functions to access SR and FPSCR	202
8.8.2	Functions to model floating-point behavior	204
8.8.3	Floating-point special cases and exceptions	206
<b>8.9</b>	<b>Abstract sequential model</b>	<b>206</b>
8.9.1	Initial conditions	207
8.9.2	Instruction execution loop	207
8.9.3	State changes	208
<b>8.10</b>	<b>Example instructions</b>	<b>209</b>
8.10.1	ADD #imm, Rn	209
8.10.2	FADD FRm, FRn	211
<b>9</b>	<b>Instruction descriptions</b>	<b>213</b>
9.1	Alphabetical list of instructions	213



<b>10</b>	<b>Pipelining</b>	<b>483</b>
10.1	Pipelines	483
10.2	Parallel executables	490
10.3	Execution cycles and pipeline stalling	494
<b>A</b>	<b>Address list</b>	<b>513</b>
<b>B</b>	<b>Instruction prefetch side effects</b>	<b>515</b>
	<b>Index</b>	<b>517</b>

# PRELIMINARY DATA

x

---



*STMicroelectronics and Hitachi, Ltd.*

SH-4 CPU Core Architecture

ADCS 7182230F



# Preface

This document is part of the SuperH Documentation Suite detailed below. Comments on this or other manuals in the SuperH Documentation Suite should be made by contacting your local STMicroelectronics Limited Sales Office or distributor.

## Document identification and control

Each book carries a unique identifier in the form:

ADCS nnnnnnnx

**Where,** nnnnnnn is the document number and x is the revision.

Whenever making comments on a document the complete identification ADCS nnnnnnnx should be quoted.

### ST40 Micro Toolset Getting Started

*ADCS 7379953.* This manual provides an introduction to the ST40 Micro Toolset and instructions for getting a simple OS21 application run on an STMicroelectronics' MediaRef platform. It also describes how to boot OS21 applications from ROM and how to port applications which use STMicroelectronics' STLite/OS20 operating systems to OS21.

### OS21 User's Manual

*ADCS 7358306.* This manual describes the generic use of OS21 across supported platforms. It describes all the core features of OS21 and their use and details the OS21 function definitions. It also explains how OS21 differs to STLite/OS20, the API targeted at ST20.

### OS21 for ST40 User Manual

*ADCS 7358673.* This manual describes the use of OS21 on ST40 platforms. It describes how specific ST40 facilities are exploited by the OS21 API. It also describes the OS21 board support packages for ST40 platforms.

### 32-Bit RISC Series, SH-4 CPU Core Architecture

*ADCS 7182230.* This manual describes the architecture and instruction set of the SH4-1xx (previously known a ST40-C200) core as used by STMicroelectronics.

### 32-Bit RISC Series, SH-4, ST40 System Architecture

This manual describes the ST40 family system architecture. It is split into four volumes:

ST40 System Architecture - Volume 1 System - *ADCS 7153464.*

ST40 System Architecture - Volume 2 Bus Interfaces - *ADCS 7171720.*

ST40 System Architecture - Volume 3 Video Devices - *ADCS 7225754.*

ST40 System Architecture - Volume 4 I/O Devices - *ADCS 7225754.*

## Conventions used in this guide

### General notation

The notation in this document uses the following conventions:

- **Sample code**, keyboard input and file names,
- *Variables* and *code variables*,
- Equations and math,
- **Screens, windows** and **dialog boxes**,
- **Instructions.**

### Hardware notation

The following conventions are used for hardware notation:

- REGISTER NAMES and FIELD NAMES,

- PIN NAMES and SIGNAL NAMES.

### Software notation

Syntax definitions are presented in a modified Backus-Naur Form (BNF). Briefly:

- 1 Terminal strings of the language, that is those not built up by rules of the language, are printed in teletype font. For example, `void`.
- 2 Nonterminal strings of the language, that is those built up by rules of the language, are printed in italic teletype font. For example, *name*.
- 3 If a nonterminal string of the language starts with a nonitalicized part, it is equivalent to the same nonterminal string without that nonitalicized part. For example, `vspace-name`.
- 4 Each phrase definition is built up using a double colon and an equals sign to separate the two sides.
- 5 Alternatives are separated by vertical bars ('|').
- 6 Optional sequences are enclosed in square brackets ('[' and ']').
- 7 Items which may be repeated appear in braces ('{' and '}').





# Overview

## 1.1 SH-4 CPU core features<sup>1</sup>

This manual describes the architecture of the SH-4 CPU core. The core is a highly encapsulated design component that can be integrated into any product, you will therefore find no references to clock speeds, system facilities, pin-outs or similar data in this manual. For this information you are referred to the *Datasheet* and/or *System Architecture Manual* of the appropriate product.

The SH-4 is a 32-bit RISC (reduced instruction set computer) microprocessor, featuring object code upward-compatibility with Hitachi SuperH SH-1, SH-2, SH-3, and SH-3E microcomputers. It includes an instruction cache, a operand cache that can be switched between copy-back and write-through modes, a 4-entry full-associative instruction TLB (translation look aside buffer), and MMU (memory management unit) with 64-entry full-associative shared TLB.

The SH-4's 16-bit fixed-length instruction set enables program code size to be reduced by almost 50% compared with 32-bit instructions.

The SH-4 200 series includes an enhanced mode which enables 2-way set associative instruction and operand cache (rather than direct mapped as for the SH-4 100 series and SH-4 200 series when running in default compatibility mode). In particular, the SH4-202 has a 32 Kbyte 2-way operand cache and a 16 Kbyte 2-way instruction cache. On power up this behaves as a 16Kbyte direct mapped operand cache and an 8Kbyte direct mapped instruction cache.

1. Naming conventions:

SH-4: for non-variant specific information

SH-4 100/200 series: for series specific features

SH4-103/202: for variant specific features

The features of the SH-4 CPU core are summarized as follows:

### CPU

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

### FPU

- On-chip floating-point coprocessor
- Supports single-precision (32 bits) and double-precision (64 bits)
- Supports IEEE754-compliant data types and exceptions
- Two rounding modes: Round to Nearest and Round to Zero
- Handling of denormalized numbers: Truncation to zero or interrupt generation for compliance with IEEE754
- Floating-point registers:
  - 2 banks of sixteen 32-bit single precision registers or,
  - 2 banks of eight 64-bit double precision registers or,
  - 2 banks of four 128-bit vector registers (each vector is 4 single precision elements)



- 32-bit CPU-FPU floating-point communication register (FPUL)
- Supports FMAC (multiply-and-accumulate) instruction
- Supports FDIV (divide) and FSQRT (square root) instructions
- Supports FLDI0/FLDI1 (load constant 0/1) instructions
- Instruction execution times
  - Latency (FMAC/FADD/FSUB/FMUL): 3 cycles (single-precision), 8 cycles (double-precision)
  - Pitch (FMAC/FADD/FSUB/FMUL): 1 cycle (single-precision), 6 cycles (double-precision)
  - Note: FMAC is supported for single-precision only.
- 3-D graphics instructions (single-precision only):
  - 4-dimensional vector conversion and matrix operations (FTRV): 4 cycles (pitch), 7 cycles (latency)
  - 4-dimensional vector (FIPR) inner product: 1 cycle (pitch), 4 cycles (latency)
- Five-stage pipeline

#### Power-down

- Power-down modes
  - Sleep mode
  - Standby mode
  - Module standby function

#### MMU

- 4-Gbyte address space, 256 address space identifiers (8-bit ASIDs)
- Single virtual mode and multiple virtual memory mode
- Supports multiple page sizes: 1 kbyte, 4 kbytes, 64 kbytes, 1 Mbyte
- 4-entry fully-associative TLB for instructions
- 64-entry fully-associative TLB for instructions and operands
- Supports software-controlled replacement and random-counter replacement algorithm
- TLB contents can be accessed directly by address mapping

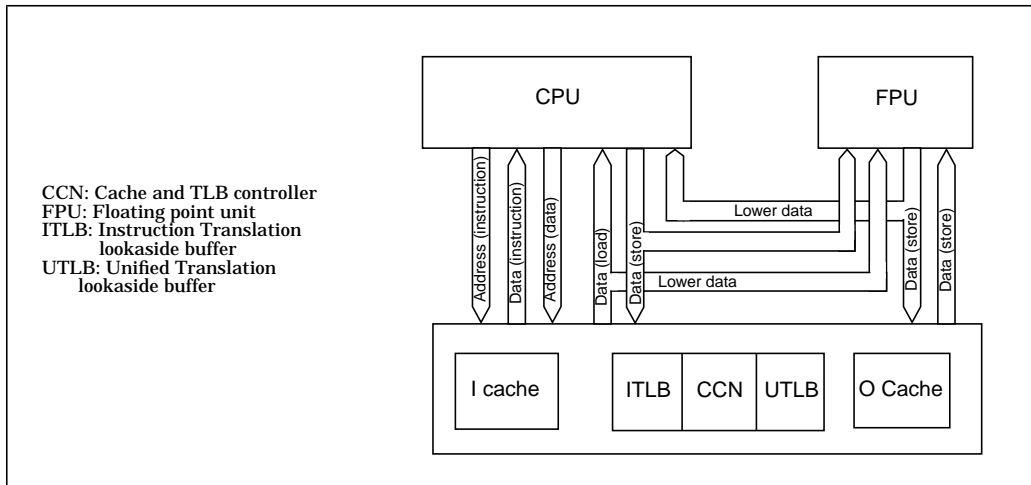
## Cache memory

- SH4-103**
- Instruction cache (IC)
    - 8 kbytes, direct mapping
    - 256 entries, 32-byte block length
    - Normal mode (8-Kbyte cache)
    - Index mode
  - Operand cache (OC)
    - 16 kbytes, direct mapping
    - 512 entries, 32-byte block length
    - Normal mode (16-kbyte cache)
    - Index mode
    - RAM mode (8-kbyte cache + 8-kbyte RAM)
    - Choice of write method (copy-back or write-through)
  - Single-stage copy-back buffer, single-stage write-through buffer
  - Cache memory contents can be accessed directly by address mapping (usable as on-chip memory)
  - Store queue (32 bytes x 2 entries)
- SH4-202**
- Instruction cache (IC):
    - 16 Kbyte, 2-way set associative
    - 512 entries, 32-bytes block length
    - Compatibility mode (8 Kbyte direct mapped)
    - Index mode<sup>a</sup>
  - - Operand cache (OC)
    - 32 Kbyte, 2-way set associative
    - 1024 entries, 32 bytes block length
    - Compatibility mode (16 Kbyte direct mapped)
    - Index mode
    - RAM mode (16 Kbyte cache + 16 Kbyte RAM)
  - Single-stage copy-back buffer, single-stage write-through buffer
  - Cache memory contents can be accessed directly by address mapping (usable as on-chip memory)
  - Store queue (32 bytes x 2 entries)

a. Index mode (IC and OC) is only supported when in SH4-1xx compatibility mode.

## 1.2 Block diagram

*Figure 1* shows an internal block diagram of the SH-4 32-Bit CPU Core .



**Figure 1 SH-4 32-Bit CPU core**





## 2

# Programming model

The SH-4 CPU core has two processor modes, user mode and privileged mode. The SH-4 normally operates in user mode, and switches to privileged mode when an exception occurs, or an interrupt is accepted.

There are four kinds of registers:

- general registers

There are 16 general registers, R0 to R15. General registers R0 to R7 are banked registers which are switched by a processor mode change.

- system registers

Access to these registers does not depend on the processor mode.

- control registers
- floating-point registers

There are thirty-two floating-point registers, FR0–FR15 and XF0–XF15. FR0–FR15 and XF0–XF15 can be assigned to either of two banks (FPR0\_BANK0–FPR15\_BANK0 or FPR0\_BANK1–FPR15\_BANK1).

The registers that can be accessed differ in the two processor modes.

Register values after a reset are shown in *Table 1*.

Type	Registers	Initial value <sup>a</sup>
General registers	R0_BANK0–R7_BANK0, R0_BANK1–R7_BANK1, R8–R15	Undefined
Control registers	SR	MD bit = 1, RB bit = 1, BL bit = 1, FD bit = 0, I3–I0 = 1111 (0xF), reserved bits = 0, others undefined
	GBR, SSR, SPC, SGR, DBR	Undefined
	VBR	0x00000000
System registers	MACH, MACL, PR, FPUL	Undefined
	PC	0xA0000000
	FPSCR	0x00040001
Floating-point registers	FR0–FR15, XF0–XF15	Undefined

**Table 1: Initial register values**

a. Initialized by a power-on reset and manual reset

## 2.1 General registers

*Figure 2* shows the relationship between the processor modes and the general registers. The SH-4 CPU core 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 either processor mode. The assignment of R0–R7, in both modes, is shown below.

- R0\_BANK0–R7\_BANK0

In user mode (SR.MD = 0), R0–R7 are always assigned to R0\_BANK0–R7\_BANK0.

In privileged mode (SR.MD = 1), R0–R7 are assigned to R0\_BANK0–R7\_BANK0 only when SR.RB = 0.

- R0\_BANK1–R7\_BANK1

In user mode, R0\_BANK1–R7\_BANK1 cannot be accessed.

In privileged mode, R0–R7 are assigned to R0\_BANK1–R7\_BANK1 only when SR.RB = 1.

SR.MD = 0 or (SR.MD = 1, SR.RB = 0)			(SR.MD = 1, SR.RB = 1)		
R0	R0_BANK0		R0_BANK0		R0_BANK0
R1	R1_BANK0		R1_BANK0		R1_BANK0
R2	R2_BANK0		R2_BANK0		R2_BANK0
R3	R3_BANK0		R3_BANK0		R3_BANK0
R4	R4_BANK0		R4_BANK0		R4_BANK0
R5	R5_BANK0		R5_BANK0		R5_BANK0
R6	R6_BANK0		R6_BANK0		R6_BANK0
R7	R7_BANK0		R7_BANK0		R7_BANK0
R0_BANK1	R0_BANK1		R0		R0
R1_BANK1	R1_BANK1		R1		R1
R2_BANK1	R2_BANK1		R2		R2
R3_BANK1	R3_BANK1		R3		R3
R4_BANK1	R4_BANK1		R4		R4
R5_BANK1	R5_BANK1		R5		R5
R6_BANK1	R6_BANK1		R6		R6
R7_BANK1	R7_BANK1		R7		R7
R8	R8		R8		R8
R9	R9		R9		R9
R10	R10		R10		R10
R11	R11		R11		R11
R12	R12		R12		R12
R13	R13		R13		R13
R14	R14		R14		R14
R15	R15		R15		R15

Figure 2: General registers

**Programming Note:**

As the user's R0–R7 are assigned to R0\_BANK0–R7\_BANK0, and after an exception or interrupt R0–R7 are assigned to R0\_BANK1–R7\_BANK1, it is not necessary for the interrupt handler to save and restore the user's R0–R7 (R0\_BANK0–R7\_BANK0).

After a reset, the values of R0\_BANK0–R7\_BANK0, R0\_BANK1–R7\_BANK1, and R8–R15 are undefined.



## 2.2 System registers

Name	Size	Initial value	Synopsis
MACH	32	Undefined	Multiply-and-accumulate register high
	Operation		MACH is used for the added value in a MAC instruction, and to store a MAC instruction or MUL instruction operation result.
MACL	32	Undefined	Multiply-and-accumulate register low
	Operation		MACL is used for the added value in a MAC instruction, and to store a MAC instruction or MUL instruction operation result.
PR	32	Undefined	Procedure register
	Operation		The return address is stored when a subroutine call using a BSR, BSRF or JSR instruction. PR is referenced by the subroutine return instruction (RTS).
PC	32	0xA000 0000	Program counter
	Operation		PC indicates the executing instruction address.
FPSCR	32	0x0004 0001	Floating-point status/control register
	Operation		Refer to <a href="#">Table 3: FPSCR register description</a>
FPUL	32	undefined	Floating-point communication register
	Operation		<p>Data transfer between FPU registers and CPU registers is carried out via the FPUL register. The FPUL register is a system register, and is accessed from the CPU side by means of LDS and STS instructions. For example, to convert the integer stored in general register R1 to a single-precision floating-point number, the processing flow is as follows:</p> <p>R1 → (LDS instruction) → FPUL → (single-precision FLOAT instruction) → FR1</p>

**Table 2: System registers**

FPSCR				
Field	Bits	Size	Synopsis	Type
RM	[0,1]	2	Rounding mode.	RW
	Operation		RM = 00: Round to Nearest. RM = 01: Round to Zero. RM = 10: Reserved. RM = 11: Reserved. For details see <a href="#">Section 6.3: Rounding</a>	
	Power-on reset		1	
Flag inexact	2	1	FPU inexact exception flag.	RW
	Operation		Set to 1 if Inexact exception occurs.	
	Power-on reset		0	
Flag underflow	3	1	FPU underflow exception flag.	RW
	Operation		Set to 1 if Underflow exception occurs	
	Power-on reset		0	
Flag overflow	4	1	FPU overflow exception flag.	RW
	Operation		Set to 1 if overflow exception occurs	
	Power-on reset		0	
Flag division by zero	5	1	FPU division by zero exception flag.	RW
	Operation		Set to 1 if division by zero exception occurs	
	Power-on reset		0	
Flag invalid operation	6	1	FPU invalid operation exception flag.	RW
	Operation		Set to 1 if Invalid operation exception occurs	
	Power-on reset		0	

Table 3: FPSCR register description

FPSCR				
Field	Bits	Size	Synopsis	Type
Enable inexact	7	1	FPU invalid exception enable field.	RW
	Operation		Set to 1 to cause a trap when an inexact exception occurs.	
	Power-on reset		0	
Enable underflow	8	1	FPU underflow exception enable field.	RW
	Operation		Set to 1 to cause a trap when an underflow exception occurs.	
	Power-on reset		0	
Enable overflow	9	1	FPU overflow exception enable field.	RW
	Operation		Set to 1 to cause a trap when an overflow exception occurs.	
	Power-on reset		0	
Enable division by zero	10	1	FPU division by zero exception enable field.	RW
	Operation		Set to 1 to cause a trap when a division by zero exception occurs.	
	Power-on reset		0	
Enable invalid	11	1	FPU invalid exception enable field.	RW
	Operation		Set to 1 to cause a trap when an Invalid exception occurs.	
	Power-on reset		0	
Cause inexact	12	1	FPU inexact exception cause field.	RW
	Operation		Set to 0 before an FPU instruction is executed. Set to 1 if an Inexact exception occurs.	
	Power-on reset		0	

Table 3: FPSCR register description

FPSCR				
Field	Bits	Size	Synopsis	Type
Cause underflow	13	1	FPU underflow exception cause field.	RW
	Operation		Set to 0 before an FPU instruction is executed. Set to 1 if an underflow exception occurs.	
	Power-on reset		0	
Cause overflow	14	1	FPU overflow exception cause field.	RW
	Operation		Set to 0 before an FPU instruction is executed. Set to 1 if an overflow exception occurs.	
	Power-on reset		0	
Cause division by zero	15	1	FPU division by zero exception cause field.	RW
	Operation		Set to 0 before an FPU instruction is executed. Set to 1 if a division by zero exception occurs.	
	Power-on reset		0	
Cause invalid	16	1	FPU invalid exception cause field.	RW
	Operation		Set to 0 before an FPU instruction is executed. Set to 1 if an invalid exception occurs.	
	Power-on reset		0	
Cause FPU error	17	1	FPU error exception cause field.	RW
	Operation		Set to 0 before an FPU instruction is executed. Set to 1 if an FPU error exception occurs.	
	Power-on reset		0	
DN	18	1	Denormalization mode.	RW
	Operation		DN = 0: A denormalizing number is treated as such. DN = 1: A denormalized number is treated as zero.	
	Power-on reset		0	

Table 3: FPSCR register description

FPSCR				
Field	Bits	Size	Synopsis	Type
PR	19	1	Precision mode.	RW
	Operation		PR = 0: Floating point instructions are executed as single precision operations. PR = 1: Floating point instructions are executed as double-precision operations (the result of instructions for which double-precision is not supported is undefined). Mode setting [SZ = 1, PR = 1] is reserved. FPU operation results are undefined in this mode.	
	Power-on reset		1	
SZ	20	1	Transfer size mode.	RW
	Operation		SZ = 0: The data size of the FMOV instruction is 32 bits. SZ = 1: The data size of the FMOV instruction is a 32-bit register pair (64 bits).  Programming note: When SZ = 1 and big endian mode is selected, FMOV can be used for double-precision floating-point data load or store operations. In little endian mode, two 32-bit data size moves must be executed, with SZ = 0, to load or store a double-precision floating-point number.	
	Power-on reset		0	
FR	21	1	Floating-point register bank.	RW
	Operation		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_BANK1 are assigned to FR0-FR15.	
	Power-on reset		0	

Table 3: FPSCR register description

FPSCR				
Field	Bits	Size	Synopsis	Type
RES	[22,31]	10	Bits reserved	RW
	Power-on reset		Undefined	

Table 3: FPSCR register description

## 2.3 Control registers

Name	Size	Initial value	Privilege protection	Synopsis
SR	32	See <a href="#">Table 5</a> for individual bits.	Yes	Status register
	Operation		Refer to <a href="#">Table 5: SR register description</a>	
SSR	32	Undefined	Yes	Saved status register
	Operation		The current contents of SR are saved to SSR in the event of an exception or interrupt.	
SPC	32	Undefined	Yes	Saved program counter
	Operation		The address of an instruction at which an interrupt or exception occurs is saved to SPC.	
GBR	32	Undefined	No	Global base register
	Operation		GBR is referenced as the base address in a GBR-referencing MOV instruction.	
VBR	32	0x0000 0000	Yes	Vector base register
	Operation		VBR is referenced as the branch destination base address in the event of an exception or interrupt. For details, see <a href="#">Chapter 5: Exceptions</a> .	
SGR	32	Undefined	Yes	Saved general register
	Operation		The contents of R15 are saved to SGR in the event of an exception or interrupt.	
DBR	32	undefined	Yes	Debug base register
	Operation		When the user break debug function is enabled (BR CR.UBDE = 1), DBR is referenced as the user break handler branch destination address instead of VBR.	

**Table 4: Control registers**

SR				
Field	Bits	Size	Synopsis	Type
T	0	1	True/False condition or carry/borrow bit.	RW
	Operation		Refer to individual instruction descriptions, which affect the T bit.	
	Power-on reset		Undefined	
S	1	1	Specifies a saturation operation for a MAC instruction.	RW
	Operation		Refer to individual instruction descriptions, which affect the S bit.	
	Power-on reset		Undefined	
IMASK	[4,7]	4	Interrupt mask level.	RW
	Operation		External interrupts of a lower level than IMASK are masked.	
	Power-on reset		1	
Q	8	1	State for divide step.	RW
	Operation		Used by the DIV0S, DIV0U and DIV1 instructions.	
	Power-on reset		Undefined	
M	9	1	State for divide step.	RW
	Operation		Used by the DIV0S, DIV0U and DIV1 instructions.	
	Power-on reset		Undefined	
FD	15	1	FPU disable bit (cleared to 0 by a reset).	RW
	Operation		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.  For further details see FPUDIS description in section <a href="#">Section 6.4: Floating-point exceptions</a>	
	Power-on reset		0	

Table 5: SR register description



SR				
Field	Bits	Size	Synopsis	Type
BL	28	1	Exception/interrupt block bit (set to 1 by a reset, exception, or interrupt).	RW
	Operation		BL = 1: Interrupt requests are masked. If a general exception, other than a user break occurs while BL = 1, the processor switches to the reset state.	
	Power-on reset		1	
RB	29	1	General register bank specifier in privileged mode (set to 1 by a reset, exception or interrupt).	RW
	Operation		RB = 0: R0_BANK0-R7_BANK0 are accessed as general registers R0-R7. (R0_BANK1-R7_BANK1 can be accessed using LDC/STC R0_BANK-R7_BANK instructions.)  RB = 1: R0_BANK1-R7_BANK1 are accessed as general registers R0-R7. (R0_BANK0-R7_BANK0 can be accessed using LDC/STC R0_BANK-R7_BANK instructions.)	
	Power-on reset		1	
MD	30	1	Processor mode.	RW
	Operation		MD = 0: User mode (Some instructions cannot be executed, and some resources cannot be accessed).  MD = 1: Privileged mode.	
	Power-on reset		1	
RES	[2,3], [10,14][ 16,27] 31	20	Bits reserved	RW
	Power-on reset		Undefined	

Table 5: SR register description

## 2.4 Floating-point registers

*Figure 3* shows the floating-point registers. There are thirty-two 32-bit floating-point registers, divided into two banks (FPR0\_BANK0–FPR15\_BANK0 and FPR0\_BANK1–FPR15\_BANK1). These 32 registers are referenced as FR0–FR15, DR0/2/4/6/8/10/12/14, FV0/4/8/12, XF0–XF15, XD0/2/4/6/8/10/12/14, or XMTRX. The correspondence between FPRn\_BANKi and the reference name is determined by the FR bit in FPSCR.

- Floating-point registers, FPRn\_BANKi (32 registers)
- Single-precision floating-point registers, FRi (16 registers)  
 FPSCR.FR = 0 : FR0–FR15 are assigned to FPR0\_BANK0–FPR15\_BANK0.  
 FPSCR.FR = 1 : FR0–FR15 are assigned to FPR0\_BANK1–FPR15\_BANK1.
- Double-precision floating-point registers or single-precision floating-point register pairs, DRi (8 registers): A DR register comprises two FR registers.  
 DR0 = {FR0, FR1}, DR2 = {FR2, FR3}, DR4 = {FR4, FR5}, DR6 = {FR6, FR7},  
 DR8 = {FR8, FR9}, DR10 = {FR10, FR11}, DR12 = {FR12, FR13},  
 DR14 = {FR14, FR15}
- Single-precision floating-point vector registers, FVi (4 registers): An FV register comprises four FR registers  
 FV0 = {FR0, FR1, FR2, FR3}, FV4 = {FR4, FR5, FR6, FR7},  
 FV8 = {FR8, FR9, FR10, FR11}, FV12 = {FR12, FR13, FR14, FR15}
- Single-precision floating-point extended registers, XFi (16 registers)  
 FPSCR.FR = 0 : XF0–XF15 are assigned to FPR0\_BANK1–FPR15\_BANK1.  
 FPSCR.FR = 1 : XF0–XF15 are assigned to FPR0\_BANK0–FPR15\_BANK0.
- Single-precision floating-point extended register pairs, XD<sub>i</sub> (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_BANK0	XF0	XD0	XMTRX
		FR1	FPR1_BANK0	XF1		
	DR2	FR2	FPR2_BANK0	XF2	XD2	
FV4	DR4	FR3	FPR3_BANK0	XF3		
		FR4	FPR4_BANK0	XF4	XD4	
		FR5	FPR5_BANK0	XF5		
FV8	DR6	FR6	FPR6_BANK0	XF6	XD6	
		FR7	FPR7_BANK0	XF7		
	DR8	FR8	FPR8_BANK0	XF8	XD8	
FV12	DR10	FR9	FPR9_BANK0	XF9		
		FR10	FPR10_BANK0	XF10	XD10	
		FR11	FPR11_BANK0	XF11		
	DR12	FR12	FPR12_BANK0	XF12	XD12	
		FR13	FPR13_BANK0	XF13		
	DR14	FR14	FPR14_BANK0	XF14	XD14	
		FR15	FPR15_BANK0	XF15		
XMTRX	XD0	XF0	FPR0_BANK1	FR0	DR0	FV0
		XF1	FPR1_BANK1	FR1		
	XD2	XF2	FPR2_BANK1	FR2	DR2	
		XF3	FPR3_BANK1	FR3		
		XF4	FPR4_BANK1	FR4	DR4	FV4
		XF5	FPR5_BANK1	FR5		
	XD6	XF6	FPR6_BANK1	FR6	DR6	
		XF7	FPR7_BANK1	FR7		
	XD8	XF8	FPR8_BANK1	FR8	DR8	FV8
		XF9	FPR9_BANK1	FR9		
		XF10	FPR10_BANK1	FR10	DR10	
		XF11	FPR11_BANK1	FR11		
	XD12	XF12	FPR12_BANK1	FR12	DR12	FV12
		XF13	FPR13_BANK1	FR13		
	XD14	XF14	FPR14_BANK1	FR14	DR14	
		XF15	FPR15_BANK1	FR15		

Figure 3: Floating-point registers

**Programming Note:**

After a reset, the values of FPR0\_BANK0–FPR15\_BANK0 and FPR0\_BANK1–FPR15\_BANK1 are undefined.

## 2.5 Memory-mapped registers

Appendix A summarizes how the control registers are mapped in to the address space. The control registers are double-mapped to the following two memory areas. All registers have two addresses.

0x1F00 0000–0x1FFF FFFF

0xFF00 0000–0xFFFF FFFF

These two areas are used as follows.

- 0x1F00 0000–0x1FFF FFFF

This area must be accessed in address translation mode using the TLB. Since external memory area is defined as a 29-bit address space in the SH-4 CPU core architecture, the TLB's physical page numbers do not cover a 32-bit address space. In address translation, the page numbers of this area can be set in the corresponding field of the TLB by accessing a memory-mapped register. The page numbers of this area should be used as the actual page numbers set in the TLB. When address translation is not performed, the operation of accesses to this area is undefined.

- 0xFF00 0000–0xFFFF FFFF

Access to area 0xFF00 0000–0xFFFF FFFF in user mode will cause an address error. Memory-mapped registers can be referenced in user mode by means of access that involves address translation.

*Note: Do not access undefined locations in either area. The operation of an access to an undefined location is undefined. Memory-mapped registers must be accessed using a load/store instruction of an equal size to that of the register. The operation of an access using an invalid data size is undefined.*

## 2.6 Data format in registers

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

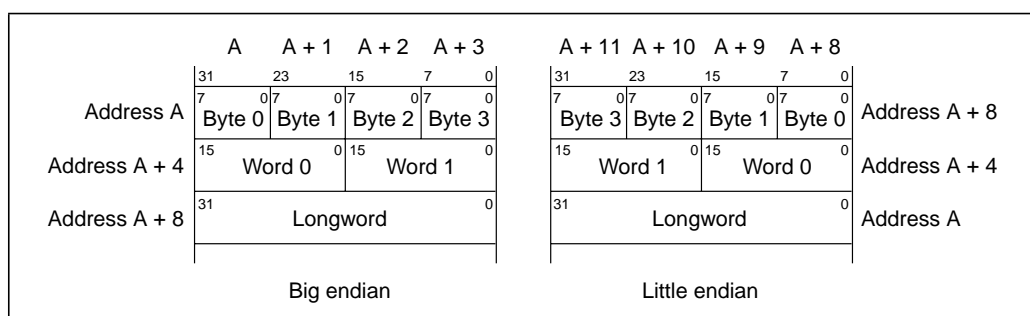
## 2.7 Data formats in memory

Memory can be accessed in 8-bit byte, 16-bit word, or 32-bit longword form. A memory operand less than 32 bits in length is sign-extended before being loaded into a register.

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

Big endian or little endian byte order can be selected for the data format. This endian selection cannot be changed dynamically and is selected by the system during power-on reset. Refer to the system architecture manual of the relevant product for details of how to perform endian selection. Bit positions are numbered left to right from most-significant to least-significant. Thus, in a 32-bit longword, the left-most bit, bit 31, is the most significant bit and the right-most bit, bit 0, is the least significant bit.

The data format in memory is shown in [Figure 4](#).



**Figure 4: Data formats in memory**

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

## 2.8 Processor states

The SH-4 CPU core has four processor states. Transitions between the states are shown in *Figure 5*

### 2.8.1 Reset state

In this state the CPU is reset. The CPU can be placed in one of two reset states, either power on reset or manual reset. Which of these is selected is determined by the system architecture. Refer to the relevant system architecture manual for details. For more information on resets, see section 5, Exceptions.

The purpose of having two reset modes is to allow some flexibility over which system components are reset. Typically:

- power-on reset will cause all system components to be reset,
- manual reset may, for example, avoid resetting DRAM controllers so that memory contents are preserved.

### 2.8.2 Exception-handling state

This is a transient state during which the CPU's processor state flow is altered by a reset, general exception, or interrupt exception source.

In the case of a reset, the CPU branches to address 0xA000 0000 and starts executing the user-coded exception handling program.

In the case of a general exception or interrupt, the program counter (PC) contents are saved in the saved program counter (SPC), the status register (SR) contents are saved in the saved status register (SSR), and the R15 contents are saved in saved general register (SGR). The CPU branches to the start address of the user-coded exception service routine, found from the sum of the contents of the vector base address and the vector offset.

See *Chapter 5: Exceptions*, for more information on resets, general exceptions, and interrupts.

### 2.8.3 Program execution state

In this state the CPU executes program instructions in sequence.

## 2.8.4 Power-down state

The power-down state is entered by executing a SLEEP instruction. In this state the CPU stops executing instructions and signals to the system that the CPU has been put to sleep. The system response to receiving this signal is described in the *System Architecture Manual* of the appropriate product.

The CPU is restarted by raising an interrupt.

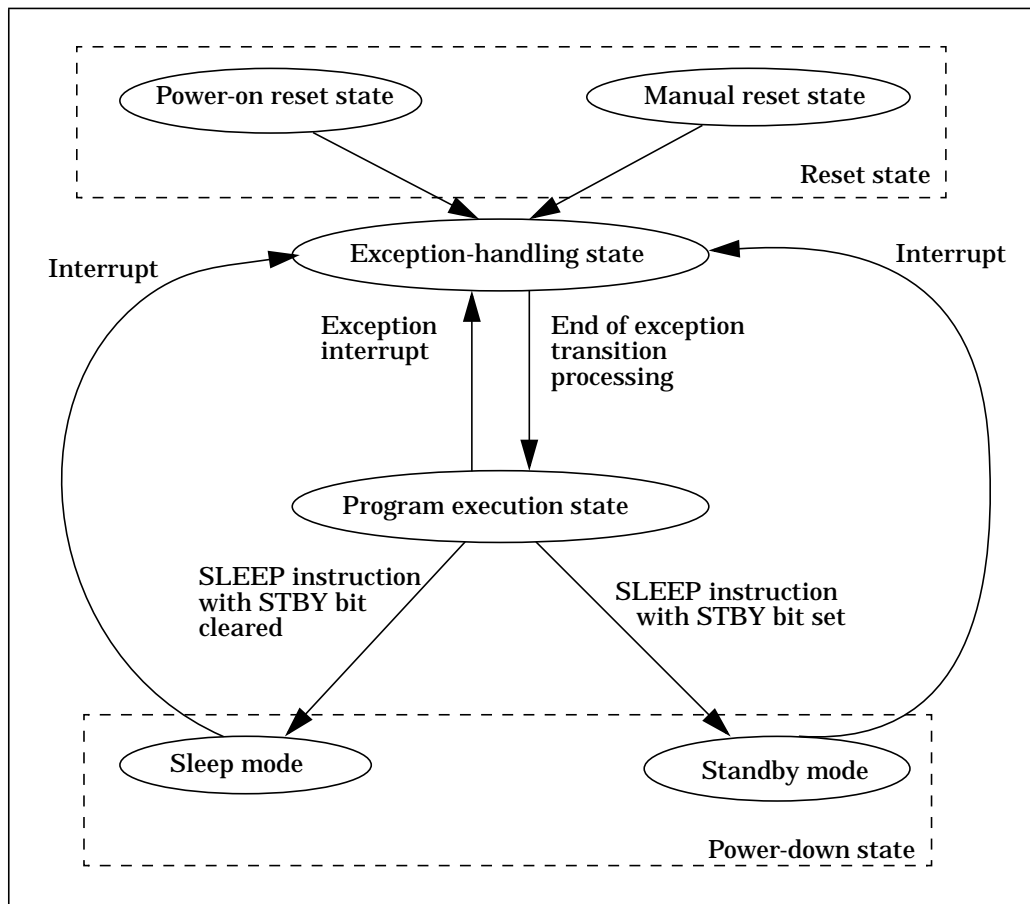


Figure 5: Processor state transitions

*Note:* For conditions determining state transitions, see the *System Architecture Manual*.

## 2.9 Processor modes

There are two processor modes: user mode and privileged mode. The processor mode is determined by the processor mode bit (MD) in the status register (SR). User mode is selected when the MD bit is cleared to 0, and privileged mode when the MD bit is set to 1. When the reset state or exception-handling state is entered, the MD bit is set to 1. When exception handling ends, the MD bit returns to the value held before the exception occurred.





## 3

# Memory management unit (MMU)

## 3.1 Overview

The SH-4 CPU core manages a 29-bit external memory space by providing 8-bit address space identifiers, and a 32-bit logical (virtual) address space. Address translation from virtual address to physical address is performed using the memory management unit (MMU), built into the SH-4 CPU core. 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. It is possible to set the virtual address space access right, and implement storage protection independently, for privileged mode and user mode.

## 3.2 Role of the MMU

The main purpose of an MMU is to ensure that efficient use is made of physical memory, which in most systems is a limiting resource. The MMU is normally managed by the OS, which allocates physical pages of memory to virtual pages of memory, as required by a task. Pages which are switched out by the OS are placed in a secondary storage device, such as a hard disk.

A page refers to a contiguous range of addresses, which can all be translated by a single translation table entry. On SH-4 there is support for 4 page sizes: 1-kbyte, 4-kbyte, 64-kbyte and 1-Mbyte.

Memory protection functions are provided to prevent physical memory from inadvertently being accessed and reset by a process.

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 (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.

### 3.3 Register descriptions

There are six MMU-related registers.

Name	Abbreviation	R/W	Initial value <sup>a</sup>	P4 address <sup>b</sup>	Area 7 address <sup>B</sup>	Access size
Page table entry high register	PTEH	R/W	Undefined	0xFF00 0000	0x1F00 0000	32
Page table entry low register	PTL	R/W	Undefined	0xFF00 0004	0x1F00 0004	32
Translation table base register	TTB	R/W	Undefined	0xFF00 0008	0x1F00 0008	32
Translation table address register	TEA	R/W	Undefined	0xFF00 000C	0x1F00 000C	32
MMU control register	MMUCR	R/W	0x0000 0000	0xFF00 0010	0x1F00 0010	32

**Table 6: MMU registers**

- a. The initial value is the value after a power-on reset or manual reset.
- b. This is the address when using the virtual/physical address space P4 region. When making an access from physical address space Area 7 using the TLB, the upper 3 bits of the address are ignored.

*Note:* Behavior is undefined if an area designated as a reserved area in this manual is accessed.

### 3.3.1 Page table entry high register (PTEH)

Longword access to PTEH can be performed from 0xFF00 0000 in the P4 region, and 0x1F00 0000 in Area 7. 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, always consists of the upper 22 bits of the virtual address which caused the exception. VPN setting can also be carried out by software. The number of the currently executing process is set in the ASID field by software. ASID is not updated by hardware. VPN and ASID are recorded in the UTLB by means of the LDLTB instruction.

PTEH				
Field	Bits	Size	Synopsis	Type
ASID	[0,7]	8	Address space identifier.	RW
	Operation		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.  See section 3.3.7 Address space identifier.	
	Power-on reset		Undefined	
VPN	[10,31]	22	Virtual page number.	RW
	Operation		For 1-kbyte: upper 22 bits of virtual address. For 4-kbyte: upper 20 bits of virtual address. For 64-kbyte: upper 16 bits of virtual address. For 1-Mbyte: upper 12 bits of virtual address.	
	Power-on reset		Undefined	

**Table 7: PTEH register description**

### 3.3.2 Page table entry low register (PTEL)

Longword access to PTEL can be performed from 0xFF00 0004 in the P4 region, and 0x1F00 0004 in Area 7. PTEL is used to hold the physical page number and page management information to be recorded in the UTLB, by means of the LDTLB instruction. The contents of this register are not changed unless a software directive is issued.

PTEL				
Field	Bits	Size	Synopsis	Type
WT	0	1	Write-through bit.	RW
	Operation		Specifies the cache write mode. 0: Copy-back mode. 1: Write-through mode.	
	Power-on reset		Undefined	
SH	1	1	Share status bit.	RW
	Operation		0: pages are not shared by processes. 1: pages are shared by processes.	
	Power-on reset		Undefined	
D	2	1	Dirty bit	RW
	Operation		Indicates whether a write has been performed to a page. 0: Write has not been performed. 1: Write has been performed.	
	Power-on reset		Undefined	
C	3	1	Cacheability bit.	RW
	Operation		Indicates whether a page is cacheable. 0: Not cacheable. 1: Cacheable. When control register is mapped, this bit must be cleared to 0.	
	Power-on reset		Undefined	

Table 8: PTEL register description

PTEL																			
Field	Bits	Size	Synopsis	Type															
SZ0	4	1	Page size bit.	RW															
	Operation		Specify page size. <div><table><tr><th>Bit SZ1</th><th>Bit SZ0</th><th>Page Size</th></tr><tr><td>0</td><td>0</td><td>1-kbyte</td></tr><tr><td>0</td><td>1</td><td>4-kbyte</td></tr><tr><td>1</td><td>0</td><td>64-kbyte</td></tr><tr><td>1</td><td>1</td><td>1-Mbyte</td></tr></table></div>		Bit SZ1	Bit SZ0	Page Size	0	0	1-kbyte	0	1	4-kbyte	1	0	64-kbyte	1	1	1-Mbyte
	Bit SZ1	Bit SZ0	Page Size																
	0	0	1-kbyte																
0	1	4-kbyte																	
1	0	64-kbyte																	
1	1	1-Mbyte																	
Power-on reset		Undefined																	
PR	[5,6]	2	Protection key data.	RW															
	Operation		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 or user mode.																
	Power-on reset		Undefined																
SZ1	7	1	Page size bit	RW															
	Operation		Refer to SZ0 for operation details.																
	Power-on reset		0																

Table 8: PTEL register description

PTEL				
Field	Bits	Size	Synopsis	Type
V	8	1	Validity bit.	RW
	Operation		Indicates whether the entry is valid. 0: Invalid 1: Valid Cleared to 0 by a power-on reset. Not affected by a manual reset.	
	Power-on reset		Undefined	
PPN	[10,28]	19	Physical page number	RW
	Operation		Upper 19 bits of the physical address. With a 1-kbyte page, PPN bits [28:10] are valid. With a 4-kbyte page, PPN bits [28:12] are valid. With a 64-kbyte page, PPN bits [28:16] are valid. With a 1-Mbyte page, PPN bits [28:20] are valid. The synonym problem must be taken into account when setting the PPN ( <a href="#">Section 3.6.5: Avoiding synonym problems on page 64</a> ).	
	Power-on reset		Undefined	
RES	9, [29,31]	4	Bits reserved	RW
	Power-on reset		Undefined	

Table 8: PTEL register description

### 3.3.3 Translation table base register (TTB)

Long word access to the TTB can be performed from 0xFF00 0008 in the P4 region, and 0x1F00 0008 in Area 7. The contents of the TTB are not changed unless a software directive is issued. This register can be freely used by software.

TTB				
Field	Bits	Size	Synopsis	Type
TTB	[0,31]	32	Translation table base register.	RW
	Operation		TTB is used, for example, to hold the base address of the currently used page table.	
	Power-on reset		Undefined	

Table 9: TTB register description

### 3.3.4 TLB exception address register (TEA)

Longword access to TEA can be performed from 0xFF00 000C in the P4 region and 0x1F00 000C in Area 7. The contents of this register can be changed by software.

TEA				
Field	Bits	Size	Synopsis	Type
TEA	[0,31]	32	TLB exception address register.	RW
	Operation		After an MMU exception or address error exception occurs, the virtual address at which the exception occurred is set in TEA by hardware.	
	Power-on reset		Undefined	

Table 10: TEA register description

### 3.3.5 MMU control register (MMUCR)

Longword access to MMUCR can be performed from 0xFF00 0010 in the P4 region, and 0x1F00 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 region. After MMUCR is updated, an instruction that performs data access to the

P0, P3, U0, or store queue region should be located at least four instructions after the MMUCR update instruction. Also, a branch instruction to the P0, P3, or U0 region should be located at least eight instructions after the MMUCR update instruction. MMUCR contents can be changed by software. The LRUI bits and URC bits may also be updated by hardware.

MMUCR				
Field	Bits	Size	Synopsis	Type
AT	0	1	Address translation bit.	RW
	Operation		Specifies MMU enabling or disabling. 0: MMU disabled. 1: MMU enabled.  MMU exceptions are not generated when the AT bit is 0. Therefore, in the case of software that does not use the MMU, the AT bit should be cleared to 0.	
	Power-on reset		0	
TI	2	1	TLB invalidate.	RW
	Operation		Writing 1 to this bit invalidates (clears to 0) all valid UTLB/ITLB bits. This bit always returns 0 when read.	
	Power-on reset		0	
SV	8	1	Single virtual mode bit.	RW
	Operation		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.	
	Power-on reset		0	

Table 11: MMUCR register description



MMUCR				
Field	Bits	Size	Synopsis	Type
SQMD	9	1	Store queue mode bit.	RW
	Operation		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).	
	Power-on reset		0	
URC	[10,15]	6	UTLB replace counter.	RW
	Operation		Random counter for indicating the UTLB entry for which replacement is to be performed with an LDTLB instruction. URC is incremented each time the UTLB is accessed. When URB > 0, URC is reset to 0 when the condition URC = URB occurs. Also note that, if a value is written to URC by software which results in the condition URC > URB, incrementing is first performed in excess of URB until URC = 0x3F. URC is not incremented by an LDTLB instruction.	
	Power-on reset		0	
URB	[18,23]	6	UTLB replace boundary.	RW
	Operation		Bits that indicate the UTLB entry boundary at which replacement is to be performed. Valid only when URB > 0.	
	Power-on reset		0	

Table 11: MMUCR register description

MMUCR																																																																																									
Field	Bits	Size	Synopsis	Type																																																																																					
LRUI	[26, 31]	6	Least recently used ITLB.	RW																																																																																					
	Operation		<p>The LRU (least recently used) method is used to decide the ITLB entry to be replaced in the event of an ITLB miss. The entry to be purged from the ITLB can be confirmed using the LRUI bits. LRUI is updated by means of the algorithm shown below. A dash in this table means that updating is not performed .</p> <table><tr><td></td><td>[5]</td><td>[4]</td><td>[3]</td><td>[2]</td><td>[1]</td><td>[0]</td></tr><tr><td>When ITLB entry 0 is used</td><td>0</td><td>0</td><td>0</td><td>-</td><td>-</td><td>-</td></tr><tr><td>When ITLB entry 1 is used</td><td>1</td><td>-</td><td>-</td><td>0</td><td>0</td><td>-</td></tr><tr><td>When ITLB entry 2 is used</td><td>-</td><td>1</td><td>-</td><td>1</td><td>-</td><td>0</td></tr><tr><td>When ITLB entry 3 is used</td><td>-</td><td>-</td><td>1</td><td>-</td><td>1</td><td>1</td></tr><tr><td>Other than the above</td><td>-</td><td>-</td><td>-</td><td>-</td><td>-</td><td>-</td></tr></table> <p>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”..</p> <table><tr><td></td><td>[5]</td><td>[4]</td><td>[3]</td><td>[2]</td><td>[1]</td><td>[0]</td></tr><tr><td>ITLB entry 0 is updated</td><td>1</td><td>1</td><td>1</td><td>*</td><td>*</td><td>*</td></tr><tr><td>ITLB entry 1 is updated</td><td>0</td><td>*</td><td>*</td><td>1</td><td>1</td><td>*</td></tr><tr><td>ITLB entry 2 is updated</td><td>*</td><td>0</td><td>*</td><td>0</td><td>*</td><td>1</td></tr><tr><td>ITLB entry 3 is updated</td><td>*</td><td>*</td><td>0</td><td>*</td><td>0</td><td>0</td></tr><tr><td>Other than the above</td><td colspan="6">Setting prohibited</td></tr></table> <p>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 manual reset the bits are initialized to 0,and therefore a prohibited setting is never made by a hardware update.</p>					[5]	[4]	[3]	[2]	[1]	[0]	When ITLB entry 0 is used	0	0	0	-	-	-	When ITLB entry 1 is used	1	-	-	0	0	-	When ITLB entry 2 is used	-	1	-	1	-	0	When ITLB entry 3 is used	-	-	1	-	1	1	Other than the above	-	-	-	-	-	-		[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				
	[5]	[4]	[3]	[2]	[1]	[0]																																																																																			
When ITLB entry 0 is used	0	0	0	-	-	-																																																																																			
When ITLB entry 1 is used	1	-	-	0	0	-																																																																																			
When ITLB entry 2 is used	-	1	-	1	-	0																																																																																			
When ITLB entry 3 is used	-	-	1	-	1	1																																																																																			
Other than the above	-	-	-	-	-	-																																																																																			
	[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																																																																																								
Power-on reset		0																																																																																							

Table 11: MMUCR register description

MMUCR				
Field	Bits	Size	Synopsis	Type
RES	1, [3,7], [16,17], [24,25]	10	Bits reserved	RW
	Power-on reset		Undefined	

Table 11: MMUCR register description

## 3.4 Address space

### 3.4.1 Physical address space

The SH-4 CPU core supports a 32-bit (4-Gbyte) physical address space. When the MMUCR.AT bit is cleared to 0 and the MMU is disabled, the address space accessed by the program is this physical address space. The physical address space is divided into a number of regions, as shown in [Figure 7](#). The region is selected using the top 3 bits of the physical address.

Bit			Region accessed	
31	30	29	Privileged mode	User mode
0	0	0	P0	U0
0	0	1		
0	1	0		
0	1	1		
1	0	0	P1	Address error
1	0	1	P2	Address error
1	1	0	P3	Address error
1	1	1	P4	Address error <sup>a</sup>

Table 12: Region selection

- a. Except for address from 0xe000 0000 - 0xe3FF FFFF which the user can use to access the store queues.

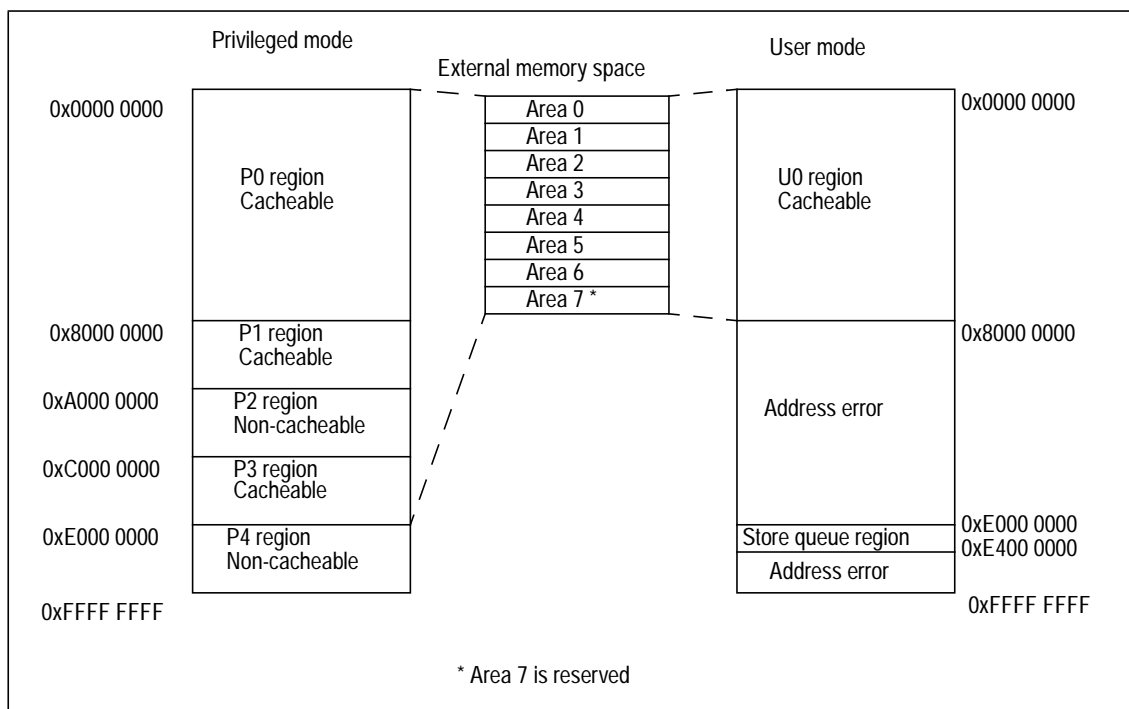
The region selected determines how the remaining 29 bits are interpreted. For example P0, P1 and P3 all access the 29 bits of external memory via the cache. P4 is used exclusively to access the cores internal devices. See the system architecture manual for more details of the internal devices available on a particular product.

### 3.4.2 External memory space

The SH-4 CPU core supports a 29-bit external memory space. The external memory space is divided into eight Areas as shown in [Figure 7](#). Areas 0 to 6 relate to memory, Area 7 is a reserved area, and is only accessed via the P4 region.

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 6: External memory Space



**Figure 7: Physical address space (MMUCR.AT = 0)**

**P0, P1, P3, U0 Regions:** The P0, P1, P3, and U0 regions can be accessed using the cache. Whether or not the cache is used is determined by the cache control register (CCR). When the cache is used, with the exception of the P1 region, switching between the copy-back method and the write-through method for write accesses is specified by the CCR.WT bit. For the P1 region, switching is specified by the CCR.CB bit. Zeroing the upper 3 bits of an address in these regions 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 regions.

**P2 Region:** The P2 region cannot be accessed using the cache. In the P2 region, zeroing 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 region.

**P4 Region:** The P4 region is mapped onto SH-4 CPU core on-chip I/O channels. This region cannot be accessed using the cache. The P4 region is shown in detail in [Table 13](#).

Start address	End address	Function
0xE000 0000	0xE3FF FFFF	Comprises addresses for accessing the store queues (SQs). When the MMU is disabled (MMUCR.AT=0), the SQ access right is specified by the MMUCR.SQMD bit. For details, see <a href="#">Section 4.6: Store queues on page 101</a> .
0xF000 0000	0xF0FF FFFF	Used for direct access to the instruction cache address array. For details, see <a href="#">Section 4.5.1: IC address array on page 95</a> .
0xF100 0000	0xF1FF FFFF	Used for direct access to the instruction cache data array. For details, see <a href="#">Section 4.5.4: IC data array on page 97</a> .
0xF200 0000	0xF2FF FFFF	Used for direct access to the instruction TLB address array. For details, see <a href="#">Section 3.8.1: ITLB address array on page 70</a> .
0xF300 0000	0xF3FF FFFF	Used for direct access to instruction TLB data arrays 1 and 2. For details, see <a href="#">Section 3.8.2: ITLB data array 1 on page 71</a> .
0xF400 0000	0xF4FF FFFF	Used for direct access to the operand cache address array. For details, see <a href="#">Section 4.5.5: OC address array on page 98</a> .
0xF500 0000	0xF5FF FFFF	Used for direct access to the operand cache data array. For details, see <a href="#">Section 4.5.6: OC data array on page 99</a> .
0xF600 0000	0xF6FF FFFF	Used for direct access to the unified TLB address array. For details, see <a href="#">Section 3.8.3: UTLB address array on page 72</a> .
0xF700 0000	0xF7FF FFFF	Used for direct access to unified TLB data arrays 1 and 2. For details, see <a href="#">Section 3.8.4: UTLB data array 1 on page 74</a> .
0xFC00 0000	0xFFFF FFFF	Control register area.

Table 13: P4 area

### 3.4.3 Virtual address space

Setting the MMUCR.AT bit to 1, enables the P0, P3, and U0 regions of the address space in the SH-4 CPU core to be mapped onto any external memory space in 1-, 4-, or 64-kbyte, or 1-Mbyte, page units. Mapping from virtual address space to 29-bit external memory space is carried out using the TLB. When accessed using virtual addressing, Area 7 is equivalent to the P4 region in physical address space. Virtual address space is illustrated in [Figure 8](#).

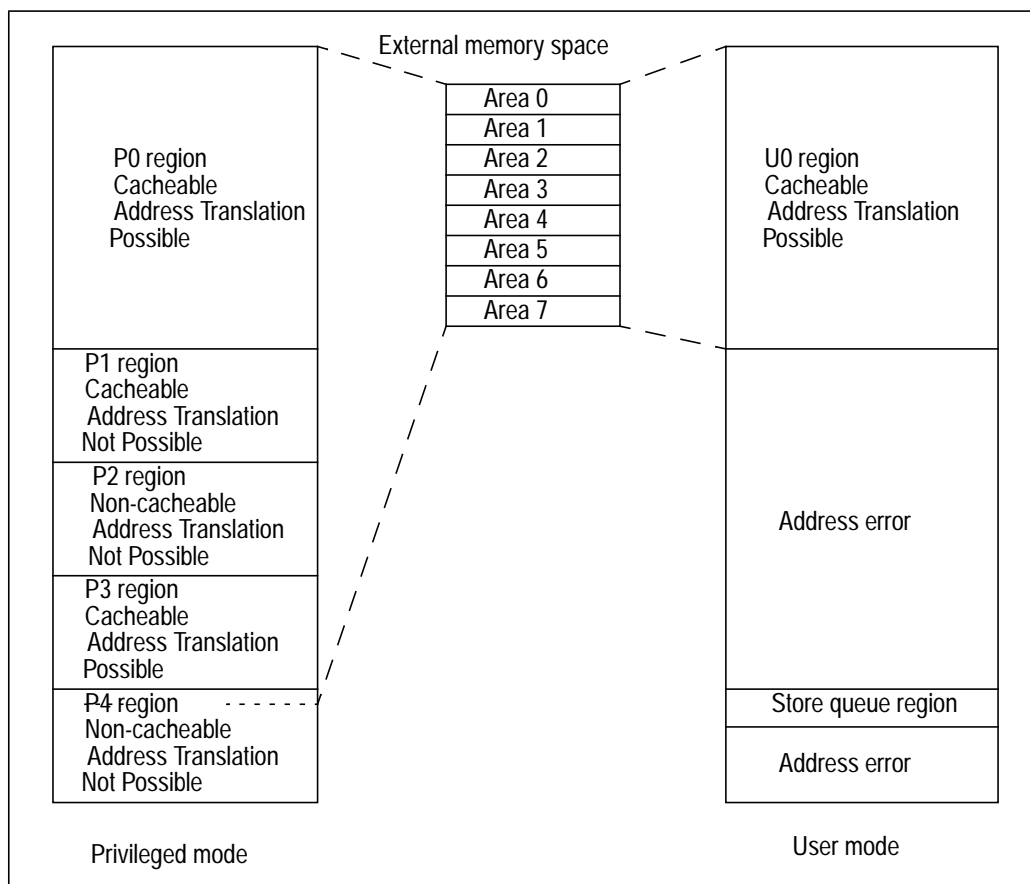


Figure 8: Virtual memory space (MMUCR.AT = 1)

**P0, P3, U0 Regions:** The P0 region (excluding addresses 0x7C00 0000 to 0x7FFF FFFF), P3 region, and U0 region, allow access using the cache, and address translation using the TLB. These regions can be mapped onto any external memory space in 1, 4, or 64-kbyte, or 1-Mbyte, page units. When CCR is in the cache-enabled state, and the TLB enable bit (C bit) is 1, accesses can be performed using the cache. In write accesses to the cache, switching between the copy-back method and the write-through method is indicated by the TLB write-through bit (WT bit), and is specified in page units.

Only when the P0, P3, and U0 regions are mapped onto external memory space by means of the TLB, are addresses 0x1C00 0000 to 0x1FFF FFFF of Area 7 in external memory space allocated to the control register area. This enables control registers to be accessed from the U0 region in user mode. In this case, the C bit for the corresponding page must be cleared to 0.

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

### 3.4.4 On-chip RAM space

In the SH-4 CPU core, half of the (16 kbyte) operand cache 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), the P0/ U0 region addresses 0x7C00 0000 to 0x7FFF FFFF are an on-chip RAM area. Data accesses (byte/word/longword/quadword) can be used in this area. This area can only be used in RAM mode.

*Note: It is not possible to execute instructions out of this on-chip RAM.*



### 3.4.5 Address translation

In the SH-4 CPU core, the ITLB is used for instruction accesses and the UTLB for data accesses. In the event of an access to a region other than the P4 region, the accessed virtual address is translated to a physical address. If the virtual address belongs to the P1 or P2 region, the physical address is uniquely determined without accessing the TLB. If the virtual address belongs to the P0, U0, or P3 region, the TLB is searched using the virtual address, and if the virtual address is recorded in the TLB, a TLB hit is made and the corresponding physical address is read from the TLB. If the accessed virtual address is not recorded in the TLB, a TLB miss exception is generated and processing switches to the TLB miss exception handling routine. In the TLB miss exception handling routine, the address translation table in external memory is searched, and the corresponding physical address and page management information are recorded in the TLB. After the return from the exception handling routine, the instruction which caused the TLB miss exception is re-executed.

### 3.4.6 Single virtual memory mode and multiple virtual memory mode

There are two virtual memory systems, either of which can be selected with the MMUCR.SV bit:

- single virtual memory  
A number of processes run simultaneously, using non-overlapping virtual address spaces, so that the physical address corresponding to a particular virtual address is uniquely determined.
- multiple virtual memory  
A number of processes run with overlapping virtual address spaces, consequently, virtual addresses may need to be translated into different physical addresses depending on the process i.d.

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 3.5.3: Address translation method on page 59](#)).



### 3.5.2 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 10* shows the overall configuration of the ITLB. The ITLB consists of 4 fully-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
Entry 1	ASID [7:0]	VPN [31:10]	V	PPN [28:10]	SZ [1:0]	SH	C	PR
Entry 2	ASID [7:0]	VPN [31:10]	V	PPN [28:10]	SZ [1:0]	SH	C	PR
Entry 3	ASID [7:0]	VPN [31:10]	V	PPN [28:10]	SZ [1:0]	SH	C	PR

Figure 10: ITLB configuration

### 3.5.3 Address translation method

*Figure 11* and *Figure 12* show flowcharts of memory accesses using the UTLB and ITLB

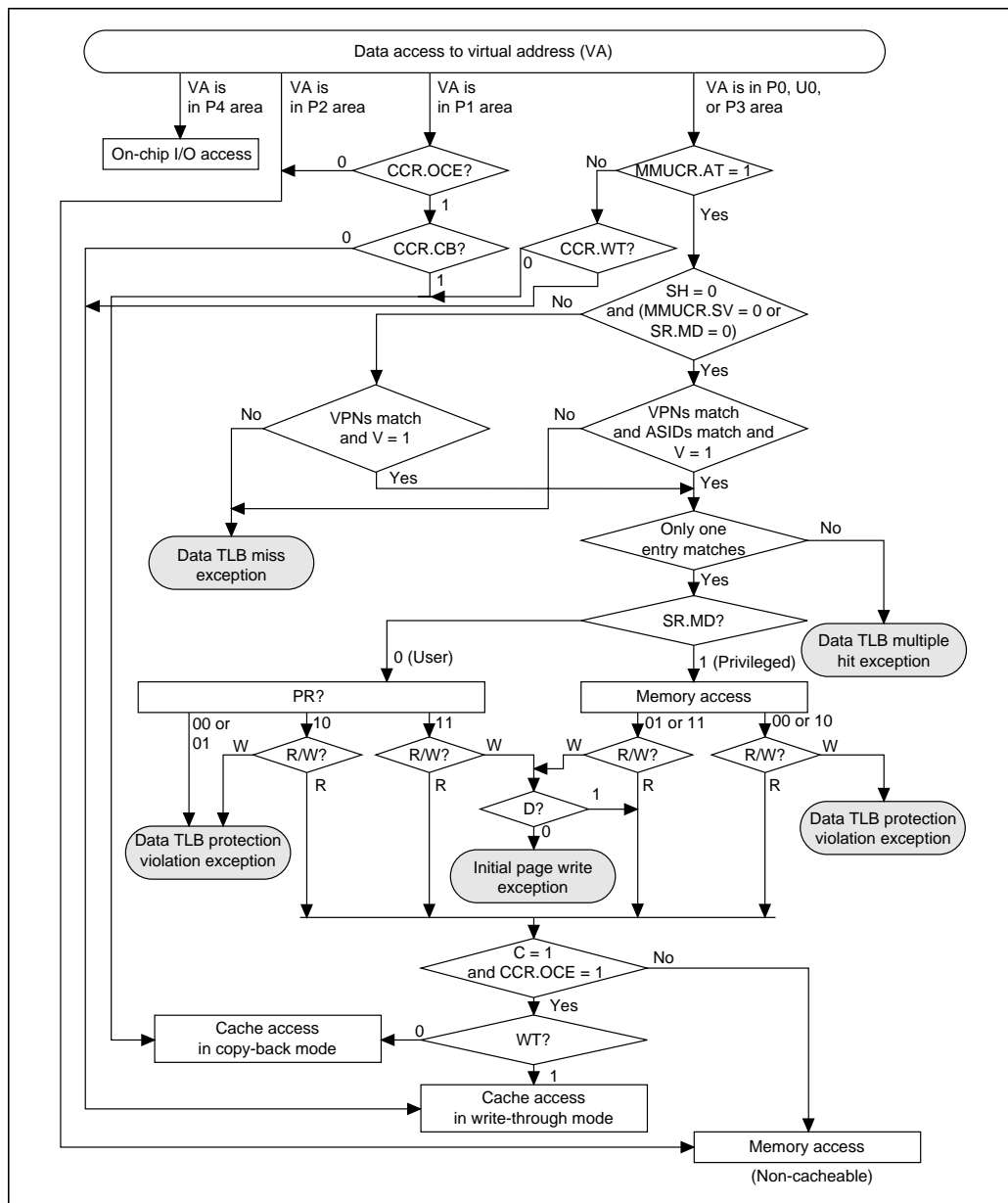


Figure 11: Flowchart of memory access using UTLB figure

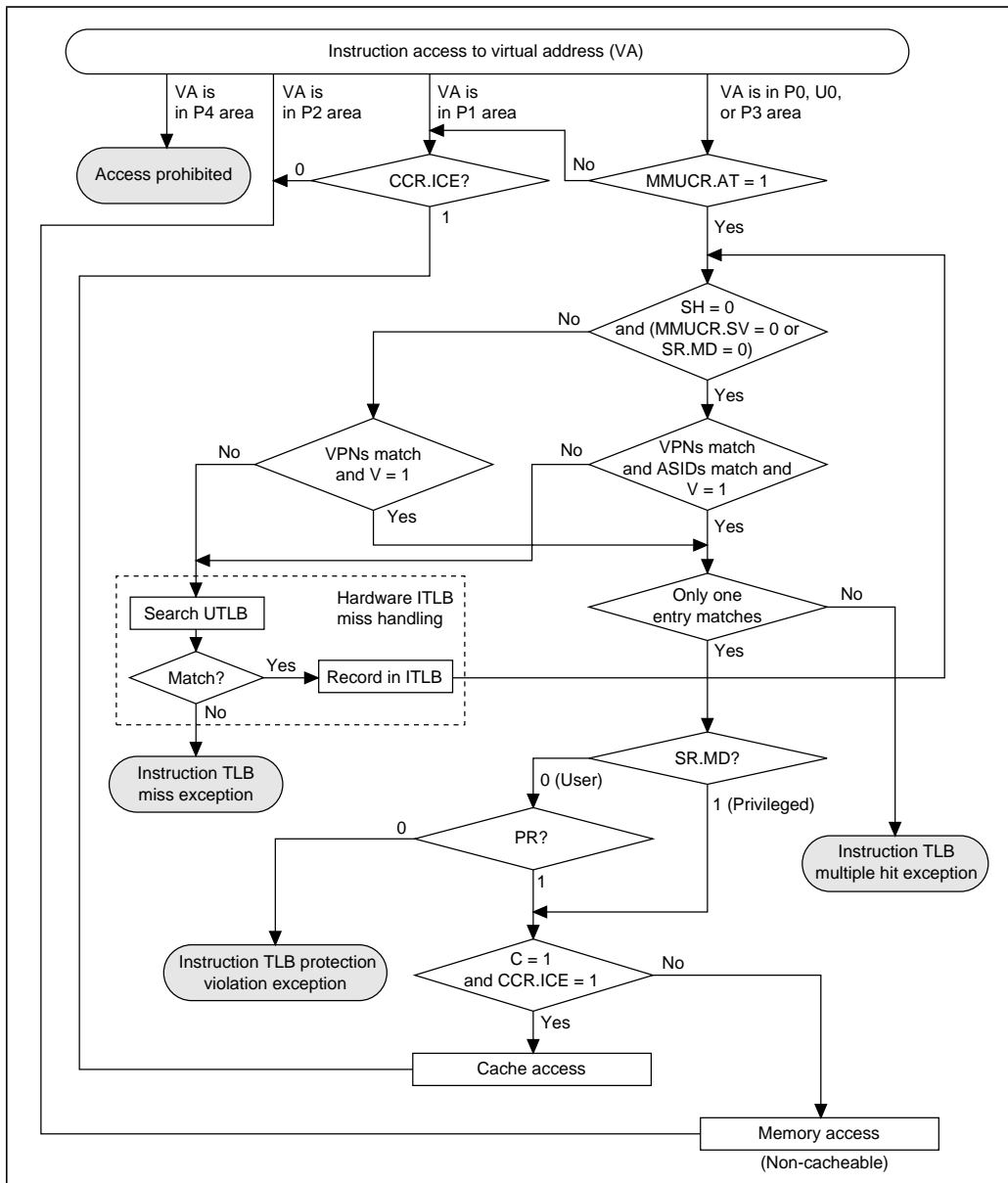


Figure 12: Flowchart of memory access using ITLB

## 3.6 MMU functions

### 3.6.1 MMU hardware management

The SH-4 CPU core supports the following MMU functions.

- 1 The MMU decodes the virtual address to be accessed by software, and performs address translation by controlling the UTLB/ITLB, in accordance with the MMUCR settings.
- 2 The MMU determines the cache access status, on the basis of the page management information read during address translation (C, WT 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.

### 3.6.2 MMU software management

Software processing for the MMU consists of the following:

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

### 3.6.3 MMU instruction (LDTLB)

A TLB load instruction (LDTLB) is provided for recording UTLB entries. When an LDTLB instruction is issued, the SH-4 CPU core copies the contents of PTEH and PTEL, 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 region. The operation of the LDTLB instruction is shown in [Figure 13](#).

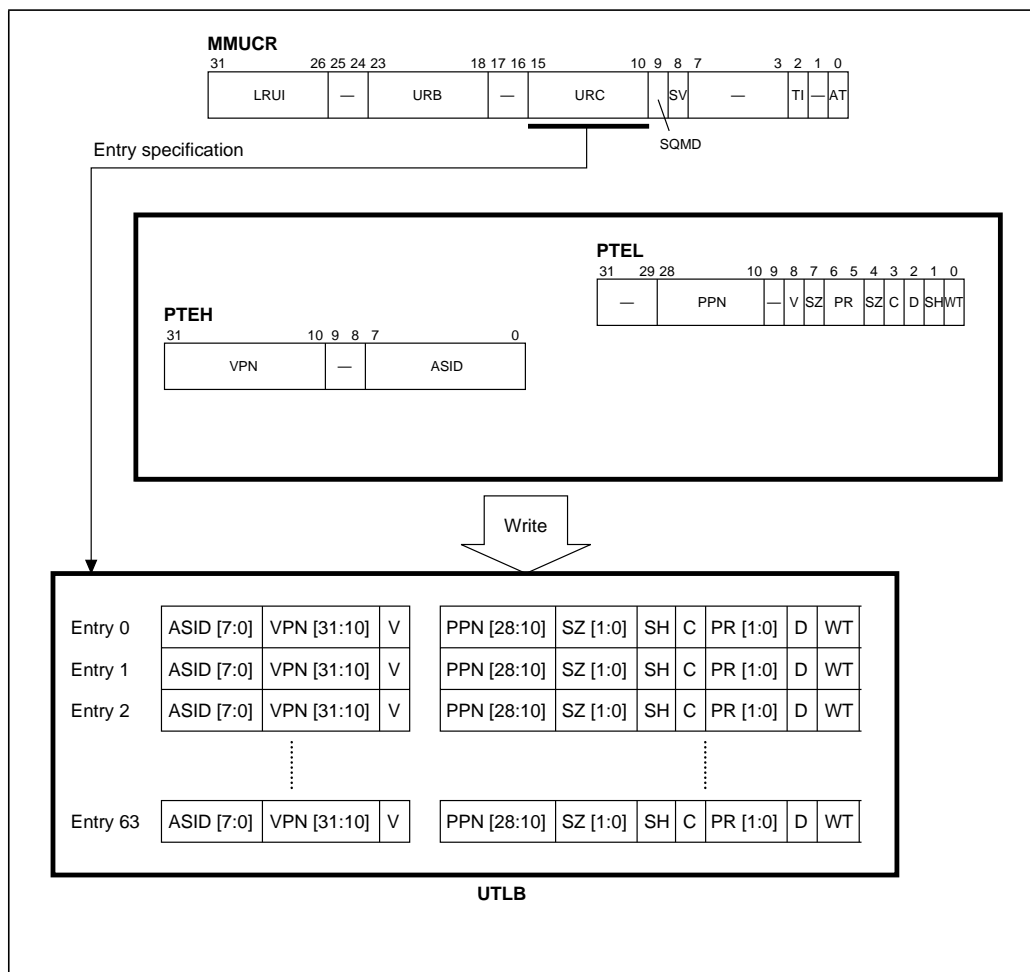


Figure 13: Operation of LDTLB instruction

### 3.6.4 Hardware ITLB miss handling

In an instruction access, the SH-4 CPU core 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.

### 3.6.5 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 may be 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 CPU core, line selection is performed using bits [13:5] of the virtual address, as this avoids the cache having to go via the TLB and thus achieves faster operand cache operation. However, bits [13:10] of the virtual address in the case of a 1-kbyte page, and bits [13:12] of the virtual address in the case of a 4-kbyte page, are subject to address translation. As a result, bits [13:10] of the physical address after translation may differ from bits [13:10] of the virtual address.

Great care must therefore be taken whenever translations are set up which could cause synonyms, in particular, if two operand translations are to the same physical page but their virtual addresses differ in their synonym bits:

- Do not allow both the translations to be active at the same time.
- Always separate activations of the two translations by an appropriate cache purge.



## 3.7 Handling MMU exceptions

There are seven MMU exceptions.

### 3.7.1 ITLBMULTIHIT

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

See *Chapter 5: Exceptions on page 105*, *ITLBMULTIHIT - Instruction TLB Multiple-Hit Exception on page 118*.

#### 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.

### 3.7.2 ITLBMISS

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 software, is shown below. This is the same as the processing for a data TLB miss exception.

#### Hardware processing

See, *Chapter 5: Exceptions on page 105*, *ITLBMISS - Instruction TLB Miss Exception on page 122*.

### Software processing (instruction TLB miss exception handling routine)

Software is responsible for searching the external memory page table and assigning the necessary page table entry. Software should carry out the following processing in order to find and assign the necessary page table entry.

- 1 Write to PTEL the values of the PPN, PR, SZ, C, D, SH, V, and WT bits in the page table entry recorded in the external memory address translation table.
- 2 When the entry to be replaced in entry replacement is specified by software, write that value to 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 to the TLB.
- 4 Finally, execute the exception handling return instruction (RTE), terminate the exception handling routine, and return control to the normal flow. The RTE instruction should be issued at least one instruction after the LDTLB instruction.

## 3.7.3 EXECPROT

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 software, is shown below.

### Hardware processing

See *Chapter 5: Exceptions on page 105*, *EXECPROT - Instruction TLB Protection Violation Exception on page 126*.

### Software processing (instruction TLB protection violation exception handling routine)

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

### 3.7.4 OTLBMULTIHIT

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

#### Hardware processing

See *Chapter 5: Exceptions on page 105*, *OTLBMULTIHIT - Operand TLB Multiple-Hit Exception on page 119*.

#### 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.

### 3.7.5 TLBMISS

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 software, is shown below.

#### Hardware processing

See *Chapter 5: Exceptions on page 105*, *RTLBMIS - Read Data TLB Miss Exception on page 120*.

#### Software processing (data TLB miss exception handling routine)

Software is responsible for searching the external memory page table and assigning the necessary page table entry. Software should carry out the following processing in order to find and assign the necessary page table entry.

- 1 Write to PTEL the values of the PPN, PR, SZ, C, D, SH, V, and WT bits in the page table entry recorded in the external memory address translation table.

- 2 When the entry to be replaced in entry replacement is specified by software, write that value to 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 to the UTLB.
- 4 Finally, execute the exception handling return instruction (RTE), terminate the exception handling routine, and return control to the normal flow. The RTE instruction should be issued at least one instruction after the LDTLB instruction.

### 3.7.6 READPROT

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 software, is shown below.

#### Hardware processing

See *Chapter 5: Exceptions on page 105, READPROT - Data TLB Protection Violation Exception on page 124*

#### Software processing (data TLB protection violation exception handling routine)

Resolve the data TLB protection violation, execute the exception handling return instruction (RTE), terminate the exception handling routine, and return control to the normal flow. The RTE instruction should be issued at least one instruction after the LDTLB instruction.

### 3.7.7 FIRSTWRITE

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 software, is shown below.

### Hardware processing

See *Chapter 5: Exceptions on page 105*, *FIRSTWRITE - Initial Page Write Exception on page 123*

### Software processing (initial page write exception handling routine)

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

- 1 Retrieve the necessary page table entry from external memory.
- 2 Write 1 to the D bit in the external memory page table entry.
- 3 Write to PTEL the values of the PPN, PR, SZ, C, D, WT, SH, and V bits in the page table entry recorded in external memory.
- 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 to the UTLB.
- 6 Finally, execute the exception handling return instruction (RTE), terminate the exception handling routine, and return control to the normal flow. The RTE instruction should be issued at least one instruction after the LDTLB instruction.

## 3.8 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 region program, with a MOV instruction in privileged mode. Operation is not guaranteed if access is made from a program in another region. A branch to a region other than the P2 region should be made at least 8 instructions after this MOV instruction. The ITLB and UTLB are allocated to the P4 region in physical address space. VPN, V and ASID in the ITLB can be accessed as an address array, PPN, V, SZ, PR, C, and SH as data array 1. VPN, D, V, and ASID in the UTLB can be accessed as an address array, PPN, V, SZ, PR, C, D, WT, and SH as data array 1. V and D can be accessed from both the address array side and the data array side. Only longword access is possible. Instruction fetches cannot be performed in these regions. For reserved bits, a write value of 0 should be specified; their read value is undefined.

### 3.8.1 ITLB address array

The ITLB address array is allocated to addresses 0xF200 0000 to 0xF2FF FFFF in the P4 region. An address array access requires a 32-bit address field specification (when reading or writing), and a 32-bit data field specification (when writing). Information for selecting the entry to be accessed is specified in the address field, and VPN, V, and ASID to be written to the address array are specified in the data field.

In the address field, bits [31:24] have the value 0xF2 indicating the ITLB address array, and the entry is selected by bits [9:8]. As longword access is used, 0 should be specified for address field bits [1:0].

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

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

#### 1 ITLB address array read

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

#### 2 ITLB address array write

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

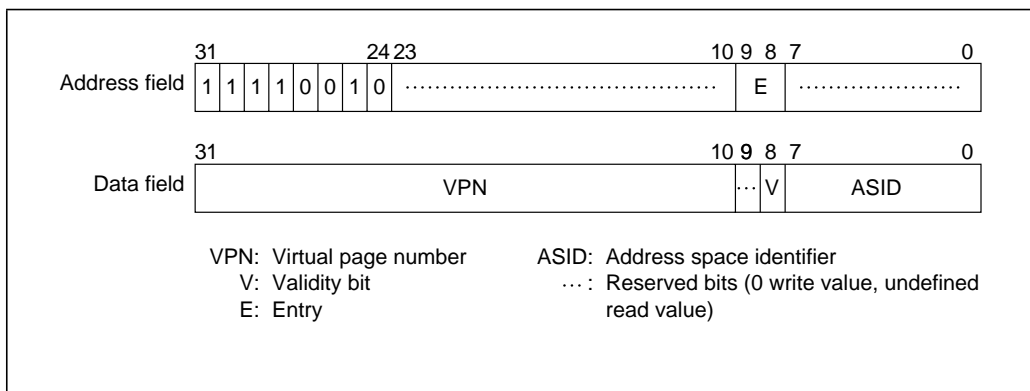


Figure 14: Memory-mapped ITLB address array

### 3.8.2 ITLB data array 1

ITLB data array 1 is allocated to addresses 0xF300 0000 to 0xF37F FFFF in the P4 region. A data array access requires a 32-bit address field specification (when reading or writing), and a 32-bit data field specification (when writing). Information for selecting the entry to be accessed is specified in the address field, and PPN, V, SZ, PR, C, and SH to be written to the data array are specified in the data field.

In the address field, bits [31:23] have the value 0xF30 indicating ITLB data array 1, and the entry is selected by bits [9:8].

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

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

#### 1 ITLB data array 1 read

PPN, V, SZ, PR, C, and SH are read into the data field from the ITLB entry corresponding to the entry set in the address field.

#### 2 ITLB data array 1 write

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

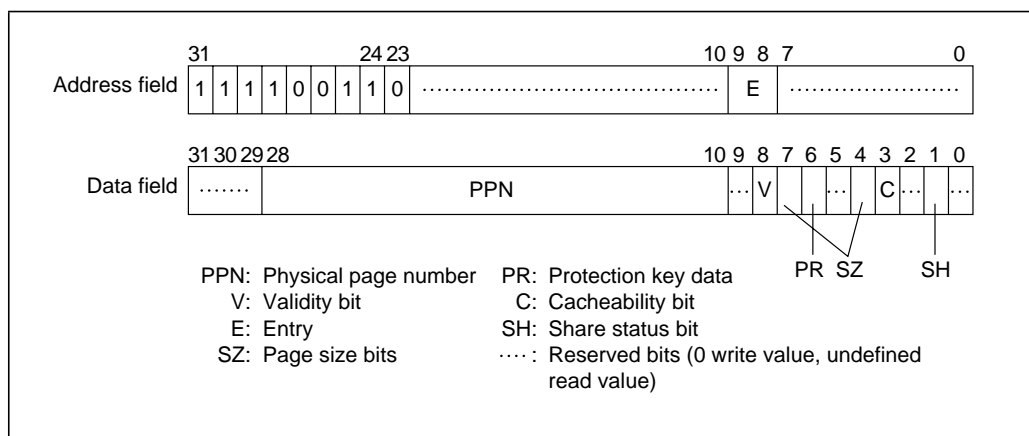


Figure 15: Memory-mapped ITLB data array 1

### 3.8.3 UTLB address array

The UTLB address array is allocated to addresses 0xF600 0000 to 0xF6FF FFFF in the P4 region. An address array access requires a 32-bit address field specification (when reading or writing), and a 32-bit data field specification (when writing). Information for selecting the entry to be accessed is specified in the address field, and VPN, D, V, and ASID to be written to the address array are specified in the data field.

In the address field, bits [31:24] have the value 0xF6 indicating the UTLB address array, and the entry is selected by bits [13:8]. The address array bit [7] association bit (A bit), specifies whether or not address comparison is performed when writing to the UTLB address array.

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



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

### 1 UTLB address array read

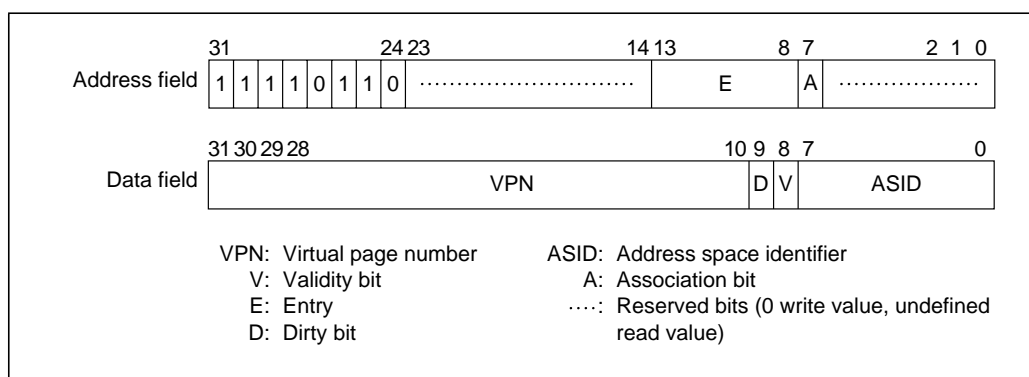
VPN, D, V, and ASID are read into the data field from the UTLB entry corresponding to the entry set in the address field. In a read, associative operation is not performed, regardless of whether the association bit specified in the address field is 1 or 0.

### 2 UTLB address array write (non-associative)

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

### 3 UTLB address array write (associative)

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



**Figure 16: Memory-mapped UTLB address array**

### 3.8.4 UTLB data array 1

UTLB data array 1 is allocated to addresses 0xF700 0000 to 0xF77F FFFF in the P4 region. A data array access requires a 32-bit address field specification (when reading or writing), and a 32-bit data field specification (when writing). Information for selecting the entry to be accessed is specified in the address field, and PPN, V, SZ, PR, C, D, SH, and WT to be written to the data array, are specified in the data field.

In the address field, bits [31:23] have the value 0xF70 indicating UTLB data array 1, and the entry is selected by bits [13:8].

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

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

#### 1 UTLB data array 1 read

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

#### 2 UTLB data array 1 write

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

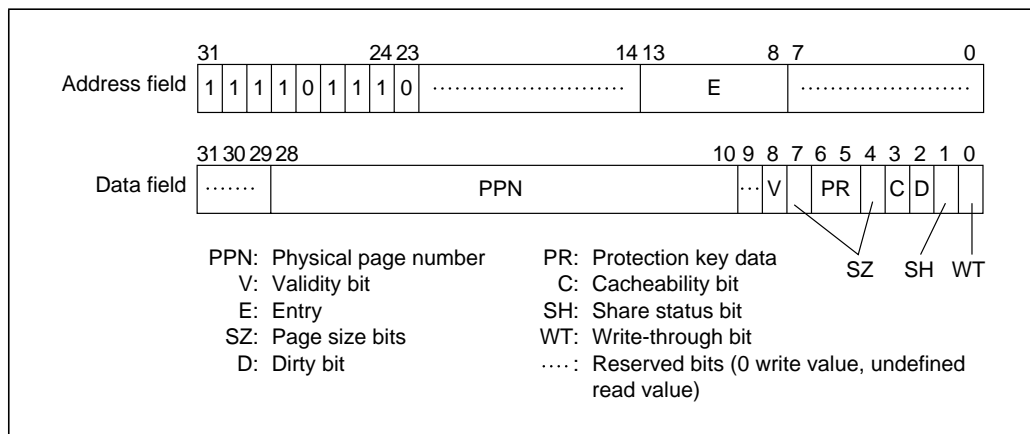


Figure 17: Memory-mapped UTLB data array 1



## 4

# Caches

## 4.1 Overview

### 4.1.1 Features

*Note:* This chapter details both the SH4-103 and SH4-202 variants. Please refer to your datasheet for specific core details.

The SH-4 CPU core has an on-chip 8-kbyte instruction cache (IC) for instructions and 16-kbyte operand cache (OC) for data. Half of the memory of the operand cache (8 kbytes) can also be used as on-chip RAM. The features of these caches are summarized in [Table 14](#).

The SH4-202 has an on-chip 16-kbyte instruction cache (IC) for instructions and 32-kbyte operand cache (OC) for data. Half of the operand cache (16 kbytes) can also be used as on-chip RAM. The features of these caches are summarized in [Table 14](#) and [Table 15](#).

The SH-4 CPU supports two 32-byte store queues (SQ) to perform high-speed writes to external memory. The features of the SQ are summarized in [Table 16](#).

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

Table 14: Cache features (SH4-103, SH4-202 in compatibility mode)

Item	Instruction cache	Operand cache
Entries	256	512
Write method		Copy-back/write-through selectable

Table 14: Cache features (SH4-103, SH4-202 in compatibility mode)

Item	Instruction cache	Operand cache
Capacity	16-kbyte cache	32-kbyte cache or 16-kbyte cache + 16-kbyte RAM
Type	2way set associative	2way set associative
Line size	32 bytes	32 bytes
Entries	256 entry /way	512 entry / way
Write method		Copy-back/write-through selectable
Replace algorithm	LRU	LRU

Table 15: Cache features (SH4-202 in the enhanced mode)

Item	Store queues
Capacity	$2 \times 32$ bytes
Addresses	0xE000 0000 to 0xE3FF FFFF
Write	Store instruction
Write-back	Prefetch instruction
Access right	MMU off: according to MMUCR.SQMD MMU on: according to individual page PR

Table 16: Store queue features

## 4.2 Register descriptions

There are three cache and store queue related control registers.

Name	Abbreviation	R/W	Initial value <sup>a</sup>	P4 address <sup>b</sup>	Area 7 address <sup>b</sup>	Access size
Cache control register	CCR	R/W	0x0000 0000	0xFF00 001C	0x1F00 001C	32
Queue address control register 0	QACR0	R/W	Undefined	0xFF00 0038	0x1F00 0038	32
Queue address control register 1	QACR1	R/W	Undefined	0xFF00 003C	0x1F00 003C	32

**Table 17: Cache control registers**

- The initial value is the value after a power-on or manual reset.
- This is the address when using the virtual/physical address space P4 area. The area 7 address is the address used when making an access from physical address space area 7 using the TLB.

### 4.2.1 Cache control register (CCR)

CCR can be accessed by longword-size access from 0xFF00001C in the P4 region and 0x1F00001C in Area 7. The CCR bits are used to modify the cache settings described below. CCR modifications must only be made by a program in the non-cached P2 region. After CCR is updated, an instruction that performs data access to the P0, P1, P3, or U0 regions, should be located at least four instructions after the CCR update instruction. Also, a branch instruction to the P0, P1, P3, or U0 regions should be located at least eight instructions after the CCR update instruction.

CCR				
Field	Bits	Size	Synopsis	Type
OCE	0	1	OC enable.	RW
	Operation		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.	
	Power-on reset		0	
WT	1	1	Write-through enable.	RW
	Operation		Indicates the P0, U0 and P3 region cache write mode. When address translation is performed, the value of the WT bit in the page management information has priority. 0: Copy-back mode. 1: Write-through mode.	
	Power-on reset		0	
CB	2	1	Copy-back bit.	RW
	Operation		Indicates the P1 region cache write mode. 0: Write-through mode. 1: Copy-back mode.	
	Power-on reset		0	
OCI	3	1	OC invalidation bit.	RW
	Operation		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.	
	Power-on reset		0	

Table 18: CCR register description

CCR				
Field	Bits	Size	Synopsis	Type
ORA	5	1	OC RAM enable bit.	RW
	Operation		0: Normal mode (all of OC is used as cache). 1: RAM mode (half of OC is used as cache, the other half is used as RAM. Please refer to <a href="#">Section 4.3.6</a> ).	
	Power-on reset		0	
OIX	7	1	OC index enable bit.	RW
	Operation		0: Address bits [13:5] used for OC entry selection. 1: Address bits [25] and [12:5] used for OC entry selection. Note: In SH4-202, when CCR.ORA is set to 1, CCR.OIX must be set to 0. Please refer to <a href="#">Section 4.3.7</a> .	
	Power-on reset		0	
ICE	8	1	IC enable bit.	RW
	Operation		Indicates whether or not the IC is to be used. When address translation is to be 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.	
	Power-on reset		0	
ICI	11	1	IC invalidation bit.	RW
	Operation		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.	
	Power-on reset		0	
IIX	15	1	IC index enable bit.	RW
	Operation		0: Address bits [12:5] used for IC entry selection. 1: Address bits [25] and [11:5] used for IC entry selection.	
	Power-on reset		0	

Table 18: CCR register description

CCR				
Field	Bits	Size	Synopsis	Type
EMODE	31	1	Enhanced mode <b>SH4-202 only</b> .	RW
	Operation		Indicates whether or not the OC is to be used in enhanced mode. 0: Compatible mode*. 1: Enhanced mode. *: SH4-202 is not compatible with SH4-103 in the following conditions: 1. OC index mode and RAM mode. 2. Address map in RAM mode.	
	Power-on reset		0	
Reserved bits	4, 6, [10:9] [14:12] [30:16]	23	For maximum forward compatibility preserve values on write, otherwise write 0. Read is undefined.	
	Power-on reset			

Table 18: CCR register description

## 4.2.2 Queue address control register 0 (QACR0)

QACR0 can be accessed by longword-size access from 0xFF000038 in the P4 region, and 0x1F000038 in Area 7.

QACR0				
Field	Bits	Size	Synopsis	Type
Area	[2,4]	3	Queue address control register 0.	RW
	Operation		QACR0 specifies the area onto which store queue 0 (SQ0) is mapped when the MMU is off.	
	Power-on reset		Undefined	

Table 19: QACR0



QACR0				
Field	Bits	Size	Synopsis	Type
Reserved bits	[0,1], [5,31]	29		
	Power-on reset			

Table 19: QACR0

### 4.2.3 Queue address control register 1 (QACR1)

QACR1 can be accessed by longword-size access from 0xFF00003C in the P4 region, and 0x1F00003C in Area 7.

QACR1				
Field	Bits	Size	Synopsis	Type
Area	[2,4]	3	Queue address control register 1.	RW
	Operation		QACR1 specifies the area onto which store queue 1 (SQ1) is mapped when the MMU is off.	
	Power-on reset		Undefined	
Reserved bits	[0,1], [5,31]	29		
	Power-on reset			

Table 20: QACR1

## 4.3 Operand cache (OC)

### 4.3.1 Configuration

*Figure 18* shows the configuration of the operand cache for the SH4-103 while *Figure 19* shows the same for the SH4-202.

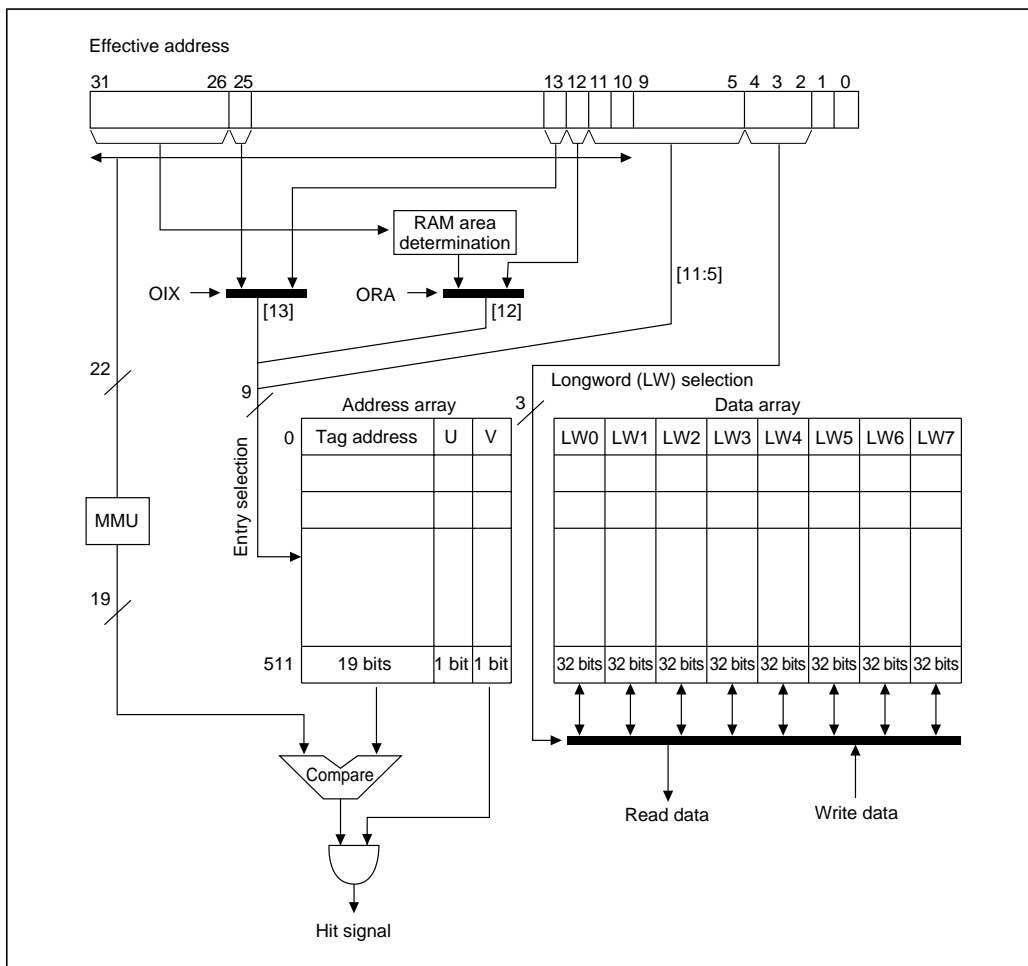
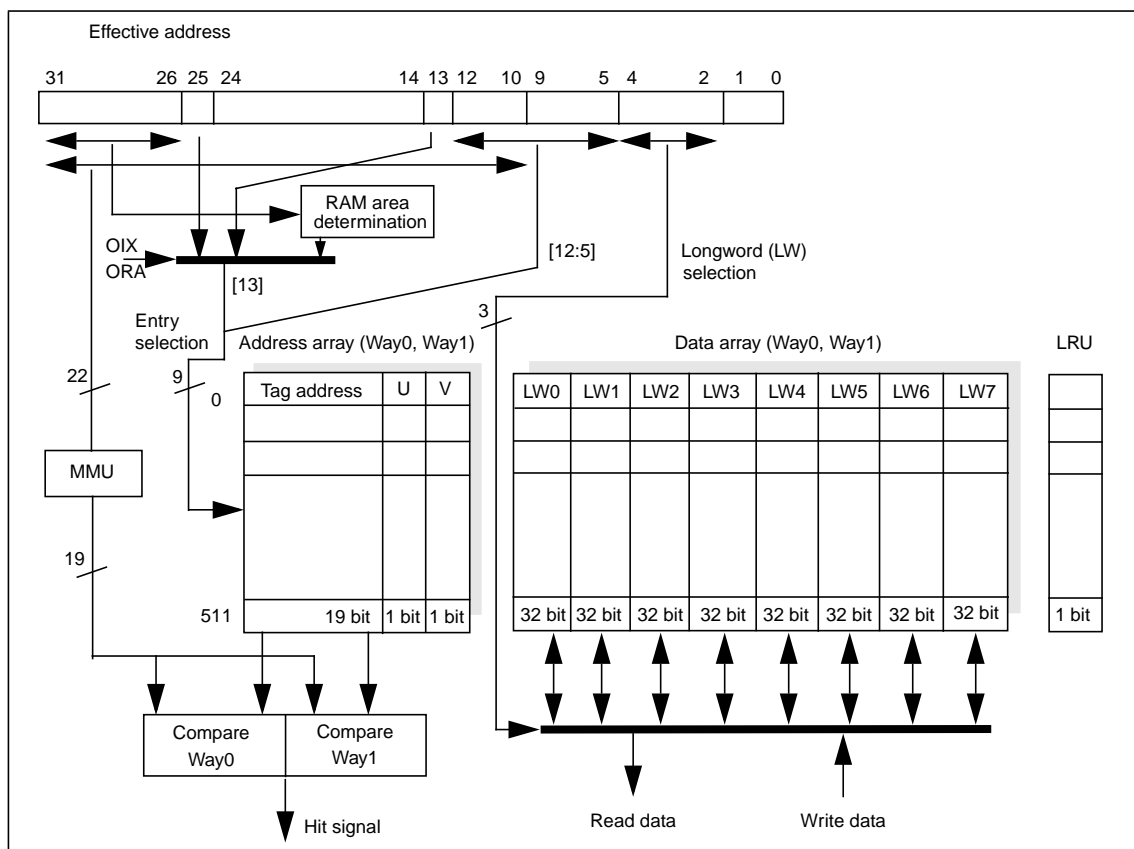


Figure 18: Configuration of operand cache on SH4-103 and SH-202 in compatibility mode



**Figure 19: Configuration of instruction cache on the SH4-202**

The operand cache for the SH4-103 consists of 512 cache lines, each composed of a 19-bit tag, V bit, U bit, and 32-byte data.

The SH4-202 operand cache is 2 way associative cache and consists of 512 cache lines/way, each composed of a 19-bit tag, V bit and 32-byte data.

- Tag

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

- V bit (validity bit)

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

- U bit (dirty bit)

The U bit is set to 1 if data is written to the cache line, while the cache is being used in copy-back mode, that is the U bit indicates a mismatch between the data in the cache line and the data in external memory. The U bit is never set to 1 while the cache is being used in write-through mode, unless it is modified by accessing the memory-mapped cache (see [Section 4.5: Memory-mapped cache configuration on page 95](#)). 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.

- LRU (SH-4 200 series only)

When a 200 series SH-4 is operating in enhanced mode, an additional state bit is deployed to keep track of which of the two ways in each cache set was least recently used (LRU). These additional LRU bits can not be read or written by software.

### 4.3.2 Read operation

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

- 1 The tag, V bit, and U bit are read from the cache line, indexed by effective address bits [13:5].
- 2 The tag is compared with bits [28:10] of the address resulting from effective address translation by the MMU. Operation is as described in [Table 2](#).

Tag match	V bit	U bit	Operation	Description
Yes	1	-	Cache hit	The data indexed by bits [4:0] of the effective address, is read from the cache line indexed by bits [13:5], in accordance with the access size (quadword/longword/word/byte).
Yes	0	-	Cache miss (no write-back)	Data from the external memory space, corresponding to the effective address, is written into the cache line. Data reading is performed, using the critical word first method, and when the data arrives in the cache, the read data is returned to the CPU. The CPU continues to execute the next process, while the cache line of data is being read. When reading of one line of data is completed, the tag corresponding to the effective address is recorded in the cache, and the V bit is set to 1.
No	0	-		
No	1	0		
No	1	1	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 from the external memory space, corresponding to the effective address, is written into the cache line. Data reading is performed, using the critical word first method, and when the data arrives in the cache, the read data is returned to the CPU. The CPU continues to execute the next process, while the cache line of data is being read. When reading of one line of data is completed, the tag corresponding to the effective address is recorded in the cache, the V bit is set to 1, and the U bit is set to 0. The data in the write-back buffer is then written back to the external memory.

Table 21: OC read operation

### 4.3.3 Write operation

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

- 1 The tag, V bit, and U bit are read from the cache line indexed by effective address bits [13:5].
- 2 The tag is compared with bits [28:10] of the address resulting from effective address translation by the MMU. In copy back, operation is per [Table 22](#). In write through mode it is per [Table 23](#).

Tag match	V bit	U bit	Operation	Description
Yes	1	-	Cache hit (copy-back)	A data write for the data indexed by bits [4:0] is performed, in accordance with the access size (quadword/longword/word/byte).
Yes	0	-	Cache miss (no copy-back/write-back)	A data write for the data indexed by bits [4:0] is performed, in accordance with the access size (quadword/longword/word/byte). Then, data from the external memory corresponding to the effective address, is read into the cache line. Data reading is performed, using the critical word first method, and one cache line of data is read, excluding the written data. The CPU continues to execute the next process, while the cache line of data is being read. When reading of one line of data is completed, the tag corresponding to the effective address is recorded in the cache, the V bit and U bit are both set to 1.
No	0	-		
No	1	0		

Table 22: OC write operation, with copy-back

Tag match	V bit	U bit	Operation	Description
No	1	1	Cache miss (with copy-back/write-back)	The tag and data field of the cache line, indexed by effective address bits [13:5] are first saved in the write-back buffer. Then, a data write for the data indexed by bits [4:0], is performed in accordance with the access size (quadword/longword/word/byte). Data from the external memory space, corresponding to the effective address, is read into the cache line. Data reading is performed, using the critical word first method, and one cache line of data is read, excluding the written data. The CPU continues to execute the next process, while the cache line of data is being read. When reading of one line of data is completed, the tag corresponding to the effective address is recorded in the cache, the V bit and U bit are both set to 1. The data in the write back buffer is then written back to external memory.

Table 22: OC write operation, with copy-back

Tag match	V bit	U bit	Operation	Description
Yes	1	-	Cache-hit (write-through)	A data write for the data indexed by bits [4:0], is performed in accordance with the access size (quadword/longword/word/byte). The U bit is set to 1.
Yes	0	-	Cache miss (write-through)	A write is performed to the external memory, corresponding to the effective address. A write to cache is not performed.
No	0	-		
No	1	0		
No	1	1		

Table 23: OC write operation, with write-through

### 4.3.4 Write-back buffer

The write-back buffer enables priority to be given to data reads, and improves performance. When a cache miss makes the purge of a dirty cache entry into external memory necessary, the cache entry is held in the write-back buffer. The write-back buffer contains one cache line of data and the physical address of the purge destination.

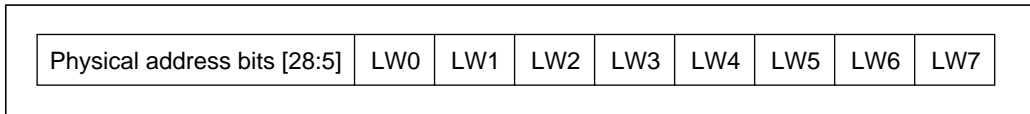


Figure 20: Configuration of write-back buffer

### 4.3.5 Write-through buffer

When writing data in write-through mode or writing to a non-cacheable area, data is held in a 64-bit buffer. 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.

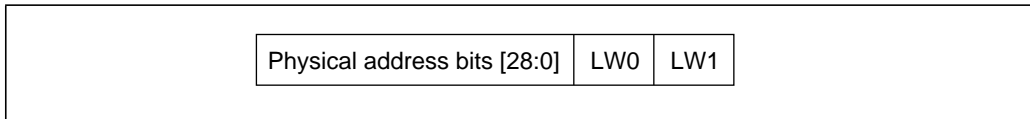


Figure 21: Configuration of write-through buffer

### 4.3.6 RAM mode

#### SH-4 100 series

Setting CCR.ORA to 1 enables 8 kbytes of the operand cache to be used as RAM. The operand cache entries used as RAM are, entries 128 to 255 and 384 to 511. Other entries can still be used as cache. RAM can be accessed using addresses 0x7C00 0000 to 0x7FFF FFFF. Byte-, word-, longword-, and quadword-size data reads and writes can be performed in the operand cache RAM area. Instruction fetches cannot be performed in this area.

*Note:* On the SH4-202, RAM mode cannot be used in conjunction with OC index mode even when in compatibility mode.



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

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

Address start	Address end	Size	RAM area
0x7C00 0000	0x7C00 0FFF	4-kbytes	1
0x7C00 1000	0x7C00 1FFF		1
0x7C00 2000	0x7C00 2FFF		2
0x7C00 3000	0x7C00 3FFF		2
0x7C00 4000	0x7C00 4FFF		1 <sup>a</sup>

**Table 24: RAM use when OC index mode is off**

- a. RAM areas 1 and 2 then repeat every 8Kbytes up to 0x7FFF FFFF.

Thus, to secure a continuous 8-kbyte RAM area, the area from 0x7C00 1000 to 0x7C00 2FFF can be used, for example.

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

Address start	Address end	Size	RAM area
0x7C00 0000	0x7C00 0FFF	4-kbytes	1
0x7C00 1000	0x7C00 1FFF		1
0x7C00 2000	0x7C00 2FFF		1
...	...		1
0x7DFF F000	0x7DFF FFFF		1
0x7E00 0000	0x7E00 0FFF		2
0x7E00 1000	0x7E00 1FFF		2
...	...		2
0x7FFF F000	0x7FFF FFFF		2

**Table 25: RAM use when OC index mode is on**

As the distinction between RAM areas 1 and 2 is indicated by address bit [25], the area from 0x7DFF F000 to 0x7E00 0FFF should be used to secure a continuous 8-kbyte RAM area.

### RAM Mode of SH4-202

Setting CCR.ORA to 1 enables half of the operand cache to be used as RAM. The operand cache entries used as RAM are entries 256 to 511 in the compatible mode. The operand cache entries used as RAM are entries 256 to 511 of each way in the enhanced mode. Other entries can still be used as cache. RAM can be accessed using addresses 0x7C00 0000 to 0x7FFF FFFF. Byte-, word-, longword-, and quadword-size data reads and writes can be performed in the operand cache RAM area. Instruction fetches cannot be performed in this area.

Even when in compatibility mode, the OC index mode cannot be used in conjunction with RAM mode on a 200 series part.

### RAM mode address map of SH4-202

An example of RAM use is shown below. Here, the 8 kbytes comprising OC entries 256 to 511 of way 0 are designated as RAM area 1, and the 8 kbytes comprising OC entries 256 to 511 of way 1 as RAM area 2.

In the compatible mode (CCR.EMODE=0)

Address start	Address end	Size	RAM area
0x7C00 0000	0x7C00 1FFF	8-Kbytes	256-511
0x7C00 2000	0x7C00 3FFF	8-Kbytes	256-511
Repeated until...			
0x7FFF E000	0x7FFF FFFF	8-Kbytes	256-511

**Table 26: Compatible mode**

In the enhanced mode (CCR.EMODE=1)

Address start	Address end	Size	RAM area
0x7C00 0000	0x7C00 1FFF	8-Kbytes	1
0x7C00 2000	0x7C00 3FFF	8-Kbytes	2
0x7C00 4000	0x7C00 5FFF	8-Kbytes	1
0x7C00 6000	0x7C00 7FFF	8-Kbytes	2
RAM areas 1 and 2 then repeat every 16Kbytes until...			
0x7FFF C000	0x7FFF FFFF	16-Kbytes	

**Table 27: Compatible mode**

### 4.3.7 OC index mode

OC index mode is only available on the SH-4 100 series or when a 200 series part is used in compatibility mode and RAM mode is not being used.

In normal mode, with CCR.OIX cleared to 0, OC indexing is performed using bits [13:5] of the effective address. Using index mode, with CCR.OIX set to 1, allows the OC to be handled as two 8-kbyte areas, by means of effective address bit [25]. This partitioning makes it possible for the software to make more efficient use of the cache.

### 4.3.8 Coherency between cache and external memory

Coherency between cache and external memory should be assured by software. In the SH-4 CPU core, the following four new instructions are supported for cache operations. Details of these instructions are given in the Instruction Descriptions chapter.

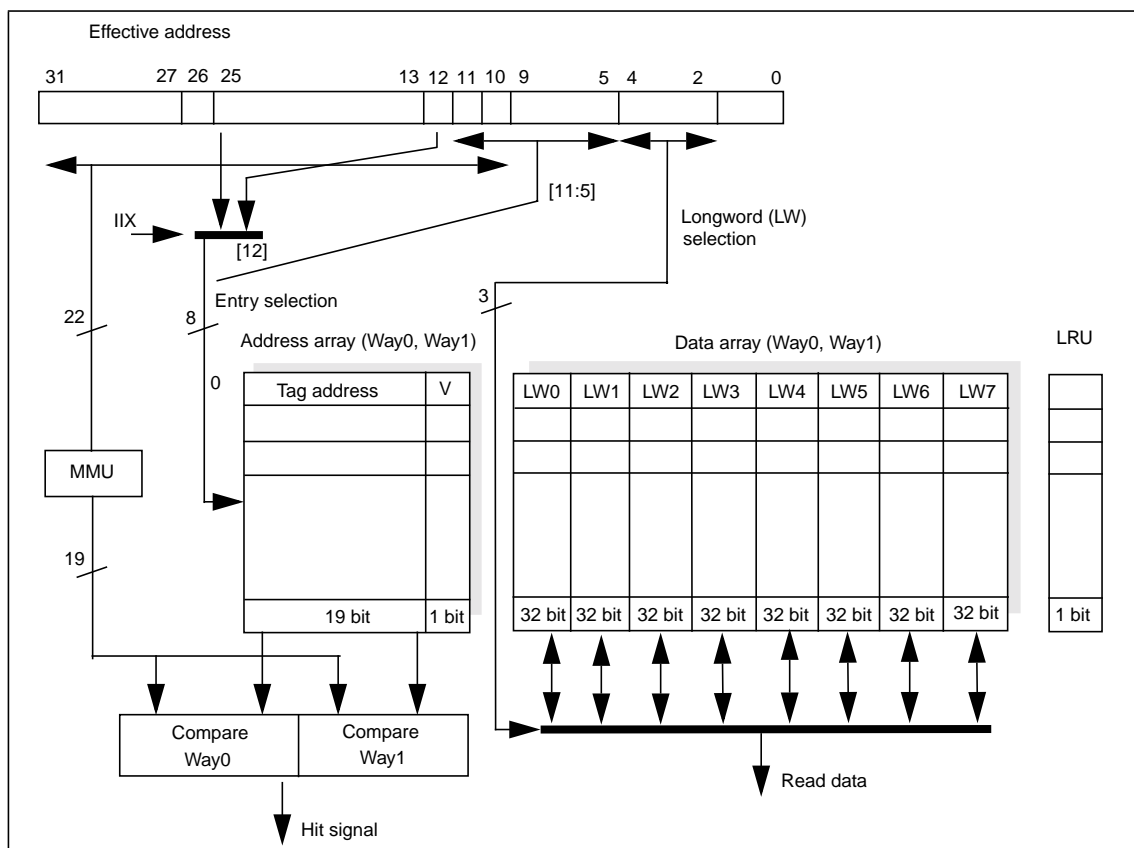
Invalidate instruction:	OCBI @Rn	Cache invalidation (no write-back)
Purge instruction:	OCBP @Rn	Cache invalidation (with write-back)
Write-back instruction:	OCBWB @Rn	Cache write-back
Allocate instruction:	MOVCA.L R0, @Rn	Cache allocation

### 4.3.9 Prefetch operation

The SH-4 CPU core 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 can be prevented by using the prefetch instruction to fill the cache with data before the operation, and so improve software performance. If a prefetch instruction is executed for data already held in the cache, or if the prefetch address results in a UTLB miss or a protection violation, the result is no operation, and an exception is not generated. Details of the prefetch instruction are given in the Instruction Descriptions chapter.

Prefetch instruction: PREF @Rn





**Figure 23: Configuration of instruction cache on the SH4-202 in enhanced mode (CCR.EMODE=1)**

The instruction cache for the SH4-103 consists of 256 cache lines, each composed of a 19-bit tag, V bit, and 32-byte data (16 instructions).

The instruction cache for the SH4-202 consists of 256 cache lines/way, each composed of a 19-bit tag, V bit, and 32-byte data (16 instructions).

- Tag

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

Setting this bit to 1 Indicates that valid data is stored in the cache line. 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.

- LRU (SH-4 200 series only)

When a 200 series SH-4 is operating in enhanced mode, an additional state bit is deployed to keep track of which of the two ways in each cache set was least recently used (LRU). These additional LRU bits can not be read or written by software.

## 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:

Tag	V bit	Operation	Description
Matches	1	Cache hit	Data indexed by effective address bits [4:2], is read as an instruction.
Matches	0	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 critical word first method, and when the data arrives in the cache, the read data is returned to the CPU as an instruction. When reading of one line of data is completed, the tag corresponding to the effective address is recorded in the cache, and 1 is written to the V bit.
Does not match	0		
Does not match	1		

Table 28: IC read operation

## 4.4.3 IC index mode

IC index mode is only available on the SH-4 100 series or when a 200 series part is used in compatibility mode.

In normal mode, with CCR.IIX cleared to 0, IC indexing is performed using bits [12:5] of the effective address. Using index mode, with CCR.IIX set to 1, allows the IC to be handled as two 4-kbyte areas by means of effective address bit [25]. This provides efficient use of the cache.

## 4.5 Memory-mapped cache configuration

To enable the IC and OC to be managed by software, IC content can be read and written by a P2 region program, with a MOV instruction in privileged mode. Behavior is undefined if access is made from a program in another region. In this case, a branch to the P0, U0, P1, or P3 regions should be made at least 8 instructions after this MOV instruction.

The OC content can be read and written by a P1 and P2 regions program, with a MOV instruction in privileged mode. Behavior is undefined if access is made from a program in another region. In this case, a branch to the P0, U0, or P3 regions should be made at least 8 instructions after this MOV instruction.

The IC and OC are allocated to the P4 region in physical memory space. Only (longword) data accesses can be used on both the IC address array and data array, and the OC address array and data array. Instruction fetches cannot be performed in these regions. For reserved bits, a write value of 0 should be specified; their read value is undefined.

### 4.5.1 IC address array

The IC address array is allocated to addresses 0xF000 0000 to 0xF0FF FFFF in the P4 region. An address array access requires a 32-bit address field specification (when reading or writing), and a 32-bit data field specification. The entry to be accessed is specified in the address field, and the write tag and V bit are specified in the data field.

In the address field, bits [31:24] have the value 0xF0 indicating the IC address array, and the entry is specified by bits [12:5]. CCR.IIX has no effect on this entry specification. The address array bit [3], the association bit (A bit), specifies whether or not association is performed when writing to the IC address array. As only longword access is used, 0 should be specified for address field bits [1:0].

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

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

1 IC address array read

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

2 IC address array write (non-associative)

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

3 IC address array write (associative)

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

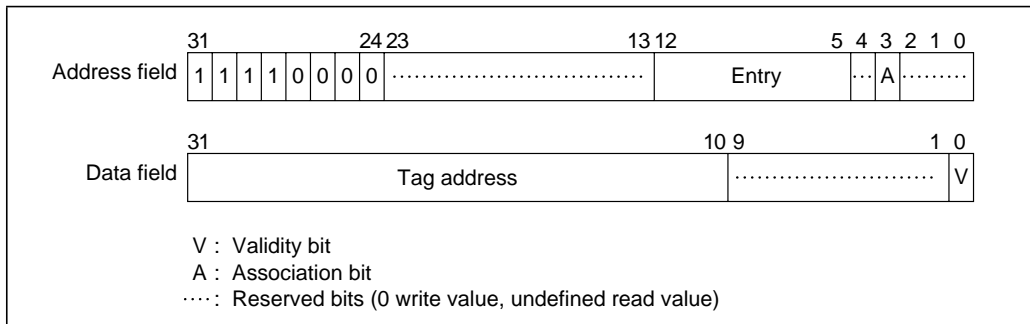


Figure 24: Memory-mapped IC address array



## 4.5.4 IC data array

The IC data array is allocated to addresses 0xF100 0000 to 0xF1FF FFFF in the P4 region. A data array access requires a 32-bit address field specification (when reading or writing), and a 32-bit data field specification. The entry to be accessed is specified in the address field, and the longword data to be written is specified in the data field.

In the address field, bits [31:24] have the value 0xF1 indicating the IC data array, and the entry is specified by bits [12:5]. CCR.IIX has no effect on this entry specification. Address field bits [4:2] are used for the longword data specification in the entry. As only longword access is used, 0 should be specified for address field bits [1:0].

The data field is used for the longword data specification.

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

### 1 IC data array read

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

### 2 IC data array write

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

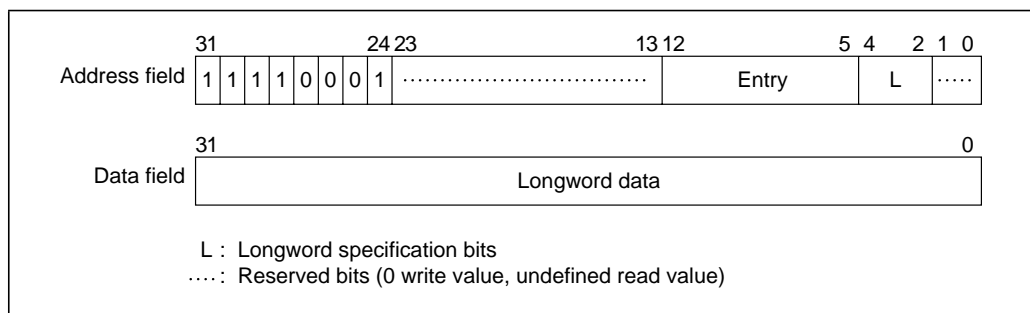


Figure 25: Memory-mapped IC data array

## 4.5.5 OC address array

The OC address array is allocated to addresses 0xF400 0000 to 0xF4FF FFFF in the P4 region. An address array access requires a 32-bit address field specification (when reading or writing), and a 32-bit data field specification. The entry to be accessed is specified in the address field, and the write tag, U bit, and V bit are specified in the data field.

In the address field, bits [31:24] have the value 0xF4 indicating the OC address array, and the entry is specified by bits [13:5]. CCR.OIX and CCR.ORA have no effect on this entry specification. The address array bit [3], association bit (A bit), specifies whether or not association is performed when writing to the OC address array. As only longword access is used, 0 should be specified for address field bits [1:0].

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

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

### 1 OC address array read

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

### 2 OC address array write (non-associative)

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

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

### 3 OC address array write (associative)

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

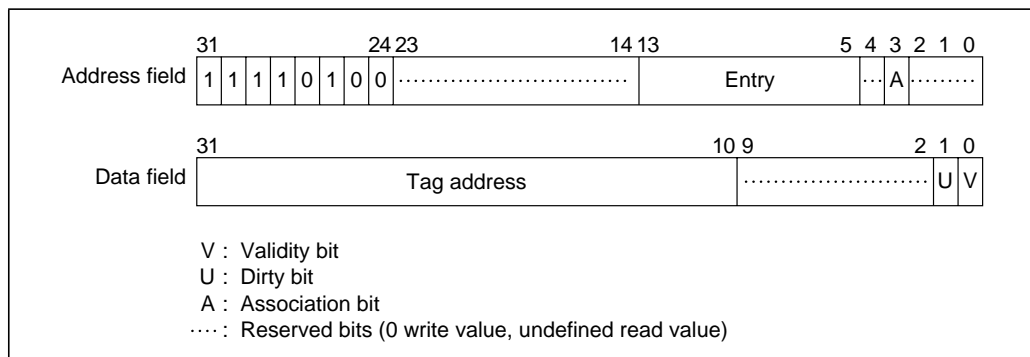


Figure 26: Memory-mapped OC address array

## 4.5.6 OC data array

The OC data array is allocated to addresses 0xF500 0000 to 0xF5FF FFFF in the P4 region. A data array access requires a 32-bit address field specification (when reading or writing), and a 32-bit data field specification. The entry to be accessed is specified in the address field, and the longword data to be written is specified in the data field.

In the address field, bits [31:24] have the value 0xF5 indicating the OC data array, and the entry is specified by bits [13:5]. CCR.OIX and CCR.ORA have no effect on this entry specification. Address field bits [4:2] are used for the longword data specification in the entry. As only longword access is used, 0 should be specified for address field bits [1:0].

The data field is used for the longword data specification.

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

### 1 OC data array read

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

### 2 OC data array write

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

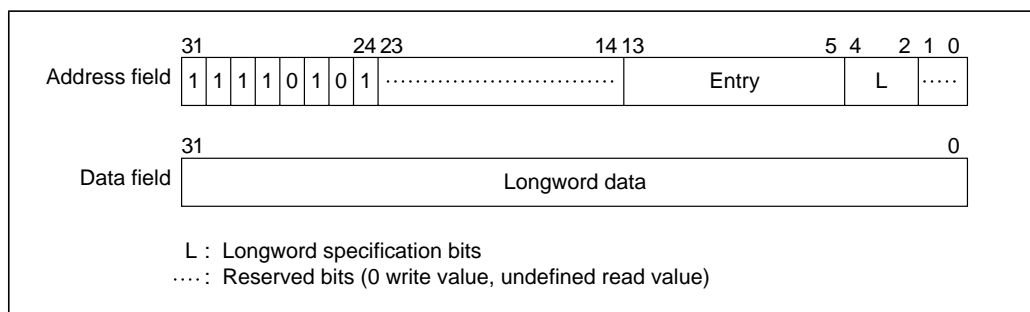


Figure 27: Memory-mapped OC data array

### Memory-mapped OC configuration in the enhanced mode (SH4-202)

- Normal mode (0xF500 3FFF (16 Kbyte): Corresponds to Way0 (entry 0 - 511)  
 0xF500 4000 to 0xF500 7FFF (16 Kbyte): Corresponds to Way1 (entry 0 - 511)  
 :::  
 Cache area then repeat every 32 kbytes up to 0xF5FF FFFF.
- RAM mode (CCR.ORA=1)  
 0xF500 0000 to 0xF500 1FFF (8 Kbyte): Corresponds to Way0 (entry 0 - 255)  
 0xF500 2000 to 0xF500 3FFF (8 Kbyte): Corresponds to Way1 (entry 0 - 255)  
 :::  
 Cache area then repeat every 16 kbytes up to 0xF5FF FFFF.

## 4.6 Store queues

Two 32-byte store queues (SQs) are supported to perform high-speed writes to external memory. When not using the SQs, the low power dissipation power-down modes, in which SQ functions are stopped, can be used. The queue address control registers (QACR0 and QACR1) cannot be accessed while SQ functions are stopped. Refer to the product level documentation of clock and power management for the details on stopping SQ functions.

Item	Store queues
Capacity	2 * 32
Addresses	0xE000 0000 to 0xE3FF FFFF
Write	Store instruction (1-cycle write)
Write-back	Prefetch instruction
Access right	MMU off: according to MMUCR.SQMD MMU on: according to individual page PR

Table 29: Store queue features

### 4.6.1 SQ configuration

There are two 32-byte store queues, SQ0 and SQ1, as shown in *Figure 28*. These two store queues can be set independently.

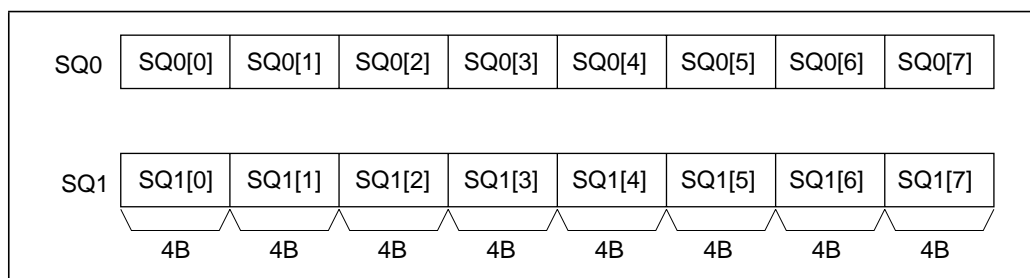


Figure 28: Store queue configuration

## 4.6.2 SQ writes

A write to the SQs can be performed using a store instruction on P4 area 0xE000 0000 to 0xE3FF FFFC. A longword or quadword access size can be used. The meaning of the address bits is as follows:

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

## 4.6.3 SQ reads (implementation dependant)

A read from the SQs can be performed using a load instruction on P4 area 0xFF00 1000 to 0xFF00 103C. A longword access size must be used. The meaning of the address bits is as follows:

[31:6]:	0xFF00100	Store queue specification
[5]:	0/1	0: SQ0 specification      1: SQ1 specification
[4:2]:	LW specification	Specifies longword position in SQ0/SQ1
[1:0]:	00	Fixed at 0

## 4.6.4 Transfer to external memory

Transfer from the SQs to external memory can be performed with the prefetch instruction (PREF). Issuing a PREF instruction for 0xE000 0000 to 0xE3FF FFFC in the P4 region, starts a burst transfer from the SQs to external memory. The burst transfer has a fixed length of 32 bytes, and the start address must be 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 incurring a penalty cycle. A write to the SQ being transferred to external memory is suspended until the transfer to external memory is completed.

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

- When MMU is on

The SQ area (0xE000 0000 to 0xE3FF FFFF) is set in VPN of the UTLB, and the transfer destination external address is set in PPN. The ASID, V, SZ, SH, PR, and D bits have the same meaning as for normal address translation, but the C and WT bits have no meaning with regard to this page.

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

If SQ access is enabled by MMUCR.SQMD, in privileged mode only, an address error will be flagged in user mode, even if address translation is successful.

- When MMU is off
- The SQ area (0xE000 0000 to 0xE3FF FFFF) is specified as the address at which a prefetch is performed. The meaning of address bits [31:0] is as follows:

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

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

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

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

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

### Determination of SQ access exception

Determination of an exception in a write to an SQ or transfer to external memory (PREF instruction) is performed as follows according to whether the MMU is on or off. In the SH7751, if an exception occurs in as SQ Write, the SQ contents may be corrupted. In the SH7751R, if an exception occurs in as SQ Write, SQ write access is cancelled and the data before the SQ write access is kept. If an exception occurs in transfer from an SQ to external memory, the transfer to external memory will be aborted.

- When MMU is on

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

- When MMU is off

Operation is in accordance with MMUCR.SQMD.

0: Privileged/user access possible

1: Privileged access possible

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





# Exceptions

## 5.1 Overview

The process of responding to an extraordinary event such as a reset, a general exception (trap) or an interrupt, is called exception handling.

Exception handling is performed by user supplied special routines, that are executed by the CPU when one of these extraordinary events is encountered.

## 5.2 Register descriptions

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

Name	Abbreviation	R/W	Initial value <sup>a</sup>	P4 address <sup>b</sup>	Area 7 address <sup>B</sup>	Access size
TRAPA exception register	TRA	R/W	Undefined	0xFF00 0020	0x1F00 0020	32
Exception event register	EXPEVT	R/W	0x0000 0000/ 0x0000 0020 <sup>A</sup>	0xFF00 0024	0x1F00 0024	32
Interrupt event register	INTEVT	R/W	Undefined	0xFF00 0028	0x1F00 0028	32

**Table 30: Exception-related registers**

- a. 0x0000 0000 is set in a power-on reset, and 0x0000 0020 in a manual reset.
- b. This is the address when using the virtual/physical address space P4 area. When making an access from physical address space area 7 using the TLB, the upper 3 bits of the address are ignored.

## 5.2.1 Exception event register (EXPEVT)

The exception event register (EXPEVT) resides at P4 address 0xFF00 0024, and contains a 12-bit exception code. The exception code set in EXPEVT is that for a reset or general exception event. The exception code is set automatically by hardware when an exception occurs. EXPEVT can also be modified by software.

EXPEVT				
Field	Bits	Size	Synopsis	Type
Exception code	[0,11]	12	Exception code	RW
	Operation		Exception code set automatically by hardware when exception occurs.	
	Power-on reset		Undefined	
RES	[12,31]	20	Bits reserved	RW
	Power-on reset		Undefined	

Table 31: EXPEVT Register Description

## 5.2.2 Interrupt event register (INTEVT)

The interrupt event register (INTEVT) resides at P4 address 0xFF00 0028, and contains a 12-bit exception code. The exception code set in INTEVT is that for an interrupt request. The exception code is set automatically by hardware when an exception occurs. INTEVT can also be modified by software.

INTEVT				
Field	Bits	Size	Synopsis	Type
Exception code	[0,11]	12	Exception code	RW
	Operation		Exception code set automatically by hardware when exception occurs.	
	Power-on reset		Undefined	

Table 32: INTEVT Register Description

INTEVT				
Field	Bits	Size	Synopsis	Type
RES	[12,31]	20	Bits reserved	RW
	Power-on reset		Undefined	

Table 32: INTEVT Register Description

### 5.2.3 TRAPA exception register (TRA)

The TRAPA exception register (TRA) resides at P4 address 0xFF00 0020. TRA is set automatically by hardware when a TRAPA instruction is executed. TRA can also be modified by software.

TRA				
Field	Bits	Size	Synopsis	Type
Imm	[2,9]	8	8-bit immediate data for the TRAPA instruction.	RW
	Operation			
	Power-on reset		Undefined	
RES	[0,1], [10,31]	24	Bits reserved	RW
	Power-on reset		Undefined	

Table 33: TRA

## 5.3 Exception handling functions

### 5.3.1 Exception handling flow

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

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

- 1 The PC, SR and R15 contents are saved in SPC, SSR and SGR.
- 2 The block bit (BL) in SR is set to 1.
- 3 The mode bit (MD) in SR is set to 1.
- 4 The register bank bit (RB) in SR is set to 1.
- 5 In a reset, the FPU disable bit (FD) in SR is cleared to 0.
- 6 The exception code is written to bits 11 to 0 of the exception event register (EXPEVT), or to bits 13 to 0 of the interrupt event register (INTEVT).
- 7 The CPU branches to the determined exception handling vector address, and the exception handling routine begins.

### 5.3.2 Exception handling vector addresses

The reset vector address is fixed at 0xA000 0000. Exception and interrupt vector addresses are determined by adding the offset for the specific event, to the vector base address, which is set by software in the vector base register (VBR). In the case of the TLB miss exception, for example, the offset is 0x0000 0400, so if 0x9C08 0000 is set in VBR, the exception handling vector address will be 0x9C08 0400. If a further exception occurs at the exception handling vector address, a duplicate exception will result, and recovery will be difficult; therefore, fixed physical addresses (P1, P2) should be specified for vector addresses.

## 5.4 Exception types and priorities

*Table 34* shows the types of exceptions, with their relative priorities, vector addresses, and exception/interrupt codes.

Exception category	Execution mode	Exception	Priority level	Priority order	Vector address	Offset	Exception code
Reset	Abort type	POWERON	1	1	0xA000 0000	-	0x000
		MANRESET	1	2	0xA000 0000	-	0x020
		HUDIRESET	1	1	0xA000 0000	-	0x000
		ITLBMULTIHIT	1	3	0xA000 0000	-	0x140
		OTLBMULTIHIT	1	4	0xA000 0000	-	0x140
General exception	Re-execution type	UBRKBEFORE* <sup>1</sup>	2	0	(VBR/DBR)	0x100/-	0x1E0
		IADDERR	2	1	(VBR)	0x100	0x0E0
		ITLBMISS	2	2	(VBR)	0x400	0x040
		EXECPROT	2	3	(VBR)	0x100	0x0A0
		RESINST	2	4	(VBR)	0x100	0x180
		ILLSLOT	2	4	(VBR)	0x100	0x1A0
		FPUDIS	2	4	(VBR)	0x100	0x800
		SLOTFPUDIS	2	4	(VBR)	0x100	0x820
		RADDERR	2	5	(VBR)	0x100	0x0E0
		WADDERR	2	5	(VBR)	0x100	0x100
		RTLBMISS	2	6	(VBR)	0x400	0x040
		WTLBMISS	2	6	(VBR)	0x400	0x060
		READPROT	2	7	(VBR)	0x100	0x0A0
		WRITEPROT	2	7	(VBR)	0x100	0x0C0
		FPUExc	2	8	(VBR)	0x100	0x120
		FIRSTWRITE	2	9	(VBR)	0x100	0x080

**Table 34: Exceptions**

Exception category	Execution mode	Exception	Priority level	Priority order	Vector address	Offset	Exception code
Interrupt	Completion type	TRAP	2	4	(VBR)	0x100	0x160
		UBRKAFTER* <sup>A</sup>	2	10	(VBR/DBR)	0x100/-	0x1E0
		NMI <sup>b</sup>	3	-	(VBR)	0x600	0x1C0
		IRLINT <sup>b</sup>	4	a	(VBR)	0x600	See the system manual <sup>b</sup>
		PERIPHINT <sup>b</sup>	4	A	(VBR)	0x600	

Table 34: Exceptions

- The priority order of external interrupts and peripheral module interrupts can be set by software.
- The set of peripheral interrupts is system-dependent. See the Interrupt chapter in the *System Architecture Manual* for the list of peripheral interrupts and their corresponding INTEVT codes.

**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 0xA000 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 to IRL0).

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

*Note:* When  $BRCR.UBDE = 1$ ,  $PC = DBR$ . In other cases,  $PC = VBR + 0x100$ .

## 5.5 Exception flow

### 5.5.1 Exception flow

Figure 29 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. 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. For details, see section 5.6, Description of Exceptions. Also, see [Section 5.6.4](#), 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.

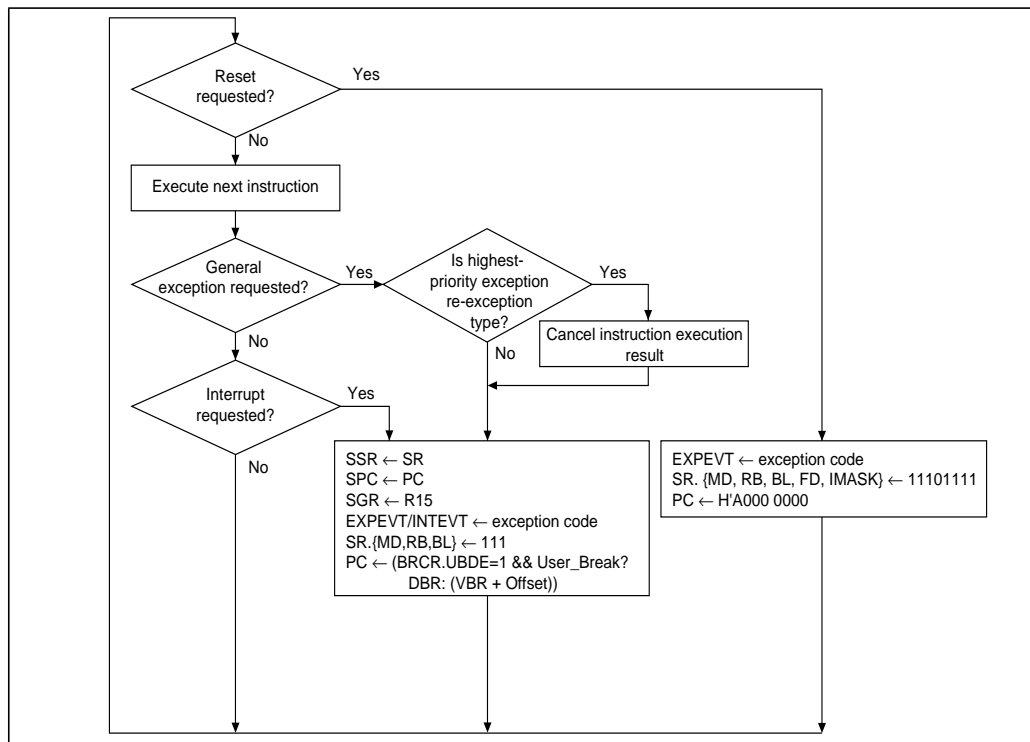


Figure 29: Instruction execution and exception handling

## 5.5.2 Exception source acceptance

A priority ranking is provided for all exceptions, for use in determining which of two or more simultaneously generated exceptions should be accepted. Five of the general exceptions:

- general illegal instruction exception
- slot illegal instruction exception
- general FPU disable exception
- slot FPU disable exception
- unconditional trap exception

are detected in the process of instruction decoding, and do not occur simultaneously in the instruction pipeline. Therefore, these exceptions all have the same priority. General exceptions are detected in the order of instruction execution. However, exception handling is performed in the order of instruction flow (program order). Thus, an exception for an earlier instruction is accepted before that for a later instruction. An example of the order of acceptance for general exceptions is shown in [Figure 30](#).



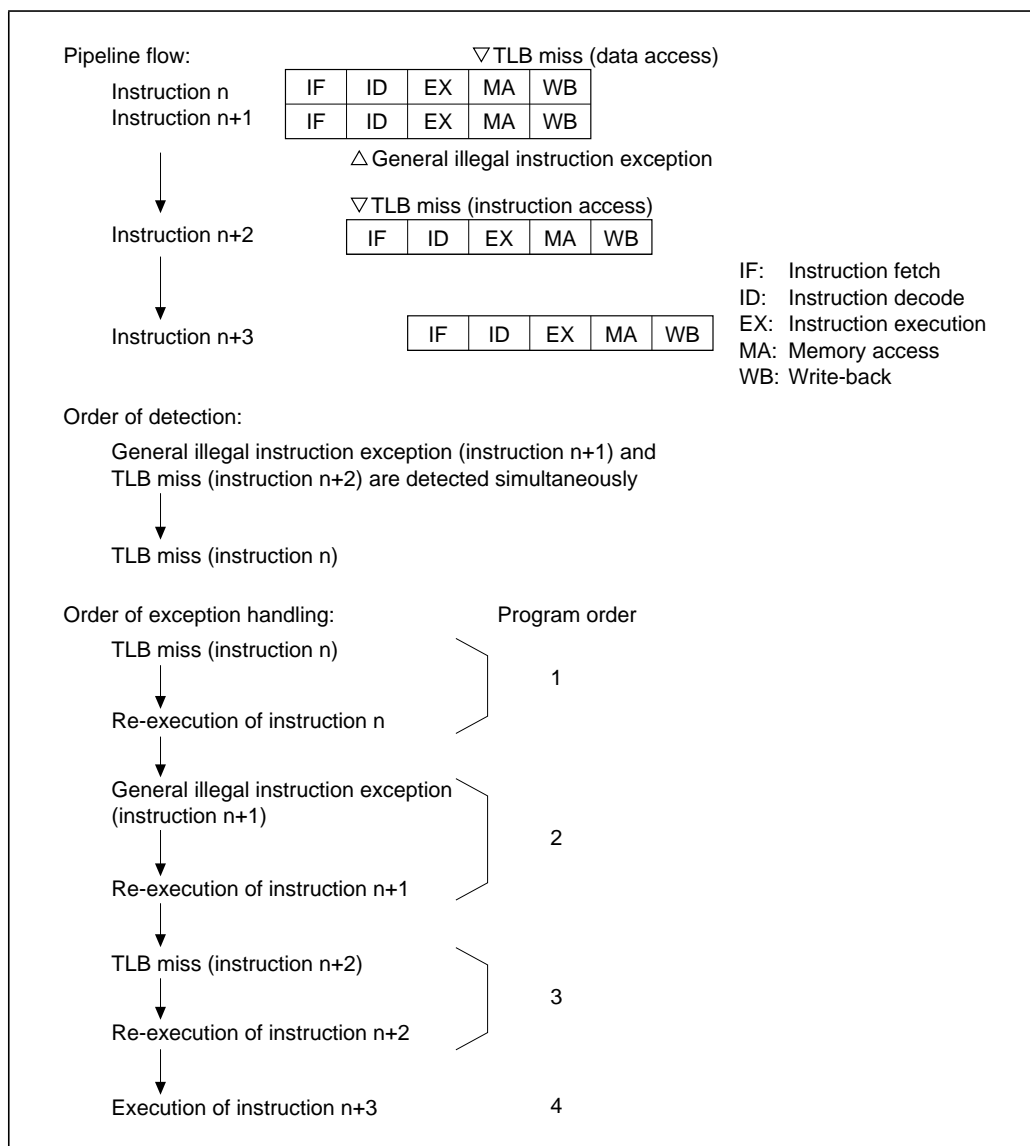


Figure 30: Example of general exception acceptance order

### 5.5.3 Exception requests and BL bit

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

When the BL bit in SR is 1 and an exception other than a user break is generated, the CPUs 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 (0xA000 0000). For the operation in the event of a user break, see 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, SPC and SSR are saved and then the BL bit in SR is cleared to 0, to enable multiple exception state acceptance.

### 5.5.4 Return from exception handling

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

## 5.6 Description of exceptions

The various exception handling operations are described here, covering exception sources, transition addresses, and processor operation, when a transition is made.

### 5.6.1 Resets

#### 1 POWERON - Power-On Reset

- Sources:  
For details of how the core is driven to the power on reset state, refer to the *System Architecture Manual* of the appropriate product.
- Transition address: 0xA000 0000
- Transition operations:  
Exception code 0x000 is set in EXPEVT, initialization of VBR and SR is performed, and a branch is made to PC = 0xA000 0000. In the initialization processing, the VBR register is set to 0x0000 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 0xF.

CPU initialization is performed. For details of the impact on the rest of the system refer to the *System Architecture Manual*.

Refer to Appendix A for power-on reset values for the various CPU core modules set by the Initialize\_Module function.

```
POWERON( )
{
    Initialize_Module(PowerOn);
    EXPEVT = 0x00000000;
    VBR = 0x00000000;
    SR.MD = 1;
    SR.RB = 1;
    SR.BL = 1;
    SR.(I0-I3) = 0xF;
    SR.FD=0;
    PC = 0xA0000000;
}
```

**MANRESET - Manual Reset**

- Sources:  
When a general exception other than a user break occurs while the BL bit is set to 1 in SR. It is also possible for the system in which the core is integrated to drive the processor in to this reset state. For details refer to the *System Architecture Manual* of the appropriate product.
- Transition address: 0xA000 0000
- Transition operations:

Exception code 0x020 is set in EXPEVT, initialization of VBR and SR is performed, and a branch is made to PC = 0xA000 0000. In the initialization processing, the VBR register is set to 0x0000 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 0xF. CPU and system initialization are performed. For details refer to the *System Architecture Manual*.

Refer to Appendix A for the manual reset values for the various CPU core modules set by the Initialize\_Module function.

```
MANRESET( )
{
    Initialize_Module(Manual);
    EXPEVT = 0x00000020;
    VBR = 0x00000000;
    SR.MD = 1;
    SR.RB = 1;
    SR.BL = 1;
    SR.(I0-I3) = 0xF;
    SR.FD = 0;
    PC = 0xA0000000;
}
```

## 2 HUDIRESET - H-UDI Reset

- Source:  
Refer to the *System Architecture Manual* for a description of how the core is placed in the H-UDI reset state.
- Transition address: 0xA000 0000

Transition operations:

Exception code 0x000 is set in EXPEVT, initialization of VBR and SR is performed, and a branch is made to PC = 0xA000 0000. In the initialization processing, the VBR register is set to 0x0000 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 0xF. CPU and system initialization are performed, for details refer to the *System Architecture Manual*.

Refer to Appendix A for the manual reset values for the various CPU core modules set by the Initialize\_Module function.

```

HUDIRESET( )
{
    Initialize_Module(PowerOn);
    EXPEVT = 0x00000000;
    VBR = 0x00000000;
    SR.MD = 1;
    SR.RB = 1;
    SR.BL = 1;
    SR.(I0-I3) = 0xF;
    SR.FD = 0;
    PC = 0xA0000000;
}

```

### 3 ITLBMULTIHIT - Instruction TLB Multiple-Hit Exception

- Source: Multiple ITLB address matches
- Transition address: 0xA000 0000
- Transition operations:  
The virtual address (32 bits) at which this exception occurred is set in TEA, and the corresponding virtual page number (22 bits) is set in PTEH [31:10]. ASID in PTEH indicates the ASID when this exception occurred.

Exception code 0x140 is set in EXPEVT, initialization of VBR and SR is performed, and a branch is made to PC = 0xA000 0000.

In the initialization processing, the VBR register is set to 0x0000 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 0xF.

CPU and system initialization are performed in the same way as in a manual reset.

Refer to Appendix A for the manual reset values for the various CPU core modules set by the Initialize\_Module function.

```
ITLBMULTIHIT( )
{
    Initialize_Module(Manual);
    TEA = EXCEPTION_ADDRESS;
    PTEH.VPN = PAGE_NUMBER;
    EXPEVT = 0x00000140;
    VBR = 0x00000000;
    SR.MD = 1;
    SR.RB = 1;
    SR.BL = 1;
    SR.(I0-I3) = 0xF;
    SR.FD = 0;
    PC = 0xA0000000;
}
```

#### 4 OTLBMULTIHIT - Operand TLB Multiple-Hit Exception

- Source: Multiple UTLB address matches
- Transition address: 0xA000 0000

Transition operations:

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

Exception code 0x140 is set in EXPEVT, initialization of VBR and SR is performed, and a branch is made to PC = 0xA000 0000.

In the initialization processing, the VBR register is set to 0x0000 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 0xF.

CPU and system initialization are performed in the same way as in a manual reset.

Refer to Appendix A for the manual reset values for the various CPU core modules set by the Initialize\_Module function.

```
OTLBMULTIHIT( )
{
    Initialize_Module(Manual);
    TEA = EXCEPTION_ADDRESS;
    PTEH.VPN = PAGE_NUMBER;
    EXPEVT = 0x00000140;
    VBR = 0x00000000;
    SR.MD = 1;
    SR.RB = 1;
    SR.BL = 1;
    SR.(I0-I3) = 0xF;
    SR.FD = 0;
    PC = 0xA0000000;
}
```

## 5.6.2 General exceptions

### 1 RTLBMIS - Read Data TLB Miss Exception

- Source: Address mismatch in UTLB address comparison
- Transition address: VBR + 0x0000 0400
- Transition operations:

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

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

Exception code 0x040 is set in EXPEVT. The BL, MD, and RB bits are set to 1 in SR, and a branch is made to PC = VBR + 0x0400.

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

```
RTLBMIS ( )
{
    TEA = EXCEPTION_ADDRESS;
    PTEH.VPN = PAGE_NUMBER;
    SPC = PC;
    SSR = SR;
    SGR = R15;
    EXPEVT = 0x00000040;
    SR.MD = 1;
    SR.RB = 1;
    SR.BL = 1;
    PC = VBR + 0x00000400;
}
```



## 2 WTLBMISS - Write Data TLB Miss Exception

- Source: Address mismatch in UTLB address comparison
- Transition address:  $VBR + 0x0000\ 0400$
- Transition operations:  
The virtual address (32 bits) at which this exception occurred is set in TEA, and the corresponding virtual page number (22 bits) is set in PTEH [31:10]. ASID in PTEH indicates the ASID when this exception occurred.

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

Exception code 0x060 is set in EXPEVT. The BL, MD, and RB bits are set to 1 in SR, and a branch is made to  $PC = VBR + 0x0400$ .

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

```
WTLBMISS( )
{
    TEA = EXCEPTION_ADDRESS;
    PTEH.VPN = PAGE_NUMBER;
    SPC = PC;
    SSR = SR;
    SGR = R15;
    EXPEVT = 0x00000060;
    SR.MD = 1;
    SR.RB = 1;
    SR.BL = 1;
    PC = VBR + 0x00000400;
}
```

### 3 ITLBMISS - Instruction TLB Miss Exception

- Source: Address mismatch in ITLB address comparison
- Transition address: VBR + 0x0000 0400

- Transition operations:

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

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

Exception code 0x040 is set in EXPEVT. The BL, MD, and RB bits are set to 1 in SR, and a branch is made to PC = VBR + 0x0400.

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

```
ITLBMISS( )
{
    TEA = EXCEPTION_ADDRESS;
    PTEH.VPN = PAGE_NUMBER;
    SPC = PC;
    SSR = SR;
    SGR = R15;
    EXPEVT = 0x00000040;
    SR.MD = 1;
    SR.RB = 1;
    SR.BL = 1;
    PC = VBR + 0x00000400;
}
```

**4 FIRSTWRITE - Initial Page Write Exception**

- Source: TLB is hit in a store access, but dirty bit D = 0
- Transition address: VBR + 0x0000 0100
- Transition operations:  
The virtual address (32 bits) at which this exception occurred is set in TEA, and the corresponding virtual page number (22 bits) is set in PTEH [31:10]. ASID in PTEH indicates the ASID when this exception occurred.

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

Exception code 0x080 is set in EXPEVT. The BL, MD, and RB bits are set to 1 in SR, and a branch is made to PC = VBR + 0x0100.

```

FIRSTWRITE( )
{
    TEA = EXCEPTION_ADDRESS;
    PTEH.VPN = PAGE_NUMBER;
    SPC = PC;
    SSR = SR;
    SGR = R15;
    EXPEVT = 0x00000080;
    SR.MD = 1; SR.RB = 1;
    SR.BL = 1;
    PC = VBR + 0x00000100;
}

```

## 5 READPROT - Data TLB Protection Violation Exception

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

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

- Transition address: VBR + 0x0000 0100

- Transition operations:

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

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

Exception code 0x0A0 is set in EXPEVT. The BL, MD, and RB bits are set to 1 in SR, and a branch is made to PC = VBR + 0x0100.

```

READPROT ( )
{
    TEA = EXCEPTION_ADDRESS;
    PTEH.VPN = PAGE_NUMBER;
    SPC = PC;
    SSR = SR;
    SGR = R15;
    EXPEVT = 0x000000A0;
    SR.MD = 1;
    SR.RB = 1;
    SR.BL = 1;
    PC = VBR + 0x00000100;
}

```

## 6 WRITEPROT - Write Data TLB Protection Violation Exception

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

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

- Transition address: VBR + 0x0000 0100

- Transition operations:

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

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

Exception code 0x0C0 is set in EXPEVT. The BL, MD, and RB bits are set to 1 in SR, and a branch is made to PC = VBR + 0x0100.

```
WRITEPROT ( )
{
    TEA = EXCEPTION_ADDRESS;
    PTEH.VPN = PAGE_NUMBER;
    SPC = PC;
    SSR = SR;
    SGR = R15;
    EXPEVT = 0x000000C0;
    SR.MD = 1;
    SR.RB = 1;
    SR.BL = 1;
    PC = VBR + 0x00000100;
}
```

## 7 EXECPROT - Instruction TLB Protection Violation Exception

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

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

- Transition address:  $VBR + 0x0000\ 0100$
- Transition operations: The virtual address (32 bits) at which this exception occurred is set in TEA, and the corresponding virtual page number (22 bits) is set in PTEH [31:10]. ASID in PTEH indicates the ASID when this exception occurred.

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

Exception code 0x0A0 is set in EXPEVT. The BL, MD, and RB bits are set to 1 in SR, and a branch is made to  $PC = VBR + 0x0100$ .

```
EXECPROT ( )
{
    TEA = EXCEPTION_ADDRESS;
    PTEH.VPN = PAGE_NUMBER;
    SPC = PC;
    SSR = SR;
    SGR = R15;
    EXPEVT = 0x000000A0;
    SR.MD = 1;
    SR.RB = 1;
    SR.BL = 1;
    PC = VBR + 0x00000100;
}
```

**8 RADDERR - Read Data Address Error**

## - Sources:

Word data access from other than a word boundary ( $2n + 1$ )

Longword data access from other than a longword data boundary ( $4n + 1$ ,  $4n + 2$ , or  $4n + 3$ )

Quadword data access from other than a quadword data boundary ( $8n + 1$ ,  $8n + 2$ ,  $8n + 3$ ,  $8n + 4$ ,  $8n + 5$ ,  $8n + 6$ , or  $8n + 7$ )

Access to area  $0x8000\ 00000x\text{FFFF}\ \text{FFFF}$  in user mode

- Transition address:  $\text{VBR} + 0x0000\ 0100$ 

## - Transition operations:

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

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

Exception code  $0x0E0$  is set in EXPEVT. The BL, MD, and RB bits are set to 1 in SR, and a branch is made to  $\text{PC} = \text{VBR} + 0x0100$ . For details, see [Chapter 3: Memory management unit \(MMU\) on page 41](#).

```
RADDERR ( )
{
    TEA = EXCEPTION_ADDRESS;
    PTEN.VPN = PAGE_NUMBER;
    SPC = PC;
    SSR = SR;
    SGR = R15;
    EXPEVT = 0x000000E0;
    SR.MD = 1;
    SR.RB = 1;
    SR.BL = 1;
    PC = VBR + 0x00000100;
}
```

## 9 WADDERR - Write Data Address Error

- Sources:

Word data access from other than a word boundary ( $2n + 1$ )

Longword data access from other than a longword data boundary ( $4n + 1$ ,  $4n + 2$ , or  $4n + 3$ )

Quadword data access from other than a quadword data boundary ( $8n + 1$ ,  $8n + 2$ ,  $8n + 3$ ,  $8n + 4$ ,  $8n + 5$ ,  $8n + 6$ , or  $8n + 7$ )

Access to area  $0x8000\ 00000 - 0xFFFF\ FFFF$  in user mode (except for the store queue area  $0xE000\ 0000 - 0xE3FF\ FFFF$ )

- Transition address:  $VBR + 0x0000\ 0100$

- Transition operations:

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

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

Exception code  $0x100$  is set in EXPEVT. The BL, MD, and RB bits are set to 1 in SR, and a branch is made to  $PC = VBR + 0x0100$ . For details, see [Chapter 3: Memory management unit \(MMU\) on page 41](#).

```
WADDERR(
{
    TEA = EXCEPTION_ADDRESS;
    PTEN.VPN = PAGE_NUMBER;
    SPC = PC;
    SSR = SR;
    SGR = R15;
    EXPEVT = 0x00000100;
    SR.MD = 1;
    SR.RB = 1;
    SR.BL = 1;
    PC = VBR + 0x00000100;
}
```



**10 IADDERR - Instruction Address Error**

## - Sources:

Instruction fetch from other than a word boundary ( $2n + 1$ )

Instruction fetch from area 0x8000 00000 - 0xFFFF FFFF in user mode

## - Transition address: VBR + 0x0000 0100

## - Transition operations:

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

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

Exception code 0x0E0 is set in EXPEVT. The BL, MD, and RB bits are set to 1 in SR, and a branch is made to PC = VBR + 0x0100. For details, see [Chapter 3: Memory management unit \(MMU\) on page 41](#).

```
IADDERR ( )
{
    TEA = EXCEPTION_ADDRESS;
    PTEN.VPN = PAGE_NUMBER;
    SPC = PC;
    SSR = SR;
    SGR = R15;
    EXPEVT = 0x000000E0;
    SR.MD = 1;
    SR.RB = 1;
    SR.BL = 1;
    PC = VBR + 0x00000100;
}
```

**11 TRAP - Unconditional trap**

- Source: Execution of TRAPA instruction
- Transition address: VBR + 0x0000 0100

- Transition operations:

As this is a processing-completion-type exception, the PC contents for the instruction following the TRAPA instruction are saved in SPC. The value of SR and R15 when the TRAPA instruction is executed are saved in SSR and SGR. The 8-bit immediate value in the TRAPA instruction is multiplied by 4, and the result is set in TRA [9]. Exception code 0x160 is set in EXPEVT. The BL, MD, and RB bits are set to 1 in SR, and a branch is made to PC = VBR + 0x0100.

```
TRAP( )
{
    SPC = PC + 2;
    SSR = SR;
    SGR = R15;
    TRA = imm << 2; EXPEVT = 0x00000160;
    SR.MD = 1;
    SR.RB = 1;
    SR.BL = 1;
    PC = VBR + 0x00000100;
}
```

**12 RESINST - General Illegal Instruction Exception**

## - Sources:

Decoding of an undefined instruction other than in a branch delay slot.

The opcode 0xFFFD is guaranteed to be defined in any SH-4 architecture revision. Other unused opcodes may be treated as reserved in any particular SH-4 implementation.

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 + 0x0000 0100

## - Transition operations:

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

Exception code 0x180 is set in EXPEVT. The BL, MD, and RB bits are set to 1 in SR, and a branch is made to PC = VBR + 0x0100.

*Note: The only undefined opcode which the architecture guarantees to cause a General Illegal Instruction Exception is 0xFFFD.*

```
RESINST ( )
{
    SPC = PC;
    SSR = SR;
    SGR = R15;
    EXPEVT = 0x00000180;
    SR.MD = 1;
    SR.RB = 1;
    SR.BL = 1;
    PC = VBR + 0x00000100;
}
```

**13 ILLSLOT - Slot Illegal Instruction Exception**

## - Sources:

Decoding of an undefined instruction in a delay slot

The branches with delay slots are JMP, JSR, BRA, BRAF, BSR, BSRF, RTS, RTE, BT/S and BF/S. The opcode 0xFFFFD is guaranteed to be undefined in any SH-4 architecture revision. Other unused opcodes may be treated as reserved in any particular SH-4 implementation.

Decoding of an instruction that modifies PC in a delay slot

Instructions that modify PC: JMP, JSR, BRA, BRAF, BSR, BSRF, RTS, RTE, BT, BF, BT/S, BF/S, TRAPA, LDC Rm, SR, LDC.L @Rm+, SR

Decoding in user mode of a privileged instruction in a delay slot

Privileged instructions: LDC, STC, RTE, LDTLB, SLEEP, but excluding LDC/STC instructions that access GBR

Decoding of a PC-relative MOV instruction or MOVA instruction in a delay slot

Transition address: VBR + 0x0000 0100

Transition operations:

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

Exception code 0x1A0 is set in EXPEVT. The BL, MD, and RB bits are set to 1 in SR, and a branch is made to PC = VBR + 0x0100.

*Note: The only undefined opcode which the architecture guarantees to cause a Slot Illegal Instruction Exception is 0xFFFFD.*

```
ILLSLOT ( )
{
    SPC = PC - 2;
    SSR = SR;
    SGR = R15;
    EXPEVT = 0x000001A0;
    SR.MD = 1;
    SR.RB = 1;
    SR.BL = 1;
    PC = VBR + 0x00000100;
}
```

**14 FPUDIS - General FPU Disable Exception**

- Source: Decoding of an FPU instruction\* not in a delay slot with SR.FD =1
- Transition address: VBR + 0x0000 0100

Transition operations:

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

Exception code 0x800 is set in EXPEVT. The BL, MD, and RB bits are set to 1 in SR, and a branch is made to PC = VBR + 0x0100.

*Note: FPU instructions are instructions in which the first 4 bits of the instruction code are F (but excluding undefined instruction 0xFFFFD), and the LDS, STS, LDS.L, and STS.L instructions corresponding to FPUL and FPSCR.*

```

FPUDIS ( )
{
    SPC = PC;
    SSR = SR;
    SGR = R15;
    EXPEVT = 0x00000800;
    SR.MD = 1;
    SR.RB = 1;
    SR.BL = 1;
    PC = VBR + 0x00000100;
}

```

**15 SLOTFPUDIS - Slot FPU Disable Exception**

- Source: Decoding of an FPU instruction in a delay slot with SR.FD = 1
- Transition address: VBR + 0x0000 0100
- Transition operations:  
The PC contents for the preceding delayed branch instruction are saved in SPC. The SR and R15 contents when this exception occurred are saved in SSR and SGR.

Exception code 0x820 is set in EXPEVT. The BL, MD, and RB bits are set to 1 in SR, and a branch is made to PC = VBR + 0x0100.

```
SLOTFPUDIS( )
{
    SPC = PC - 2;
    SSR = SR;
    SGR = R15;
    EXPEVT = 0x00000820;
    SR.MD = 1;
    SR.RB = 1;
    SR.BL = 1;
    PC = VBR + 0x00000100;
}
```

**16 UBRKBEFORE - User Breakpoint Pre-execution Trap**

- Source: Fulfilling of a break condition set in the user break controller
- Transition address: VBR + 0x0000 0100, or DBR
- Transition operations:

The PC contents for the instruction at which the breakpoint is set are set in SPC. The SR and R15 contents when the break occurred are saved in SSR and SGR. Exception code 0x1E0 is set in EXPEVT.

The BL, MD, and RB bits are set to 1 in SR, and a branch is made to PC = VBR + 0x0100. It is also possible to branch to PC = DBR. For details of PC, etc., when a data break is set, see *User Break Controller (UBC) Chapter in the ST40 System Architecture Manual*.

```
UBRKBEFORE( )
{
    SPC = PC;
    SSR = SR;
    SGR = R15;
    EXPEVT = 0x000001E0;
    SR.MD = 1;
    SR.RB = 1;
    SR.BL = 1;
    PC = (BRCCR.UBDE==1 ? DBR : VBR + H00000100);
}
```

**17 UBRKAFTER - User Breakpoint Post-Execution Trap**

- Source: Fulfilling of a break condition set in the user break controller
- Transition address: VBR + 0x0000 0100, or DBR
- Transition operations:

The PC of the instruction following that at which the breakpoint is set is placed in SPC. The SR and R15 contents when the break occurred are saved in SSR and SGR. Exception code 0x1E0 is set in EXPEVT.

The BL, MD, and RB bits are set to 1 in SR, and a branch is made to PC = VBR + 0x0100. It is also possible to branch to PC = DBR. For details of PC, etc., when a data break is set, see *User Break Controller (UBC) Chapter in the ST40 System Architecture Manual*.

```
UBRKAFTER ( )
{
    SPC = PC + 2;
    SSR = SR;
    SGR = R15;
    EXPEVT = 0x000001E0;
    SR.MD = 1;
    SR.RB = 1;
    SR.BL = 1;
    PC = (BRCR.UBDE==1 ? DBR : VBR + H00000100);
}
```



**18 FPUEXC - FPU Exception**

- Source: Exception due to execution of a floating-point operation
- Transition address: VBR + 0x0000 0100
- Transition operations:

The PC and SR contents for the instruction at which this exception occurred are saved in SPC and SSR. Exception code 0x120 is set in EXPEVT. The BL, MD, and RB bits are set to 1 in SR, and a branch is made to PC = VBR + 0x0100. The contents of R15 are saved to SGR.

```
FPUEXC( )
{
    SPC = PC;
    SSR = SR;
    SGR = R15;
    EXPEVT = 0x00000120;
    SR.MD = 1;
    SR.RB = 1;
    SR.BL = 1;
    PC = VBR + 0x00000100;
}
```

## 5.6.3 Interrupts

### 1 NMI - Non-Maskable Interrupt

- Source: Refer to relevant *System Architecture Manual* for details of non-maskable interrupt generation (NMI).
- Transition address: VBR + 0x0000 0600

Transition operations:

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

Exception code 0x1C0 is set in INTEVT. The BL, MD, and RB bits are set to 1 in SR, and a branch is made to PC = VBR + 0x0600.

When the BL bit in SR is 0, this interrupt is not masked by the interrupt mask bits in SR, and is accepted at the highest priority level. When the BL bit in SR is 1, a software setting can specify whether this interrupt is to be masked or accepted. For details refer to the description of interrupt programming in the appropriate *System Architecture Manual*.

```
NMI ( )
{
    SPC = PC;
    SSR = SR;
    SGR = R15;
    INTEVT = 0x000001C0;
    SR.MD = 1;
    SR.RB = 1;
    SR.BL = 1;
    PC = VBR + 0x00000600;
}
```

**2 IRLINT - IRL Interrupts**

- Source: The interrupt mask bit setting in SR is smaller than the IRL (3-0) level, and the BL bit in SR is 0 (accepted at instruction boundary).
- Transition address: VBR + 0x0000 0600
- Transition operations:

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

The code corresponding to the IRL (3-0) level is set in INTEVT. For further details of the interrupt handling behavior, refer to the product level documentation of the interrupt controller. The BL, MD, and RB bits are set to 1 in SR, and a branch is made to VBR + 0x0600. The acceptance level is not set in the interrupt mask bits in SR. When the BL bit in SR is 1, the interrupt is masked. For further details of the interrupt handling behavior, refer to the product level documentation of the interrupt controller.

```
IRLINT( )
{
    SPC = PC;
    SSR = SR;
    SGR = R15;
    INTEVT = 0x00000200 ~ 0x000003C0;
    SR.MD = 1;
    SR.RB = 1;
    SR.BL = 1;
    PC = VBR + 0x00000600;
}
```

### 3 PERIPHINT - Peripheral Module Interrupts

- Source: The interrupt mask bit setting in SR is smaller than the peripheral module (Hitachi-UDI for example) interrupt level, and the BL bit in SR is 0 (accepted at instruction boundary).
- Transition address: VBR + 0x0000 0600
- Transition operations:  
The PC contents immediately after the instruction at which the interrupt is accepted are set in SPC. The SR and R15 contents at the time of acceptance are set in SSR and SGR.

The code corresponding to the interrupt source is set in INTEVT. The BL, MD, and RB bits are set to 1 in SR, and a branch is made to VBR + 0x0600. The module interrupt levels should be set as values between 0x0 and 0xF in the interrupt priority registers (IPRA-IPRC) in the interrupt controller. For further details of the interrupt handling behavior, refer to the product level documentation of the interrupt controller.

```
Module_interruption()
{
    SPC = PC;
    SSR = SR;
    SGR = R15;
    INTEVT = 0x00000400 ~ 0x00000B80;
    SR.MD = 1;
    SR.RB = 1;
    SR.BL = 1;
    PC = VBR + 0x00000600;
}
```

## 5.6.4 Priority order with multiple exceptions

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

- 1 Instructions that make two accesses to memory.  
With MAC instructions, memory-to-memory arithmetic/logic instructions, and TAS instructions, two data transfers are performed by a single instruction, and an exception will be detected for each of these data transfers. In these cases, therefore, the following order is used to determine priority.
  - 1.1 Data address error in first data transfer.
  - 1.2 TLB miss in first data transfer.
  - 1.3 TLB protection violation in first data transfer.
  - 1.4 Data address error in second data transfer.
  - 1.5 TLB miss in second data transfer.
  - 1.6 TLB protection violation in second data transfer.
  - 1.7 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.
  - 2.1 The delayed branch instruction is checked for priority levels 1 and 2.
  - 2.2 The delay slot instruction is checked for priority levels 1 and 2.
  - 2.3 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.)
  - 2.4 A check is performed for priority level 4 in the delayed branch instruction and priority level 4 in the delay slot instruction. (There is no priority ranking between these two.)

If the delay slot instruction has a second data transfer, two checks are performed in step b, as in 1 above.

If the accepted exception (the highest-priority exception) is a delay slot instruction re-execution type exception, the branch instruction PR register write operation (PC PR operation performed in BSR, BSRF, JSR) is inhibited.

## 5.7 Usage notes

### 1 Return from exception handling

#### 1.1 Check the BL bit in SR with software.

If SPC and SSR have been saved to external memory, set the BL bit in SR to 1 before restoring them.

#### 1.2 Issue an RTE instruction.

When RTE is executed, the SPC contents are set in PC, the SSR contents are set in SR, and branch is made to the SPC address to return from the exception handling routine.

### 2 If an exception or interrupt occurs when SR.BL = 1

#### 2.1 Exception

When an exception other than a user break occurs, 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 (0xA000 0000). The value in EXPEVT at this time is 0x0000 0020. The value of the SPC and SSR registers is undefined.

#### 2.2 Interrupt

If an ordinary interrupt occurs, the interrupt request is held pending and is accepted after the BL bit in SR has been cleared to 0 by software. If a nonmaskable interrupt (NMI) occurs, it can be held pending or accepted according to the setting made by software. In the sleep or standby state, an interrupt is accepted even if the BL bit in SR is set to 1.

### 3 SPC when an exception occurs

#### 3.1 Re-execution type exception

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

#### 3.2 Completion type exception or interrupt

The PC value for the instruction following that in which the exception

occurred is set in SPC. If an exception occurs in a branch instruction with delay slot, the PC value for the branch destination is saved in SPC.

- 4 An exception must not be generated in an RTE instruction delay slot, as the operation will be undefined in this case.







## 6

# Floating-point unit

## 6.1 Overview

The floating-point unit (FPU) has the following features:

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

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

## 6.2 Floating-point format

An IEEE754 floating-point number contains three fields: a sign (s), an exponent (e) and a fraction (f) in the format given in [Figure 31](#).

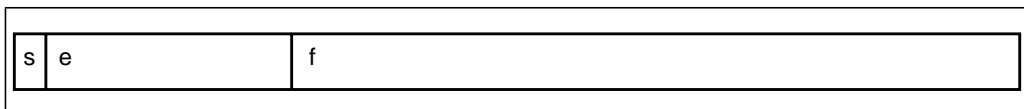


Figure 31: IEEE754 floating-point representations

The sign, s, is the sign of the represented number. If s is 0, the number is positive. If s is 1, the number is negative.

The exponent, e, is held as a biased value. The relationship between the biased exponent, e, and the unbiased exponent, E, is given by  $e = E + \text{bias}$ , where bias is a fixed positive number. The unbiased exponent, E, takes any value in the range  $[E_{\min}-1, E_{\max}+1]$ . The minimum and maximum values in that range,  $E_{\min}-1$  and  $E_{\max}+1$ , designate special values such as positive zero, negative zero, positive infinity, negative infinity, denormalized numbers and “Not a Number” (NaN).

The fraction, f, specifies the binary digits that lie to the right of the binary point. A normalized floating-point number has a leading bit of 1 which lies to the left of the binary point. A denormalized floating-point number has a leading bit of 0 which lies to the left of the binary point. The leading bit is implicitly represented; it is determined by whether the number is normalized or denormalized, and is not explicitly encoded. The implicit leading bit and the explicit fraction bits together form the significance of the floating-point number.

Floating-point number value v is determined as follows:

The value, v, of a floating-point number is determined as follows:

NaN: if  $E = E_{\max} + 1$  and  $f \neq 0$ , then v is Not a Number irrespective of the sign s

Positive or negative infinity: if  $E = E_{\max} + 1$  and  $f = 0$ , then  $v = (-1)^s (\infty)$

Normalized number: if  $E_{\min} \leq E \leq E_{\max}$ , then  $v = (-1)^s 2^E (1.f)$

Denormalized number: if  $E = E_{\min} - 1$  and  $f \neq 0$ , then  $v = (-1)^s 2^{E_{\min}} (0.f)$

Positive or negative zero: if  $E = E_{\min} - 1$  and  $f = 0$ , then  $v = (-1)^s 0$

The architecture supports two IEEE754 basic floating-point number formats: single-precision and double-precision.

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

**Table 35: Floating-point number formats and parameters**

*Table 36* shows the ranges of the various numbers in hexadecimal notation.

Type	Single-precision	Double-precision
sNaN (Signaling not-a-number)	0x7FFFFFFF to 0x7FC00000 and 0xFFC00000 to 0xFFFFFFFF	0x7FFFFFFF 0xFFFFFFFF to 0x7FF80000 0x00000000 and 0xFFF80000 0x00000000 to 0xFFFFFFFF 0xFFFFFFFF
qNaN (Quiet not-a-number)	0x7FBFFFFF to 0x7F800001 and 0xFF800001 to 0xFFBFFFFF	0x7FF7FFFF 0xFFFFFFFF to 0x7FF00000 0x00000001 and 0xFFF00000 0x00000001 to 0xFFF7FFFF 0xFFFFFFFF
+INF (Positive infinity)	0x7F800000	0x7FF00000 0x000000
+NORM (Positive normalized number)	0x7F7FFFFF to 0x00800000	0x7FEFFFFFFF 0xFFFFFFFF to 0x00100000 0x00000000

**Table 36: Floating-point ranges**

Type	Single-precision	Double-precision
+DENORM (Positive denormalized number)	0x007FFFFFFF to 0x00000001	0x000FFFFFFF 0xFFFFFFFF to 0x00000000 0x00000001
+0.0 (Positive zero)	0x00000000	0x00000000 0x00000000
- 0.0 (Negative zero)	0x80000000	0x80000000 0x00000000
-DENORM (Negative denormalized number)	0x80000001 to 0x807FFFFFFF	0x80000000 0x00000001 to 0x800FFFFFFF 0xFFFFFFFF
-NORM (Negative normalized number)	0x80800000 to 0xFF7FFFFFFF	0x80100000 0x00000000 to 0xFFEFFFFFFF 0xFFFFFFFF
-INF (Negative infinity)	0xFF800000	0xFFF00000 0x00000000

Table 36: Floating-point ranges

## 6.2.1 Non-numbers (NaN)

Figure 32 shows the bit pattern of a non-number (NaN).

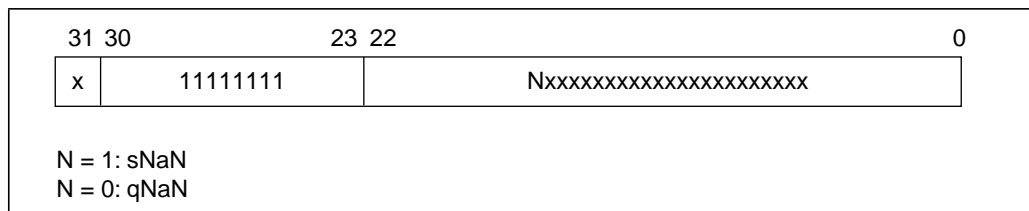


Figure 32: Single-precision NaN bit pattern

A floating-point number is a NaN if the exponent field contains the maximum representable value and the fraction is non-zero, regardless of the value of the sign. In the figure above, x can have a value of 0 or 1. If the most significant bit of the fraction (N, in the figure above) is 1, the value is a signaling NaN (sNaN), otherwise the value is a quiet NaN (qNaN).

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

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

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

See the individual instruction descriptions for details of floating-point operations when a non-number (NaN) is input.

## 6.2.2 Denormalized numbers

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

When the DN bit in the FPU's status register FPSCR is 1, a denormalized number (source operand or operation result) is always flushed to 0 in a floating-point operation that generates a value (an operation other than copy, FNEG, or FABS).

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

## 6.3 Rounding

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

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

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

**Round to Nearest:**

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

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

**Round to Zero:**

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

If the unrounded value is larger than the maximum expressible absolute value, the value will be the maximum expressible absolute value.

## 6.4 Floating-point exceptions

FPU-related exceptions are as follows:

- General illegal instruction/slot illegal instruction exception

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

- FPU exceptions

The exception sources are as follows:

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

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

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

- Enable/disable exception handling

The SH-4 CPU core supports enable exception handling and disable exception handling.

Enable exception handling is initiated in the following cases:

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

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

Except for the above, the FPU disables exception handling. In all processing, the bit corresponding to source V, Z, O, U, or I is set to 1, and disable exception handling is provided for each exception.

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

When rounding mode = RZ, the maximum normalized number, with the same sign as the unrounded value, is generated.

When rounding mode = RN, infinity with the same sign as the unrounded value is generated.

- Underflow (U):

When FPSCR.DN = 0, a denormalized number with the same sign as the unrounded value, or zero with the same sign as the unrounded value, is generated.

When FPSCR.DN = 1, zero with the same sign as the unrounded value, is generated.

- Inexact exception (I): An inexact result is generated.

## 6.5 Graphics support functions

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

### 6.5.1 Geometric operation instructions

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

Maximum error = MAX (individual multiplication result  $\times 2^{-\text{MIN (number of multiplier significant digits1, number of multiplicand significant digits1)}}$ ) + MAX (result value  $\times 2^{-23}, 2^{-149}$ )

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

In future versions of the SuperH series, the above error is guaranteed, but the same result as SH-4 is not guaranteed.

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

- Inner product (m does not= n):

This operation is generally used for surface/rear surface determination for polygon surfaces.

- Sum of square of elements (m = n):

This operation is generally used to find the length of a vector.



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

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

- Matrix (4 x 4). vector (4):

This operation is generally used for viewpoint changes, angle changes, or movements called vector transformations (4-dimensional). Since affine transformation processing for angle + parallel movement basically requires a 4 x 4 matrix, the SH-4 CPU core supports 4-dimensional operations.

- Matrix (4 x 4) x matrix (4 x 4):

This operation requires the execution of four FTRV instructions.

Since approximate-value computations are performed to enable high-speed computation, the inexact exception (I) bit in the cause field and flag field is always set to 1 when an FTRV instruction is executed. Therefore, if the corresponding bit is set in the enable field, enable exception handling will be executed. For the same reason, it is not possible to check all data types in the registers beforehand when executing an FTRV instruction. If the V bit is set in the enable field, enable exception handling will be executed.

**FRCHG:** This instruction modifies banked registers. For example, when the FTRV instruction is executed, matrix elements must be set in an array in the background bank. However, to create the actual elements of a translation matrix, it is easier to use registers in the foreground bank. When the LDC instruction is used on FPSCR, this instruction expends 4 to 5 cycles in order to maintain the FPU state. With the FRCHG instruction, an FPSCR.FR bit modification can be performed in one cycle.

## 6.5.2 Pair single-precision data transfer

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

When FPSCR.SZ = 1, the SH-4 CPU core can perform data transfer by means of pair single-precision data transfer instructions.

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

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

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



# Instruction set

## 7.1 Execution environment

### PC

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

Data sizes and data types: The SH-4 instruction set is implemented with 16-bit fixed-length instructions. The SH-4 CPU core can use byte (8-bit), word (16-bit), longword (32-bit), and quadword (64-bit) data sizes for memory access. Single-precision floating-point data (32 bits) can be moved to and from memory using longword or quadword size. Double-precision floating-point data (64 bits) can be moved to and from memory using longword size. When a double-precision floating-point operation is specified (FPSCR.PR = 1), the result of an operation using quadword access will be undefined. When the SH-4 CPU core moves byte-size or word-size data from memory to a register, the data is sign-extended.

### Load-store architecture

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

### Delayed branches

Except for the two branch instructions BF and BT, the SH-4's branch instructions and RTE are delayed branches. In a delayed branch, the instruction following the branch is executed before the branch destination instruction. This execution slot following a delayed branch is called a delay slot. For example, the BRA execution sequence is as follows:

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

### Delay slot

An illegal instruction exception may occur when a specific instruction is executed in a delay slot. See section 5, Exceptions. The instruction following BF/S or BT/S for which the branch is not taken is also a delay slot instruction.

### T bit

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

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

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

### Constant values

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

MOV.W                      @(disp, PC), Rn

MOV.L                      @(disp, PC), Rn

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

## 7.2 Addressing modes

Addressing modes and effective address calculation methods are shown in [Table 37](#). When a location in virtual memory space is accessed (MMUCR.AT = 1), the effective address is translated into a physical memory address. If multiple virtual memory space systems are selected (MMUCR.SV = 0), the least significant bit of PTEH is also referenced as the access ASID. See [Chapter 3: Memory management unit \(MMU\) on page 41](#).

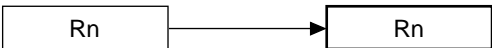
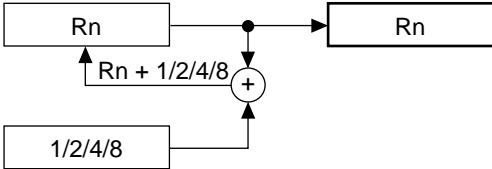
Addressing mode	Instruction format	Effective address calculation method	Calculation formula
Register direct	Rn	Effective address is register Rn. (Operand is register Rn contents.)	—
Register indirect	@Rn	Effective address is register Rn contents. 	Rn → EA (EA: effective address)
Register indirect with post-increment	@Rn+	Effective address is register Rn contents. A constant is added to Rn after instruction execution: 1 for a byte operand, 2 for a word operand, 4 for a longword operand, 8 for a quadword operand. 	Rn → EA After instruction execution Byte: Rn + 1 → Rn Word: Rn + 2 → Rn Longword: Rn + 4 → Rn Quadword: Rn + 8 → Rn

Table 37: Addressing modes and effective addresses

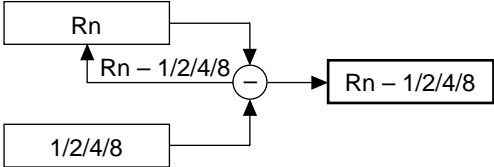
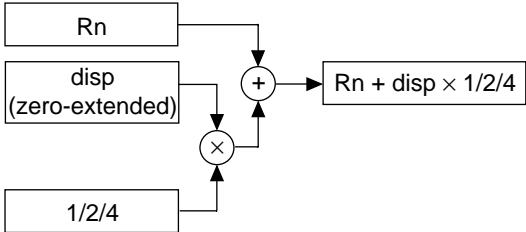
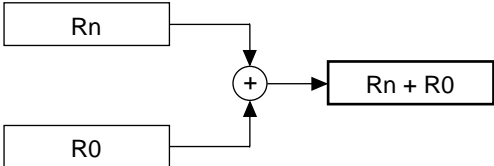
Addressing mode	Instruction format	Effective address calculation method	Calculation formula
Register indirect with pre-decrement	@-Rn	<p>Effective address is register Rn contents, decremented by a constant beforehand:            1 for a byte operand, 2 for a word operand,            4 for a longword operand, 8 for a quadword operand.</p> 	<p>Byte:  <math>Rn - 1 \rightarrow Rn</math>            Word:  <math>Rn - 2 \rightarrow Rn</math>            Longword:  <math>Rn - 4 \rightarrow Rn</math>            Quadword:  <math>Rn - 8 \rightarrow Rn</math>  <math>Rn \rightarrow EA</math>            (Instruction executed with Rn after calculation)</p>
Register indirect with displacement	@(disp:4, Rn)	<p>Effective address is register Rn contents with 4-bit displacement disp added. After disp is zero-extended, it is multiplied by 1 (byte), 2 (word), or 4 (longword), according to the operand size.</p> 	<p>Byte: <math>Rn + disp \rightarrow EA</math>            Word: <math>Rn + disp \times 2 \rightarrow EA</math>            Longword: <math>Rn + disp \times 4 \rightarrow EA</math></p>
Indexed register indirect	@(R0, Rn)	<p>Effective address is sum of register Rn and R0 contents.</p> 	<p><math>Rn + R0 \rightarrow EA</math></p>

Table 37: Addressing modes and effective addresses

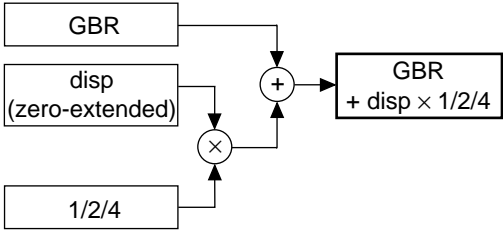
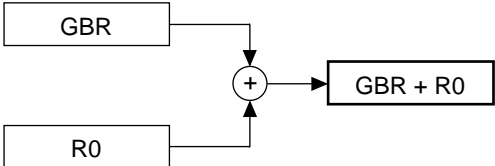
Addressing mode	Instruction format	Effective address calculation method	Calculation formula
GBR indirect with displacement	@(disp:8, GBR)	<p>Effective address is register GBR contents with 8-bit displacement disp added. After disp is zero-extended, it is multiplied by 1 (byte), 2 (word), or 4 (longword), according to the operand size.</p>  <pre> graph LR     GBR[GBR] --&gt; Add((+))     disp[disp (zero-extended)] --&gt; Mult((×))     mult[1/2/4] --&gt; Mult     Mult --&gt; Add     Add --&gt; EA[GBR + disp × 1/2/4] </pre>	<p>Byte: <math>\text{GBR} + \text{disp} \rightarrow \text{EA}</math></p> <p>Word: <math>\text{GBR} + \text{disp} \times 2 \rightarrow \text{EA}</math></p> <p>Longword: <math>\text{GBR} + \text{disp} \times 4 \rightarrow \text{EA}</math></p>
Indexed GBR indirect	@(R0, GBR)	<p>Effective address is sum of register GBR and R0 contents.</p>  <pre> graph LR     GBR[GBR] --&gt; Add((+))     R0[R0] --&gt; Add     Add --&gt; EA[GBR + R0] </pre>	$\text{GBR} + \text{R0} \rightarrow \text{EA}$

Table 37: Addressing modes and effective addresses



Addressing mode	Instruction format	Effective address calculation method	Calculation formula
PC-relative with displacement	@(disp:8, PC)	<p>Effective address is PC+4 with 8-bit displacement disp added. After disp is zero-extended, it is multiplied by 2 (word), or 4 (longword), according to the operand size. With a longword operand, the lower 2 bits of PC are masked.</p> <p>PC</p> <p>0xFFFFF0</p> <p>4</p> <p>disp (zero-extended)</p> <p>2/4</p> <p>* With longword operand</p> <p>PC + 4 + disp × 2 or PC &amp; 0xFFFFF0 + 4 + disp × 4</p>	<p>Word: PC + 4 + disp × 2 → EA</p> <p>Longword: PC &amp; 0xFFFFF0 + 4 + disp × 4 → EA</p>
PC-relative	disp:8	<p>Effective address is PC+4 with 8-bit displacement disp added after being sign-extended and multiplied by 2.</p> <p>PC</p> <p>4</p> <p>disp (sign-extended)</p> <p>2</p> <p>PC + 4 + disp × 2</p>	PC + 4 + disp × 2 → Branch-Target

Table 37: Addressing modes and effective addresses

Addressing mode	Instruction format	Effective address calculation method	Calculation formula
PC-relative	disp:12	<p>Effective address is PC+4 with 12-bit displacement disp added after being sign-extended and multiplied by 2.</p> <pre> graph TD     PC[PC] --&gt; A((+))     4[4] --&gt; A     A --&gt; B((+))     disp["disp (sign-extended)"] --&gt; C((x))     2[2] --&gt; C     C --&gt; B     B --&gt; Result["PC + 4 + disp × 2"]           </pre>	$PC + 4 + \text{disp} \times 2 \rightarrow$ Branch-Target
	Rn	<p>Effective address is sum of PC+4 and Rn.</p> <pre> graph TD     PC[PC] --&gt; A((+))     4[4] --&gt; A     A --&gt; B((+))     Rn[Rn] --&gt; B     B --&gt; Result["PC + 4 + Rn"]           </pre>	$PC + 4 + Rn \rightarrow$ Branch-Target
Immediate	#imm:8	8-bit immediate data imm of TST, AND, OR, or XOR instruction is zero-extended.	—
	#imm:8	8-bit immediate data imm of MOV, ADD, or CMP/EQ instruction is sign-extended.	—

Table 37: Addressing modes and effective addresses

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

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

## 7.3 Instruction set summary

*Table 38* shows the notation used in the following SH instruction list.

Item	Format	Description
Instruction mnemonic	OP.Sz SRC, DEST	OP: Operation code Sz: Size SRC: Source DEST: Source and/or destination operand
Summary of operation		→, ← Transfer direction (xx) Memory operand M/Q/T SR flag bits & Logical AND of individual bits   Logical OR of individual bits ^ Logical exclusive-OR of individual bits ~ Logical NOT of individual bits <<n, >>n n-bit shift
Instruction code	MSB ↔ LSB	mmmm: Register number (Rm, FRm) nnnn: Register number (Rn, FRn) 0000: R0, FR0 0001: R1, FR1 : 1111: R15, FR15 mmm: Register number (DRm, XDm, Rm_BANK) nnn: Register number (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 data dddd: Displacement
Privileged mode		“Privileged” means the instruction can only be executed in privileged mode.

**Table 38: Notation used in instruction list**

Item	Format	Description
T bit	Value of T bit after instruction execution	—: No change

Table 38: Notation used in instruction list

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

Instruction		Operation	Instruction code	Privileged	T bit
MOV	#imm,Rn	imm $\rightarrow$ sign extension $\rightarrow$ Rn	1110nnnniiiiiii	—	—
MOV.W	@(disp,PC),Rn	(disp $\times 2$ + PC + 4) $\rightarrow$ sign extension $\rightarrow$ Rn	1001nnnnnddddddd	—	—
MOV.L	@(disp,PC),Rn	(disp $\times 4$ + PC & 0xFFFFFC + 4) $\rightarrow$ Rn	1101nnnnnddddddd	—	—
MOV	Rm,Rn	Rm $\rightarrow$ Rn	0110nnnnnnmmmm0011	—	—
MOV.B	Rm,@Rn	Rm $\rightarrow$ (Rn)	0010nnnnnnmmmm0000	—	—
MOV.W	Rm,@Rn	Rm $\rightarrow$ (Rn)	0010nnnnnnmmmm0001	—	—
MOV.L	Rm,@Rn	Rm $\rightarrow$ (Rn)	0010nnnnnnmmmm0010	—	—
MOV.B	@Rm,Rn	(Rm) $\rightarrow$ sign extension $\rightarrow$ Rn	0110nnnnnnmmmm0000	—	—
MOV.W	@Rm,Rn	(Rm) $\rightarrow$ sign extension $\rightarrow$ Rn	0110nnnnnnmmmm0001	—	—
MOV.L	@Rm,Rn	(Rm) $\rightarrow$ Rn	0110nnnnnnmmmm0010	—	—
MOV.B	Rm,@-Rn	Rn-1 $\rightarrow$ Rn, Rm $\rightarrow$ (Rn)	0010nnnnnnmmmm0100	—	—
MOV.W	Rm,@-Rn	Rn-2 $\rightarrow$ Rn, Rm $\rightarrow$ (Rn)	0010nnnnnnmmmm0101	—	—
MOV.L	Rm,@-Rn	Rn-4 $\rightarrow$ Rn, Rm $\rightarrow$ (Rn)	0010nnnnnnmmmm0110	—	—
MOV.B	@Rm+,Rn	(Rm) $\rightarrow$ sign extension $\rightarrow$ Rn, Rm + 1 $\rightarrow$ Rm	0110nnnnnnmmmm0100	—	—
MOV.W	@Rm+,Rn	(Rm) $\rightarrow$ sign extension $\rightarrow$ Rn, Rm + 2 $\rightarrow$ Rm	0110nnnnnnmmmm0101	—	—
MOV.L	@Rm+,Rn	(Rm) $\rightarrow$ Rn, Rm + 4 $\rightarrow$ Rm	0110nnnnnnmmmm0110	—	—

Table 39: Fixed-point transfer instructions

Instruction		Operation	Instruction code	Privileged	T bit
MOV.B	R0, @(disp,Rn)	$R0 \rightarrow (\text{disp} + Rn)$	10000000nnnnndddd	—	—
MOV.W	R0, @(disp,Rn)	$R0 \rightarrow (\text{disp} \times 2 + Rn)$	10000001nnnnndddd	—	—
MOV.L	Rm, @(disp,Rn)	$Rm \rightarrow (\text{disp} \times 4 + Rn)$	0001nnnnmmmmddddd	—	—
MOV.B	@(disp,Rm),R0	$(\text{disp} + Rm) \rightarrow \text{sign extension} \rightarrow R0$	10000100mmmmddddd	—	—
MOV.W	@(disp,Rm),R0	$(\text{disp} \times 2 + Rm) \rightarrow \text{sign extension} \rightarrow R0$	10000101mmmmddddd	—	—
MOV.L	@(disp,Rm),Rn	$(\text{disp} \times 4 + Rm) \rightarrow Rn$	0101nnnnmmmmddddd	—	—
MOV.B	Rm, @(R0,Rn)	$Rm \rightarrow (R0 + Rn)$	0000nnnnmmmm0100	—	—
MOV.W	Rm, @(R0,Rn)	$Rm \rightarrow (R0 + Rn)$	0000nnnnmmmm0101	—	—
MOV.L	Rm, @(R0,Rn)	$Rm \rightarrow (R0 + Rn)$	0000nnnnmmmm0110	—	—
MOV.B	@(R0,Rm),Rn	$(R0 + Rm) \rightarrow \text{sign extension} \rightarrow Rn$	0000nnnnmmmm1100	—	—
MOV.W	@(R0,Rm),Rn	$(R0 + Rm) \rightarrow \text{sign extension} \rightarrow Rn$	0000nnnnmmmm1101	—	—
MOV.L	@(R0,Rm),Rn	$(R0 + Rm) \rightarrow Rn$	0000nnnnmmmm1110	—	—
MOV.B	R0, @(disp,GBR)	$R0 \rightarrow (\text{disp} + \text{GBR})$	11000000ddddddddd	—	—
MOV.W	R0, @(disp,GBR)	$R0 \rightarrow (\text{disp} \times 2 + \text{GBR})$	11000001ddddddddd	—	—
MOV.L	R0, @(disp,GBR)	$R0 \rightarrow (\text{disp} \times 4 + \text{GBR})$	11000010ddddddddd	—	—
MOV.B	@(disp,GBR),R0	$(\text{disp} + \text{GBR}) \rightarrow \text{sign extension} \rightarrow R0$	11000100ddddddddd	—	—
MOV.W	@(disp,GBR),R0	$(\text{disp} \times 2 + \text{GBR}) \rightarrow \text{sign extension} \rightarrow R0$	11000101ddddddddd	—	—
MOV.L	@(disp,GBR),R0	$(\text{disp} \times 4 + \text{GBR}) \rightarrow R0$	11000110ddddddddd	—	—
MOVA	@(disp,PC),R0	$\text{disp} \times 4 + \text{PC} \ \& \ 0\text{FFFFFFFC} + 4 \rightarrow R0$	11000111ddddddddd	—	—
MOVT	Rn	$T \rightarrow Rn$	0000nnnn00101001	—	—

Table 39: Fixed-point transfer instructions

Instruction		Operation	Instruction code	Privileged	T bit
SWAP.B	Rm,Rn	Rm $\rightarrow$ swap lower 2 bytes $\rightarrow$ REG	0110nnnnmmmm1000	—	—
SWAP.W	Rm,Rn	Rm $\rightarrow$ swap upper/lower words $\rightarrow$ Rn	0110nnnnmmmm1001	—	—
XTRCT	Rm,Rn	Rm:Rn middle 32 bits $\rightarrow$ Rn	0010nnnnmmmm1101	—	—

Table 39: Fixed-point transfer instructions

Instruction		Operation	Instruction code	Privileged	T Bit
ADD	Rm,Rn	Rn + Rm $\rightarrow$ Rn	0011nnnnmmmm1100	—	—
ADD	#imm,Rn	Rn + imm $\rightarrow$ Rn	0111nnnniiiiiii	—	—
ADDC	Rm,Rn	Rn + Rm + T $\rightarrow$ Rn, carry $\rightarrow$ T	0011nnnnmmmm1110	—	Carry
ADDV	Rm,Rn	Rn + Rm $\rightarrow$ Rn, overflow $\rightarrow$ T	0011nnnnmmmm1111	—	Overflow
CMP/EQ	#imm,R0	When R0 = imm, 1 $\rightarrow$ T Otherwise, 0 $\rightarrow$ T	10001000iiiiiii	—	Comparison result
CMP/EQ	Rm,Rn	When Rn = Rm, 1 $\rightarrow$ T Otherwise, 0 $\rightarrow$ T	0011nnnnmmmm0000	—	Comparison result
CMP/HS	Rm,Rn	When Rn $\geq$ Rm (unsigned), 1 $\rightarrow$ T Otherwise, 0 $\rightarrow$ T	0011nnnnmmmm0010	—	Comparison result
CMP/GE	Rm,Rn	When Rn $\geq$ Rm (signed), 1 $\rightarrow$ T Otherwise, 0 $\rightarrow$ T	0011nnnnmmmm0011	—	Comparison result
CMP/HI	Rm,Rn	When Rn > Rm (unsigned), 1 $\rightarrow$ T Otherwise, 0 $\rightarrow$ T	0011nnnnmmmm0110	—	Comparison result
CMP/GT	Rm,Rn	When Rn > Rm (signed), 1 $\rightarrow$ T Otherwise, 0 $\rightarrow$ T	0011nnnnmmmm0111	—	Comparison result

Table 40: Arithmetic operation instructions

Instruction		Operation	Instruction code	Privileged	T Bit
CMP/PZ	Rn	When $Rn \geq 0$ , $1 \rightarrow T$ Otherwise, $0 \rightarrow T$	0100nnnn00010001	—	Comparison result
CMP/PL	Rn	When $Rn > 0$ , $1 \rightarrow T$ Otherwise, $0 \rightarrow T$	0100nnnn00010101	—	Comparison result
CMP/STR	Rm,Rn	When any bytes are equal, $1 \rightarrow T$ Otherwise, $0 \rightarrow T$	0010nnnnmmmm1100	—	Comparison result
DIV1	Rm,Rn	1-step division ( $Rn \div Rm$ )	0011nnnnmmmm0100	—	Calculation result
DIV0S	Rm,Rn	MSB of $Rn \rightarrow Q$ , MSB of $Rm \rightarrow M$ , $M \wedge Q \rightarrow T$	0010nnnnmmmm0111	—	Calculation result
DIV0U		$0 \rightarrow M/Q/T$	0000000000011001	—	0
DMULS.L	Rm,Rn	Signed, $Rn \times Rm \rightarrow MAC$ , $32 \times 32 \rightarrow 64$ bits	0011nnnnmmmm1101	—	—
DMULU.L	Rm,Rn	Unsigned, $Rn \times Rm \rightarrow MAC$ , $32 \times 32 \rightarrow 64$ bits	0011nnnnmmmm0101	—	—
DT	Rn	$Rn - 1 \rightarrow Rn$ ; when $Rn = 0$ , $1 \rightarrow T$ When $Rn \neq 0$ , $0 \rightarrow T$	0100nnnn00010000	—	Comparison result
EXTS.B	Rm,Rn	Rm sign-extended from byte $\rightarrow Rn$	0110nnnnmmmm1110	—	—
EXTS.W	Rm,Rn	Rm sign-extended from word $\rightarrow Rn$	0110nnnnmmmm1111	—	—
EXTU.B	Rm,Rn	Rm zero-extended from byte $\rightarrow Rn$	0110nnnnmmmm1100	—	—
EXTU.W	Rm,Rn	Rm zero-extended from word $\rightarrow Rn$	0110nnnnmmmm1101	—	—
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	0000nnnnmmmm1111	—	—

Table 40: Arithmetic operation instructions

Instruction		Operation	Instruction code	Privileged	T Bit
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	0100nnnnnnmmmm1111	—	—
MUL.L	Rm,Rn	$Rn \times Rm \rightarrow MACL$ $32 \times 32 \rightarrow 32$ bits	0000nnnnnnmmmm0111	—	—
MULS.W	Rm,Rn	Signed, $Rn \times Rm \rightarrow MACL$ $16 \times 16 \rightarrow 32$ bits	0010nnnnnnmmmm1111	—	—
MULU.W	Rm,Rn	Unsigned, $Rn \times Rm \rightarrow MACL$ $16 \times 16 \rightarrow 32$ bits	0010nnnnnnmmmm1110	—	—
NEG	Rm,Rn	$0 - Rm \rightarrow Rn$	0110nnnnnnmmmm1011	—	—
NEGC	Rm,Rn	$0 - Rm - T \rightarrow Rn$ , borrow $\rightarrow T$	0110nnnnnnmmmm1010	—	Borrow
SUB	Rm,Rn	$Rn - Rm \rightarrow Rn$	0011nnnnnnmmmm1000	—	—
SUBC	Rm,Rn	$Rn - Rm - T \rightarrow Rn$ , borrow $\rightarrow T$	0011nnnnnnmmmm1010	—	Borrow
SUBV	Rm,Rn	$Rn - Rm \rightarrow Rn$ , underflow $\rightarrow T$	0011nnnnnnmmmm1011	—	Underflow

Table 40: Arithmetic operation instructions



Instruction		Operation	Instruction 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	When (Rn) = 0, $1 \rightarrow T$ Otherwise, $0 \rightarrow T$ In both cases, $1 \rightarrow$ MSB of (Rn)	0100nnnn00011011	—	Test result
TST	Rm,Rn	$Rn \& Rm$ ; when result = 0, $1 \rightarrow T$ Otherwise, $0 \rightarrow T$	0010nnnnmmmm1000	—	Test result
TST	#imm,R0	$R0 \& imm$ ; when result = 0, $1 \rightarrow T$ Otherwise, $0 \rightarrow T$	11001000iiiiiii	—	Test result
TST.B	#imm,@(R0,GBR)	$(R0 + GBR) \& imm$ ; when result = 0, $1 \rightarrow T$ Otherwise, $0 \rightarrow T$	11001100iiiiiii	—	Test 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 41: Logic operation instructions

Instruction		Operation	Instruction 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	When $Rn \geq 0$ , $Rn \ll Rm \rightarrow Rn$ When $Rn < 0$ , $Rn \gg Rm \rightarrow$ [MSB $\rightarrow Rn$ ]	0100nnnnnnmm1100	—	—
SHAL	Rn	$T \leftarrow Rn \leftarrow 0$	0100nnnn00100000	—	MSB
SHAR	Rn	$\text{MSB} \rightarrow Rn \rightarrow T$	0100nnnn00100001	—	LSB
SHLD	Rm,Rn	When $Rn \geq 0$ , $Rn \ll Rm \rightarrow Rn$ When $Rn < 0$ , $Rn \gg Rm \rightarrow$ [0 $\rightarrow Rn$ ]	0100nnnnnnmm1101	—	—
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 42: Shift instructions

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

Table 43: Branch instructions

Instruction		Operation	Instruction code	Privileged	T bit
CLRMACH		$0 \rightarrow \text{MACH}, \text{MACL}$	0000000000101000	—	—
CLRS		$0 \rightarrow S$	0000000001001000	—	—
CLRT		$0 \rightarrow T$	0000000000001000	—	0
LDC	Rm,SR	$Rm \rightarrow SR$	0100mmmm00001110	Privileged	LSB
LDC	Rm,GBR	$Rm \rightarrow GBR$	0100mmmm00011110	—	—
LDC	Rm,VBR	$Rm \rightarrow VBR$	0100mmmm00101110	Privileged	—
LDC	Rm,SSR	$Rm \rightarrow SSR$	0100mmmm00111110	Privileged	—
LDC	Rm,SPC	$Rm \rightarrow SPC$	0100mmmm01001110	Privileged	—
LDC	Rm,DBR	$Rm \rightarrow DBR$	0100mmmm11111010	Privileged	—
LDC	Rm,Rn_BANK	$Rm \rightarrow Rn\_BANK$ ( $n = 0$ to $7$ )	0100mmmm1nnn1110	Privileged	—
LDC.L	@Rm+,SR	$(Rm) \rightarrow SR, Rm + 4 \rightarrow Rm$	0100mmmm00000111	Privileged	LSB
LDC.L	@Rm+,GBR	$(Rm) \rightarrow GBR, Rm + 4 \rightarrow Rm$	0100mmmm00010111	—	—
LDC.L	@Rm+,VBR	$(Rm) \rightarrow VBR, Rm + 4 \rightarrow Rm$	0100mmmm00100111	Privileged	—
LDC.L	@Rm+,SSR	$(Rm) \rightarrow SSR, Rm + 4 \rightarrow Rm$	0100mmmm00110111	Privileged	—
LDC.L	@Rm+,SPC	$(Rm) \rightarrow SPC, Rm + 4 \rightarrow Rm$	0100mmmm01000111	Privileged	—
LDC.L	@Rm+,DBR	$(Rm) \rightarrow DBR, Rm + 4 \rightarrow Rm$	0100mmmm11110110	Privileged	—
LDC.L	@Rm+,Rn_BANK	$(Rm) \rightarrow Rn\_BANK,$ $Rm + 4 \rightarrow Rm$	0100mmmm1nnn0111	Privileged	—
LDS	Rm,MACH	$Rm \rightarrow MACH$	0100mmmm00001010	—	—
LDS	Rm,MACL	$Rm \rightarrow MACL$	0100mmmm00011010	—	—
LDS	Rm,PR	$Rm \rightarrow PR$	0100mmmm00101010	—	—
LDS.L	@Rm+,MACH	$(Rm) \rightarrow MACH, Rm + 4 \rightarrow Rm$	0100mmmm00000110	—	—
LDS.L	@Rm+,MACL	$(Rm) \rightarrow MACL, Rm + 4 \rightarrow Rm$	0100mmmm00010110	—	—
LDS.L	@Rm+,PR	$(Rm) \rightarrow PR, Rm + 4 \rightarrow Rm$	0100mmmm00100110	—	—

Table 44: System control instructions

Instruction		Operation	Instruction code	Privileged	T bit
LDTLB		PTEH/PTEL → TLB	000000000111000	Privileged	—
MOVCA.L	R0,@Rn	R0 → (Rn) (without fetching cache block)	0000nnnn11000011	—	—
NOP		No operation	0000000000001001	—	—
OCBI	@Rn	Invalidates operand cache block	0000nnnn10010011	—	—
OCBP	@Rn	Writes back and invalidates operand cache block	0000nnnn10100011	—	—
OCBWB	@Rn	Writes back operand cache block	0000nnnn10110011	—	—
PREF	@Rn	(Rn) → operand cache	0000nnnn10000011	—	—
RTE		Delayed branch, SSR/SPC → SR/PC	0000000000101011	Privileged	—
SETS		1 → S	0000000001011000	—	—
SETT		1 → T	0000000000011000	—	1
SLEEP		Sleep or standby	0000000000011011	Privileged	—
STC	SR,Rn	SR → Rn	0000nnnn00000010	Privileged	—
STC	GBR,Rn	GBR → Rn	0000nnnn00010010	—	—
STC	VBR,Rn	VBR → Rn	0000nnnn00100010	Privileged	—
STC	SSR,Rn	SSR → Rn	0000nnnn00110010	Privileged	—
STC	SPC,Rn	SPC → Rn	0000nnnn01000010	Privileged	—
STC	SGR,Rn	SGR → Rn	0000nnnn00111010	Privileged	—
STC	DBR,Rn	DBR → Rn	0000nnnn11111010	Privileged	—
STC	Rm_BANK,Rn	Rm_BANK → Rn (m = 0 to 7)	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	—

Table 44: System control instructions

Instruction		Operation	Instruction code	Privileged	T bit
STC.L	SSR,@-Rn	$Rn - 4 \rightarrow Rn$ , $SSR \rightarrow (Rn)$	0100nnnn00110011	Privileged	—
STC.L	SPC,@-Rn	$Rn - 4 \rightarrow Rn$ , $SPC \rightarrow (Rn)$	0100nnnn01000011	Privileged	—
STC.L	SGR,@-Rn	$Rn - 4 \rightarrow Rn$ , $SGR \rightarrow (Rn)$	0100nnnn00110010	Privileged	—
STC.L	DBR,@-Rn	$Rn - 4 \rightarrow Rn$ , $DBR \rightarrow (Rn)$	0100nnnn11110010	Privileged	—
STC.L	Rm_BANK,@-Rn	$Rn - 4 \rightarrow Rn$ , $Rm\_BANK \rightarrow (Rn)$ ( $m = 0$ to 7)	0100nnnn1mmm0011	Privileged	—
STS	MACH,Rn	$MACH \rightarrow Rn$	0000nnnn00001010	—	—
STS	MACL,Rn	$MACL \rightarrow Rn$	0000nnnn00011010	—	—
STS	PR,Rn	$PR \rightarrow Rn$	0000nnnn00101010	—	—
STS.L	MACH,@-Rn	$Rn - 4 \rightarrow Rn$ , $MACH \rightarrow (Rn)$	0100nnnn00000010	—	—
STS.L	MACL,@-Rn	$Rn - 4 \rightarrow Rn$ , $MACL \rightarrow (Rn)$	0100nnnn00010010	—	—
STS.L	PR,@-Rn	$Rn - 4 \rightarrow Rn$ , $PR \rightarrow (Rn)$	0100nnnn00100010	—	—
TRAPA	#imm	$PC + 2 \rightarrow SPC$ , $SR \rightarrow SSR$ , #imm < 2 $\rightarrow TRA$ , 0x160 $\rightarrow EXPEVT$ , $VBR + 0x0100 \rightarrow PC$	11000011iiiiiiii	—	—

Table 44: System control instructions

Instruction		Operation	Instruction code	Privileged	T bit
FLDI0	FRn	0x00000000 → FRn	1111nnnn10001101	—	—
FLDI1	FRn	0x3F800000 → FRn	1111nnnn10011101	—	—
FMOV	FRm,FRn	FRm → FRn	1111nnnnnnmm1100	—	—
FMOV.S	@Rm,FRn	(Rm) → FRn	1111nnnnnnmm1000	—	—
FMOV.S	@(R0,Rm),FRn	(R0 + Rm) → FRn	1111nnnnnnmm0110	—	—
FMOV.S	@Rm+,FRn	(Rm) → FRn, Rm + 4 → Rm	1111nnnnnnmm1001	—	—
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	1111nnn0mm01100	—	—
FMOV	@Rm,DRn	(Rm) → DRn	1111nnn0mmmm1000	—	—
FMOV	@(R0,Rm),DRn	(R0 + Rm) → DRn	1111nnn0mmmm0110	—	—
FMOV	@Rm+,DRn	(Rm) → DRn, Rm + 8 → Rm	1111nnn0mmmm1001	—	—
FMOV	DRm,@Rn	DRm → (Rn)	1111nnnnmm01010	—	—
FMOV	DRm,@-Rn	Rn-8 → Rn, DRm → (Rn)	1111nnnnmm01011	—	—
FMOV	DRm,@(R0,Rn)	DRm → (R0 + Rn)	1111nnnnmm00111	—	—
FLDS	FRm,FPUL	FRm → FPUL	1111mmmm00011101	—	—
FSTS	FPUL,FRn	FPUL → FRn	1111nnnn00001101	—	—
FABS	FRn	FRn & 0x7FFF FFFF → FRn	1111nnnn01011101	—	—
FADD	FRm,FRn	FRn + FRm → FRn	1111nnnnnnmm0000	—	—
FCMP/EQ	FRm,FRn	When FRn = FRm, 1 → T Otherwise, 0 → T	1111nnnnnnmm0100	—	Comparison result

Table 45: Floating-point single-precision instructions

Instruction		Operation	Instruction code	Privileged	T bit
FCMP/GT	FRm,FRn	When $FRn > FRm$ , $1 \rightarrow T$ Otherwise, $0 \rightarrow T$	1111nnnnnnmmmm0101	—	Comparison result
FDIV	FRm,FRn	$FRn/FRm \rightarrow FRn$	1111nnnnnnmmmm0011	—	—
FLOAT	FPUL,FRn	(float) $FPUL \rightarrow FRn$	1111nnnn00101101	—	—
FMAC	FR0,FRm,FRn	$FR0*FRm + FRn \rightarrow FRn$	1111nnnnnnmmmm1110	—	—
FMUL	FRm,FRn	$FRn*FRm \rightarrow FRn$	1111nnnnnnmmmm0010	—	—
FNEG	FRn	$FRn \wedge 0x80000000 \rightarrow FRn$	1111nnnn01001101	—	—
FSQRT	FRn	$\sqrt{FRn} \rightarrow FRn$	1111nnnn01101101	—	—
FSUB	FRm,FRn	$FRn - FRm \rightarrow FRn$	1111nnnnnnmmmm0001	—	—
FTRC	FRm,FPUL	(long) $FRm \rightarrow FPUL$	1111mmmm00111101	—	—

Table 45: Floating-point single-precision instructions

Instruction		Operation	Instruction code	Privileged	T bit
FABS	DRn	$DRn \& 0x7FFF\ FFFF\ FFFF\ FFFF \rightarrow DRn$	1111nnnn001011101	—	—
FADD	DRm,DRn	$DRn + DRm \rightarrow DRn$	1111nnnn0mmmm00000	—	—
FCMP/EQ	DRm,DRn	When $DRn = DRm$ , $1 \rightarrow T$ Otherwise, $0 \rightarrow T$	1111nnnn0mmmm00100	—	Comparison result
FCMP/GT	DRm,DRn	When $DRn > DRm$ , $1 \rightarrow T$ Otherwise, $0 \rightarrow T$	1111nnnn0mmmm00101	—	Comparison result
FDIV	DRm,DRn	$DRn / DRm \rightarrow DRn$	1111nnnn0mmmm00011	—	—
FCNVDS	DRm,FPUL	$\text{double\_to\_float}[DRm] \rightarrow FPUL$	1111mmmm010111101	—	—
FCNVSD	FPUL,DRn	$\text{float\_to\_double}[FPUL] \rightarrow DRn$	1111nnnn010101101	—	—

Table 46: Floating-point double-precision instructions



Instruction		Operation	Instruction code	Privileged	T bit
FLOAT	FPUL,DRn	(float)FPUL $\rightarrow$ DRn	1111nnn000101101	—	—
FMUL	DRm,DRn	DRn *DRm $\rightarrow$ DRn	1111nnn0mmm00010	—	—
FNEG	DRn	DRn $\wedge$ 0x8000 0000 0000 0000 $\rightarrow$ DRn	1111nnn001001101	—	—
FSQRT	DRn	$\sqrt{\text{DRn}}$ $\rightarrow$ DRn	1111nnn001101101	—	—
FSUB	DRm,DRn	DRn – DRm $\rightarrow$ DRn	1111nnn0mmm00001	—	—
FTRC	DRm,FPUL	(long) DRm $\rightarrow$ FPUL	1111mmm000111101	—	—

Table 46: Floating-point double-precision instructions

Instruction		Operation	Instruction code	Privileged	T bit
LDS	Rm,FPSCR	Rm $\rightarrow$ FPSCR	0100mmmm01101010	—	—
LDS	Rm,FPUL	Rm $\rightarrow$ FPUL	0100mmmm01011010	—	—
LDS.L	@Rm+,FPSCR	(Rm) $\rightarrow$ FPSCR, Rm+4 $\rightarrow$ Rm	0100mmmm01100110	—	—
LDS.L	@Rm+,FPUL	(Rm) $\rightarrow$ FPUL, Rm+4 $\rightarrow$ Rm	0100mmmm01010110	—	—
STS	FPSCR,Rn	FPSCR $\rightarrow$ Rn	0000nnnn01101010	—	—
STS	FPUL,Rn	FPUL $\rightarrow$ Rn	0000nnnn01011010	—	—
STS.L	FPSCR,@-Rn	Rn – 4 $\rightarrow$ Rn, FPSCR $\rightarrow$ (Rn)	0100nnnn01100010	—	—
STS.L	FPUL,@-Rn	Rn – 4 $\rightarrow$ Rn, FPUL $\rightarrow$ (Rn)	0100nnnn01010010	—	—

Table 47: Floating-point control instructions

Instruction		Operation	Instruction Code	Privileged	T Bit
FMOV	DRm, XDn	DRm $\rightarrow$ XDn	1111nnn1mmm01100	—	—
FMOV	XDm, DRn	XDm $\rightarrow$ DRn	1111nnn0mmm11100	—	—
FMOV	XDm, XDn	XDm $\rightarrow$ XDn	1111nnn1mmm11100	—	—
FMOV	@Rm, XDn	(Rm) $\rightarrow$ XDn	1111nnn1mmmm1000	—	—
FMOV	@Rm+, XDn	(Rm) $\rightarrow$ XDn, Rm + 8 $\rightarrow$ Rm	1111nnn1mmmm1001	—	—
FMOV	@(R0,Rm), XDn	(R0 + Rm) $\rightarrow$ XDn	1111nnn1mmmm0110	—	—
FMOV	XDm, @Rn	XDm $\rightarrow$ (Rn)	1111nnnnmmmm11010	—	—
FMOV	XDm, @-Rn	Rn - 8 $\rightarrow$ Rn, XDm $\rightarrow$ (Rn)	1111nnnnmmmm11011	—	—
FMOV	XDm, @(R0,Rn)	XDm $\rightarrow$ (R0+Rn)	1111nnnnmmmm10111	—	—
FIPR	FVm, FVn	inner_product [FVm, FVn] $\rightarrow$ FR[n+3]	1111nnmm11101101	—	—
FTRV	XMTRX, FVn	transform_vector [XMTRX, FVn] $\rightarrow$ FVn	1111nn0111111101	—	—
FRCHG		$\sim$ FPSCR.FR $\rightarrow$ SPFCR.FR	1111101111111101	—	—
FSCHG		$\sim$ FPSCR.SZ $\rightarrow$ SPFCR.SZ	1111001111111101	—	—

Table 48: Floating-point graphics acceleration instructions



## 8

# Instruction specification

## 8.1 Overview

The behavior of instructions is specified using a simple notational language to describe the effects of each instruction on the architectural state of the machine.

The language consists of the following features:

- A simple variable and type system.
- Expressions.
- Statements.
- Notation for the architectural state of the machine.
- An abstract sequential model of instruction execution.

These features are described in the following sections. Additional mechanisms are defined to model memory, synchronization instructions, cache instructions and floating-point. The final section gives example instruction specifications.

Each instruction is described using informal text as well as the formal notational language. Sometimes it is inappropriate for one of these descriptions to convey the full semantics. In such cases these two descriptions must be taken together to constitute the full specification.

## 8.2 Variables and types

Variables are used to hold state. The type of a variable determines the set of values that the variable can take and the available operators to manipulate that variable. The supported scalar types are integers, booleans and bit-fields. One-dimensional arrays of the scalar types are also supported.

The architectural state of the machine is represented by a set of variables. Each of these variables has an associated type, which is either a bit-field or an array of bit-fields. Bit-fields are used to give a bit-accurate representation.

Additional variables are used to hold temporary values. The type of temporary variables is implicit, and determined by their context rather than explicit declaration. The type of a temporary variable is an integer, a boolean or an array of these.

### 8.2.1 Integer

An integer variable can take the value of any mathematical integer. No limits are imposed on the range of integers supported. Integers obey their standard mathematical properties. Integer operations do not overflow. The integer operators are defined so that singularities do not occur. For example, no definition is given to the result of divide by zero; the operator is simply not available when the divisor is zero.

The representation of literal integer values is achieved using the following notations:

- Decimal numbers are represented by the regular expression: `{0-9}+`
- Hexadecimal numbers are represented by the regular expression: `0x{0-9a-fA-F}+`
- Binary numbers are represented by the regular expression: `0b{0-1}+`

These notations are standard and map onto integer values in the obvious way. Underscore characters ('\_') can be inserted into any of the above literal representations. These do not change the represented value but can be used as spacers to aid readability.

The notations allow only zero and positive numbers to be represented directly. A monadic integer negation operator can subsequently be used to derive a negative value.

## 8.2.2 Boolean

A boolean variable can take two values:

- Boolean false. The literal representation of boolean false is 'FALSE'.
- Boolean true. The literal representation of boolean true is 'TRUE'.

## 8.2.3 Bit-fields

Bit-fields are provided to define 'bit-accurate' storage.

Bit-fields containing arbitrary numbers of bits are supported. A bit-field of  $b$  bits contains bits numbered from 0 (the least significant bit) up to  $b-1$  (the most significant bit). Each bit can take the value 0 or the value 1. Bit-fields are mapped to, and from, integers in the usual way. If bit  $i$  of a  $b$ -bit, bit-field, where  $i$  is in  $[0, b)$ , is set then it contributes  $2^i$  to the integral value of the bit-field. The integral value of the bit-field as a whole is an integer in the range  $[0, 2^b)$ .

When a bit-field is read, it gives its integral value. When a bit-field is written with an integral value, the integer must be in the range of values supported by the bit-field. Typically, the only operations applied directly to bit-fields are conversions to other types.

## 8.2.4 Arrays

One-dimensional arrays of the above types are also available. Indexing into an  $n$ -element array  $A$  is achieved using the notation  $A[i]$  where  $A$  is an array of some type and  $i$  is an integer in the range  $[0, n)$ . This selects the  $i^{\text{th}}$  element of the array  $A$ . If  $i$  is zero this selects the first entry, and if  $i$  is  $n-1$  then this selects the last entry. The type of the selected element is the base type of the array.

Multi-dimensional arrays are not provided.

## 8.2.5 Floating point values

Floating-point types and operators are not provided. Instead, the value in a floating-point register is represented as a bit-field. The organization of the bit-field is consistent with an IEEE754 format.

When a floating-point register is read, an integral representation of that bit-pattern is returned. When an integral value is written into a floating-point register, the value written is the bit-pattern of that integer. Thus, reading and writing is achieved as bit-pattern transfers, and not by interpreting the bit-patterns as real numbers.

The language does not provide direct means to interpret these bit-patterns as real numbers. Instead, functions are provided which give the required functionality. For example, arithmetic on real numbers is represented using a function notation.

## 8.3 Expressions

Expressions are constructed from monadic operators, dyadic operators and functions applied to variable and literal values.

There are no defined precedence and associativity rules for the operators. Parentheses are used to specify the expression unambiguously.

Sub-expressions can be evaluated in any order. If a particular evaluation order is required, then sub-expressions must be split into separate statements.

### 8.3.1 Integer arithmetic operators

Since the notation uses straightforward mathematical integers, the set of standard mathematical operators is available and already defined.

The standard dyadic operators are listed in [Table 49](#).

Operation	Description
$i + j$	Integer addition
$i - j$	Integer subtraction
$i \times j$	Integer multiplication

**Table 49: Standard dyadic operators**

Operation	Description
$i / j$	Integer division
$i \setminus j$	Integer remainder

Table 49: Standard dyadic operators

The standard monadic operators are described in [Table 50](#).

Operator	Description
$-i$	Integer negation
$ i $	Integer modulus

Table 50: Standard monadic operators

The division operator truncates towards zero. The remainder operator is consistent with this. The sign of the result of the remainder operator follows the sign of the dividend. Division or remainder with a divisor of zero results in a singularity, and its behavior is not defined.

For a numerator ( $n$ ) and a denominator ( $d$ ), the following properties hold where  $d \neq 0$ :

$$\begin{aligned}
 n &= d \times (n/d) + (n \setminus d) \\
 (-n)/d &= -(n/d) = n/(-d) \\
 (-n) \setminus d &= -(n \setminus d) \\
 n \setminus (-d) &= n \setminus d \\
 0 \leq (n \setminus d) < d &\text{ where } n \geq 0 \text{ and } d > 0
 \end{aligned}$$

### 8.3.2 Integer shift operators

The available integer shift operators are listed in [Table 51](#).

Operation	Description
$n \ll b$	Integer left shift
$n \gg b$	Integer right shift

**Table 51: Shift operators**

The shift operators are defined on integers as follows where  $b \geq 0$ :

$$n \ll b = n \times 2^b$$

$$n \gg b = \begin{cases} n/2^b & \text{where } n \geq 0 \\ (n - 2^b + 1)/2^b & \text{where } n < 0 \end{cases}$$

Note that right shifting rounds the result towards minus infinity. This contrasts with division, which rounds towards zero, and is the reason why the right shift definition is separate for positive and negative  $n$ .

### 8.3.3 Integer bitwise operators

The available integer bitwise operators are listed in [Table 52](#).

Operation	Description
$i \wedge j$	Integer bitwise AND
$i \vee j$	Integer bitwise OR
$i \oplus j$	Integer bitwise XOR
$\sim i$	Integer bitwise NOT
$n_{<b \text{ FOR } m>}$	Integer bit-field extraction: extract $m$ bits starting at bit $b$ from integer $n$
$n_{<b>}$	Integer bit-field extraction: extract 1 bit starting at bit $b$ from integer $n$

**Table 52: Bitwise operators**



In order to define bitwise operations all integers are considered as having an infinitely long two's complement representation. Bit 0 is the least significant bit of this representation, bit 1 is the next higher bit, and so on. The value of bit  $b$ , where  $b \geq 0$ , in integer  $n$  is given by:

$$\begin{aligned} \text{BIT}(n, b) &= (n/2^b) \setminus 2 & \text{where } n \geq 0 \\ \text{BIT}(-n, b) &= 1 - \text{BIT}(n - 1, b) & \text{where } n > 0 \end{aligned}$$

Care must be taken whenever the infinitely long two's complement representation of a negative number is constructed. This representation will contain an infinite number of higher bits with the value 1 representing the sign. Typically, a subsequent conversion operation is used to discard these upper bits and return the result back to a finite value.

Bitwise AND ( $\wedge$ ), OR ( $\vee$ ), XOR ( $\oplus$ ) and NOT ( $\sim$ ) are defined on integers as follows, where  $b$  takes all values such that  $b \geq 0$ :

$$\begin{aligned} \text{BIT}(i \wedge j, b) &= \text{BIT}(i, b) \times \text{BIT}(j, b) \\ \text{BIT}(i \vee j, b) &= \text{BIT}(i \wedge j, b) + \text{BIT}(i \oplus j, b) \\ \text{BIT}(i \oplus j, b) &= (\text{BIT}(i, b) + \text{BIT}(j, b)) \setminus 2 \\ \text{BIT}(\sim i, b) &= 1 - \text{BIT}(i, b) \end{aligned}$$

*Note: Bitwise NOT of any finite positive  $i$  will result in a value containing an infinite number of higher bits with the value 1 representing the sign.*

Bitwise extraction is defined on integers as follows, where  $b \geq 0$  and  $m > 0$ :

$$\begin{aligned} n_{\langle b \text{ FOR } m \rangle} &= (n \gg b) \wedge (2^m - 1) \\ n_{\langle b \rangle} &= n_{\langle b \text{ FOR } 1 \rangle} \end{aligned}$$

The result of  $n_{\langle b \text{ FOR } m \rangle}$  is an integer in the range  $[0, 2^m)$ .

### 8.3.4 Relational operators

Relational operators are defined to compare integral values and give a boolean result.

Operation	Description
$i = j$	Result is true if $i$ is equal to $j$ , otherwise false
$i \neq j$	Result is true if $i$ is not equal to $j$ , otherwise false
$i < j$	Result is true if $i$ is less than $j$ , otherwise false
$i > j$	Result is true if $i$ is greater than $j$ , otherwise false
$i \leq j$	Result is true if $i$ is less than or equal to $j$ , otherwise false
$i \geq j$	Result is true if $i$ is greater than or equal to $j$ , otherwise false

**Table 53: Relational operators**

### 8.3.5 Boolean operators

Boolean operators are defined to perform logical AND, OR, XOR and NOT. These operators have boolean sources and result. Additionally, the conversion operator INT is defined to convert a boolean source into an integer result.

Operation	Description
$i \text{ AND } j$	Result is true if $i$ and $j$ are both true, otherwise false
$i \text{ OR } j$	Result is true if either/both $i$ and $j$ are true, otherwise false
$i \text{ XOR } j$	Result is true if exactly one of $i$ and $j$ are true, otherwise false
NOT $i$	Result is true if $i$ is false, otherwise false
INT $i$	Result is 0 if $i$ is false, otherwise 1

**Table 54: Boolean operators**

### 8.3.6 Single-value functions

In some cases it is inconvenient or inappropriate to describe an expression directly in the specification language. In these cases a function call is used to reference the undescribed behavior.

A single-value function evaluates to a single value (the result), which can be used in an expression. The type of the result value can be determined by the expression context from which the function is called. There are also multiple-value functions which evaluate to multiple values. These are only available in an assignment context, and are described in [Section 8.4.2: Assignment on page 190](#).

Functions can contain side-effects.

#### Scalar conversions

Two monadic functions are defined to support conversions between integral representations of finite-precision signed and unsigned number spaces. These functions are often used to convert between bit-fields and integer values.

Function	Description
ZeroExtend <sub>n</sub> (i)	Convert integer i to an n-bit 2's complement unsigned range
SignExtend <sub>n</sub> (i)	Convert integer i to an n-bit 2's complement signed range

**Table 55: Integer conversion operators**

These two functions are defined as follows, where  $n > 0$ :

$$\begin{aligned} \text{ZeroExtend}_n(i) &= i_{\langle 0 \text{ FOR } n \rangle} \\ \text{SignExtend}_n(i) &= \begin{cases} i_{\langle 0 \text{ FOR } n \rangle} & \text{where } i_{\langle n-1 \rangle} = 0 \\ i_{\langle 0 \text{ FOR } (n-1) \rangle} - 2^n & \text{where } i_{\langle n-1 \rangle} = 1 \end{cases} \end{aligned}$$

For syntactic convenience, conversion functions are also defined for converting an integer to a single bit and to a 32-bit register. [Table 56](#) shows the additional functions provided.

Operation	Description
Bit(i)	Convert lowest bit of integer i to a 1-bit value This is a convenient notation for $i_{<0>}$
Register(i)	Convert lowest 32 bits of integer i to a 32-bit value This is a convenient notation for $i_{<0 \text{ FOR } 32>}$

**Table 56: Conversion operators from integers to bit-fields**

### Floating-point conversions

The specification language manipulates floating-point values as integers containing the associated IEEE754 bit-pattern. The layout of these bit-patterns is described in [Chapter 6: Floating-point unit on page 145](#). The language does not support a floating-point type.

Conversion functions are defined to support floating-point. Floating-point values are held as either scalar values in a single register, or vector values in multiple registers. The available register formats are:

- One 32-bit value in a single-precision register.
- One 64-bit value in a double-precision register.
- Two 32-bit values in a pair of single-precision registers.
- Four 32-bit values in a four-entry vector of single-precision registers.
- Sixteen 32-bit values in a four-by-four matrix of single-precision registers.

Conversions are available to convert between register bit-fields in these formats and integers or arrays of integers holding the appropriate IEEE754 bit-patterns.

The following conversions are provided to convert from floating-point registers:

Operation	Description
FloatValue <sub>32</sub> (r)	Convert a single-precision floating-point register into a 32-bit integer bit-pattern.
FloatValue <sub>64</sub> (r)	Convert a double-precision floating-point register into a 64-bit integer bit-pattern.
FloatValuePair <sub>32</sub> (r)	Convert a pair of single-precision floating-point registers into an array of 2 x 32-bit integer bit-patterns.
FloatValueVector <sub>32</sub> (r)	Convert a 4-entry vector of single-precision floating-point registers into an array of 4 x 32-bit integer bit-patterns.
FloatValueMatrix <sub>32</sub> (r)	Convert a 16-entry matrix of single-precision floating-point registers into an array of 16 x 32-bit integer bit-patterns.

**Table 57: Conversion from floating-point register formats**

The following conversions are provided to convert to floating-point registers:

Operation	Description
FloatRegister <sub>32</sub> (i)	Convert a 32-bit integer bit-pattern into a single-precision floating-point register.
FloatRegister <sub>64</sub> (i)	Convert a 64-bit integer bit-pattern into a double-precision floating-point register.
FloatRegisterPair <sub>32</sub> (a)	Convert an array of 2 x 32-bit integer bit-patterns into a pair of single-precision floating-point registers.
FloatRegisterVector <sub>32</sub> (a)	Convert an array of 4 x 32-bit integer bit-patterns into a 4-entry vector of single-precision floating-point registers.
FloatRegisterMatrix <sub>32</sub> (a)	Convert an array of 16 x 32-bit integer bit-patterns into a 16-entry matrix of single-precision floating-point registers.

**Table 58: Conversion to floating-point register formats**

## 8.4 Statements

An instruction specification consists of a sequence of statements. These statements are processed sequentially in order to specify the effect of the instruction on the architectural state of the machine. The available statements are discussed in this section.

Each statement has a semi-colon terminator. A sequence of statements can be aggregated into a statement block using '{' to introduce the block and '}' to terminate the block. A statement block can be used anywhere that a statement can.

### 8.4.1 Undefined behavior

The statement:

```
UNDEFINED( ) ;
```

indicates that the resultant behavior is architecturally undefined.

A particular implementation can choose to specify an implementation-defined behavior in such cases. It is very likely that any implementation-defined behavior will vary from implementation to implementation. Exploitation of implementation-defined behavior should be avoided to allow software to be portable between implementations.

In cases where architecturally undefined behavior can occur in user mode, the implementation will ensure that implemented behavior does not break the protection model. Thus, the implemented behavior will be some execution flow that is permitted for that user mode thread.

### 8.4.2 Assignment

The '←' operator is used to denote assignment of an expression to a variable. An example assignment statement is:

```
variable ← expression;
```

The expression can be constructed from variables, literals, operators and functions as described in [Section 8.3: Expressions on page 182](#). The expression is fully evaluated before the assignment takes place. The variable can be an integer, a boolean, a bit-field or an array of one of these types.

### Assignment to architectural state

This is where the variable is part of the architectural state (as described in *Table 59: Scalar architectural state on page 194*). The type of the expression and the type of the variable must match.

### Assignment to a temporary

Alternatively, if the variable is not part of the architectural state, then it is a temporary variable. The type of the variable is determined by the type of expression. A temporary variable must be assigned to, before it is used in the instruction specification.

### Assignment of an undefined value

An assignment of the following form results in a variable being initialized with an architecturally undefined value:

```
variable ← UNDEFINED;
```

After assignment the variable will hold a value which is valid for its type. However, the value is architecturally undefined. The actual value can be unpredictable; that is to say the value indicated by UNDEFINED can vary with each use of UNDEFINED. Architecturally-undefined values can occur in both user and privileged modes.

A particular implementation can choose to specify an implementation-defined value in such cases. It is very likely that any implementation-defined values will vary from implementation to implementation. Exploitation of implementation-defined values should be avoided to allow software to be portable between implementations.

### Assignment of multiple values

Multi-value functions are used to return multiple values, and are only available when used in a multiple assignment context. The syntax consists of a list of comma-separated variables, an assignment symbol followed by a function call. The function is evaluated and returns multiple results into the variables listed. The number of variables and the number of results of the function must match. The assigned variables must all be distinct (i.e. no aliases).

For example, a two-valued assignment from a function call with 3 parameters can be represented as:

```
variable1, variable2 ← call(param1, param2, param3);
```

### 8.4.3 Conditional

Conditional behavior is specified using 'IF', 'ELSE IF' and 'ELSE'.

Conditions are expressions that result in a boolean value. If the condition after an 'IF' is true, then its block of statements is executed and the whole conditional then completes. If the condition is false, then any 'ELSE IF' clauses are processed, in turn, in the same fashion. If no conditions are met and there is an 'ELSE' clause then its block of statements is executed. Finally, if no conditions are met and there is no 'ELSE' clause, then the statement has no effect apart from the evaluation of the condition expressions.

The 'ELSE IF' and 'ELSE' clauses are optional. In ambiguous cases, the 'ELSE' matches with the nearest 'IF'.

For example:

```
IF (condition1)
    block1
ELSE IF (condition2)
    block2
ELSE
    block3
```

### 8.4.4 Repetition

Repetitive behavior is specified using the following construct:

```
REPEAT i FROM m FOR n STEP s block
```

The block of statements is iterated n times, with the integer i taking the values:

m, m + s, m + 2s, m + 3s, up to m + (n - 1) × s.

The behavior is equivalent to textually writing the block n times with i being substituted with the appropriate value in each copy of the block.

The value of n must be greater or equal to 0, and the value of s must be non-zero. The values of the expressions for m, n and s must be constant across the iteration. The integer i must not be assigned to within the iterated block. The 'STEP s' can be omitted in which case the step-size takes the default value of 1.



## 8.4.5 Exceptions

Exception handling is triggered by a **THROW** statement. When an exception is thrown, no further statements are executed from the instruction specification and control passes to an exception handler. The actions associated with the launch of the handler are not shown in the instruction specification, but are described separately in *Chapter 5: Exceptions on page 105*.

There are two forms of throw statement:

```
THROW type;
```

and:

```
THROW type, value;
```

where **type** indicates the type of exception which is launched, and **value** is an optional argument to the exception handling sequence.

The full set of exceptions is described in *Chapter 5: Exceptions on page 105*.

## 8.4.6 Procedures

Procedure statements contain a procedure name followed by a list of comma-separated arguments contained within parentheses followed by a semi-colon. The execution of procedures typically causes side-effects to the architectural state of the machine.

Procedures are generally used where it is difficult or inappropriate to specify the effect of an instruction using the abstract execution model. A fuller description of the effect of the instruction will be given in the surrounding text.

An example procedure with two parameters is:

```
proc(param1, param2);
```

## 8.5 Architectural state

The architectural state is described in [Chapter 2: Programming model on page 21](#). The notations used in the model to refer to this state are summarized in [Table 59](#) and [Table 60](#). Each item of scalar architectural state is a bit-field of a particular width. Each item of array architectural state is an array of bit-fields of a particular width.

Architectural state	Type is a bit-field containing:	Description
MD (SR.MD)	1 bit	User (0) or privileged (1) mode
PC	32 bits	32-bit program counter
MMUCR	32 bits	For details of the MMU control register see <a href="#">Chapter 3: Memory management unit (MMU) on page 41</a> .
FPSCR	32 bits	32-bit floating-point status and control register
GBR	32 bits	Global base register
MACL	32 bits	Multiply-accumulate low
MACH	32 bits	Multiply-accumulate high
PR	32 bits	Procedure link register
T	1 bit	Condition code flag
S	1 bit	Multiply-accumulate saturation flag
M	1 bit	Divide-step M flag
Q	1 bit	Divide-step Q flag
FPUL	32 bits	FPU communication register
$R_i$ where $i$ is in $[0, 15]$	32 bits	16 x 32-bit general purpose registers
$FR_i$ where $i$ is in $[0, 31]$	32 bits	32 x 32-bit floating-point registers
$DR_{2i}$ where $i$ is in $[0, 15]$	64 bits	16 x 64-bit floating-point registers

**Table 59: Scalar architectural state**

Architectural state	Type is an array of bit-fields each containing:	Description
$FP_{2i}$ where $i$ is in $[0, 31]$	32 bits	32 pairs of 32-bit floating-point registers
$FV_{4i}$ where $i$ is in $[0, 15]$	32 bits	16 vectors of 4 x 32-bit floating-point registers
$MTRX_{16i}$ where $i$ is in $[0, 3]$	32 bits	4 matrices of 16 x 32-bit floating-point registers
$MEM[i]$ where $i$ is in $[0, 2^{32})$	8 bits	$2^{32}$ bytes of memory
$UTLB[i]$ where $i$ is in $[0, 63]$	a UTLB entry	Used for translation, for further details see <a href="#">Chapter 3: Memory management unit (MMU)</a> on page 41.

Table 60: Array architectural state

*Note:*  $FR$ ,  $FP$ ,  $FV$ ,  $MTRX$  and  $DR$  provide different views of the same architectural state.

There is no implicit meaning to the value held by the collection of bits in a register. The interpretation of the register is supplied by each instruction that reads or writes the register value.

PC denotes the program counter of the currently executing instruction. PC' denotes the program counter of the next instruction that is to be executed.

## 8.6 Memory model

Instruction specification uses a simple model of memory. It assumes, for example, that any caches have no architectural visibility. For typical well-disciplined instruction sequences these effects will not be architecturally visible. However, a fuller description of the behavior in other cases is defined by the text of the architecture manual.

MEM is an array of bytes indexed by an effective address. Elements in arrays are selected using array indexing notation: MEM[i] selects the  $i^{\text{th}}$  entry in the MEM array. The total range of array indices into MEM is  $[0, 2^{32})$ , though not all of this memory is available on all implementations.

Array slicing can be used to view an array as consisting of elements of a larger size. The notation MEM[s FOR n], where  $n > 0$ , denotes a memory slice containing the elements MEM[s], MEM[s+1] through to MEM[s+n-1]. The type of this slice is a bit-field exactly large enough to contain a concatenation of the n selected elements. In this case it contain  $8n$  bits since the base type of MEM is byte.

The order of the concatenation depends on the endianness of the processor:

- If the processor is operating in a little-endian mode, the concatenation order obeys the following condition as  $i$  (the byte number) varies in the range  $[0, n)$ :

$$(\text{MEM}[s \text{ FOR } n])_{\langle 8i \text{ FOR } 8 \rangle} = \text{MEM}[s + i]$$

This equivalence states that byte number  $i$ , using little-endian byte numbering (i.e. byte 0 is bits 0 to 7), in the bit-field MEM[s FOR n] is the  $i^{\text{th}}$  byte in memory counting upwards from MEM[s].

- If the processor is operating in a big-endian mode, the concatenation order obeys the following condition as  $i$  (the byte number) varies in the range  $[0, n)$ :

$$(\text{MEM}[s \text{ FOR } n])_{\langle 8(n-1-i) \text{ FOR } 8 \rangle} = \text{MEM}[s + i]$$

This equivalence states that byte number  $i$ , using big-endian byte numbering (i.e. byte 0 is bits  $8n-8$  to  $8n-1$ ), in the bit-field MEM[s FOR n] is the  $i^{\text{th}}$  byte in memory counting upwards from MEM[s].

For syntactic convenience, functions and procedures are provided to read, write and swap memory. The basic primitives support aligned accesses. Misaligned read and write primitives support the instructions for misaligned load and store.

Additionally, mechanisms are provided for reading and writing pairs of values. Pair access requires that each half of the pair is endianness converted separately, and that the lower half is written into memory at the provided address while the upper half is written into that address plus the object size. This maintains the ordering of the halves of the pair as they are transferred between registers and memory. Pair access is used only for loading and storing pairs of single-precision floating-point registers (see [Chapter 6: Floating-point unit on page 145](#)).

## 8.6.1 Support functions

The specification of the memory instructions relies on the support functions listed in [Table 61](#). These functions are used to model the behavior of the memory management unit described in [Chapter 3: Memory management unit \(MMU\) on page 41](#).

Function	Description
AddressUnavailable(address)	Returns true if the provided address is outside of the available part of the effective address space. For further details refer to <a href="#">Chapter 3: Memory management unit (MMU) on page 41</a> .
MMU()	Returns true if the MMU is enabled.
DataAccessMiss(address)	Returns true if the provided address does not have a mapping for a data access.
InstFetchMiss(address)	Returns true if the provided address does not have a mapping for an instruction fetch.
InstInvalidateMiss(address)	Returns true if the provided address does not have a mapping for an instruction invalidation.
ReadProhibited(address)	Returns true if the provided address has no read permission for the current privilege.
WriteProhibited(address)	Returns true if the provided address has no write permission for the current privilege.
ExecuteProhibited(address)	Returns true if the provided address has no execute permission for the current privilege.

**Table 61: Support functions for memory access**

Function	Description
DirtyBit(address)	Returns the value of the dirty bit (D) in the UTLB for the translation used for the specified address.
IsLittleEndian()	Returns true if processor is little-endian.

**Table 61: Support functions for memory access**

More detailed properties of translation miss detection are not modelled here. The conditions that determine whether an access is a translation miss or a hit depend on the MMU and cache.

DataAccessMiss is used to check for the absence of a data translation. This function is used for all data accesses when the MMU is enabled. InstFetchMiss is used to check for instruction fetches.

## 8.6.2 Reading memory

Functions are provided to read memory.

Function	Description
ReadMemory <sub>n</sub> (address)	Aligned memory read of an n-bit value
ReadMemoryPair <sub>n</sub> (address)	Aligned memory read of a pair of n-bit values

**Table 62: Support functions to read memory**

The ReadMemory<sub>n</sub> function takes an integer parameter to indicate the address being accessed. The number of bits being read (n) is one of 8, 16 or 32 bits. The required bytes are read from memory, interpreted according to endianness, and an integer result returns the read bit-field value. If the read memory value is to be interpreted as signed, then a sign-extension should be used on the result.

The assignment:

```
result ← ReadMemoryn(a);
```

is equivalent to:

```
width ← n >> 3;
IF (AddressUnavailable(a) OR ((a^(width-1)) ≠ 0)) THROW
RADDERR,a;
IF (MMU() AND DataAccessMiss(a)) THROW RTLBMISSE,a;
IF (MMU() AND ReadProhibited(a)) THROW READPROT,a;
result ← MEM[a FOR width];
```

ReadMemoryPair<sub>n</sub> reads a pair of n-bit values. The alignment check requires alignment for a 2n-bit access. The access maintains the ordering of the two halves of the pair, with endianness applied separately to each half. The assignment:

```
result ← ReadMemoryPairn(a);
```

is equivalent to:

```
width ← n >> 3;
pairwidth ← n << 1;
IF (AddressUnavailable(a) OR ((a^(pairwidth-1)) ≠ 0)) THROW
RADDERR,a;
IF (MMU() AND DataAccessMiss(a)) THROW RTLBMISSE,a;
IF (MMU() AND ReadProhibited(a)) THROW READPROT,a;
low ← MEM[a FOR width];
high ← MEM[a+width FOR width];
result ← low + (high << n);
```

### 8.6.3 Prefetching memory

A function is provided to denote memory prefetch.

Function	Description
PrefetchMemory(address)	Memory prefetch

**Table 63: Support procedure to prefetch memory**

This is used for a software-directed data prefetch from a specified effective address. This is a hint to give advance notice that particular data will be required. It is implementation-specific as to whether a prefetch will be performed.

The statement:

```
result ← PrefetchMemory(a);
```

is equivalent to:

```
IF (NOT AddressUnavailable(address))
  IF (NOT (MMU() AND DataAccessMiss(address)))
    IF (NOT (MMU() AND ReadProhibited(address)))
      PREF(address);
result ← 0;
```

where PREF is a cache operation defined in [Section 8.7: Cache model on page 202](#). This function does not raise exceptions. PrefetchMemory evaluates to zero for syntactic convenience.

### 8.6.4 Writing memory

Procedures are provided to write memory.

Function	Description
WriteMemory <sub>n</sub> (address, value)	Aligned memory write to an n-bit value
WriteMemoryPair <sub>n</sub> (address, value)	Aligned memory write to a pair of n-bit values

**Table 64: Support procedures to write memory**

The WriteMemory<sub>n</sub> procedure takes an integer parameter to indicate the address being accessed, followed by an integer parameter containing the value to be written.



The number of bits being written ( $n$ ) is one of 8, 16 or 32 bits. The written value is interpreted as a bit-field of the required size; all higher bits of the value are discarded. The bytes are written to memory, ordered according to endianness. The statement:

```
WriteMemoryn(a, value);
```

is equivalent to:

```
width ← n >> 3;
IF (AddressUnavailable(a) OR ((a^(width-1)) ≠ 0)) THROW
WADDERR,a;
IF (MMU() AND DataAccessMiss(a)) THROW WTLBMISS,a;
IF (MMU() AND WriteProhibited(a)) THROW WRITEPROT,a;
IF (MMU() AND NOT DirtyBit(a)) THROW FIRSTWRITE,a;
MEM[a FOR width] ← value<0 FOR n>;
```

WriteMemoryPair<sub>n</sub> writes a pair of  $n$ -bit values. The alignment check requires alignment for a  $2n$ -bit access. The access maintains the ordering of the two halves of the pair, with endianness applied separately to each half. The statement:

```
WriteMemoryPairn(a, value);
```

is equivalent to:

```
width ← n >> 3;
pairwidth ← n << 1;
IF (AddressUnavailable(a) OR ((a^(pairwidth-1)) ≠ 0)
) THROW WADDERR,a;
IF (MMU() AND DataAccessMiss(a)) THROW WTLBMISS,a;
IF (MMU() AND WriteProhibited(a)) THROW WRITEPROT,a;
IF (MMU() AND NOT DirtyBit(a)) THROW FIRSTWRITE,a;
MEM[a FOR width] ← value<0 FOR n>;
MEM[a+width FOR width] ← value<n FOR n>;
Sleep operations
```

The SLEEP operation is used to enter sleep mode. The effects of this operation is beyond the scope of the specification language, and it is therefore modelled using

procedure calls. The behavior of these procedure calls is elaborated in the text of the manual.

Procedure	Description
SLEEP()	Procedure to enter sleep mode

**Table 65: Procedures to model sleep operation**

## 8.7 Cache model

Cache operations are used to allocate, prefetch and cohere lines in caches. The effects of these operations are beyond the scope of the specification language, and are therefore modelled using procedure calls. The behavior of these procedure calls is elaborated in the text of the manual.

Procedure	Description
ALLOCO(address)	Procedure to allocate an operand cache block.
OCBI(address)	Procedure to invalidate an operand cache block.
OCBP(address)	Procedure to purge an operand cache block.
OCBWB(address)	Procedure to write-back an operand cache block.
PREF (address)	Procedure to prefetch an operand cache block.

**Table 66: Procedures to model cache operations**

## 8.8 Floating-point model

The floating-point specification is abstracted using functions to hide the low-level details. Additional information is provided in a tabular form to describe special and exceptional cases. [Chapter 6: Floating-point unit on page 145](#) provides a textual description of floating-point operation.

### 8.8.1 Functions to access SR and FPSCR

The floating-point instruction specifications use a function notation to access SR and FPSCR state. The used functions are described in [Table 67](#).

Function	Description
FpuIsDisabled(SR)	True if SR.FD is 1, otherwise false
FpuFlagI(FPSCR)	True if FPSCR.FLAG.I (sticky flag for inexact) is 1, otherwise false
FpuFlagU(FPSCR)	True if FPSCR.FLAG.U (sticky flag for underflow) is 1, otherwise false
FpuFlagO(FPSCR)	True if FPSCR.FLAG.O (sticky flag for overflow) is 1, otherwise false
FpuFlagZ(FPSCR)	True if FPSCR.FLAG.Z (sticky flag for divide by zero) is 1, otherwise false
FpuFlagV(FPSCR)	True if FPSCR.FLAG.V (sticky flag for invalid) is 1, otherwise false
FpuCauseI(FPSCR)	True if FPSCR.CAUSE.I (cause flag for inexact) is 1, otherwise false
FpuCauseU(FPSCR)	True if FPSCR.CAUSE.U (cause flag for underflow) is 1, otherwise false
FpuCauseO(FPSCR)	True if FPSCR.CAUSE.O (cause flag for overflow) is 1, otherwise false
FpuCauseZ(FPSCR)	True if FPSCR.CAUSE.Z (cause flag for divide by zero) is 1, otherwise false
FpuCauseV(FPSCR)	True if FPSCR.CAUSE.V (cause flag for invalid) is 1, otherwise false
FpuCauseE(FPSCR)	True if FPSCR.CAUSE.E (cause flag for FPU error) is 1, otherwise false
FpuEnableI(FPSCR)	True if FPSCR.ENABLE.I (exception enable for inexact) is 1, otherwise false
FpuEnableU(FPSCR)	True if FPSCR.ENABLE.U (exception enable for underflow) is 1, otherwise false
FpuEnableO(FPSCR)	True if FPSCR.ENABLE.O (exception enable for overflow) is 1, otherwise false
FpuEnableZ(FPSCR)	True if FPSCR.ENABLE.Z (exception enable for divide by zero) is 1, otherwise false
FpuEnableV(FPSCR)	True if FPSCR.ENABLE.V (exception enable for invalid) is 1, otherwise false

Table 67: SR and FPSCR access

## 8.8.2 Functions to model floating-point behavior

Functions are used to model almost all of the floating-point behavior. Each function is associated with a list of results and a list of parameters. The functions encapsulate the computation associated with the instruction. This includes handling of input denormalized values, special case detection, exceptional cases and the floating-point arithmetic.

The following tables summarize the functions used by each instruction. The table shows how the parameters are interpreted and how the results are computed. The  $n^{\text{th}}$  parameter is denoted as  $P_n$  and the  $n^{\text{th}}$  result as  $RES_n$ .

The parameters and results of these functions are all modeled as integer values. For floating-point parameters and results, these values are integer bit-patterns representing the IEEE754 formats. Multi-value results are used to return two results: the computed result and a new value for FPSCR. If the new value of FPSCR causes an exception to be raised, then the destination register will not be updated with the computed result.

Instruction	Function	RES0	RES1	P0, P1	P2
FADD.S	FADD_S	Single result of $(P0 +_{\text{IEEE754}} P1)$	New FPSCR	Single	Old FPSCR
FADD.D	FADD_D	Double result of $(P0 +_{\text{IEEE754}} P1)$	New FPSCR	Double	Old FPSCR
FSUB.S	FSUB_S	Single result of $(P0 -_{\text{IEEE754}} P1)$	New FPSCR	Single	Old FPSCR
FSUB.D	FSUB_D	Double result of $(P0 -_{\text{IEEE754}} P1)$	New FPSCR	Double	Old FPSCR
FMUL.S	FMUL_S	Single result of $(P0 \times_{\text{IEEE754}} P1)$	New FPSCR	Single	Old FPSCR
FMUL.D	FMUL_D	Double result of $(P0 \times_{\text{IEEE754}} P1)$	New FPSCR	Double	Old FPSCR
FDIV.S	FDIV_S	Single result of $(P0 /_{\text{IEEE754}} P1)$	New FPSCR	Single	Old FPSCR
FDIV.D	FDIV_D	Double result of $(P0 /_{\text{IEEE754}} P1)$	New FPSCR	Double	Old FPSCR

**Table 68: Floating-point dyadic arithmetic**

Instruction	Function	RES0	RES1	P0	P1
FABS.S	FABS_S	Single result of absolute P0	(not used)	Single	Old FPSCR
FABS.D	FABS_D	Double result of absolute P0	(not used)	Double	Old FPSCR

**Table 69: Floating-point monadic arithmetic**

Instruction	Function	RES0	RES1	P0	P1
FNEG.S	FNEG_S	Single result of negating P0	(not used)	Single	Old FPSCR
FNEG.D	FNEG_D	Double result of negating of P0	(not used)	Double	Old FPSCR
FSQRT.S	FSQRT_S	Single result of $\sqrt{\text{IEEE754}} P0$	New FPSCR	Single	Old FPSCR
FSQRT.D	FSQRT_D	Double result of $\sqrt{\text{IEEE754}} P0$	New FPSCR	Double	Old FPSCR

Table 69: Floating-point monadic arithmetic

Instruction	Function	RES0	RES1	P0, P1	P2
FCMPEQ.S	FCMPEQ_S	Boolean result of $(P0 =_{\text{IEEE754}} P1)$	New FPSCR	Single	Old FPSCR
FCMPEQ.D	FCMPEQ_D	Boolean result of $(P0 =_{\text{IEEE754}} P1)$	New FPSCR	Double	Old FPSCR
FCMPGT.S	FCMPGT_S	Boolean result of $(P0 >_{\text{IEEE754}} P1)$	New FPSCR	Single	Old FPSCR
FCMPGT.D	FCMPGT_D	Boolean result of $(P0 >_{\text{IEEE754}} P1)$	New FPSCR	Double	Old FPSCR

Table 70: Floating-point comparisons

Instruction	Function	RES0	RES1	P0	P1
FCNV.SD	FCNV_SD	P0 is converted to double result	New FPSCR	Single	Old FPSCR
FCNV.DS	FCNV_DS	P0 is converted to single result	New FPSCR	Double	Old FPSCR
FTRC.SL	FTRC_SL	P0 is converted to signed 32-bit integer result	New FPSCR	Single	Old FPSCR
FTRC.DL	FTRC_DL	P0 is converted to signed 32-bit integer result	New FPSCR	Double	Old FPSCR
FLOAT.LS	FLOAT_LS	P0 is converted to single result	New FPSCR	32-bit int	Old FPSCR
FLOAT.LD	FLOAT_LD	P0 is converted to double result	New FPSCR	32-bit int	Old FPSCR

Table 71: Floating-point conversions

Instruction	Function	RES0	RES1	P0, P1, P2	P3
FMAC.S	FMAC_S	Single result of fused $(P0 \times P1) + P2$	New FPSCR	Single	Old FPSCR

Table 72: Floating-point multiply-accumulate

Instruction	Function	RES0	RES1	P0	P1	P2
FIPR.S	FIPR_S	Single result of inner product of P0 with P1	New FPSCR	Array of 4 singles	Array of 4 singles	Old FPSCR
FTRV.S	FTRV_S	Array of 4 single results of matrix transform of P0 with P1	New FPSCR	Array of 16 singles	Array of 4 singles	Old FPSCR

Table 73: Special-purpose floating-point dyadic arithmetic

### 8.8.3 Floating-point special cases and exceptions

A special-case table is provided for each floating-point instruction that is considered an operation and has at least one input that is interpreted as a floating-point value. This table enumerates all different possible combinations of input values and the results returned by the instruction in the absence of an exception being raised.

The entries in the table are IEEE754 floating-point values as described in [Chapter 6: Floating-point unit on page 145](#). Each cell entry in the table describes the result returned for a particular combination of floating-point inputs. If the result is invariant, its value is given in the cell. If the result is variable, the name of the appropriate operation is entered in the cell. If the cell contains 'n/a' then this indicates that an exception is always raised for that combination of inputs and that the implementation does not associate any value with the result.

## 8.9 Abstract sequential model

This section describes the abstract sequential model that is used to specify how instructions are executed on the SH4. It is described in terms of transitions in the explicit architectural state of the device plus some hidden internal state held in PC” and PR” which are used to keep track of delayed state changes.

[Section 8.9.1](#) describes the initial values taken by the internal state.

[Section 8.9.2](#) describes the steps taken to execute each SHcompact instruction in the abstract sequential model. [Section](#) describes the mechanisms used to model delayed branching.

## 8.9.1 Initial conditions

The hidden internal state used to keep track of delayed state changes are automatically set to appropriate initial conditions at the beginning of a sequence of instructions.

The initial state is set as follows:

- PC" is set to PC+2
- PR" is set to the same value as PR

## 8.9.2 Instruction execution loop

The steps associated with executing each instruction are:

- 1 Check for asynchronous events, such as interrupt or reset, and initiate handling if required. Asynchronous events are not accepted between a delayed branch and a delay slot. They are delayed until after the delay slot.
- 2 Check the current program counter (PC) for instruction address exceptions, and initiate handling if required.
- 3 Fetch the instruction bytes from the address in memory, as indicated by the current program counter, 2 bytes need to be fetched for each instruction.
- 4 Calculate the default values of PC' and PR'. PC' is set to the value of PC", PR' is set to the value of PR".
- 5 Calculate the default values of PC" and PR" assuming continued sequential execution without procedure call or mode switch: PC" is PC'+2, while PR" is unchanged.
- 6 Decode and execute the instruction. This includes checks for synchronous events, such as exceptions and panics, and initiation of handling if required. Synchronous events are not accepted between a delayed branch and a delay slot. They are detected either before the delayed branch or after the delay slot.

The execution of an instruction can update the PC and PR state as follows:

- The instruction can change PC' to achieve a branch after this instruction has completed. It must also update PC" to the value of PC'+2 to ensure correct sequential execution after the control flow.
- The instruction can change PR' to load the procedure link register. It must also update PR" to the same value as PR'.

- The instruction can change PC" and PR" to achieve a branch or procedure call after the next instruction has completed.

Any changes made to PC', PR', PC" or PR" over-ride the default values.

- 7 Set the current program counter (PC) to the value of the next program counter (PC') and PR to the value of PR'.

The actions associated with the handling of asynchronous and synchronous events are described in [Chapter 5: Exceptions on page 105](#). The actions required by step 6 depend on the instruction, and are specified by the instruction specification for that instruction. Step 7 specifies the behavior for PC overflow. Non-delayed And Delayed

### 8.9.3 State changes

Non-delayed and delayed state changes are used to model the branch mechanism. These correspond to non-delayed and delayed branches.

In the model, PC and PR are never written directly by an instruction. Instead, an instruction writes to PC' or PR' to cause a non-delayed state change, or to PC" or PR" to cause a delayed state change:

- A non-delayed state change is achieved by updating PC' or PR' to over-ride their default values. After the execution of this instruction, PC' and PR' get copied to PC and PR respectively, and then influence instruction execution. Hence, there is no delay slot before the values of PC' and PR' propagate through to PC and PR.
- A delayed state change is achieved by updating PC" or PR" to over-ride their default values. After the execution of this instruction, PC" and PR" get copied to PC' and PR' respectively. After the execution of the next instruction, PC' and PR' get copied to PC and PR respectively, and then influence instruction execution. Hence, there is a delay slot before the values of PC' and PR" propagate through to PC and PR.

There are potential ambiguities when one instruction makes a delayed state change and the immediately following instruction (which is in a delay slot) makes a non-delayed state change. These are handled as follows:

- The case of a delayed state change to PC immediately followed by a non-delayed state change to PC does not occur. This is because delay slot instructions that write to PC are illegal and cause an ILLSLOT exception.



- The case of a delayed state change to PR immediately followed by a non-delayed state change to PR can occur. The ambiguous cases are when a BSR, BSRF or JSR instruction is followed by an LDS that writes to PR. In this case the PR, observed by the instruction that dynamically follows the LDS instruction, is the value written by LDS not the value written by the sub-routine call. This behavior follows from the model described above.

There are also potential ambiguities when one instruction makes a delayed state change and the immediately following instruction (which is in a delay slot) reads from that state. These are handled as follows:

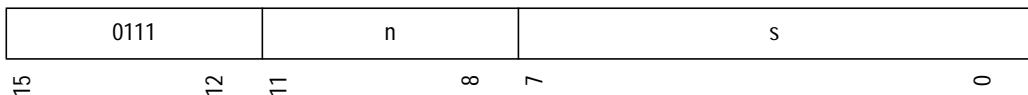
- The case of a delayed state change to PC immediately followed by a read of PC does not occur. This is because delay slot instructions that read from PC are illegal and cause an ILLSLOT exception.
- The case of a delayed state change to PR immediately followed by a read from PR can occur. The ambiguous cases are when a BSR, BSRF or JSR instruction is followed by an STS that reads from PR. In this case the PR, observed by the STS instruction, is the value written by the sub-routine call and not the previous value. This behavior is modeled explicitly in the definition of the STS instruction. It reads the value from PR' (rather than the intuitive read from PR).

## 8.10 Example instructions

### 8.10.1 ADD #imm, Rn

An example specification for this instruction is shown below.

**ADD #imm, Rn**



```

imm ← SignExtend8(s);
op2 ← SignExtend32(Rn);
op2 ← op2 + imm;
Rn ← Register(op2);

```

The top half of this figure shows the assembly syntax and the binary encoding of the instruction. Particular fields within the encoding are identified by single characters. The opcode field, and any extension field, contain the literal encoding values associated with that instruction. Reserved fields must be encoded with the literal value given in the figure. Operand fields contain register designators or immediate constants.

The lower half of this figure specifies the effects of the execution of the instruction on the architectural state of the machine. The specification statements are organized into 3 stages as follows:

- 1 The first two statements read all required source information:

```
imm ← SignExtend8(s);
op2 ← SignExtend32(Rn);
```

The first statement reads the value of *s*, interprets it as a sign-extended 8-bit integer value and assigns this to a temporary integer called 'imm'. The name 'imm' corresponds to the name of the immediate used in the assembly syntax. The second statement reads the value of *R<sub>n</sub>* register, interprets it as a sign-extended 32-bit integer value and assigns this to a temporary integer called *op2*.

- 2 The next statement implements the addition:

```
op2 ← op2 + imm;
```

This statement does not refer to any architectural state. It adds the 2 integers 'imm' and 'op2' together, and assigns the result to a temporary integer called 'op2'. Note that since this is a conventional mathematical addition, the result can contain more significant bits of information than the sources.

- 3 The final statement updates the architectural state:

```
Rn ← Register(op2);
```

The integer 'op2' is converted back to a register bit-field, assigned to the *R<sub>n</sub>* register.

## 8.10.2 FADD FR<sub>m</sub>, FR<sub>n</sub>

An example specification for this instruction is shown below.

### FADD FR<sub>m</sub>, FR<sub>n</sub>

1111	n	m	0000
15	12	11	8
		7	4
			3
			0

Available only when PR=0

```

sr ← ZeroExtend32(SR);
fps ← ZeroExtend32(FPSCR);
op1 ← FloatValue32(FRm);
op2 ← FloatValue32(FRn);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
    THROW FPUDIS;
op2, fps ← FADD_S(op1, op2, fps);
IF (FpuEnableV(fps) AND FpuCauseV(fps))
    THROW FPUExc, fps;
IF (FpuCauseE(fps))
    THROW FPUExc, fps;
IF ((FpuEnableI(fps) OR FpuEnableO(fps)) OR FpuEnableU(fps))
    THROW FPUExc, fps;
FRn ← FloatRegister32(op2);
FPSCR ← ZeroExtend32(fps);

```

The specification statements are organized as follows:

#### 1 Read all required source information:

```

sr ← ZeroExtend32(SR);
fps ← ZeroExtend32(FPSCR);
op1 ← FloatValue32(FRm);
op2 ← FloatValue32(FRn);

```

Execute the instruction:

```

IF (FpuIsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))

```

```

    THROW FPU DIS;
    op2, fps ← FADD_S(op1, op2, fps);
    IF (FpuEnableV(fps) AND FpuCauseV(fps))
    THROW FPU EXC, fps;
    IF (FpuCauseE(fps))
    THROW FPU EXC, fps;
    IF ((FpuEnableI(fps) OR FpuEnableO(fps)) OR FpuEnableU(fps))
    THROW FPU EXC, fps;

```

The behavior of the floating-point single-precision addition is modelled by the FADD\_S procedure. This procedure is given the two source operands and the current value of FPSCR, and calculates the result and the new value of FPSCR. It is responsible for detecting special cases and exceptions, and setting the result and new FPSCR values accordingly.

This instruction contains exception cases. These are detected by IF statements and are raised by THROW statements. When a THROW statement is executed, no further statements from the specification are processed. In exception cases, this specification makes no updates to architectural state. Instead, a handler is launched for the exception as described in [Chapter 5: Exceptions on page 105](#). The THROW statement includes arguments to specify the kind of exception and any necessary parameters of that exception. For an FPU EXC exception, the THROW statement includes an updated value of 'fps' which the exception handler uses to initialize FPSCR during the launch sequence.

2 Update the architectural state:

```

    FRn ← FloatRegister32(op2);
    FPSCR ← ZeroExtend32(fps);

```



# 9

## Instruction descriptions

### 9.1 Alphabetical list of instructions

Instructions are listed in this section in alphabetical order.

# ADD Rm, Rn

## Description

This instruction adds  $R_m$  to  $R_n$  and places the result in  $R_n$ .

## Operation

### ADD Rm, Rn

0011	n	m	1100
15	12	11	8
		7	4
			3
			0

```

op1 ← SignExtend32(Rm);
op2 ← SignExtend32(Rn);
op2 ← op2 + op1;
Rn ← Register(op2);

```

## Note

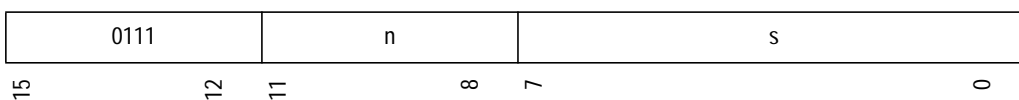
# ADD #imm, Rn

## Description

This instruction adds  $R_n$  to the sign-extended 8-bit immediate  $s$  and places the result in  $R_n$ .

## Operation

### ADD #imm, Rn



```

imm ← SignExtend8(s);
op2 ← SignExtend32(Rn);
op2 ← op2 + imm;
Rn ← Register(op2);

```

## Note

The '#imm' in the assembly syntax represents the immediate  $s$  after sign extension.

# ADDC Rm, Rn

## Description

This instruction adds  $R_m$ ,  $R_n$  and the T-bit. The result of the addition is placed in  $R_n$ , and the carry-out from the addition is placed in the T-bit.

## Operation

### ADDC Rm, Rn

0011	n	m	1110
15	12	11	8
		7	4
		3	0

```

t ← ZeroExtend1(T);
op1 ← ZeroExtend32(SignExtend32(Rm));
op2 ← ZeroExtend32(SignExtend32(Rn));
op2 ← (op2 + op1) + t;
t ← op2 < 32 FOR 1 >;
Rn ← Register(op2);
T ← Bit(t);

```

## Note



# ADDV Rm, Rn

## Description

This instruction adds  $R_m$  to  $R_n$  and places the result in  $R_n$ . The T-bit is set to 1 if the addition result is outside the 32-bit signed range, otherwise the T-bit is set to 0.

## Operation

**ADDV Rm, Rn**

0011	n	m	1111
15	12	11	8
		7	4
		3	0

```

op1 ← SignExtend32(Rm);
op2 ← SignExtend32(Rn);
op2 ← op2 + op1;
t ← INT ((op2 < (- 231)) OR (op2 ≥ 231));
Rn ← Register(op2);
T ← Bit(t);

```

## Note

# AND Rm, Rn

## Description

This instruction performs bitwise AND of  $R_m$  with  $R_n$  and places the result in  $R_n$ .

## Operation

### AND Rm, Rn

0010	n	m	1001
15	12	11	8
		7	4
			3
			0

```

op1 ← ZeroExtend32(Rm);
op2 ← ZeroExtend32(Rn);
op2 ← op2 ∧ op1;
Rn ← Register(op2);

```

## Note

This instruction performs a 32-bit bitwise AND.

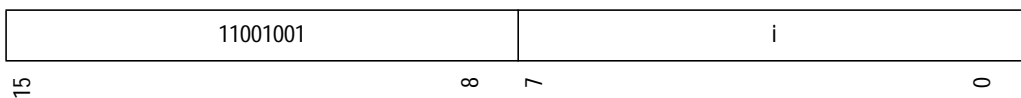
# AND #imm, R0

## Description

This instruction performs bitwise AND of  $R_0$  with the zero-extended 8-bit immediate  $i$  and places the result in  $R_0$ .

## Operation

**AND #imm, R0**



```

r0 ← ZeroExtend32(R0);
imm ← ZeroExtend8(i);
r0 ← r0 ∧ imm;
R0 ← Register(r0);

```

## Note

This instruction performs a 32-bit bitwise AND. The '#imm' in the assembly syntax represents the immediate  $i$  after zero extension.

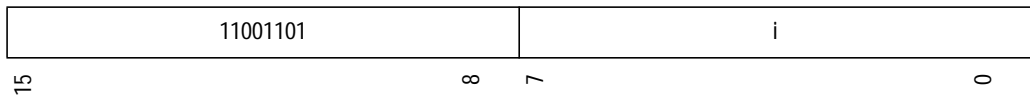
# AND.B #imm, @(R0, GBR)

## Description

This instruction performs a bitwise AND of an immediate constant with 8 bits of data held in memory. The effective address is calculated by adding  $R_0$  and GBR. The 8 bits of data at the effective address are read. A bitwise AND is performed of the read data with the zero-extended 8-bit immediate  $i$ . The result is written back to the 8 bits of data at the same effective address.

## Operation

**AND.B #imm, @(R0, GBR)**



```

r0 ← SignExtend32(R0);
gbr ← SignExtend32(GBR);
imm ← ZeroExtend8(i);
address ← ZeroExtend32(r0 + gbr);
value ← ZeroExtend8(ReadMemory8(address));
value ← value ∧ imm;
WriteMemory8(address, value);

```

## Exceptions

WADDERR, WTLBMISS, READPROT, WRITEPROT, FIRSTWRITE

## Note

Zero-extension is performed on the effective address computation allowing wrap around to occur.

The '#imm' in the assembly syntax represents the immediate  $i$  after zero extension.

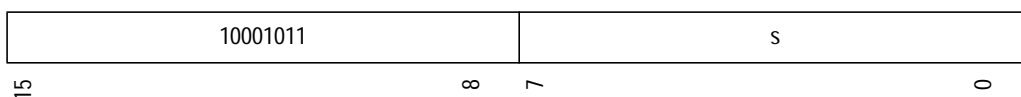
# BF label

## Description

This instruction is a conditional branch. The 8-bit displacement  $s$  is sign-extended, doubled and added to  $PC+4$  to form the target address. If the T-bit is 1, the branch is not taken. If the T-bit is 0, the target address is copied to the PC.

## Operation

### BF label



```

t ← ZeroExtend1(T);
pc ← SignExtend32(PC);
newpc ← SignExtend32(PC');
delayedpc ← SignExtend32(PC'');
label ← SignExtend8(s) << 1;
IF (IsDelaySlot())
    THROW ILLSLOT;
IF (t = 0)
{
    temp ← ZeroExtend32(pc + 4 + label);
    newpc ← temp;
    delayedpc ← temp + 2;
}
PC' ← Register(newpc);
PC'' ← Register(delayedpc);

```

## Exceptions

ILLSLOT

## Note

The effective address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.

This is not a delayed branch instruction. An ILLSLOT exception is raised if this instruction is executed in a delay slot.

The 'label' in the assembly syntax represents the immediate *s* after sign extension and scaling.

If the branch target address is invalid then the IADDERR trap is not delivered until after the branch instruction completes its execution and the PC has advanced to the target address, that is the exception is associated with the target instruction not the branch.

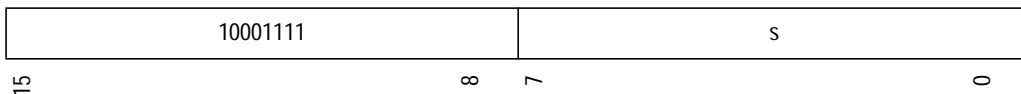
# BF/S label

## Description

This instruction is a delayed conditional branch. The 8-bit displacement  $s$  is sign-extended, doubled and added to  $PC+4$  to form the target address. If the T-bit is 1, the branch is not taken. If the T-bit is 0, the delay slot is executed and then the target address is copied to the PC.

## Operation

### BF/S label



```

t ← ZeroExtend1(T);
pc ← SignExtend32(PC);
delayedpc ← SignExtend32(PC");
label ← SignExtend8(s) << 1;
IF (IsDelaySlot())
    THROW ILLSLOT;
IF (t = 0)
{
    temp ← ZeroExtend32(pc + 4 + label);
    delayedpc ← temp;
}
PC" ← Register(delayedpc);

```

## Exceptions

ILLSLOT

### Note

The effective address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.

The delay slot is executed before branching. An ILLSLOT exception is raised if this instruction is executed in a delay slot.

The 'label' in the assembly syntax represents the immediate s after sign extension and scaling.

If the branch target address is invalid then IADDERR trap is not delivered until after the instruction in the delay slot has executed and the PC has advanced to the target address, that is the exception is associated with the target instruction not the branch.



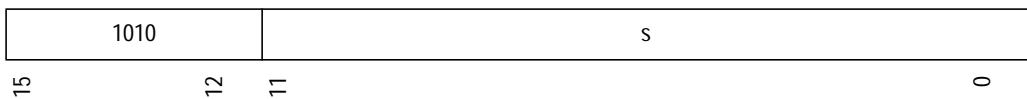
# BRA label

## Description

This instruction is a delayed unconditional branch. The 12-bit displacement  $s$  is sign-extended, doubled and added to PC+4 to form the target address. The delay slot is executed and then the target address is copied to the PC.

## Operation

### BRA label



```

pc ← SignExtend32(PC);
label ← SignExtend12(s) << 1;
IF (IsDelaySlot())
    THROW ILLSLOT;
temp ← ZeroExtend32(pc + 4 + label);
delayedpc ← temp;
PC" ← Register(delayedpc);

```

## Exceptions

### ILLSLOT

### Note

The effective address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.

The delay slot is executed before branching. An ILLSLOT exception is raised if this instruction is executed in a delay slot.

The 'label' in the assembly syntax represents the immediate  $s$  after sign extension and scaling.

If the branch target address is invalid then IADDERR trap is not delivered until after the instruction in the delay slot has executed and the PC has advanced to the target address, that is the exception is associated with the target instruction not the branch.

# BRAF Rn

## Description

This instruction is a delayed unconditional branch. The target address is calculated by adding  $R_n$  to  $PC+4$ . If the least significant bit of the target address is set, an IADDERR exception is raised, otherwise, the delay slot is executed.

## Operation

### BRAF Rn

0000	n	00100011
15	12	0

```

pc ← SignExtend32(PC);
op1 ← SignExtend32(Rn);
IF (IsDelaySlot())
    THROW ILLSLOT;
target ← ZeroExtend32(pc + 4 + op1);
delayedpc ← target ∧ (~ 0x1);
PC" ← Register(delayedpc);

```

## Exceptions

### ILLSLOT

### Note

The effective address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.

The delay slot is executed before branching occurs. An ILLSLOT exception is raised if this instruction is executed in a delay slot.

If the branch target address is invalid then IADDERR trap is not delivered until after the instruction in the delay slot has executed and the PC has advanced to the target address, that is the exception is associated with the target instruction not the branch.

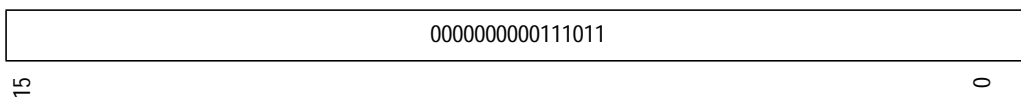
# BRK

## Description

The BRK instruction causes a pre-execution BREAK exception. This exception is generated even BRK is executed in a delay slot. BRK is typically reserved for use by the debugger.

## Operation

### BRK



THROW BREAK;
--------------

## Exceptions

BREAK

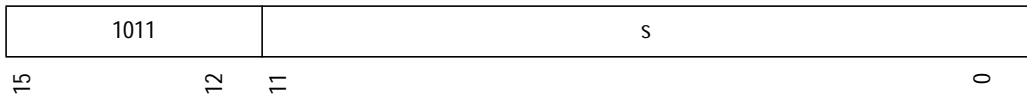
# BSR label

## Description

This instruction is a delayed unconditional branch used for branching to a subroutine. The 12-bit displacement  $s$  is sign-extended, doubled and added to  $PC+4$  to form the target address. The delay slot is executed and then the target address is copied to the PC. The address of the instruction immediately following the delay slot is copied to PR to indicate the return address.

## Operation

### BSR label



```

pc ← SignExtend32(PC);
label ← SignExtend12(s) << 1;
IF (IsDelaySlot())
    THROW ILLSLOT;
delayedpr ← pc + 4;
temp ← ZeroExtend32(pc + 4 + label);
delayedpc ← temp;
PR" ← Register(delayedpr);
PC" ← Register(delayedpc);

```

## Exceptions

ILLSLOT

## Note

The effective address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.

The delay slot is executed before branching. An ILLSLOT exception is raised if this instruction is executed in a delay slot. The 'label' in the assembly syntax represents the immediate  $s$  after sign extension and scaling.

If the branch target address is invalid then IADDERR trap is not delivered until after the instruction in the delay slot has executed and the PC has advanced to the target address, that is the exception is associated with the target instruction not the branch.

# BSRF R<sub>n</sub>

## Description

This instruction is a delayed unconditional branch used for branching to a far subroutine. The target address is calculated by adding R<sub>n</sub> to PC+4. If the least significant bit of the target address is set, an IADDERR exception is raised, otherwise, the delay slot is executed. The address of the instruction immediately following the delay slot is copied to PR to indicate the return address.

## Operation

### BSRF R<sub>n</sub>

0000				n				00000011						
15		12		11		8		7						0

```

pc ← SignExtend32(PC);
op1 ← SignExtend32(Rn);
IF (IsDelaySlot())
    THROW ILLSLOT;
delayedpr ← pc + 4;
target ← ZeroExtend32(pc + 4 + op1);
delayedpc ← target ∧ (~ 0x1);
PR" ← Register(delayedpr);
PC" ← Register(delayedpc);

```

## Exceptions

### ILLSLOT

### Note

The effective address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.

The delay slot is executed before branching and before PR is updated. An ILLSLOT exception is raised if this instruction is executed in a delay slot.

If the branch target address is invalid then IADDERR trap is not delivered until after the instruction in the delay slot has executed and the PC has advanced to the target address, that is the exception is associated with the target instruction not the branch.

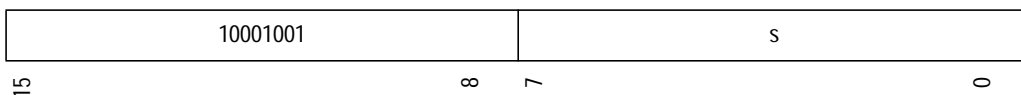
# BT label

## Description

This instruction is a conditional branch. The 8-bit displacement  $s$  is sign-extended, doubled and added to  $PC+4$  to form the target address. If the T-bit is 0, the branch is not taken. If the T-bit is 1, the target address is copied to the PC.

## Operation

### BT label



```

t ← ZeroExtend1(T);
pc ← SignExtend32(PC);
newpc ← SignExtend32(PC');
delayedpc ← SignExtend32(PC'');
label ← SignExtend8(s) << 1;
IF (IsDelaySlot())
    THROW ILLSLOT;
IF (t = 1)
{
    temp ← ZeroExtend32(pc + 4 + label);
    newpc ← temp;
    delayedpc ← temp + 2;
}
PC' ← Register(newpc);
PC'' ← Register(delayedpc);

```

## Exceptions

ILLSLOT

## Note

The effective address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.

This is not a delayed branch instruction. An ILLSLOT exception is raised if this instruction is executed in a delay slot.

The 'label' in the assembly syntax represents the immediate *s* after sign extension and scaling.

If the branch target address is invalid then the IADDERR trap is not delivered until after the branch instruction completes its execution and the PC has advanced to the target address, that is the exception is associated with the target instruction not the branch.



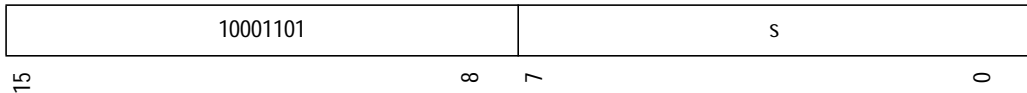
# BT/S label

## Description

This instruction is a delayed conditional branch. The 8-bit displacement  $s$  is sign-extended, doubled and added to  $PC+4$  to form the target address. If the T-bit is 0, the branch is not taken. If the T-bit is 1, the delay slot is executed and then the target address is copied to the PC.

## Operation

### BT/S label



```

t ← ZeroExtend1(T);
pc ← SignExtend32(PC);
delayedpc ← SignExtend32(PC");
label ← SignExtend8(s) << 1;
IF (IsDelaySlot())
    THROW ILLSLOT;
IF (t = 1)
{
    temp ← ZeroExtend32(pc + 4 + label);
    delayedpc ← temp;
}
PC" ← Register(delayedpc);

```

## Exceptions

ILLSLOT

## Note

The effective address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.

The delay slot is executed before branching. An ILLSLOT exception is raised if this instruction is executed in a delay slot.

The 'label' in the assembly syntax represents the immediate s after sign extension and scaling.

If the branch target address is invalid then IADDERR trap is not delivered until after the instruction in the delay slot has executed and the PC has advanced to the target address, that is the exception is associated with the target instruction not the branch.

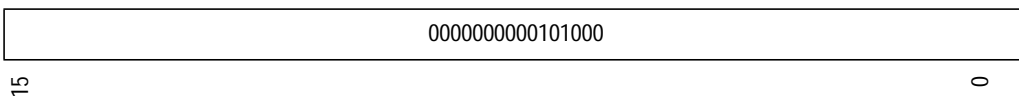
# CLRMAC

## Description

This instruction clears MACL and MACH.

## Operation

### CLRMAC



```

mac1 ← 0;
mach ← 0;
MACL ← ZeroExtend32(mac1);
MACH ← ZeroExtend32(mach);

```

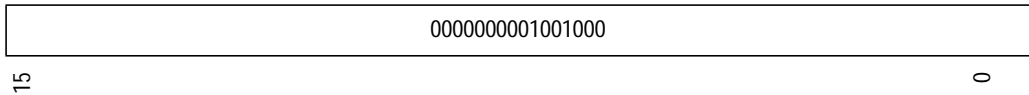
# CLRS

## Description

This instruction clears the S-bit.

## Operation

### CLRS



$s \leftarrow 0;$   
 $S \leftarrow \text{Bit}(s);$

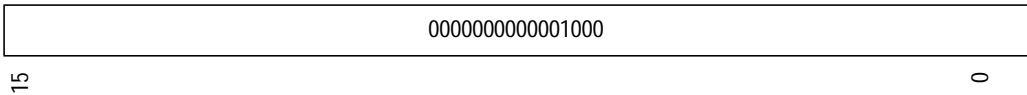
# CLRT

## Description

This instruction clears the T-bit.

## Operation

**CLRT**



$t \leftarrow 0;$   
 $T \leftarrow \text{Bit}(t);$

# CMP/EQ Rm, Rn

## Description

This instruction sets the T-bit if the value of  $R_n$  is equal to the value of  $R_m$ , otherwise it clears the T-bit.

## Operation

### CMP/EQ Rm, Rn

0011	n	m	0000
15	12	11	8
		7	4
		3	0

```

op1 ← SignExtend32(Rm);
op2 ← SignExtend32(Rn);
t ← INT (op2 = op1);
T ← Bit(t);

```

## Note

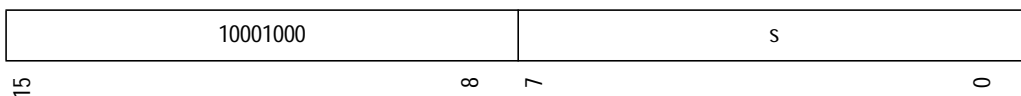
# CMP/EQ #imm, R0

## Description

This instruction sets the T-bit if the value of  $R_0$  is equal to the sign-extended 8-bit immediate  $s$ , otherwise it clears the T-bit.

## Operation

**CMP/EQ #imm, R0**



```

r0 ← SignExtend32(R0);
imm ← SignExtend8(s);
t ← INT (r0 = imm);
T ← Bit(t);

```

## Note

The '#imm' in the assembly syntax represents the immediate  $s$  after sign extension.

# CMP/GE Rm, Rn

## Description

This instruction sets the T-bit if the signed value of  $R_n$  is greater than or equal to the signed value of  $R_m$ , otherwise it clears the T-bit.

## Operation

### CMP/GE Rm, Rn

0011	n	m	0011
15	12	11	8
		7	4
			3
			0

```

op1 ← SignExtend32(Rm);
op2 ← SignExtend32(Rn);
t ← INT (op2 ≥ op1);
T ← Bit(t);

```

## Note



# CMP/GT Rm, Rn

## Description

This instruction sets the T-bit if the signed value of  $R_n$  is greater than the signed value of  $R_m$ , otherwise it clears the T-bit.

## Operation

### CMP/GT Rm, Rn

0011	n	m	0111
15	12	11	8
		7	4
			3
			0

```

op1 ← SignExtend32(Rm);
op2 ← SignExtend32(Rn);
t ← INT (op2 > op1);
T ← Bit(t);

```

## Note

# CMP/HI Rm, Rn

## Description

This instruction sets the T-bit if the unsigned value of  $R_n$  is greater than the unsigned value of  $R_m$ , otherwise it clears the T-bit.

## Operation

### CMP/HI Rm, Rn

0011	n	m	0110
15	12	11	8
		7	4
			3
			0

```

op1 ← ZeroExtend32(SignExtend32(Rm));
op2 ← ZeroExtend32(SignExtend32(Rn));
t ← INT (op2 > op1);
T ← Bit(t);

```

## Note

# CMP/HS Rm, Rn

## Description

This instruction sets the T-bit if the unsigned value of  $R_n$  is greater than or equal to the unsigned value of  $R_m$ , otherwise it clears the T-bit.

## Operation

### CMP/HS Rm, Rn

0011	n	m	0010
15	12	11	8
		7	4
			3
			0

```

op1 ← ZeroExtend32(SignExtend32(Rm));
op2 ← ZeroExtend32(SignExtend32(Rn));
t ← INT (op2 ≥ op1);
T ← Bit(t);

```

## Note

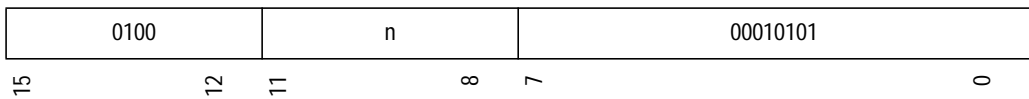
# CMP/PL R<sub>n</sub>

## Description

This instruction sets the T-bit if the signed value of R<sub>n</sub> is greater than 0, otherwise it clears the T-bit.

## Operation

**CMP/PL R<sub>n</sub>**



```

op1 ← SignExtend32(Rn);
t ← INT (op1 > 0);
T ← Bit(t);

```

## Note

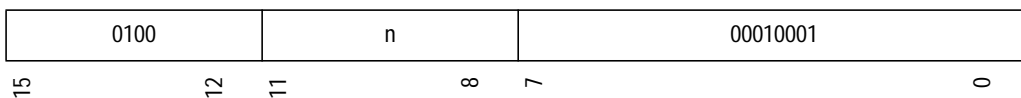
# CMP/PZ R<sub>n</sub>

## Description

This instruction sets the T-bit if the signed value of R<sub>n</sub> is greater than or equal to 0, otherwise it clears the T-bit.

## Operation

### CMP/PZ R<sub>n</sub>



```

op1 ← SignExtend32(Rn);
t ← INT (op1 ≥ 0);
T ← Bit(t);

```

## Note

# CMP/STR Rm, Rn

## Description

This instruction sets the T-bit if any byte in  $R_n$  has the same value as the corresponding byte in  $R_m$ , otherwise it clears the T-bit.

## Operation

### CMP/STR Rm, Rn

0010				n				m				1100																			
15				12				11				8				7				4				3				0			

```

op1 ← SignExtend32(Rm);
op2 ← SignExtend32(Rn);
temp ← op1 ⊕ op2;
t ← INT (temp< 0 FOR 8 > = 0);
t ← (INT (temp< 8 FOR 8 > = 0)) ∨ t;
t ← (INT (temp< 16 FOR 8 > = 0)) ∨ t;
t ← (INT (temp< 24 FOR 8 > = 0)) ∨ t;
T ← Bit(t);

```

## Note

# DIV0S Rm, Rn

## Description

This instruction initializes the divide-step state for a signed division. The Q-bit is initialized with the sign-bit of the dividend, and the M-bit with the sign-bit of the divisor. The T-bit is initialized to 0 if the Q-bit and the M-bit are the same, otherwise it is initialized to 1.

## Operation

### DIV0S Rm, Rn

0010	n	m	0111
15	12	11	8
		7	4
			3
			0

```

op1 ← SignExtend32(Rm);
op2 ← SignExtend32(Rn);
q ← op2< 31 FOR 1 >;
m ← op1< 31 FOR 1 >;
t ← m ⊕ q;
Q ← Bit(q);
M ← Bit(m);
T ← Bit(t);

```

## Note

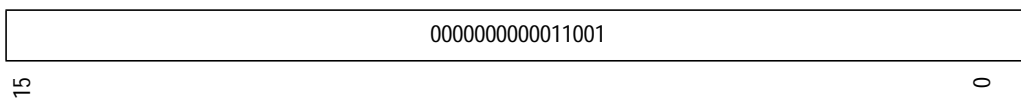
# DIV0U

## Description

This instruction initializes the divide-step state for an unsigned division. The Q-bit, M-bit and T-bit are all set to 0.

## Operation

### DIV0U



```

q ← 0;
m ← 0;
t ← 0;
Q ← Bit(q);
M ← Bit(m);
T ← Bit(t);

```



# DIV1 Rm, Rn

## Description

This instruction is used to perform a single-bit divide-step for the division of a dividend held in  $R_n$  by a divisor held in  $R_m$ . The Q-bit, M-bit and T-bit are used to hold additional state through a divide-step sequence. Each DIV1 consumes 1 bit of the dividend from  $R_n$ , and produces 1 bit of result. The divide initialization and step instructions do not detect divide-by-zero nor overflow. If required, these cases should be checked using additional instructions.

## Operation

### DIV1 Rm, Rn

0011	n	m	0100
15	12	11	8
		7	4
		3	0

```

q ← ZeroExtend1(Q);
m ← ZeroExtend1(M);
t ← ZeroExtend1(T);
op1 ← ZeroExtend32(SignExtend32(Rm));
op2 ← ZeroExtend32(SignExtend32(Rn));
oldq ← q;
q ← op2< 31 FOR 1 >;
op2 ← ZeroExtend32(op2 << 1) ∨ t;
IF (oldq = m)
    op2 ← op2 - op1;
ELSE
    op2 ← op2 + op1;
q ← (q ⊕ m) ⊕ op2< 32 FOR 1 >;
t ← 1 - (q ⊕ m);
Rn ← Register(op2);
Q ← Bit(q);
T ← Bit(t);

```

## Note

# DMULS.L Rm, Rn

## Description

This instruction multiplies the signed 32-bit value held in  $R_m$  with the signed 32-bit value held in  $R_n$  to give a full 64-bit result. The lower half of the result is placed in MACL and the upper half in MACH.

## Operation

### DMULS.L Rm, Rn

0011	n	m	1101
15	12	11	8
		7	4
		3	0

```

op1 ← SignExtend32(Rm);
op2 ← SignExtend32(Rn);
mac ← op2 × op1;
macl ← mac;
mach ← mac >> 32;
MACL ← ZeroExtend32(macl);
MACH ← ZeroExtend32(mach);

```

## Note

# DMULU.L Rm, Rn

## Description

This instruction multiplies the unsigned 32-bit value held in  $R_m$  with the unsigned 32-bit value held in  $R_n$  to give a full 64-bit result. The lower half of the result is placed in MACL and the upper half in MACH.

## Operation

### DMULU.L Rm, Rn

0011	n	m	0101
15	12	11	8
		7	4
		3	0

```

op1 ← ZeroExtend32(SignExtend32(Rm));
op2 ← ZeroExtend32(SignExtend32(Rn));
mac ← op2 × op1;
macl ← mac;
mach ← mac >> 32;
MACL ← ZeroExtend32(macl);
MACH ← ZeroExtend32(mach);

```

## Note

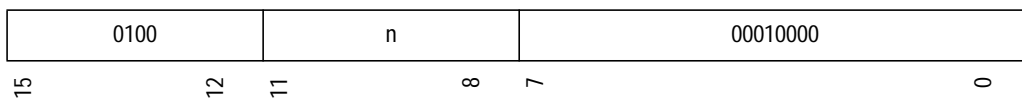
# DT Rn

## Description

This instruction subtracts 1 from  $R_n$  and placed the result in  $R_n$ . The T-bit is set if the result is zero, otherwise the T-bit is cleared.

## Operation

### DT Rn



```

op1 ← SignExtend32(Rn);
op1 ← op1 - 1;
t ← INT (op1 = 0);
Rn ← Register(op1);
T ← Bit(t);

```

## Note

# EXTS.B Rm, Rn

## Description

This instruction reads the 8 least significant bits of  $R_m$ , sign-extends, and places the result in  $R_n$ .

## Operation

### EXTS.B Rm, Rn

0110	n	m	1110
15	12	11	8
		7	4
		3	0

```

op1 ← SignExtend8(Rm);
op2 ← op1;
Rn ← Register(op2);

```

## Note

# EXTS.W Rm, Rn

## Description

This instruction reads the 16 least significant bits of  $R_m$ , sign-extends, and places the result in  $R_n$ .

## Operation

**EXTS.W Rm, Rn**

0110				n				m				1111			
15				12				8				0			
11				7				4				3			

```

op1 ← SignExtend16(Rm);
op2 ← op1;
Rn ← Register(op2);

```

## Note

# EXTU.B Rm, Rn

## Description

This instruction reads the 8 least significant bits of  $R_m$ , zero-extends, and places the result in  $R_n$ .

## Operation

**EXTU.B Rm, Rn**

0110				n				m				1100			
15				12				8				0			
11				7				4				3			

```

op1 ← ZeroExtend8(Rm);
op2 ← op1;
Rn ← Register(op2);

```

## Note

# EXTU.W Rm, Rn

## Description

This instruction reads the 16 least significant bits of  $R_m$ , zero-extends, and places the result in  $R_n$ .

## Operation

EXTU.W Rm, Rn

0110				n				m				1101			
15				12				8				0			
11				7				4				3			

```

op1 ← ZeroExtend16(Rm);
op2 ← op1;
Rn ← Register(op2);

```

## Note



# FABS DR<sub>n</sub>

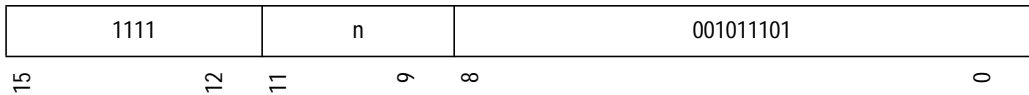
## Description

This floating-point instruction computes the absolute value of a double-precision floating-point number. It reads DR<sub>n</sub>, clears the sign bit and places the result in DR<sub>n</sub>.

This instruction is not considered an arithmetic operation, and it does not signal invalid operations. There are no special floating-point cases for this instruction.

## Operation

### FABS DR<sub>n</sub>



Available only when PR=1 and SZ=0

```

sr ← ZeroExtend32(SR);
op1 ← FloatValue64(DR2n);
IF (FpulsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpulsDisabled(sr))
    THROW FPUDIS;
op1 ← FABS_D(op1);
DR2n ← FloatRegister64(op1);

```

## Exceptions

SLOTFPUDIS, FPUDIS

# FABS FR<sub>n</sub>

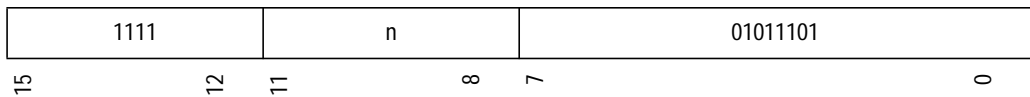
## Description

This floating-point instruction computes the absolute value of a single-precision floating-point number. It reads FR<sub>n</sub>, clears the sign bit and places the result in FR<sub>n</sub>.

This instruction is not considered an arithmetic operation, and it does not signal invalid operations. There are no special floating-point cases for this instruction.

## Operation

### FABS FR<sub>n</sub>



Available only when PR=0

```

sr ← ZeroExtend32(SR);
op1 ← FloatValue32(FRn);
IF (FpulsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpulsDisabled(sr))
    THROW FPUDIS;
op1 ← FABS_S(op1);
FRn ← FloatRegister32(op1);

```

## Exceptions

SLOTFPUDIS, FPUDIS

# FADD DRm, DRn

## Description

This floating-point instruction performs a double-precision floating-point addition. It adds  $DR_m$  to  $DR_n$  and places the result in  $DR_n$ . The rounding mode is determined by FPSCR.RM.

## Operation

### FADD DRm, DRn

1111				n		0	m		00000							
15				12		11	9		8	7		5		4	0	

Available only when PR=1 and SZ=0

```

sr ← ZeroExtend32(SR);
fps ← ZeroExtend32(FPSCR);
op1 ← FloatValue64(DR2m);
op2 ← FloatValue64(DR2n);
IF (FpulsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpulsDisabled(sr))
    THROW FPUDIS;
op2, fps ← FADD_D(op1, op2, fps);
IF (FpuEnableV(fps) AND FpuCauseV(fps))
    THROW FPUEXC, fps;
IF (FpuCauseE(fps))
    THROW FPUEXC, fps;
IF ((FpuEnableI(fps) OR FpuEnableO(fps)) OR FpuEnableU(fps))
    THROW FPUEXC, fps;
DR2n ← FloatRegister64(op2);
FPSCR ← ZeroExtend32(fps);

```

## Exceptions

SLOTFPUDIS, FPUDIS, FPUEXC

# FADD FR<sub>m</sub>, FR<sub>n</sub>

## Description

This floating-point instruction performs a single-precision floating-point addition. It adds FR<sub>m</sub> to FR<sub>n</sub> and places the result in FR<sub>n</sub>. The rounding mode is determined by FPSCR.RM.

## Operation

### FADD FR<sub>m</sub>, FR<sub>n</sub>

1111				n				m				0000																			
15				12				11				8				7				4				3				0			

Available only when PR=0

```

sr ← ZeroExtend32(SR);
fps ← ZeroExtend32(FPSCR);
op1 ← FloatValue32(FRm);
op2 ← FloatValue32(FRn);
IF (FpulsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpulsDisabled(sr))
    THROW FPUDIS;
op2, fps ← FADD_S(op1, op2, fps);
IF (FpuEnableV(fps) AND FpuCauseV(fps))
    THROW FPUExc, fps;
IF (FpuCauseE(fps))
    THROW FPUExc, fps;
IF ((FpuEnableI(fps) OR FpuEnableO(fps)) OR FpuEnableU(fps))
    THROW FPUExc, fps;
FRn ← FloatRegister32(op2);
FPSCR ← ZeroExtend32(fps);

```

## Exceptions

SLOTFPUDIS, FPUDIS, FPUExc

**FADD Special Cases:**

When FPSCR.DN is 1, a positive denormalized number is treated as +0 and a negative denormalized number as -0. This flush-to-zero treatment is applied before exception detection and special case handling.

Exceptional conditions are checked in the order given below. Execution of the instruction is terminated once any check detects an exceptional condition.

- 1 Disabled: an exception is raised if the FPU is disabled.
- 2 Invalid: an invalid operation is signaled if either input is a signaling NaN, or if the inputs are differently signed infinities.
- 3 Error: an FPU error is signaled if FPSCR.DN is zero, neither input is a NaN and either input is a denormalized number.
- 4 Inexact, underflow and overflow: these are checked together and can be signaled in combination. When inexact, underflow or overflow exceptions are requested by the user, an exception is always raised regardless of whether that condition arose.

If the instruction does not raise an exception, a result is generated according to the following table.

op1 → ↓ op2	+NORM, -NORM	+0	-0	+INF	-INF	+DNORM, -DNORM	qNaN	sNaN
+, -NORM	ADD	op2		+INF	-INF	n/a	qNaN	qNaN
+0	op1	+0	+0	+INF	-INF	n/a	qNaN	qNaN
-0	op1	+0	-0	+INF	-INF	n/a	qNaN	qNaN
+INF	+INF	+INF	+INF	+INF	qNaN	n/a	qNaN	qNaN
-INF	-INF	-INF	-INF	qNaN	-INF	n/a	qNaN	qNaN
+, -DNORM	n/a	n/a	n/a	n/a	n/a	n/a	qNaN	qNaN
qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN
sNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN

FPU error is indicated by heavy shading and always raises an exception. Invalid operations are indicated by light shading and raise an exception if enabled. FPU disabled, inexact, underflow and overflow cases are not shown.

The behavior of the normal 'ADD' case is described by the IEEE754 specification.

# FCMP/EQ DR<sub>m</sub>, DR<sub>n</sub>

## Description

This floating-point instruction performs a double-precision floating-point equality comparison. It sets the T-bit to 1 if DR<sub>m</sub> is equal to DR<sub>n</sub>, and otherwise sets the T-bit to 0.

## Operation

### FCMP/EQ DR<sub>m</sub>, DR<sub>n</sub>

1111				n		0	m		00100			
15	12			11	9	8	7	5	4	0		

Available only when PR=1 and SZ=0

```

sr ← ZeroExtend32(SR);
fps ← ZeroExtend32(FPSCR);
op1 ← FloatValue64(DR2m);
op2 ← FloatValue64(DR2n);
IF (FpulsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpulsDisabled(sr))
    THROW FPUDIS;
t, fps ← FCMPEQ_D(op1, op2, fps);
IF (FpuEnableV(fps) AND FpuCauseV(fps))
    THROW FPUEXC, fps;
FPSCR ← ZeroExtend32(fps);
T ← Bit(t);

```

## Exceptions

SLOTFPUDIS, FPUDIS, FPUEXC

# FCMP/EQ FR<sub>m</sub>, FR<sub>n</sub>

## Description

This floating-point instruction performs a single-precision floating-point equality comparison. It sets the T-bit to 1 if FR<sub>m</sub> is equal to FR<sub>n</sub>, and otherwise sets the T-bit to 0.

## Operation

### FCMP/EQ FR<sub>m</sub>, FR<sub>n</sub>

1111	n	m	0100
15	12	11	8
		7	4
			3
			0

Available only when PR=0

```

sr ← ZeroExtend32(SR);
fps ← ZeroExtend32(FPSCR);
op1 ← FloatValue32(FRm);
op2 ← FloatValue32(FRn);
IF (FpulsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpulsDisabled(sr))
    THROW FPUDIS;
t, fps ← FCMPEQ_S(op1, op2, fps);
IF (FpuEnableV(fps) AND FpuCauseV(fps))
    THROW FPUExc, fps;
FPSCR ← ZeroExtend32(fps);
T ← Bit(t);

```

## Exceptions

SLOTFPUDIS, FPUDIS, FPUExc

### FCMP/EQ Special Cases:

When FPSCR.DN is 1, a positive denormalized number is treated as +0 and a negative denormalized number as -0. This flush-to-zero treatment is applied before exception detection and special case handling.

Exceptional conditions are checked in the order given below. Execution of the instruction is terminated once any check detects an exceptional condition.

- 1 Disabled: an exception is raised if the FPU is disabled.
- 2 Invalid: an invalid operation is signaled if either input is a signaling NaN.

If the instruction does not raise an exception, a result is generated according to the following table.

op1 → ↓ op2	+NORM, -NORM	+0	-0	+INF	-INF	+DNORM, -DNORM	qNaN	sNaN
+, -NORM	CMPEQ	false	false	false	false	false	false	false
+0	false	true	true	false	false	false	false	false
-0	false	true	true	false	false	false	false	false
+INF	false	false	false	true	false	false	false	false
-INF	false	false	false	false	true	false	false	false
+, -DNORM	false	false	false	false	false	CMPEQ	false	false
qNaN	false	false	false	false	false	false	false	false
sNaN	false	false	false	false	false	false	false	false

Invalid operations are indicated by light shading and raise an exception if enabled. FPU disabled cases are not shown.

The behavior of the normal 'CMPEQ' case is described by the IEEE754 specification.



# FCMP/GT DR<sub>m</sub>, DR<sub>n</sub>

## Description

This floating-point instruction performs a double-precision floating-point greater-than comparison. It sets the T-bit to 1 if DR<sub>n</sub> is greater than DR<sub>m</sub>, and otherwise sets the T-bit to 0.

## Operation

### FCMP/GT DR<sub>m</sub>, DR<sub>n</sub>

1111				n		0	m		00101			
15		12		11	9	8	7	5		4	0	

Available only when PR=1 and SZ=0

```

sr ← ZeroExtend32(SR);
fps ← ZeroExtend32(FPSCR);
op1 ← FloatValue64(DR2m);
op2 ← FloatValue64(DR2n);
IF (FpulsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpulsDisabled(sr))
    THROW FPUDIS;
t, fps ← FCMPGT_D(op2, op1, fps);
IF (FpuEnableV(fps) AND FpuCauseV(fps))
    THROW FPUEXC, fps;
FPSCR ← ZeroExtend32(fps);
T ← Bit(t);

```

## Exceptions

SLOTFPUDIS, FPUDIS, FPUEXC

# FCMP/GT FR<sub>m</sub>, FR<sub>n</sub>

## Description

This floating-point instruction performs a single-precision floating-point greater-than comparison. It sets the T-bit to 1 if FR<sub>n</sub> is greater than FR<sub>m</sub>, and otherwise sets the T-bit to 0.

## Operation

### FCMP/GT FR<sub>m</sub>, FR<sub>n</sub>

1111	n	m	0101
15	12	11	8
		7	4
			3
			0

Available only when PR=0

```

sr ← ZeroExtend32(SR);
fps ← ZeroExtend32(FPSCR);
op1 ← FloatValue32(FRm);
op2 ← FloatValue32(FRn);
IF (FpulsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpulsDisabled(sr))
    THROW FPUDIS;
t, fps ← FCMPGT_S(op2, op1, fps);
IF (FpuEnableV(fps) AND FpuCauseV(fps))
    THROW FPUExc, fps;
FPSCR ← ZeroExtend32(fps);
T ← Bit(t);

```

## Exceptions

SLOTFPUDIS, FPUDIS, FPUExc

### FCMP/GT Special Cases:

When FPSCR.DN is 1, a positive denormalized number is treated as +0 and a negative denormalized number as -0. This flush-to-zero treatment is applied before exception detection and special case handling.

Exceptional conditions are checked in the order given below. Execution of the instruction is terminated once any check detects an exceptional condition.

- 1 Disabled: an exception is raised if the FPU is disabled.
- 2 Invalid: an invalid operation is signaled if either input is a NaN.

If the instruction does not raise an exception, a result is generated according to the following table.

op2 → ↓ op1	+NORM, -NORM	+0	-0	+INF	-INF	+DNORM, -DNORM	qNaN	sNaN
+, -NORM	CMPGT	CMPGT	CMPGT	true	false	CMPGT	false	false
+0	CMPGT	false	false	true	false	CMPGT	false	false
-0	CMPGT	true	false	true	false	CMPGT	false	false
+INF	false	false	false	false	false	false	false	false
-INF	true	true	true	true	false	true	false	false
+, -DNORM	CMPGT	CMPGT	CMPGT	true	false	CMPGT	false	false
qNaN	false	false	false	false	false	false	false	false
sNaN	false	false	false	false	false	false	false	false

Invalid operations are indicated by light shading and raise an exception if enabled. FPU disabled cases are not shown.

The behavior of the normal 'CMPGT' case is described by the IEEE754 specification.

# FCNVDS DR<sub>m</sub>, FPUL

## Description

This floating-point instruction performs a double-precision to single-precision floating-point conversion. It reads a double-precision value from DR<sub>m</sub>, converts it to single-precision and places the result in FPUL. The rounding mode is determined by FPSCR.RM.

## Operation

### FCNVDS DR<sub>m</sub>, FPUL

1111	m	010111101
15	12	8

Available only when PR=1 and SZ=0

```

sr ← ZeroExtend32(SR);
fps ← ZeroExtend32(FPSCR);
op1 ← FloatValue64(DR2m);
IF (FpulsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpulsDisabled(sr))
    THROW FPUDIS;
fpul, fps ← FCNV_DS(op1, fps);
IF (FpuEnableV(fps) AND FpuCauseV(fps))
    THROW FPUEXC, fps;
IF (FpuCauseE(fps))
    THROW FPUEXC, fps;
IF ((FpuEnableI(fps) OR FpuEnableO(fps)) OR FpuEnableU(fps))
    THROW FPUEXC, fps;
FPSCR ← ZeroExtend32(fps);
FPUL ← ZeroExtend32(fpul);

```

## Exceptions

SLOTFPUDIS, FPUDIS, FPUEXC

# FCNVSD FPUL, DR<sub>n</sub>

## Description

This floating-point instruction performs a single-precision to double-precision floating-point conversion. It reads a single-precision value from FPUL, converts it to double-precision and places the result in DR<sub>n</sub>. FPSCR.RM has no effect since the conversion is exact.

## Operation

### FCNVSD FPUL, DR<sub>n</sub>

1111	n	010101101
15	12	0

Available only when PR=1 and SZ=0

```

sr ← ZeroExtend32(SR);
fps ← ZeroExtend32(FPSCR);
fpul ← SignExtend32(FPUL);
IF (FpulsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpulsDisabled(sr))
    THROW FPUDIS;
op1, fps ← FCNV_SD(fpul, fps);
IF (FpuEnableV(fps) AND FpuCauseV(fps))
    THROW FPUEXC, fps;
IF (FpuCauseE(fps))
    THROW FPUEXC, fps;
DR2n ← FloatRegister64(op1);
FPSCR ← ZeroExtend32(fps);

```

## Exceptions

SLOTFPUDIS, FPUDIS, FPUEXC

**FCNVDS and FCNVSD Special Cases:**

When FPSCR.DN is 1, a positive denormalized number is treated as +0 and a negative denormalized number as -0. This flush-to-zero treatment is applied before exception detection and special case handling.

Exceptional conditions are checked in the order given below. Execution of the instruction is terminated once any check detects an exceptional condition.

- 1 Disabled: an exception is raised if the FPU is disabled.
- 2 Invalid: an invalid operation is signaled if the input is a signaling NaN.
- 3 Error: an FPU error is signaled if FPSCR.DN is zero and the input is a denormalized number.
- 4 Inexact, underflow and overflow: these are checked together and can be signaled in combination. These cases occur for FCNVDS but not for FCNVSD. When inexact, underflow or overflow exceptions are requested by the user, an exception is always raised for FCNVDS regardless of whether that condition arose.

If the instruction does not raise an exception, a result is generated according to the following table.

op1 →	+NORM	-NORM	+0	-0	+INF	-INF	+DNORM, -DNORM	qNaN	sNaN
	CNV	CNV	+0	-0	+INF	-INF	n/a	qNaN	qNaN

FPU error is indicated by heavy shading and always raises an exception. Invalid operations are indicated by light shading and raise an exception if enabled. FPU disabled, inexact, underflow and overflow cases are not shown.

The behavior of the normal 'CNV' case is described by the IEEE754 specification.

# FDIV DRm, DRn

## Description

This floating-point instruction performs a double-precision floating-point division. It divides  $DR_n$  by  $DR_m$  and places the result in  $DR_n$ . The rounding mode is determined by FPSCR.RM.

## Operation

### FDIV DRm, DRn

1111				n		0	m		00011		
15	12	11	9	8	7	5	4			0	

Available only when PR=1 and SZ=0

```

sr ← ZeroExtend32(SR);
fps ← ZeroExtend32(FPSCR);
op1 ← FloatValue64(DR2m);
op2 ← FloatValue64(DR2n);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
    THROW FPUDIS;
op2, fps ← FDIV_D(op2, op1, fps);
IF (FpuEnableV(fps) AND FpuCauseV(fps))
    THROW FPUEXC, fps;
IF (FpuEnableZ(fps) AND FpuCauseZ(fps))
    THROW FPUEXC, fps;
IF (FpuCauseE(fps))
    THROW FPUEXC, fps;
IF ((FpuEnableI(fps) OR FpuEnableO(fps)) OR FpuEnableU(fps))
    THROW FPUEXC, fps;
DR2n ← FloatRegister64(op2);
FPSCR ← ZeroExtend32(fps);

```

## Exceptions

SLOTFPUDIS, FPUDIS, FPUEXC

# FDIV FR<sub>m</sub>, FR<sub>n</sub>

## Description

This floating-point instruction performs a single-precision floating-point division. It divides FR<sub>n</sub> by FR<sub>m</sub> and places the result in FR<sub>n</sub>. The rounding mode is determined by FPSCR.RM.

## Operation

### FDIV FR<sub>m</sub>, FR<sub>n</sub>

1111				n				m				0011																			
15				12				11				8				7				4				3				0			

Available only when PR=0

```

sr ← ZeroExtend32(SR);
fps ← ZeroExtend32(FPSCR);
op1 ← FloatValue32(FRm);
op2 ← FloatValue32(FRn);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
    THROW FPUDIS;
op2, fps ← FDIV_S(op2, op1, fps);
IF (FpuEnableV(fps) AND FpuCauseV(fps))
    THROW FPUExc, fps;
IF (FpuEnableZ(fps) AND FpuCauseZ(fps))
    THROW FPUExc, fps;
IF (FpuCauseE(fps))
    THROW FPUExc, fps;
IF ((FpuEnableI(fps) OR FpuEnableO(fps)) OR FpuEnableU(fps))
    THROW FPUExc, fps;
FRn ← FloatRegister32(op2);
FPSCR ← ZeroExtend32(fps);

```

## Exceptions

SLOTFPUDIS, FPUDIS, FPUExc



**FDIV Special Cases:**

When FPSCR.DN is 1, a positive denormalized number is treated as +0 and a negative denormalized number as -0. This flush-to-zero treatment is applied before exception detection and special case handling.

Exceptional conditions are checked in the order given below. Execution of the instruction is terminated once any check detects an exceptional condition.

- 1 Disabled: an exception is raised if the FPU is disabled.
- 2 Invalid: an invalid operation is signaled if either input is a signaling NaN, or if the division is of a zero by a zero, or of an infinity by an infinity.
- 3 Divide-by-zero: a divide-by-zero is signaled if the divisor is zero and the dividend is a finite non-zero number.
- 4 Error: an FPU error is signaled if FPSCR.DN is zero, neither input is a NaN and either of the following conditions is true: the divisor is a denormalized number, or the dividend is a denormalized number and the divisor is not a zero.
- 5 Inexact, underflow and overflow: these are checked together and can be signaled in combination. When inexact, underflow or overflow exceptions are requested by the user, an exception is always raised regardless of whether that condition arose.

If the instruction does not raise an exception, a result is generated as follows:

op2 → ↓ op1	+NORM, -NORM	+0	-0	+INF	-INF	+DNORM, -DNORM	qNaN	sNaN
+, -NORM	DIV	+0, -0	-0, +0	+INF, -INF	-INF, +INF	n/a	qNaN	qNaN
+0	+INF, -INF	qNaN	qNaN	+INF	-INF	+INF, -INF	qNaN	qNaN
-0	-INF, +INF	qNaN	qNaN	-INF	+INF	-INF, +INF	qNaN	qNaN
+INF	+0, -0	+0	-0	qNaN	qNaN	n/a	qNaN	qNaN
-INF	-0, +0	-0	+0	qNaN	qNaN	n/a	qNaN	qNaN
+, -DNORM	n/a	n/a	n/a	n/a	n/a	n/a	qNaN	qNaN
qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN
sNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN

FPU error is indicated by heavy shading and always raises an exception. Invalid operations and divide-by-zero are indicated by light shading and raise an exception if enabled. FPU disabled, inexact, underflow and overflow cases are not shown.

The behavior of the normal 'DIV' case is described by the IEEE754 specification.

# FIPR FV<sub>m</sub>, FV<sub>n</sub>

## Description

This floating-point instruction computes dot-product of two vectors, FV<sub>m</sub> and FV<sub>n</sub>, and places the result in element 3 of FV<sub>n</sub>. Each vector contains four single-precision floating-point values. The dot-product is specified as:

$$FR_{n+3} = \sum_{i=0}^3 FR_{m+i} \times FR_{n+i}$$

This is an approximate computation. The specified error in the result value:

$$\text{spec\_error} = \begin{cases} 0 & \text{if}(epm = ez) \\ 2^{epm-24} + 2^{E-24+rm} & \text{if}(epm \neq ez) \end{cases}$$

where

$$rm = \begin{cases} 0 & \text{if}(\text{round-to-nearest}) \\ 1 & \text{if}(\text{round-to-zero}) \end{cases}$$

E = unbiased exponent value of the result

ez < -252

epm = max (ep<sub>0</sub>, ep<sub>1</sub>, ep<sub>2</sub>, ep<sub>3</sub>)

ep<sub>i</sub> = pre-normalized exponent of the product FR<sub>m+i</sub> and FR<sub>n+i</sub>

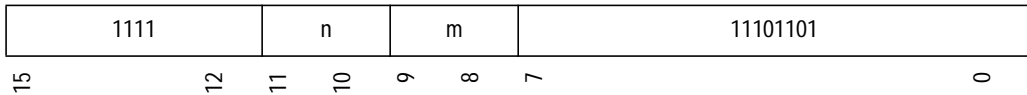
eFR<sub>m+i</sub> = biased exponent value of FR<sub>m+i</sub>

eFR<sub>n+i</sub> = biased exponent value of FR<sub>n+i</sub>

$$ep_i = \begin{cases} ez & \text{if}((FR_{m+i} = 0.0) \text{OR} (FR_{n+i} = 0.0)) \\ \max(eFR_{m+i}, 1) + \max(eFR_{n+i}, 1) - 254 & \text{otherwise} \end{cases}$$

## Operation

### FIPR FV<sub>m</sub>, FV<sub>n</sub>



Available only when PR=0

```

sr ← ZeroExtend32(SR);
fps ← ZeroExtend32(FPSCR);
op1 ← FloatValueVector32(FV4m);
op2 ← FloatValueVector32(FV4n);
IF (FpulsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpulsDisabled(sr))
    THROW FPUDIS;
op2[3], fps ← FIPR_S(op1, op2, fps);
IF (FpuEnableV(fps) AND FpuCauseV(fps))
    THROW FPUExc, fps;
IF ((FpuEnableI(fps) OR FpuEnableO(fps)) OR FpuEnableU(fps))
    THROW FPUExc, fps;
FV4n ← FloatRegisterVector32(op2);
FPSCR ← ZeroExtend32(fps);

```

## Exceptions

SLOTFPUDIS, FPUDIS, FPUExc

### FIPR Special Cases:

FIPR is an approximate instruction. Denormalized numbers are supported:

- When FPSCR.DN is 0, denormalized numbers are treated as their denormalized value in the FIPR.S calculation. This instruction never signals an FPU error.
- When FPSCR.DN is 1, a positive denormalized number is treated as +0 and a negative denormalized number as -0. This flush-to-zero treatment is applied before exception detection and special case handling.

Exceptional conditions are checked in the order given below. Execution of the instruction is terminated once any check detects an exceptional condition.

- 1 Disabled: an exception is raised if the FPU is disabled.
- 2 Invalid: an invalid operation is signaled if any of the following arise:
  - Any of the inputs is a signaling NaN.
  - Multiplication of a zero by an infinity.
  - Addition of differently signed infinities where none of the inputs is a qNaN.

The multiplication is performed with sufficient precision to avoid overflow, and therefore the multiplication of any two finite numbers does not produce an infinity. The multiplication result will be an infinity only if there is a multiplication of an infinity with a normalized number, an infinity with a denormalized number or an infinity with an infinity.

The addition of differently signed infinities is detected if there is (at least) one positive infinity and (at least) one negative infinity in the set of 4 multiplication results.

- 3 Inexact, underflow and overflow: these are checked together and can be signaled in combination. This is an approximate instruction and inexact is signaled except where special cases occur. Precise details of the approximate inner-product algorithm, including the detection of underflow and overflow cases, are implementation dependent. When inexact, underflow or overflow exceptions are requested by the user, an exception is always raised regardless of whether that condition arose.

If the instruction does not raise an exception, a result is generated according to the following tables. Where the behavior is not a special case, the instruction computes an approximate result using an implementation-dependent algorithm.

**FIPR Special Cases (continued):**

Each of the 4 pairs of multiplication operands (op1 and op2) is selected from corresponding elements of the two 4-element source vectors and multiplied:

op1 → ↓ op2	+, -NORM, +, -DENORM	+0	-0	+INF	-INF	qNaN	sNaN
+, -NORM    +, -DENORM	FIPRMUL	+0, -0	-0, +0	+INF, -INF	-INF, +INF	qNaN	qNaN
+0	+0, -0	+0	-0	qNaN	qNaN	qNaN	qNaN
-0	-0, +0	-0	+0	qNaN	qNaN	qNaN	qNaN
+INF	+INF, -INF	qNaN	qNaN	+INF	-INF	qNaN	qNaN
-INF	-INF, +INF	qNaN	qNaN	-INF	+INF	qNaN	qNaN
qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN
sNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN

If any of the multiplications evaluates to qNaN, then the result of the instruction is qNaN and no further analysis need be performed. In the 'FIPRMUL', +0, -0, +INF and -INF cases, the 4 addition operands (labelled intermediate 0 to 3) are summed:

intermediate 0 →		FIPRMUL, +0, -0			+INF			-INF		
intermediate 1 →		FIPRMUL, +0, -0			+INF			-INF		
↓ intermediate 2	↓ intermediate 3	FIPRMUL, +0, -0	+INF	-INF	FIPRMUL, +0, -0	+INF	-INF	FIPRMUL, +0, -0	+INF	-INF
FIPRMUL, +0, -0	FIPRMUL, +0, -0	FIPRADD	+INF	-INF	+INF	+INF	qNaN	-INF	qNaN	-INF
	+INF	+INF	+INF	qNaN	+INF	+INF	qNaN	qNaN	qNaN	qNaN
	-INF	-INF	qNaN	-INF	qNaN	qNaN	qNaN	-INF	qNaN	-INF
+INF	FIPRMUL, +0, -0	+INF	+INF	qNaN	+INF	+INF	qNaN	qNaN	qNaN	qNaN
	+INF	+INF	+INF	qNaN	+INF	+INF	qNaN	qNaN	qNaN	qNaN
	-INF	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN
-INF	FIPRMUL, +0, -0	-INF	qNaN	-INF	qNaN	qNaN	qNaN	-INF	qNaN	-INF
	+INF	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN
	-INF	-INF	qNaN	-INF	qNaN	qNaN	qNaN	-INF	qNaN	-INF

Inexact is signaled in the 'FIPRADD' case. Invalid operations are indicated by light shading and raise an exception if enabled. FPU disabled, inexact, underflow and overflow cases are not shown.

# FLDS FR<sub>m</sub>, FPUL

## Description

This floating-point instruction copies FR<sub>m</sub> to FPUL.

This instruction is not considered an arithmetic operation, and it does not signal invalid operations.

## Operation

### FLDS FR<sub>m</sub>, FPUL

1111	m	00011101
15	11	0

```

sr ← ZeroExtend32(SR);
op1 ← FloatValue32(FRm);
IF (FpulsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpulsDisabled(sr))
    THROW FPUDIS;
fpul ← op1;
FPUL ← ZeroExtend32(fpul);

```

## Exceptions

SLOTFPUDIS, FPUDIS

# FLDI0 FR<sub>n</sub>

## Description

This floating-point instruction loads a constant representing the single-precision floating-point value of 0.0 into FR<sub>n</sub>.

## Operation

### FLDI0 FR<sub>n</sub>

1111				n				10001101						
15	12	11		8	7									0

Available only when PR=0

```

sr ← ZeroExtend32(SR);
IF (FpulsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpulsDisabled(sr))
    THROW FPUDIS;
op1 ← 0x00000000;
FRn ← FloatRegister32(op1);

```

## Exceptions

SLOTFPUDIS, FPUDIS



# FLDI1 FR<sub>n</sub>

## Description

This floating-point instruction loads a constant representing the single-precision floating-point value of 1.0 into FR<sub>n</sub>.

## Operation

### FLDI1 FR<sub>n</sub>

1111	n	10011101
15	12	0

Available only when PR=0

```

sr ← ZeroExtend32(SR);
IF (FpulsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpulsDisabled(sr))
    THROW FPUDIS;
op1 ← 0x3F800000;
FRn ← FloatRegister32(op1);

```

## Exceptions

SLOTFPUDIS, FPUDIS

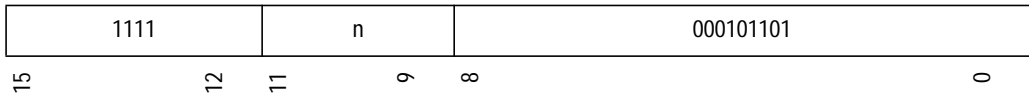
# FLOAT FPUL, DR<sub>n</sub>

## Description

This floating-point instruction performs a signed 32-bit integer to double-precision floating-point conversion. It reads a signed 32-bit integer value from FPUL, converts it to a double-precision range and places the result in DR<sub>n</sub>. In all cases the provided integer value will be exactly represented in the destination floating-point format. FPSCR.RM has no effect since the conversion is exact.

## Operation

### FLOAT FPUL, DR<sub>n</sub>



Available only when PR=1 and SZ=0

```

fpul ← SignExtend32(FPUL);
sr ← ZeroExtend32(SR);
fps ← ZeroExtend32(FPSCR);
IF (FpulsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpulsDisabled(sr))
    THROW FPUDIS;
op1, fps ← FLOAT_LD(fpul, fps);
DR2n ← FloatRegister64(op1);
  
```

## Exceptions

SLOTFPUDIS, FPUDIS, FPUEXC

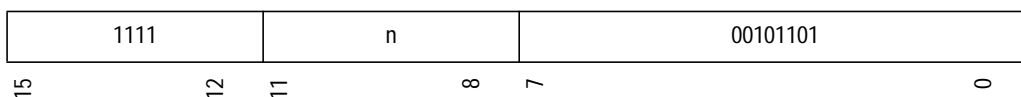
# FLOAT FPUL, FR<sub>n</sub>

## Description

This floating-point instruction performs a signed 32-bit integer to single-precision floating-point conversion. It reads a signed 32-bit integer value from FPUL, converts it to a single-precision range and places the result in FR<sub>n</sub>. In cases where the integer value cannot be exactly represented in the destination floating-point format, the rounding mode is determined by FPSCR.RM.

## Operation

### FLOAT FPUL, FR<sub>n</sub>



Available only when PR=0

```

sr ← ZeroExtend32(SR);
fps ← ZeroExtend32(FPSCR);
fpul ← SignExtend32(FPUL);
IF (FpulsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpulsDisabled(sr))
    THROW FPUDIS;
op1, fps ← FLOAT_LS(fpul, fps);
IF (FpuEnableI(fps))
    THROW FPUEXC, fps;
FRn ← FloatRegister32(op1);
FPSCR ← ZeroExtend32(fps);

```

## Exceptions

SLOTFPUDIS, FPUDIS, FPUEXC

## FLOAT Special Cases:

Exceptional conditions are checked in the order given below. Execution of the instruction is terminated once any check detects an exceptional condition.

- 1 Disabled: an exception is raised if the FPU is disabled.
- 2 Inexact: inexact can occur for FLOAT FPUL,  $FR_n$  but not for FLOAT FPUL,  $DR_n$ . When inexact exceptions are requested by the user, an exception is always raised for FLOAT FPUL,  $FR_n$  regardless of whether that condition arose. Overflow and underflow do not occur for either of these instructions.

If the instruction does not raise an exception, the conversion is performed as indicated by the IEEE754 specification.

# FMAC FR0, FRm, FRn

## Description

This floating-point instruction performs a single-precision floating-point multiply-accumulate. It multiplies  $FR_0$  by  $FR_m$ , adds this intermediate to  $FR_n$  and places the result back to  $FR_n$ . The multiplication and addition are performed as if the exponent and precision ranges were unbounded, followed by one rounding down to single-precision format. The rounding mode is determined by  $FPSCR.RM$ .

## Operation

### FMAC FR0, FRm, FRn

1111	n	m	1110
15	12	11	8
		7	4
			3
			0

Available only when  $PR=0$

```

sr ← ZeroExtend32(SR);
fps ← ZeroExtend32(FPSCR);
fr0 ← FloatValue32(FR0);
op1 ← FloatValue32(FRm);
op2 ← FloatValue32(FRn);
IF (FpulsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpulsDisabled(sr))
    THROW FPUDIS;
op2, fps ← FMAC_S(fr0, op1, op2, fps);
IF (FpuEnableV(fps) AND FpuCauseV(fps))
    THROW FPUEXC, fps;
IF (FpuCauseE(fps))
    THROW FPUEXC, fps;
IF ((FpuEnableI(fps) OR FpuEnableO(fps)) OR FpuEnableU(fps))
    THROW FPUEXC, fps;
FRn ← FloatRegister32(op2);
FPSCR ← ZeroExtend32(fps);

```

## Exceptions

SLOTFPUDIS, FPUDIS, FPUEXC

**FMAC Special Cases:**

When FPSCR.DN is 1, a positive denormalized number is treated as +0 and a negative denormalized number as -0. This flush-to-zero treatment is applied before exception detection and special case handling.

Exceptional conditions are checked in the order given below. Execution of the instruction is terminated once any check detects an exceptional condition.

- 1 Disabled: an exception is raised if the FPU is disabled.
- 2 Invalid: an invalid operation is signaled if any of the three inputs is a signaling NaN, there is a multiplication of a zero by an infinity, or there is an addition of differently signed infinities.

The multiplication is performed with sufficient precision to avoid overflow, and therefore the multiplication of any two finite numbers does not produce an infinity. The multiplication result will be an infinity only if there is a multiplication of an infinity with a normalized number, an infinity with a denormalized number or an infinity with an infinity.

- 3 Error: an FPU error is signaled if FPSCR.DN is 0 and none of the inputs are a NaN and at least one of the inputs is a denormalized number.
- 4 Inexact, underflow and overflow: these are checked together and can be signaled in combination. The multiply-accumulate is implemented using a fused-mac algorithm, and these are detected during the conversion of the exactly evaluated intermediate to the single-precision result. When inexact, underflow or overflow exceptions are requested by the user, an exception is always raised regardless of whether that condition arose.

If the instruction does not raise an exception, a result is generated according to the following tables.

Firstly, the operands are checked for sNaN:

fr0 →	other		sNaN	
op1 → ↓ op2	other	sNaN	other	sNaN
other		qNaN	qNaN	qNaN
sNaN	qNaN	qNaN	qNaN	qNaN

**FMAC Special Cases (continued):**

If the result of the previous table is a qNaN, no further analysis is performed. In all other cases, fr0 and op1 are checked for a zero multiplied by an infinity:

fr0 → ↓ op1	other	+0	-0	+INF	-INF
other					
+0				qNaN	qNaN
-0				qNaN	qNaN
+INF	qNaN		qNaN		
-INF	qNaN		qNaN		

If the result of the previous table is a qNaN, no further analysis is performed. In all other cases, the operands are checked for input qNaN values:

fr0 → op1 → ↓ op2	other		qNaN	
	other	qNaN	other	qNaN
other		qNaN	qNaN	qNaN
qNaN	qNaN	qNaN	qNaN	qNaN

By this stage all operations involving sNaN or qNaN operands have been dealt with. If the result of the previous table is a qNaN, no further analysis is performed. In all other cases, the operands are checked for the addition of differently signed infinities:

fr0 → op1 → ↓ op2	+other				-other				+INF				-INF			
	+other	-other	+INF	-INF	+other	-other	+INF	-INF	+other	-other	+INF	-INF	+other	-other	+INF	-INF
+other																
-other																
+INF				qNaN				qNaN				qNaN	qNaN	qNaN	qNaN	
-INF			qNaN						qNaN	qNaN			qNaN			qNaN

**FMAC Special Cases (continued):**

If the result of the previous table is a qNaN, no further analysis is performed. In all other cases, fr0 and op1 are multiplied:

fr0 → ↓ op1	+NORM, -NORM	+0	-0	+INF	-INF	+DNORM, -DNORM
+, -NORM	FULLMUL	+0, -0	-0, +0	+INF, -INF	-INF, +INF	n/a
+0	+0, -0	+0	-0			n/a
-0	-0, +0	-0	+0			n/a
+INF	+INF, -INF			+INF	-INF	n/a
-INF	-INF, +INF			-INF	+INF	n/a
+, -DNORM	n/a	n/a	n/a	n/a	n/a	n/a

The empty cells in this table correspond to cases that have already been dealt with. If either source is denormalized, no further analysis is performed. In the 'FULLMUL' case, a multiplication is performed without loss of precision. There is no rounding nor overflow, and this multiplication cannot produce an intermediate infinity.

In the 'FULLMUL', +0, -0, +INF and -INF cases, the 2 addition operands (fr0\*op1 and op2) are summed:

(fr0*op1) → ↓ op2	FULLMUL	+0	-0	+INF	-INF
+, -NORM	FULLADD	op2	op2	+INF	-INF
+0	FULLADD	+0	+0	+INF	-INF
-0	FULLADD	+0	-0	+INF	-INF
+INF	+INF	+INF	+INF	+INF	
-INF	-INF	-INF	-INF		-INF
+, -DNORM	n/a	n/a	n/a	n/a	n/a

The two empty cells in this table correspond to cases that have already been dealt with. In the 'FULLADD' cases the fully-precise addition intermediate is rounded to give a single-precision result.



---

In the above tables, FPU error is indicated by heavy shading and always raises an exception. Invalid operations are indicated by light shading and raise an exception if enabled. FPU disabled, inexact, underflow and overflow cases are not shown.

# FMOV DR<sub>m</sub>, DR<sub>n</sub>

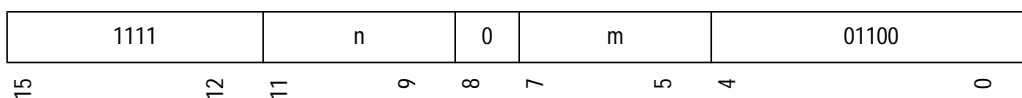
## Description

This floating-point instruction reads a pair of single-precision floating-point values from DR<sub>m</sub> and copies them to DR<sub>n</sub>. This is a bit-by-bit copy with no interpretation or conversion of the values.

This instruction is not considered an arithmetic operation, and it does not signal invalid operations. There are no special floating-point cases for this instruction.

## Operation

### FMOV DR<sub>m</sub>, DR<sub>n</sub>



Available only when PR=0 and SZ=1

```

sr ← ZeroExtend32(SR);
op1 ← FloatValuePair32(FP2m);
IF (FpulsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpulsDisabled(sr))
    THROW FPUDIS;
op2 ← op1;
FP2n ← FloatRegisterPair32(op2);

```

## Exceptions

SLOTFPUDIS, FPUDIS

# FMOV DR<sub>m</sub>, XD<sub>n</sub>

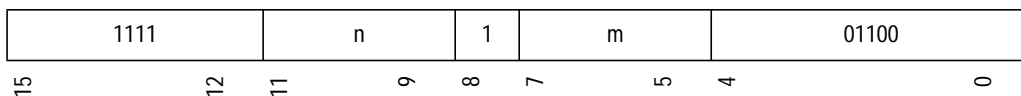
## Description

This floating-point instruction reads a pair of single-precision floating-point values from DR<sub>m</sub> and copies them to XD<sub>n</sub>. This is a bit-by-bit copy with no interpretation or conversion of the values.

This instruction is not considered an arithmetic operation, and it does not signal invalid operations. There are no special floating-point cases for this instruction.

## Operation

### FMOV DR<sub>m</sub>, XD<sub>n</sub>



Available only when PR=0 and SZ=1

```

sr ← ZeroExtend32(SR);
op1 ← FloatValuePair32(DR2m);
IF (FpulsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpulsDisabled(sr))
    THROW FPUDIS;
op2 ← op1;
XD2n ← FloatRegisterPair32(op2);

```

## Exceptions

SLOTFPUDIS, FPUDIS

# FMOV DRm, @Rn

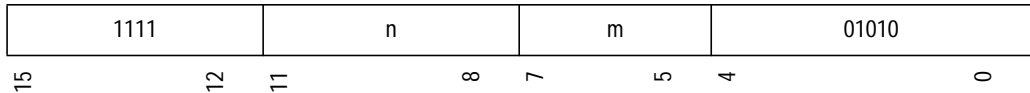
## Description

This floating-point instruction stores a pair of single-precision floating-point registers to memory using register indirect with zero-displacement addressing.  $DR_m$  is written as two consecutive 32-bit values to the effective address specified in  $R_n$ .

This instruction is not considered an arithmetic operation, and it does not signal invalid operations. There are no special floating-point cases for this instruction.

## Operation

### FMOV DRm, @Rn



Available only when PR=0 and SZ=1

```

sr ← ZeroExtend32(SR);
fps ← ZeroExtend32(FPSCR);
op1 ← FloatValuePair32(FP2m);
op2 ← SignExtend32(Rn);
IF (FpulsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpulsDisabled(sr))
    THROW FPUDIS;
address ← ZeroExtend32(op2);
WriteMemoryPair32(address, op1);

```

## Exceptions

SLOTFPUDIS, FPUDIS, WADDERR, WTLBMISS, WRITEPROT, FIRSTWRITE

## Note

# FMOV DRm, @-Rn

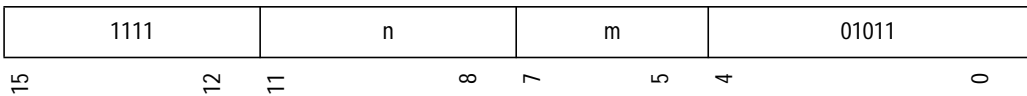
## Description

This floating-point instruction stores a pair of single-precision floating-point registers to memory using register indirect with pre-decrement addressing.  $R_n$  is pre-decremented by 8 to give the effective address.  $DR_m$  is written as two consecutive 32-bit values to the effective address.

This instruction is not considered an arithmetic operation, and it does not signal invalid operations. There are no special floating-point cases for this instruction.

## Operation

### FMOV DRm, @-Rn



Available only when PR=0 and SZ=1

```

sr ← ZeroExtend32(SR);
fps ← ZeroExtend32(FPSCR);
op1 ← FloatValuePair32(FP2m);
op2 ← SignExtend32(Rn);
IF (FpulsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpulsDisabled(sr))
    THROW FPUDIS;
address ← ZeroExtend32(op2 - 8);
WriteMemoryPair32(address, op1);
op2 ← address;
Rn ← Register(op2);
  
```

**Exceptions**

SLOTFPUDIS, FPUDIS, WADDERR, WTLBMISS, WRITEPROT, FIRSTWRITE

**Note**

The effective address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.

# FMOV DRm, @(R0, Rn)

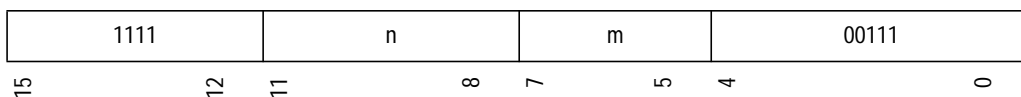
## Description

This floating-point instruction stores a pair of single-precision floating-point registers to memory using register indirect addressing. The effective address is formed by adding  $R_0$  to  $R_n$ .  $DR_m$  is written as two consecutive 32-bit values to the effective address.

This instruction is not considered an arithmetic operation, and it does not signal invalid operations. There are no special floating-point cases for this instruction.

## Operation

**FMOV DRm, @(R0, Rn)**



Available only when PR=0 and SZ=1

```

sr ← ZeroExtend32(SR);
r0 ← SignExtend32(R0);
op1 ← FloatValuePair32(FP2m);
op2 ← SignExtend32(Rn);
IF (FpulsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpulsDisabled(sr))
    THROW FPUDIS;
address ← ZeroExtend32(r0 + op2);
WriteMemoryPair32(address, op1);

```

## Exceptions

SLOTFPUDIS, FPUDIS, WADDERR, WTLBMISS, WRITEPROT, FIRSTWRITE

## Note

The effective address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.

# FMOV.S FR<sub>m</sub>, FR<sub>n</sub>

## Description

This floating-point instruction reads a single-precision floating-point value from FR<sub>m</sub> and copies it to FR<sub>n</sub>. This is a bit-by-bit copy with no interpretation or conversion of the value.

This instruction is not considered an arithmetic operation, and it does not signal invalid operations. There are no special floating-point cases for this instruction.

## Operation

### FMOV.S FR<sub>m</sub>, FR<sub>n</sub>

1111				n		m		1100	
15	12	11		8	7		4	3	0

Available only when SZ=0

```

sr ← ZeroExtend32(SR);
fps ← ZeroExtend32(FPSCR);
op1 ← FloatValue32(FRm);
IF (FpulsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpulsDisabled(sr))
    THROW FPUDIS;
op2 ← op1;
FRn ← FloatRegister32(op2);

```

## Exceptions

SLOTFPUDIS, FPUDIS



# FMOV.S FR<sub>m</sub>, @R<sub>n</sub>

## Description

This floating-point instruction stores a single-precision floating-point register to memory using register indirect with zero-displacement addressing. The 32-bit value of FR<sub>m</sub> is written to the effective address specified in R<sub>n</sub>.

This instruction is not considered an arithmetic operation, and it does not signal invalid operations. There are no special floating-point cases for this instruction.

## Operation

### FMOV.S FR<sub>m</sub>, @R<sub>n</sub>

1111				n		m		1010	
15	12	11		8	7		4	3	0

Available only when SZ=0

```

sr ← ZeroExtend32(SR);
fps ← ZeroExtend32(FPSCR);
op1 ← FloatValue32(FRm);
op2 ← SignExtend32(Rn);
IF (FpulsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpulsDisabled(sr))
    THROW FPUDIS;
address ← ZeroExtend32(op2);
WriteMemory32(address, op1);

```

## Exceptions

SLOTFPUDIS, FPUDIS, WADDERR, WTLBMISS, WRITEPROT, FIRSTWRITE

## Note

# FMOV.S FR<sub>m</sub>, @-R<sub>n</sub>

## Description

This floating-point instruction stores a single-precision floating-point register to memory using register indirect with pre-decrement addressing. R<sub>n</sub> is pre-decremented by 4 to give the effective address. The 32-bit value of FR<sub>m</sub> is written to the effective address.

This instruction is not considered an arithmetic operation, and it does not signal invalid operations. There are no special floating-point cases for this instruction.

## Operation

### FMOV.S FR<sub>m</sub>, @-R<sub>n</sub>

15	12	11	8	7	4	3	0
1111	n	m	1011				

Available only when SZ=0

```

sr ← ZeroExtend32(SR);
fps ← ZeroExtend32(FPSCR);
op1 ← FloatValue32(FRm);
op2 ← SignExtend32(Rn);
IF (FpulsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpulsDisabled(sr))
    THROW FPUDIS;
address ← ZeroExtend32(op2 - 4);
WriteMemory32(address, op1);
op2 ← address;
Rn ← Register(op2);

```

**Exceptions**

SLOTFPUDIS, FPUDIS, WADDERR, WTLBMISS, WRITEPROT, FIRSTWRITE

**Note**

The effective address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.

# FMOV.S FR<sub>m</sub>, @(R0, R<sub>n</sub>)

## Description

This floating-point instruction stores a single-precision floating-point register to memory using register indirect addressing. The effective address is formed by adding R<sub>0</sub> to R<sub>n</sub>. The 32-bit value of FR<sub>m</sub> is written to the effective address.

This instruction is not considered an arithmetic operation, and it does not signal invalid operations. There are no special floating-point cases for this instruction.

## Operation

### FMOV.S FR<sub>m</sub>, @(R0, R<sub>n</sub>)

15	12	11	8	7	4	3	0
1111	n				m	0111	

Available only when SZ=0

```

sr ← ZeroExtend32(SR);
r0 ← SignExtend32(R0);
op1 ← FloatValue32(FRm);
op2 ← SignExtend32(Rn);
IF (FpulsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpulsDisabled(sr))
    THROW FPUDIS;
address ← ZeroExtend32(r0 + op2);
WriteMemory32(address, op1);

```

## Exceptions

SLOTFPUDIS, FPUDIS, WADDERR, WTLBMISS, WRITEPROT, FIRSTWRITE

## Note

The effective address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.

# FMOV XD<sub>m</sub>, DR<sub>n</sub>

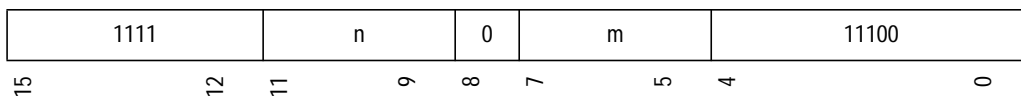
## Description

This floating-point instruction reads a pair of single-precision floating-point values from XD<sub>m</sub> and copies them to DR<sub>n</sub>. This is a bit-by-bit copy with no interpretation or conversion of the values.

This instruction is not considered an arithmetic operation, and it does not signal invalid operations. There are no special floating-point cases for this instruction.

## Operation

### FMOV XD<sub>m</sub>, DR<sub>n</sub>



Available only when PR=0 and SZ=1

```

sr ← ZeroExtend32(SR);
fps ← ZeroExtend32(FPSCR);
op1 ← FloatValuePair32(XD2m);
IF (FpulsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpulsDisabled(sr))
    THROW FPUDIS;
op2 ← op1;
DR2n ← FloatRegisterPair32(op2);

```

## Exceptions

SLOTFPUDIS, FPUDIS

# FMOV XD<sub>m</sub>, XD<sub>n</sub>

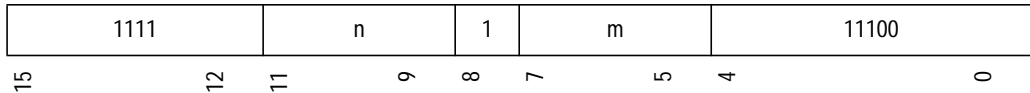
## Description

This floating-point instruction reads a pair of single-precision floating-point values from XD<sub>m</sub> and copies them to XD<sub>n</sub>. This is a bit-by-bit copy with no interpretation or conversion of the values.

This instruction is not considered an arithmetic operation, and it does not signal invalid operations. There are no special floating-point cases for this instruction.

## Operation

### FMOV XD<sub>m</sub>, XD<sub>n</sub>



Available only when PR=0 and SZ=1

```

sr ← ZeroExtend32(SR);
fps ← ZeroExtend32(FPSCR);
op1 ← FloatValue64(XD2m);
IF (FpulsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpulsDisabled(sr))
    THROW FPUDIS;
op2 ← op1;
XD2n ← FloatRegister64(op2);
  
```

## Exceptions

SLOTFPUDIS, FPUDIS

# FMOV XDm, @Rn

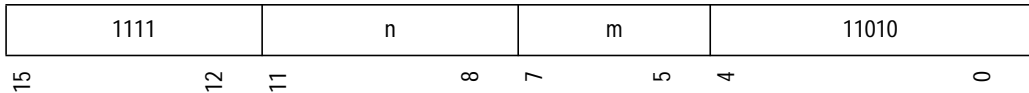
## Description

This floating-point instruction stores a pair of single-precision floating-point registers to memory using register indirect with zero-displacement addressing. XD<sub>m</sub> is written as two consecutive 32-bit values to the effective address specified in R<sub>n</sub>.

This instruction is not considered an arithmetic operation, and it does not signal invalid operations. There are no special floating-point cases for this instruction.

## Operation

**FMOV XDm, @Rn**



Available only when PR=0 and SZ=1

```

sr ← ZeroExtend32(SR);
fps ← ZeroExtend32(FPSCR);
op1 ← FloatValuePair32(XD2m);
op2 ← SignExtend32(Rn);
IF (FpulsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpulsDisabled(sr))
    THROW FPUDIS;
address ← ZeroExtend32(op2);
WriteMemoryPair32(address, op1);

```

## Exceptions

SLOTFPUDIS, FPUDIS, WADDERR, WTLBMISS, WRITEPROT, FIRSTWRITE

## Note

# FMOV XDm, @-Rn

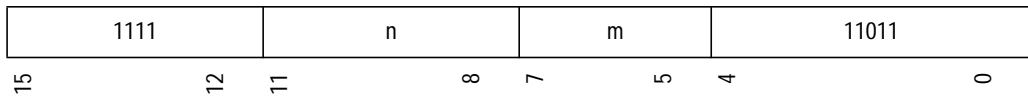
## Description

This floating-point instruction stores a pair of single-precision floating-point registers to memory using register indirect with pre-decrement addressing.  $R_n$  is pre-decremented by 8 to give the effective address.  $XD_m$  is written as two consecutive 32-bit values to the effective address.

This instruction is not considered an arithmetic operation, and it does not signal invalid operations. There are no special floating-point cases for this instruction.

## Operation

### FMOV XDm, @-Rn



Available only when PR=0 and SZ=1

```

sr ← ZeroExtend32(SR);
fps ← ZeroExtend32(FPSCR);
op1 ← FloatValuePair32(XD2m);
op2 ← SignExtend32(Rn);
IF (FpulsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpulsDisabled(sr))
    THROW FPUDIS;
address ← ZeroExtend32(op2 - 8);
WriteMemoryPair32(address, op1);
op2 ← address;
Rn ← Register(op2);
FPSCR ← ZeroExtend32(fps);
  
```



**Exceptions**

SLOTFPUDIS, FPUDIS, WADDERR, WTLBMISS, WRITEPROT, FIRSTWRITE

**Note**

The effective address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.

# FMOV XD<sub>m</sub>, @(R0, R<sub>n</sub>)

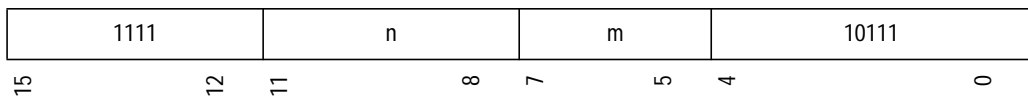
## Description

This floating-point instruction stores a pair of single-precision floating-point registers to memory using register indirect addressing. The effective address is formed by adding R<sub>0</sub> to R<sub>n</sub>. XD<sub>m</sub> is written as two consecutive 32-bit values to the effective address.

This instruction is not considered an arithmetic operation, and it does not signal invalid operations. There are no special floating-point cases for this instruction.

## Operation

**FMOV XD<sub>m</sub>, @(R0, R<sub>n</sub>)**



Available only when PR=0 and SZ=1

```

sr ← ZeroExtend32(SR);
r0 ← SignExtend32(R0);
op1 ← FloatValuePair32(XD2m);
op2 ← SignExtend32(Rn);
IF (FpulsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpulsDisabled(sr))
    THROW FPUDIS;
address ← ZeroExtend32(r0 + op2);
WriteMemoryPair32(address, op1);

```

## Exceptions

SLOTFPUDIS, FPUDIS, WADDERR, WTLBMISS, WRITEPROT, FIRSTWRITE

## Note

The effective address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.

# FMOV @Rm, DRn

## Description

This floating-point instruction loads a pair of single-precision floating-point registers from memory using register indirect with zero-displacement addressing. Two consecutive 32-bit values are read from the effective address specified in  $R_m$  and loaded into  $DR_n$ .

This instruction is not considered an arithmetic operation, and it does not signal invalid operations. There are no special floating-point cases for this instruction.

## Operation

### FMOV @Rm, DRn

15	12	11	9	8	7	4	3	0
1111	n	0	m	1000				

Available only when PR=0 and SZ=1

```

sr ← ZeroExtend32(SR);
op1 ← SignExtend32(Rm);
IF (FpulsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpulsDisabled(sr))
    THROW FPUDIS;
address ← ZeroExtend32(op1);
op2 ← ReadMemoryPair32(address);
FP2n ← FloatRegisterPair32(op2);

```

## Exceptions

SLOTFPUDIS, FPUDIS, RADDERR, RTLBMIS, READPROT

## Note

# FMOV @Rm+, DRn

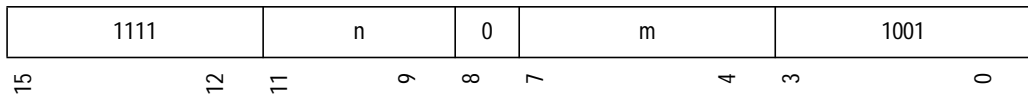
## Description

This floating-point instruction loads a pair of single-precision floating-point registers from memory using register indirect with post-increment addressing. Two consecutive 32-bit values are read from the effective address specified in  $R_m$  and loaded into  $DR_n$ .  $R_m$  is post-incremented by 8.

This instruction is not considered an arithmetic operation, and it does not signal invalid operations. There are no special floating-point cases for this instruction.

## Operation

### FMOV @Rm+, DRn



Available only when PR=0 and SZ=1

```

sr ← ZeroExtend32(SR);
fps ← ZeroExtend32(FPSCR);
op1 ← SignExtend32(Rm);
IF (FpulsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpulsDisabled(sr))
    THROW FPUDIS;
address ← ZeroExtend32(op1);
op2 ← ReadMemoryPair32(address);
op1 ← op1 + 8;
Rm ← Register(op1);
FP2n ← FloatRegisterPair32(op2);

```

## Exceptions

SLOTFPUDIS, FPUDIS, RADDERR, RTLBMIS, READPROT

## Note

# FMOV @(R0, Rm), DRn

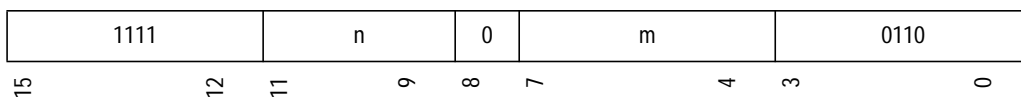
## Description

This floating-point instruction loads a pair of single-precision floating-point registers from memory using register indirect addressing. The effective address is formed by adding  $R_0$  to  $R_n$ . Two consecutive 32-bit values are read from the effective address and loaded into  $DR_n$ .

This instruction is not considered an arithmetic operation, and it does not signal invalid operations. There are no special floating-point cases for this instruction.

## Operation

**FMOV @(R0, Rm), DRn**



Available only when PR=0 and SZ=1

```

sr ← ZeroExtend32(SR);
r0 ← SignExtend32(R0);
op1 ← SignExtend32(Rm);
IF (FpulsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpulsDisabled(sr))
    THROW FPUDIS;
address ← ZeroExtend32(r0 + op1);
op2 ← ReadMemoryPair32(address);
FP2n ← FloatRegisterPair32(op2);

```

## Exceptions

SLOTFPUDIS, FPUDIS, RADDERR, RTLBMIS, READPROT

The effective address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.

# FMOV.S @Rm, FRn

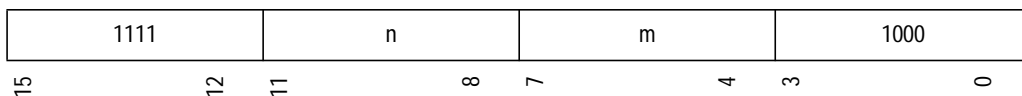
## Description

This floating-point instruction loads a single-precision floating-point register from memory using register indirect with zero-displacement addressing. A 32-bit value is read from the effective address specified in  $R_m$  and loaded into  $FR_n$ .

This instruction is not considered an arithmetic operation, and it does not signal invalid operations. There are no special floating-point cases for this instruction.

## Operation

### FMOV.S @Rm, FRn



Available only when SZ=0

```

sr ← ZeroExtend32(SR);
fps ← ZeroExtend32(FPSCR);
op1 ← SignExtend32(Rm);
IF (FpulsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpulsDisabled(sr))
    THROW FPUDIS;
address ← ZeroExtend32(op1);
op2 ← ReadMemory32(address);
FR2n ← FloatRegister32(op2);

```

## Exceptions

SLOTFPUDIS, FPUDIS, RADDERR, RTLBMIS, READPROT

## Note

# FMOV.S @Rm+, FRn

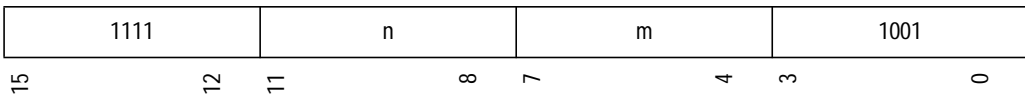
## Description

This floating-point instruction loads a single-precision floating-point register from memory using register indirect with post-increment addressing. A 32-bit value is read from the effective address specified in  $R_m$  and loaded into  $FR_n$ .  $R_m$  is post-incremented by 4.

This instruction is not considered an arithmetic operation, and it does not signal invalid operations. There are no special floating-point cases for this instruction.

## Operation

### FMOV.S @Rm+, FRn



Available only when SZ=0

```

sr ← ZeroExtend32(SR);
op1 ← SignExtend32(Rm);
IF (FpulsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpulsDisabled(sr))
    THROW FPUDIS;
address ← ZeroExtend32(op1);
op2 ← ReadMemory32(address);
op1 ← op1 + 4;
Rm ← Register(op1);
FRn ← FloatRegister32(op2);

```

## Exceptions

SLOTFPUDIS, FPUDIS, RADDERR, RTLBMIS, READPROT

## Note

# FMOV.S @(R0, Rm), FRn

## Description

This floating-point instruction loads a single-precision floating-point register from memory using register indirect addressing. The effective address is formed by adding  $R_0$  to  $R_n$ . A 32-bit value is read from the effective address and loaded into  $FR_n$ .

This instruction is not considered an arithmetic operation, and it does not signal invalid operations. There are no special floating-point cases for this instruction.

## Operation

### FMOV.S @(R0, Rm), FRn

1111				n				m				0110																			
15				12				11				8				7				4				3				0			

Available only when SZ=0

```

sr ← ZeroExtend32(SR);
r0 ← SignExtend32(R0);
op1 ← SignExtend32(Rm);
IF (FpulsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpulsDisabled(sr))
    THROW FPUDIS;
address ← ZeroExtend32(r0 + op1);
op2 ← ReadMemory32(address);
FRn ← FloatRegister32(op2);

```

## Exceptions

SLOTFPUDIS, FPUDIS, RADDERR, RTLBMIS, READPROT

## Note

The effective address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.



# FMOV @Rm, XDn

## Description

This floating-point instruction loads a pair of single-precision floating-point registers from memory using register indirect with zero-displacement addressing. Two consecutive 32-bit values are read from the effective address specified in  $R_m$  and loaded into  $XD_n$ .

This instruction is not considered an arithmetic operation, and it does not signal invalid operations. There are no special floating-point cases for this instruction.

## Operation

### FMOV @Rm, XDn

15	12	11	9	8	7	4	3	0
1111	n	1	m	1000				

Available only when PR=0 and SZ=1

```

sr ← ZeroExtend32(SR);
fps ← ZeroExtend32(FPSCR);
op1 ← SignExtend32(Rm);
IF (FpulsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpulsDisabled(sr))
    THROW FPUDIS;
address ← ZeroExtend32(op1);
op2 ← ReadMemoryPair32(address);
XD2n ← FloatRegisterPair32(op2);

```

## Exceptions

SLOTFPUDIS, FPUDIS, RADDERR, RTLBMIS, READPROT

## Note

# FMOV @Rm+, XDn

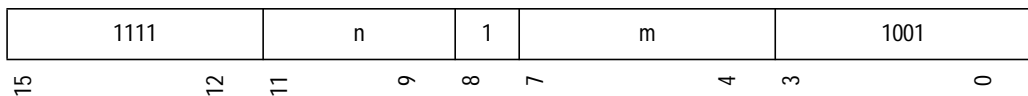
## Description

This floating-point instruction loads a pair of single-precision floating-point registers from memory using register indirect with post-increment addressing. Two consecutive 32-bit values are read from the effective address specified in  $R_m$  and loaded into  $XD_n$ .  $R_m$  is post-incremented by 8.

This instruction is not considered an arithmetic operation, and it does not signal invalid operations. There are no special floating-point cases for this instruction.

## Operation

### FMOV @Rm+, XDn



Available only when PR=0 and SZ=1

```

sr ← ZeroExtend32(SR);
fps ← ZeroExtend32(FPSCR);
op1 ← SignExtend32(Rm);
IF (FpulsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpulsDisabled(sr))
    THROW FPUDIS;
address ← ZeroExtend32(op1);
op2 ← ReadMemoryPair32(address);
op1 ← op1 + 8;
Rm ← Register(op1);
XD2n ← FloatRegisterPair32(op2);

```

## Exceptions

SLOTFPUDIS, FPUDIS, RADDERR, RTLBMIS, READPROT

## Note

# FMOV @(R0, Rm), XDn

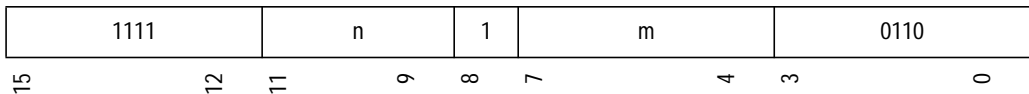
## Description

This floating-point instruction loads a pair of single-precision floating-point registers from memory using register indirect addressing. The effective address is formed by adding  $R_0$  to  $R_n$ . Two consecutive 32-bit values are read from the effective address and loaded into  $XD_n$ .

This instruction is not considered an arithmetic operation, and it does not signal invalid operations. There are no special floating-point cases for this instruction.

## Operation

**FMOV @(R0, Rm), XDn**



Available only when PR=0 and SZ=1

```

sr ← ZeroExtend32(SR);
r0 ← SignExtend32(R0);
op1 ← SignExtend32(Rm);
IF (FpulsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpulsDisabled(sr))
    THROW FPUDIS;
address ← ZeroExtend32(r0 + op1);
op2 ← ReadMemoryPair32(address);
XD2n ← FloatRegisterPair32(op2);
  
```

## Exceptions

SLOTFPUDIS, FPUDIS, RADDERR, RTLBMIS, READPROT

## Note

The effective address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.

# FMUL DR<sub>m</sub>, DR<sub>n</sub>

## Description

This floating-point instruction performs a double-precision floating-point multiplication. It multiplies DR<sub>m</sub> by DR<sub>n</sub> and places the result in DR<sub>n</sub>. The rounding mode is determined by FPSCR.RM.

## Operation

### FMUL DR<sub>m</sub>, DR<sub>n</sub>

1111				n		0	m		00010							
15				12		11	9		8	7		5		4	0	

Available only when PR=1 and SZ=0

```

sr ← ZeroExtend32(SR);
fps ← ZeroExtend32(FPSCR);
op1 ← FloatValue64(DR2m);
op2 ← FloatValue64(DR2n);
IF (FpulsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpulsDisabled(sr))
    THROW FPUDIS;
op2, fps ← FMUL_D(op1, op2, fps);
IF (FpuEnableV(fps) AND FpuCauseV(fps))
    THROW FPUEXC, fps;
IF (FpuCauseE(fps))
    THROW FPUEXC, fps;
IF ((FpuEnableI(fps) OR FpuEnableO(fps)) OR FpuEnableU(fps))
    THROW FPUEXC, fps;
DR2n ← FloatRegister64(op2);
FPSCR ← ZeroExtend32(fps);

```

## Exceptions

SLOTFPUDIS, FPUDIS, FPUEXC

# FMUL FR<sub>m</sub>, FR<sub>n</sub>

## Description

This floating-point instruction performs a single-precision floating-point multiplication. It multiplies FR<sub>m</sub> by FR<sub>n</sub> and places the result in FR<sub>n</sub>. The rounding mode is determined by FPSCR.RM.

## Operation

### FMUL FR<sub>m</sub>, FR<sub>n</sub>

1111	n	m	0010
15	12	11	8
		7	4
		3	0

Available only when PR=0

```

sr ← ZeroExtend32(SR);
fps ← ZeroExtend32(FPSCR);
op1 ← FloatValue32(FRm);
op2 ← FloatValue32(FRn);
IF (FpulsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpulsDisabled(sr))
    THROW FPUDIS;
op2, fps ← FMUL_S(op1, op2, fps);
IF (FpuEnableV(fps) AND FpuCauseV(fps))
    THROW FPUEXC, fps;
IF (FpuCauseE(fps))
    THROW FPUEXC, fps;
IF ((FpuEnableI(fps) OR FpuEnableO(fps)) OR FpuEnableU(fps))
    THROW FPUEXC, fps;
FRn ← FloatRegister32(op2);
FPSCR ← ZeroExtend32(fps);

```

## Exceptions

SLOTFPUDIS, FPUDIS, FPUEXC

**FMUL Special Cases:**

When FPSCR.DN is 1, a positive denormalized number is treated as +0 and a negative denormalized number as -0. This flush-to-zero treatment is applied before exception detection and special case handling.

Exceptional conditions are checked in the order given below. Execution of the instruction is terminated once any check detects an exceptional condition.

- 1 Disabled: an exception is raised if the FPU is disabled.
- 2 Invalid: an invalid operation is signaled if either input is a signaling NaN, or if this is a multiplication of a zero by an infinity.
- 3 Error: an FPU error is signaled if FPSCR.DN is zero, neither input is a NaN and either input is a denormalized number.
- 4 Inexact, underflow and overflow: these are checked together and can be signaled in combination. When inexact, underflow or overflow exceptions are requested by the user, an exception is always raised regardless of whether that condition arose.

If the instruction does not raise an exception, a result is generated according to the following table.

op1 → ↓ op2	+NORM, -NORM	+0	-0	+INF	-INF	+DNORM, -DNORM	qNaN	sNaN
+, -NORM	MUL	+0, -0	-0, +0	+INF, -INF	-INF, +INF	n/a	qNaN	qNaN
+0	+0, -0	+0	-0	qNaN	qNaN	n/a	qNaN	qNaN
-0	-0, +0	-0	+0	qNaN	qNaN	n/a	qNaN	qNaN
+INF	+INF, -INF	qNaN	qNaN	+INF	-INF	n/a	qNaN	qNaN
-INF	-INF, +INF	qNaN	qNaN	-INF	+INF	n/a	qNaN	qNaN
+, -DNORM	n/a	n/a	n/a	n/a	n/a	n/a	qNaN	qNaN
qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN
sNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN

FPU error is indicated by heavy shading and always raises an exception. Invalid operations are indicated by light shading and raise an exception if enabled. FPU disabled, inexact, underflow and overflow cases are not shown.

The behavior of the normal 'MUL' case is described by the IEEE754 specification.

# FNEG DR<sub>n</sub>

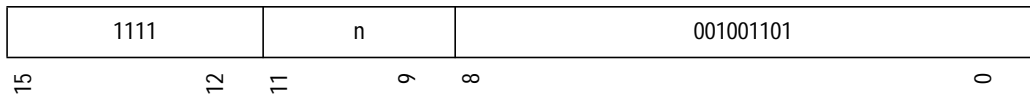
## Description

This floating-point instruction computes the negated value of a double-precision floating-point number. It reads DR<sub>n</sub>, inverts the sign bit and places the result in DR<sub>n</sub>.

This instruction is not considered an arithmetic operation, and it does not signal invalid operations. There are no special floating-point cases for this instruction.

## Operation

### FNEG DR<sub>n</sub>



Available only when PR=1 and SZ=0

```

sr ← ZeroExtend32(SR);
op1 ← FloatValue64(DR2n);
IF (FpulsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpulsDisabled(sr))
    THROW FPUDIS;
op1 ← FNEG_D(op1);
DR2n ← FloatRegister64(op1);

```

## Exceptions

SLOTFPUDIS, FPUDIS

# FNEG FRn

## Description

This floating-point instruction computes the negated value of a single-precision floating-point number. It  $FR_n$ , inverts the sign bit and places the result in  $FR_n$ .

This instruction is not considered an arithmetic operation, and it does not signal invalid operations. There are no special floating-point cases for this instruction.

## Operation

### FNEG FRn

15	12	11	8	7	0
1111		n		01001101	

Available only when PR=0

```

sr ← ZeroExtend32(SR);
fps ← ZeroExtend32(FPSCR);
op1 ← FloatValue32(FRn);
IF (FpulsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpulsDisabled(sr))
    THROW FPUDIS;
op1 ← FNEG_S(op1);
FRn ← FloatRegister32(op1);

```

## Exceptions

SLOTFPUDIS, FPUDIS



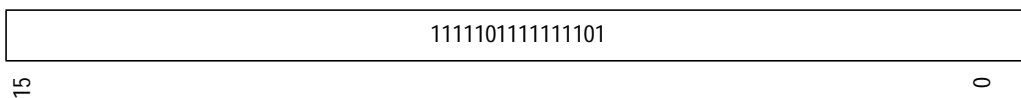
# FRCHG

## Description

This floating-point instruction toggles the FPSCR.FR bit. This has the effect of switching the basic and extended banks of the floating-point register file.

## Operation

### FRCHG



Available only when PR=0

```

sr ← ZeroExtend32(SR);
fr ← ZeroExtend1(SR.FR);
IF (FpulsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpulsDisabled(sr))
    THROW FPUDIS;
fr ← fr ⊕ 1;
SR.FR ← Bit(fr);

```

## Exceptions

SLOTFPUDIS, FPUDIS

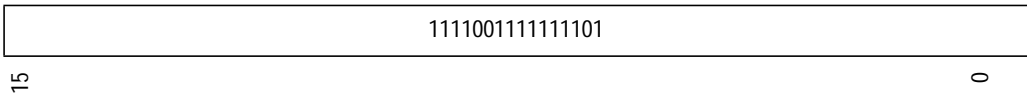
# FSCHG

## Description

This floating-point instruction toggles the FPSCR.SZ bit. This has the effect of changing the size of the data transfer for subsequent floating-point loads, stores and moves. Two transfer sizes are available: FPSCR.SZ = 0 indicates 32-bit transfer and FPSCR.SZ = 1 indicates 64-bit transfer.

## Operation

### FSCHG



Available only when PR=0

```

sr ← ZeroExtend32(SR);
sz ← ZeroExtend1(SR.SZ);
IF (FpulsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpulsDisabled(sr))
    THROW FPUDIS;
sz ← sz ⊕ 1;
SR.SZ ← Bit(sz);

```

## Exceptions

SLOTFPUDIS, FPUDIS

# FSQRT DR<sub>n</sub>

## Description

This floating-point instruction performs a double-precision floating-point square root. It extracts the square root of DR<sub>n</sub> and places the result in DR<sub>n</sub>. The rounding mode is determined by FPSCR.RM.

## Operation

### FSQRT DR<sub>n</sub>

1111	n	001101101
15	12	0

Available only when PR=1 and SZ=0

```

sr ← ZeroExtend32(SR);
fps ← ZeroExtend32(FPSCR);
op1 ← FloatValue64(DR2n);
IF (FpulsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpulsDisabled(sr))
    THROW FPUDIS;
op1, fps ← FSQRT_D(op1, fps);
IF (FpuEnableV(fps) AND FpuCauseV(fps))
    THROW FPUEXC, fps;
IF (FpuCauseE(fps))
    THROW FPUEXC, fps;
IF (FpuEnableI(fps))
    THROW FPUEXC, fps;
DR2n ← FloatRegister64(op1);
FPSCR ← ZeroExtend32(fps);

```

## Exceptions

SLOTFPUDIS, FPUDIS, FPUEXC

# FSQRT FR<sub>n</sub>

## Description

This floating-point instruction performs a single-precision floating-point square root. It extracts the square root of FR<sub>n</sub> and places the result in FR<sub>n</sub>. The rounding mode is determined by FPSCR.RM.

## Operation

### FSQRT FR<sub>n</sub>

1111	n	01101101
15	12	0

Available only when PR=0

```

sr ← ZeroExtend32(SR);
fps ← ZeroExtend32(FPSCR);
op1 ← FloatValue32(FRn);
IF (FpulsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpulsDisabled(sr))
    THROW FPUDIS;
op1, fps ← FSQRT_S(op1, fps);
IF (FpuEnableV(fps) AND FpuCauseV(fps))
    THROW FPUEXC, fps;
IF (FpuCauseE(fps))
    THROW FPUEXC, fps;
IF (FpuEnableI(fps))
    THROW FPUEXC, fps;
FRn ← FloatRegister32(op1);
FPSCR ← ZeroExtend32(fps);

```

## Exceptions

SLOTFPUDIS, FPUDIS, FPUEXC

**FSQRT Special Cases:**

When FPSCR.DN is 1, a positive denormalized number is treated as +0 and a negative denormalized number as -0. This flush-to-zero treatment is applied before exception detection and special case handling.

Exceptional conditions are checked in the order given below. Execution of the instruction is terminated once any check detects an exceptional condition.

- 1 Disabled: an exception is raised if the FPU is disabled.
- 2 Invalid: an invalid operation is signaled if the input is a signaling NaN, or if this is a square root of a number less than zero (including negative infinity and negative normalized/denormalized numbers, but excluding negative zero).
- 3 Error: an FPU error is signaled if FPSCR.DN is zero and the input is a positive denormalized number.
- 4 Inexact: only inexact is checked. When inexact exceptions are requested by the user, an exception is always raised regardless of whether that condition arose. Overflow and underflow do not occur.

If the instruction does not raise an exception, a result is generated according to the following table.

op1 →	+NORM	-NORM	+0	-0	+INF	-INF	+DNORM	-DNORM	qNaN	sNaN
	SQRT	qNaN	+0	-0	+INF	qNaN	n/a	qNaN	qNaN	qNaN

FPU error is indicated by heavy shading and always raises an exception. Invalid operations are indicated by light shading and raise an exception if enabled. FPU disabled and inexact cases are not shown.

The behavior of the normal 'SQRT' case is described by the IEEE754 specification.

# FSTS FPUL, FR<sub>n</sub>

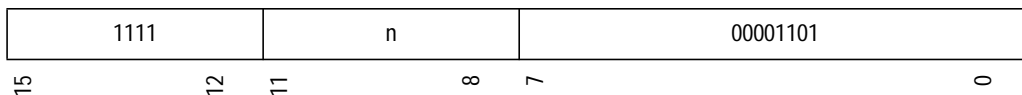
## Description

This floating-point instruction copies FPUL to FR<sub>n</sub>.

This instruction is not considered an arithmetic operation, and it does not signal invalid operations.

## Operation

### FSTS FPUL, FR<sub>n</sub>



```

sr ← ZeroExtend32(SR);
fpul ← SignExtend32(FPUL);
IF (FpulsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpulsDisabled(sr))
    THROW FPUDIS;
op1 ← fpul;
FRn ← FloatRegister32(op1);

```

## Exceptions

SLOTFPUDIS, FPUDIS

# FSUB DRm, DRn

## Description

This floating-point instruction performs a double-precision floating-point subtraction. It subtracts  $DR_m$  from  $DR_n$  and places the result in  $DR_n$ . The rounding mode is determined by  $FPSCR.RM$ .

## Operation

### FSUB DRm, DRn

1111	n	0	m	00001				
15	12	11	9	8	7	5	4	0

Available only when PR=1 and SZ=0

```

sr ← ZeroExtend32(SR);
fps ← ZeroExtend32(FPSCR);
op1 ← FloatValue64(DR2m);
op2 ← FloatValue64(DR2n);
IF (FpulsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpulsDisabled(sr))
    THROW FPUDIS;
op2, fps ← FSUB_D(op2, op1, fps);
IF (FpuEnableV(fps) AND FpuCauseV(fps))
    THROW FPUEXC, fps;
IF (FpuCauseE(fps))
    THROW FPUEXC, fps;
IF ((FpuEnableI(fps) OR FpuEnableO(fps)) OR FpuEnableU(fps))
    THROW FPUEXC, fps;
DR2n ← FloatRegister64(op2);
FPSCR ← ZeroExtend32(fps);

```

## Exceptions

SLOTFPUDIS, FPUDIS, FPUEXC

# FSUB FR<sub>m</sub>, FR<sub>n</sub>

## Description

This floating-point instruction performs a single-precision floating-point subtraction. It subtracts FR<sub>m</sub> from FR<sub>n</sub> and places the result in FR<sub>n</sub>. The rounding mode is determined by FPSCR.RM.

## Operation

**FSUB FR<sub>m</sub>, FR<sub>n</sub>**

1111	n	m	0001
15	12	11	8
		7	4
		3	0

Available only when PR=0

```

sr ← ZeroExtend32(SR);
fps ← ZeroExtend32(FPSCR);
op1 ← FloatValue32(FRm);
op2 ← FloatValue32(FRn);
IF (FpulsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpulsDisabled(sr))
    THROW FPUDIS;
op2, fps ← FSUB_S(op2, op1, fps);
IF (FpuEnableV(fps) AND FpuCauseV(fps))
    THROW FPUEXC, fps;
IF (FpuCauseE(fps))
    THROW FPUEXC, fps;
IF ((FpuEnableI(fps) OR FpuEnableO(fps)) OR FpuEnableU(fps))
    THROW FPUEXC, fps;
FRn ← FloatRegister32(op2);
FPSCR ← ZeroExtend32(fps);

```

## Exceptions

SLOTFPUDIS, FPUDIS, FPUEXC



**FSUB Special Cases:**

When FPSCR.DN is 1, a positive denormalized number is treated as +0 and a negative denormalized number as -0. This flush-to-zero treatment is applied before exception detection and special case handling.

Exceptional conditions are checked in the order given below. Execution of the instruction is terminated once any check detects an exceptional condition.

- 1 Disabled: an exception is raised if the FPU is disabled.
- 2 Invalid: an invalid operation is signaled if either input is a signaling NaN, or if the inputs are similarly signed infinities.
- 3 Error: an FPU error is signaled if FPSCR.DN is zero, neither input is a NaN and either input is a denormalized number.
- 4 Inexact, underflow and overflow: these are checked together and can be signaled in combination. When inexact, underflow or overflow exceptions are requested by the user, an exception is always raised regardless of whether that condition arose.

If the instruction does not raise an exception, a result is generated according to the following table.

op2 → ↓ op1	+NORM, -NORM	+0	-0	+INF	-INF	+DNORM, -DNORM	qNaN	sNaN
+, -NORM	SUB	SUB	SUB	+INF	-INF	n/a	qNaN	qNaN
+0	op2	+0	-0	+INF	-INF	n/a	qNaN	qNaN
-0	op2	+0	+0	+INF	-INF	n/a	qNaN	qNaN
+INF	-INF	-INF	-INF	qNaN	-INF	n/a	qNaN	qNaN
-INF	+INF	+INF	+INF	+INF	qNaN	n/a	qNaN	qNaN
+, -DNORM	n/a	n/a	n/a	n/a	n/a	n/a	qNaN	qNaN
qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN
sNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN

FPU error is indicated by heavy shading and always raises an exception. Invalid operations are indicated by light shading and raise an exception if enabled. FPU disabled, inexact, underflow and overflow cases are not shown.

The behavior of the normal 'SUB' case is described by the IEEE754 specification.

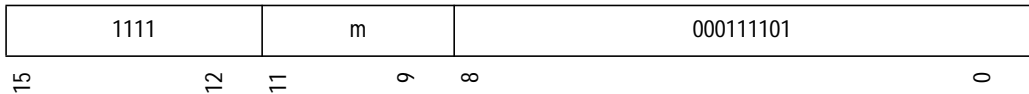
# FTRC DR<sub>m</sub>, FPUL

## Description

This floating-point instruction performs a double-precision floating-point to signed 32-bit integer conversion. It reads a double-precision value from DR<sub>m</sub>, converts it to a signed 32-bit integral range and places the result in FPUL. The conversion is achieved by rounding to zero (truncation) with saturation to the limits of the target signed integral range. The value of FPSCR.RM is ignored.

## Operation

### FTRC DR<sub>m</sub>, FPUL



Available only when PR=1 and SZ=0

```

sr ← ZeroExtend32(SR);
fps ← ZeroExtend32(FPSCR);
op1 ← FloatValue64(DR2m);
IF (FpulsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpulsDisabled(sr))
    THROW FPUDIS;
fpul, fps ← FTRC_DL(op1, fps);
IF (FpuEnableV(fps) AND FpuCauseV(fps))
    THROW FPUEXC, fps;
FPUL ← ZeroExtend32(fpul);
FPSCR ← ZeroExtend32(fps);
  
```

## Exceptions

SLOTFPUDIS, FPUDIS, FPUEXC

# FTRC FR<sub>m</sub>, FPUL

## Description

This floating-point instruction performs a single-precision floating-point to signed 32-bit integer conversion. It reads a single-precision value from FR<sub>m</sub>, converts it to a signed 32-bit integral range and places the result in FPUL. The conversion is achieved by rounding to zero (truncation) with saturation to the limits of the target signed integral range. The value of FPSCR.RM is ignored.

## Operation

### FTRC FR<sub>m</sub>, FPUL

1111	m	00111101
15	12	0

Available only when PR=0

```

sr ← ZeroExtend32(SR);
fps ← ZeroExtend32(FPSCR);
op1 ← FloatValue32(FRm);
IF (FpulsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpulsDisabled(sr))
    THROW FPUDIS;
fpul, fps ← FTRC_SL(op1, fps);
IF (FpuEnableV(fps) AND FpuCauseV(fps))
    THROW FPUExc, fps;
FPSCR ← ZeroExtend32(fps);
FPUL ← ZeroExtend32(fpul);

```

## Exceptions

SLOTFPUDIS, FPUDIS, FPUExc

### FTRC Special Cases:

Regardless of FPSCR.DN, denormalized numbers are treated as 0. These instructions do not cause FPU Error.

Exceptional conditions are checked in the order given below. Execution of the instruction is terminated once any check detects an exceptional condition.

- 1 Disabled: an exception is raised if the FPU is disabled.
- 2 Invalid: an invalid operation is signaled if the conversion overflows the target range. This is caused by out-of-range normalized numbers, infinities and NaNs.

If the instruction does not raise an exception, a result is generated according to the following table.

op1 →	+NORM (in range)	-NORM (in range)	+0	-0	+INF or +NORM (out of range)	-INF or -NORM (out of range)	+DNORM, -DNORM	qNaN	sNaN
	TRC	TRC	0	0	$+2^{31} - 1$	$-2^{31}$	0	$-2^{31}$	$-2^{31}$

Invalid operations are indicated by light shading and raise an exception if enabled. FPU disabled cases are not shown.

The behavior of the normal 'TRC' case is described by the IEEE754 specification, though only the round to zero rounding mode is supported by this instruction.

# FTRV XMTRX, FVn

## Description

This floating-point instruction multiplies the matrix, XMTRX, with a vector, FV<sub>n</sub>, and places the resulting vector in FV<sub>n</sub>. The matrix contains sixteen single-precision floating-point values. The vector contains four single-precision floating-point values. The matrix-vector multiplication is specified as:

$$FR_n = \sum_{i=0}^3 XF_{i \times 4} \times FR_{n+i}$$

$$FR_{n+1} = \sum_{i=0}^3 XF_{1+i \times 4} \times FR_{n+i}$$

$$FR_{n+2} = \sum_{i=0}^3 XF_{2+i \times 4} \times FR_{n+i}$$

$$FR_{n+3} = \sum_{i=0}^3 XF_{3+i \times 4} \times FR_{n+i}$$

This is an approximate computation. The specified error in the  $p^{th}$  element value of the result vector:

$$\text{spec\_error}_p = \begin{cases} 0 & \text{if}(epm = ez) \\ 2^{epm-24} + 2^{E-24+rm} & \text{if}(epm \neq ez) \end{cases}$$

where

$$rm = \begin{cases} 0 & \text{if}(\text{round-to-nearest}) \\ 1 & \text{if}(\text{round-to-zero}) \end{cases}$$

E = unbiased exponent value of the result

ez < -252

epm = max (ep<sub>0</sub>, ep<sub>1</sub>, ep<sub>2</sub>, ep<sub>3</sub>)

ep<sub>i</sub> = pre-normalized exponent of the product XF<sub>p+ix4</sub> and FR<sub>n+i</sub>

$eX_{F_{p+i \times 4}} = \text{biased exponent value of } X_{F_{p+i \times 4}}$

$eF_{R_{n+i}} = \text{biased exponent value of } F_{R_{n+i}}$

$$eP_i = \begin{cases} ez & \text{if } ((X_{F_{p+i \times 4}} = 0.0) \text{OR} (F_{R_{n+i}} = 0.0)) \\ \max(eX_{F_{p+i \times 4}}, 1) + \max(eF_{R_{n+i}}, 1) - 254 & \text{otherwise} \end{cases}$$

## Operation

### FTRV XMTRX, FVn

1111	n	0111111101
15	12 11 10 9	0

Available only when PR=0

```

sr ← ZeroExtend32(SR);
fps ← ZeroExtend32(FPSCR);
xmtrx ← FloatValueMatrix32(XMTRX);
op1 ← FloatValueVector32(FV4n);
IF (FpulsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpulsDisabled(sr))
    THROW FPUDIS;
op1, fps ← FTRV_S(xmtrx, op1, fps);
IF (((FpuEnableV(fps) OR FpuEnableI(fps)) OR FpuEnableO(fps)) OR FpuEnableU(fps))
    THROW FPUExc, fps;
FV4n ← FloatRegisterVector32(op1);
FPSCR ← ZeroExtend32(fps);

```

## Exceptions

SLOTFPUDIS, FPUDIS, FPUExc

### FTRV Special Cases:

FTRV is an approximate instruction. Denormalized numbers are supported:

- When FPSCR.DN is 0, denormalized numbers are treated as their denormalized value in the FTRV.S calculation. This instruction never signals an FPU error.

- When FPSCR.DN is 1, a positive denormalized number is treated as +0 and a negative denormalized number as -0. This flush-to-zero treatment is applied before exception detection and special case handling.

Exceptional conditions are checked in the order given below. Execution of the instruction is terminated once any check detects an exceptional condition.

- 1 Disabled: an exception is raised if the FPU is disabled.
- 2 Invalid: an invalid operation is signaled if any of the inputs is a signaling NaN, there is a multiplication of a zero by an infinity, or there is an addition of differently signed infinities where none of the inputs is a qNaN.

The multiplication is performed with sufficient precision to avoid overflow, and therefore the multiplication of any two finite numbers does not produce an infinity. The multiplication result will be an infinity only if there is a multiplication of an infinity with a normalized number, an infinity with a denormalized number or an infinity with an infinity.

The addition of differently signed infinities is detected if there is (at least) one positive infinity and (at least) one negative infinity in the set of 4 multiplication results in any of the 4 inner-products calculated by this instruction.

This instruction is not capable of checking its inputs for invalid operations and raising an invalid operation exception accordingly. Instead, this instruction always raises an invalid operation exception if this exception is requested by the user. If this exception is not requested by the user, then qNaN results are correctly produced for invalid operations as described above.

- 3 Inexact, underflow and overflow: these are checked together and can be signaled in combination. This is an approximate instruction and inexact is signaled except where special cases occur. Precise details of the approximate inner-product algorithm, including the detection of underflow and overflow cases, are implementation dependent. When inexact, underflow or overflow exceptions are requested by the user, an exception is always raised regardless of whether that condition arose.

If the instruction does not raise an exception, results are generated according to the following tables. The special case tables are applied separately with the appropriate vector operands to each of the four inner-products calculated by this instruction.

**FTRV Special Cases (continued):**

Each of the 4 pairs of multiplication operands (op1 and op2) is selected from corresponding elements of the two 4-element source vectors and multiplied:

op1 → ↓ op2		+, -NORM, +, -DENORM	+0	-0	+INF	-INF	qNaN	sNaN
+, -NORM    +, -DENORM		FTRVMUL	+0, -0	-0, +0	+INF, -INF	-INF, +INF	qNaN	qNaN
+0		+0, -0	+0	-0	qNaN	qNaN	qNaN	qNaN
-0		-0, +0	-0	+0	qNaN	qNaN	qNaN	qNaN
+INF		+INF, -INF	qNaN	qNaN	+INF	-INF	qNaN	qNaN
-INF		-INF, +INF	qNaN	qNaN	-INF	+INF	qNaN	qNaN
qNaN		qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN
sNaN		qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN

If any of the multiplications evaluates to qNaN, then the result of the instruction is qNaN and no further analysis need be performed. In the 'FTRVMUL', +0, -0, +INF and -INF cases, the 4 addition operands (labelled intermediate 0 to 3) are summed:

intermediate 0 → ↓ intermediate 2		FTRVMUL, +0, -0			+INF			-INF		
intermediate 1 → ↓ intermediate 3		FTRVMUL, +0, -0	+INF	-INF	FTRVMUL, +0, -0	+INF	-INF	FTRVMUL, +0, -0	+INF	-INF
FTRVMUL, +0, -0	FTRVMUL, +0, -0	FTRVADD	+INF	-INF	+INF	+INF	qNaN	-INF	qNaN	-INF
	+INF	+INF	+INF	qNaN	+INF	+INF	qNaN	qNaN	qNaN	qNaN
	-INF	-INF	qNaN	-INF	qNaN	qNaN	qNaN	-INF	qNaN	-INF
+INF	FTRVMUL, +0, -0	+INF	+INF	qNaN	+INF	+INF	qNaN	qNaN	qNaN	qNaN
	+INF	+INF	+INF	qNaN	+INF	+INF	qNaN	qNaN	qNaN	qNaN
	-INF	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN
-INF	FTRVMUL, +0, -0	-INF	qNaN	-INF	qNaN	qNaN	qNaN	-INF	qNaN	-INF
	+INF	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN
	-INF	-INF	qNaN	-INF	qNaN	qNaN	qNaN	-INF	qNaN	-INF

Inexact is signaled in the 'FTRVADD' case. Exception cases are not indicated by shading for this instruction. Where the behavior is not a special case, the instruction computes an approximate result using an implementation-dependent algorithm.



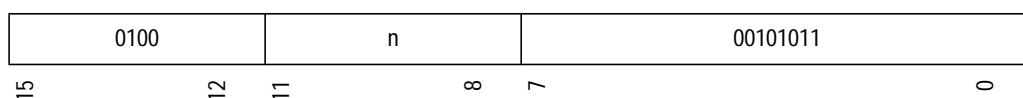
# JMP @Rn

## Description

This instruction is a delayed unconditional branch used for jumping to the target address specified in  $R_n$ .

## Operation

### JMP @Rn



```

op1 ← SignExtend32(Rn);
IF (IsDelaySlot())
    THROW ILLSLOT;
target ← op1;
delayedpc ← target ∧ (~ 0x1);
PC" ← Register(delayedpc);

```

## Exceptions

### ILLSLOT

### Note

The delay slot is executed before branching. An ILLSLOT exception is raised if this instruction is executed in a delay slot.

If the branch target address is invalid then IADDERR trap is not delivered until after the instruction in the delay slot has executed and the PC has advanced to the target address, that is the exception is associated with the target instruction not the branch.

# JSR @Rn

## Description

This instruction is a delayed unconditional branch used for jumping to the subroutine starting at the target address specified in  $R_n$ . The address of the instruction immediately following the delay slot is copied to PR to indicate the return address.

## Operation

### JSR @Rn

0100	n	00001011
15	12	11
	8	7
		0

```

pc ← SignExtend32(PC);
op1 ← SignExtend32(Rn);
IF (IsDelaySlot())
    THROW ILLSLOT;
delayedpr ← pc + 4;
target ← op1;
delayedpc ← target ∧ (~ 0x1);
PR" ← Register(delayedpr);
PC" ← Register(delayedpc);

```

## Exceptions

ILLSLOT

## Note

The delay slot is executed before branching and before PR is updated. An ILLSLOT exception is raised if this instruction is executed in a delay slot.

If the branch target address is invalid then IADDERR trap is not delivered until after the instruction in the delay slot has executed and the PC has advanced to the target address, that is the exception is associated with the target instruction not the branch.

# LDC R<sub>m</sub>, GBR

## Description

This instruction copies R<sub>m</sub> to GBR.

## Operation

### LDC R<sub>m</sub>, GBR

0100				m		00011110			
15	12	11		8	7				0

```

op1 ← SignExtend32(Rm);
gbr ← op1;
GBR ← Register(gbr);

```

## Note

# LDC Rm, SR

## Description

This instruction copies  $R_m$  to SR, it is a privileged instruction.

## Operation

### LDC Rm, SR

0100	m	00001110
15	12	11
	8	7
		0

```

md ← ZeroExtend1(MD);
IF (md = 0)
    THROW RESINST;
op1 ← SignExtend32(Rm);
sr ← op1;
SR ← Register(sr);

```

## Exceptions

RESINST

## Note

# LDC Rm, VBR

## Description

This instruction copies  $R_m$  to VBR, it is a privileged instruction.

## Operation

### LDC Rm, VBR

0100				m				00101110															
15				12				11				8				7				0			

```
md ← ZeroExtend1(MD);
IF (md = 0)
  THROW RESINST;
op1 ← SignExtend32(Rm);
vbr ← op1;
VBR ← Register(vbr);
```

## Exceptions

RESINST

## Note

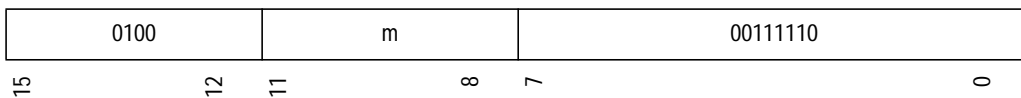
# LDC Rm, SSR

## Description

This instruction copies  $R_m$  to SSR, it is a privileged instruction.

## Operation

### LDC Rm, SSR



```

md ← ZeroExtend1(MD);
IF (md = 0)
    THROW RESINST;
op1 ← SignExtend32(Rm);
ssr ← op1;
SSR ← Register(ssr);

```

## Exceptions

RESINST

## Note

# LDC Rm, SPC

## Description

This instruction copies  $R_m$  to SPC, it is a privileged instruction.

## Operation

### LDC Rm, SPC

0100	m	01001110
15	12	11
	8	7
		0

```

md ← ZeroExtend1(MD);
IF (md = 0)
    THROW RESINST;
op1 ← SignExtend32(Rm);
spc ← op1;
SPC ← Register(spc);

```

## Exceptions

RESINST

## Note

# LDC Rm, DBR

## Description

This instruction copies  $R_m$  to DBR, it is a privileged instruction.

## Operation

### LDC Rm, SPC

0100	m	11111010
15	12	11
		8
		7
		0

```

md ← ZeroExtend1(MD);
IF (md = 0)
    THROW RESINST;
op1 ← SignExtend32(Rm);
dbr ← op1;
DBR ← Register(dbr);

```

## Exceptions

RESINST

## Note



# LDC Rm, Rn\_BANK

## Description

This instruction copies  $R_m$  to  $Rn\_BANK$ , it is a privileged instruction.

## Operation

### LDC Rm, Rn\_BANK

0100			m			1	n		1110		
15	12		11	8		7	6	4		3	0

```

md ← ZeroExtend1(MD);
IF (md = 0)
    THROW RESINST;
op1 ← SignExtend32(Rm);
rn_bank ← op1;
Rn_BANK ← Register(rn_bank);

```

## Exceptions

RESINST

## Note

# LDC.L @Rm+, GBR

## Description

This instruction loads GBR from memory using register indirect with post-increment addressing. A 32-bit value is read from the effective address specified in  $R_m$  and loaded into GBR.  $R_m$  is post-incremented by 4.

## Operation

### LDC.L @Rm+, GBR

0100	m	00010111
15	12	11
	8	7
		0

```

op1 ← SignExtend32(Rm);
address ← ZeroExtend32(op1);
gbr ← SignExtend32(ReadMemory32(address));
op1 ← op1 + 4;
Rm ← Register(op1);
GBR ← Register(gbr);

```

## Exceptions

RADDERR, RTLBMIS, READPROT

## Note

# LDC.L @Rm+, SR

## Description

This instruction loads SR from memory using register indirect with post-increment addressing. A 32-bit value is read from the effective address specified in  $R_m$  and loaded into SR.  $R_m$  is post-incremented by 4. This is a privileged instruction.

## Operation

### LDC.L @Rm+, SR

0100	m	00000111
15	12	11
	8	7
		0

```

md ← ZeroExtend1(MD);
IF (md = 0)
  THROW RESINST;
op1 ← SignExtend32(Rm);
address ← ZeroExtend32(op1);
sr ← SignExtend32(ReadMemory32(address));
op1 ← op1 + 4;
Rm ← Register(op1);
SR ← Register(sr);

```

## Exceptions

RESINST, RADDERR, RTLBMIS, READPROT

## Note

# LDC.L @Rm+, VBR

## Description

This instruction loads VBR from memory using register indirect with post-increment addressing. A 32-bit value is read from the effective address specified in  $R_m$  and loaded into VBR.  $R_m$  is post-incremented by 4. This is a privileged instruction.

## Operation

### LDC.L @Rm+, VBR

0100	m	00100111
15	12	11
	8	7
		0

```

md ← ZeroExtend1(MD);
IF (md = 0)
    THROW RESINST;
op1 ← SignExtend32(Rm);
address ← ZeroExtend32(op1);
vbr ← SignExtend32(ReadMemory32(address));
op1 ← op1 + 4;
Rm ← Register(op1);
VBR ← Register(vbr);

```

## Exceptions

RESINST, RADDERR, RTLBMIS, READPROT

## Note

# LDC.L @Rm+, SSR

## Description

This instruction loads SSR from memory using register indirect with post-increment addressing. A 32-bit value is read from the effective address specified in  $R_m$  and loaded into SSR.  $R_m$  is post-incremented by 4. This is a privileged instruction.

## Operation

### LDC.L @Rm+, SR

0100	m	00110111
15	12	11
	8	7
		0

```

md ← ZeroExtend1(MD);
IF (md = 0)
  THROW RESINST;
op1 ← SignExtend32(Rm);
address ← ZeroExtend32(op1);
ssr ← SignExtend32(ReadMemory32(address));
op1 ← op1 + 4;
Rm ← Register(op1);
SSR ← Register(ssr);

```

## Exceptions

RESINST, RADDERR, RTLBMIS, READPROT

## Note

# LDC.L @Rm+, SPC

## Description

This instruction loads SPC from memory using register indirect with post-increment addressing. A 32-bit value is read from the effective address specified in  $R_m$  and loaded into SPC.  $R_m$  is post-incremented by 4. This is a privileged instruction.

## Operation

### LDC.L @Rm+, SPC

0100	m	01000111
15	12	11
	8	7
		0

```

md ← ZeroExtend1(MD);
IF (md = 0)
  THROW RESINST;
op1 ← SignExtend32(Rm);
address ← ZeroExtend32(op1);
spc ← SignExtend32(ReadMemory32(address));
op1 ← op1 + 4;
Rm ← Register(op1);
SPC ← Register(spc);

```

## Exceptions

RESINST, RADDERR, RTLBMIS, READPROT

## Note

# LDC.L @Rm+, DBR

## Description

This instruction loads SR from memory using register indirect with post-increment addressing. A 32-bit value is read from the effective address specified in  $R_m$  and loaded into DBR.  $R_m$  is post-incremented by 4. This is a privileged instruction.

## Operation

### LDC.L @Rm+, DBR

0100	m	11110110
15	12	11
	8	7
		0

```

md ← ZeroExtend1(MD);
IF (md = 0)
    THROW RESINST;
op1 ← SignExtend32(Rm);
address ← ZeroExtend32(op1);
dbr ← SignExtend32(ReadMemory32(address));
op1 ← op1 + 4;
Rm ← Register(op1);
DBR ← Register(dbr);

```

## Exceptions

RESINST, RADDERR, RTLBMIS, READPROT

## Note

# LDC.L @Rm+, Rn\_BANK

## Description

This instruction loads Rn\_BANK from memory using register indirect with post-increment addressing. A 32-bit value is read from the effective address specified in R<sub>m</sub> and loaded into Rn\_BANK. R<sub>m</sub> is post-incremented by 4. This is a privileged instruction.

## Operation

**LDC.L @Rm+, Rn\_BANK**

0100	m	1	n	0111
15	12	11	8	7
				0

```

md ← ZeroExtend1(MD);
IF (md = 0)
  THROW RESINST;
op1 ← SignExtend32(Rm);
address ← ZeroExtend32(op1);
rn_bank ← SignExtend32(ReadMemory32(address));
op1 ← op1 + 4;
Rm ← Register(op1);
Rn_BANK ← Register(rn_bank);

```

## Exceptions

RESINST, RADDERR, RTLBMIS, READPROT

## Note



# LDS R<sub>m</sub>, FPSCR

## Description

This floating-point instruction copies R<sub>m</sub> to FPSCR. The setting of FPSCR does not cause any floating-point exceptional conditions to be signaled.

## Operation

### LDS R<sub>m</sub>, FPSCR

0100	m	01101010
15	12	11
	8	7
		0

```

sr ← ZeroExtend32(SR);
op1 ← SignExtend32(Rm);
IF (FpulsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpulsDisabled(sr))
    THROW FPUDIS;
fps, pr, sz, fr ← UnpackFPSCR(op1);
FPSCR ← ZeroExtend32(fps);
SR.PR ← Bit(pr);
SR.SZ ← Bit(sz);
SR.FR ← Bit(fr);

```

## Exceptions

SLOTFPUDIS, FPUDIS

## Note

# LDS.L @Rm+, FPSCR

## Description

This floating-point instruction loads FPSCR from memory using register indirect with post-increment addressing. A 32-bit value is read from the effective address specified in  $R_m$  and loaded into FPSCR.  $R_m$  is post-incremented by 4. The setting of FPSCR does not cause any floating-point exceptional conditions to be signaled.

## Operation

### LDS.L @Rm+, FPSCR

0100	m	01100110
15	12	0

```

sr ← ZeroExtend32(SR);
op1 ← SignExtend32(Rm);
IF (FpulsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpulsDisabled(sr))
    THROW FPUDIS;
address ← ZeroExtend32(op1);
value ← ReadMemory32(address);
fps, pr, sz, fr ← UnpackFPSCR(value);
op1 ← op1 + 4;
Rm ← Register(op1);
FPSCR ← ZeroExtend32(fps);
SR.PR ← Bit(pr);
SR.SZ ← Bit(sz);
SR.FR ← Bit(fr);

```

## Exceptions

SLOTFPUDIS, FPUDIS, RADDERR, RTLBMIS, READPROT

## Note

# LDS R<sub>m</sub>, FPUL

## Description

This floating-point instruction copies R<sub>m</sub> to FPUL.

## Operation

### LDS R<sub>m</sub>, FPUL



```

sr ← ZeroExtend32(SR);
op1 ← SignExtend32(Rm);
IF (FpulsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpulsDisabled(sr))
    THROW FPUDIS;
fpul ← op1;
FPUL ← ZeroExtend32(fpul);
  
```

## Exceptions

SLOTFPUDIS, FPUDIS

## Note

# LDS.L @Rm+, FPUL

## Description

This floating-point instruction loads FPUL from memory using register indirect with post-increment addressing. A 32-bit value is read from the effective address specified in  $R_m$  and loaded into FPUL.  $R_m$  is post-incremented by 4.

## Operation

### LDS.L @Rm+, FPUL

0100	m	01010110
15	12	0

```

sr ← ZeroExtend32(SR);
op1 ← SignExtend32(Rm);
IF (FpulsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpulsDisabled(sr))
    THROW FPUDIS;
address ← ZeroExtend32(op1);
fpul ← ReadMemory32(address);
op1 ← op1 + 4;
Rm ← Register(op1);
FPUL ← ZeroExtend32(fpul);

```

## Exceptions

SLOTFPUDIS, FPUDIS, RADDERR, RTLBMIS, READPROT

## Note

# LDS R<sub>m</sub>, MACH

## Description

This instruction copies R<sub>m</sub> to MACH.

## Operation

### LDS R<sub>m</sub>, MACH

0100	m	00001010
15	12	11
	8	7
		0

```

op1 ← SignExtend32(Rm);
mach ← op1;
MACH ← ZeroExtend32(mach);

```

## Note

# LDS.L @Rm+, MACH

## Description

This instruction loads MACH from memory using register indirect with post-increment addressing. A 32-bit value is read from the effective address specified in  $R_m$  and loaded into MACH.  $R_m$  is post-incremented by 4.

## Operation

### LDS.L @Rm+, MACH

0100	m	0000110
15	12 11	8 7 0

```

op1 ← SignExtend32(Rm);
address ← ZeroExtend32(op1);
mach ← SignExtend32(ReadMemory32(address));
op1 ← op1 + 4;
Rm ← Register(op1);
MACH ← ZeroExtend32(mach);

```

## Exceptions

RADDERR, RTLBMIS, READPROT

## Note

# LDS R<sub>m</sub>, MACL

## Description

This instruction copies R<sub>m</sub> to MACL.

## Operation

### LDS R<sub>m</sub>, MACL

0100	m	00011010
15	12	11
		8
		7
		0

```

op1 ← SignExtend32(Rm);
mac1 ← op1;
MACL ← ZeroExtend32(mac1);

```

## Note

# LDS.L @Rm+, MACL

## Description

This instruction loads MACL from memory using register indirect with post-increment addressing. A 32-bit value is read from the effective address specified in  $R_m$  and loaded into MACL.  $R_m$  is post-incremented by 4.

## Operation

### LDS.L @Rm+, MACL

0100	m	00010110
15	12	11
	8	7
		0

```

op1 ← SignExtend32(Rm);
address ← ZeroExtend32(op1);
mac1 ← SignExtend32(ReadMemory32(address));
op1 ← op1 + 4;
Rm ← Register(op1);
MACL ← ZeroExtend32(mac1);

```

## Exceptions

RADDERR, RTLBMIS, READPROT

## Note



# LDS R<sub>m</sub>, PR

## Description

This instruction copies R<sub>m</sub> to PR.

## Operation

### LDS R<sub>m</sub>, PR

0100	m	00101010
15	12	11
	8	7
		0

```

op1 ← SignExtend32(Rm);
newpr ← op1;
delayedpr ← newpr;
PR' ← Register(newpr);
PR'' ← Register(delayedpr);

```

## Note

# LDS.L @Rm+, PR

## Description

This instruction loads PR from memory using register indirect with post-increment addressing. A 32-bit value is read from the effective address specified in  $R_m$  and loaded into PR.  $R_m$  is post-incremented by 4.

## Operation

### LDS.L @Rm+, PR

0100	m	00100110
15	12	0

```

op1 ← SignExtend32(Rm);
address ← ZeroExtend32(op1);
newpr ← SignExtend32(ReadMemory32(address));
delayedpr ← newpr;
op1 ← op1 + 4;
Rm ← Register(op1);
PR' ← Register(newpr);
PR'' ← Register(delayedpr);

```

## Exceptions

RADDERR, RTLBMIS, READPROT

## Note

# LDTLB

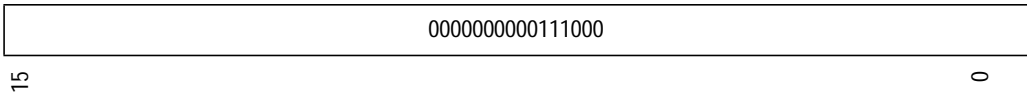
## Description

This instruction loads the contents of the PTEH/PTEL registers into the UTLB (unified translation lookaside buffer) specified by MMUCR.URC (random counter field in the MMC control register).

LDTLB is a privileged instruction, and can only be used in privileged mode. Use of this instruction in user mode will cause a RESINST trap.

## Operation

### LDTLB



```
md ← ZeroExtend1(MD);
IF (md = 0)
  THROW RESINST;
UTLB[MMUCR.URC].ASID ← PTEH.ASID
UTLB[MMUCR.URC].VPN ← PTEH.VPN
UTLB[MMUCR.URC].PPN ← PTEH.PPN
UTLB[MMUCR.URC].SZ ← PTEL.SZ1<<1 + PTEL.SZ0
UTLB[MMUCR.URC].SH ← PTEL.SH
UTLB[MMUCR.URC].PR ← PTEL.PR
UTLB[MMUCR.URC].WT ← PTEL.WT
UTLB[MMUCR.URC].C ← PTEL.C
UTLB[MMUCR.URC].D ← PTEL.D
UTLB[MMUCR.URC].V ← PTEL.V
```

## Exceptions

RESINST

## Note

As this instruction loads the contents of the PTEH/PTEL registers into a UTLB entry, it should be used either with the MMU disabled, or in the P1 or P2 virtual space with the MMU enabled (see [Chapter 3: Memory management unit \(MMU\) on page 41](#), for details). After this instruction is issued, there must be at least one

instruction between the LDTLB instruction and the execution of an instruction from the areas P0, U0, and P3 (i.e. via a BRAF, BSRF, JMP, JSR, RTS, or RTE).

# MAC.L @Rm+, @Rn+

## Description

This instruction reads the signed 32-bit value at the effective address specified in  $R_n$ , and then post-increments  $R_n$  by 4. It also reads the signed 32-bit value at the effective address specified in  $R_m$ , and then post-increments  $R_m$  by 4. These 2 values are multiplied together to give a 64-bit result, and this result is added to the 64-bit accumulator held in MACL and MACH. This accumulation gives an output with 65 bits of precision.

If the S-bit is 0, the result is the lower 64 bits of the accumulation. If the S-bit is 1, the result is calculated by saturating the accumulation to the signed range  $[-2^{48}, 2^{48}]$ . In either case, the 64-bit result is split into low and high halves, which are placed into MACL and MACH respectively.

## Exceptions

RADDERR, RTLBMIS, READPROT

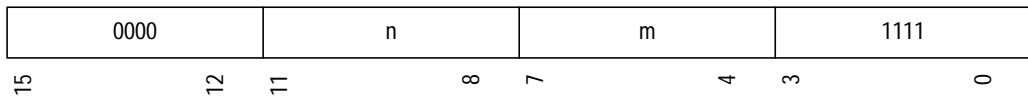
## Note

The effective address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.

If  $R_m$  and  $R_n$  refer to the same register (i.e.  $m = n$ ), then this register will be post-incremented twice. The instruction will read two long-words from consecutive memory locations.

## Operation

MAC.L @Rm+, @Rn+



**MAC.L @Rm+, @Rn+**

```

macl ← ZeroExtend32(MACL);
mach ← ZeroExtend32(MACH);
s ← ZeroExtend1(S);
m_field ← ZeroExtend4(m);
n_field ← ZeroExtend4(n);
m_address ← SignExtend32(Rm);
n_address ← SignExtend32(Rn);
value2 ← SignExtend32(ReadMemory32(ZeroExtend32(n_address)));
n_address ← n_address + 4;
IF (n_field = m_field)
{
    m_address ← m_address + 4;
    n_address ← n_address + 4;
}
value1 ← SignExtend32(ReadMemory32(ZeroExtend32(m_address)));
m_address ← m_address + 4;
mul ← value2 × value1;
mac ← (mach << 32) + macl;
result ← mac + mul;
IF (s = 1)
    IF (((result ⊕ mac) ∧ (result ⊕ mul))<63 FOR 1 > = 1)
        IF (mac<63 FOR 1 > = 0)
            result ← 247 - 1;
        ELSE
            result ← - 247;
    ELSE
        result ← SignedSaturate48(result);
macl ← result;
mach ← result >> 32;
Rm ← Register(m_address);
Rn ← Register(n_address);
MACL ← ZeroExtend32(macl);
MACH ← ZeroExtend32(mach);

```

# MAC.W @Rm+, @Rn+

## Description

This instruction reads the signed 16-bit value at the effective address specified in  $R_n$ , and then post-increments  $R_n$  by 2. It also reads the signed 16-bit value at the effective address specified in  $R_m$ , and then post-increments  $R_m$  by 2. These 2 values are multiplied together to give a 32-bit result.

If the S-bit is 0, the 32-bit multiply result is added to the 64-bit accumulator held in MACL and MACH. This accumulation gives an output with 65 bits of precision, and the result is the lower 64 bits of the accumulation. The result is split into low and high halves, which are placed into MACL and MACH respectively.

If the S-bit is 1, the 32-bit multiply result is added to the 32-bit accumulator held in MACL. This accumulation gives an output with 33 bits of precision, and is saturated to the signed range  $[-2^{31}, 2^{31})$ , and then placed in MACL. If the accumulation overflows this signed range, then MACH is set to 1 to denote overflow otherwise MACH is unchanged.

## Exceptions

RADDERR, RTLBMIS, READPROT

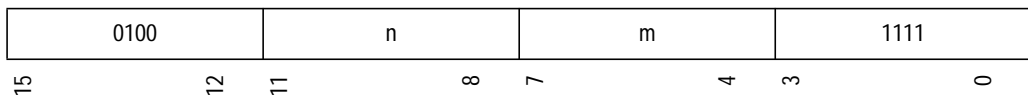
## Note

The effective address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.

If  $R_m$  and  $R_n$  refer to the same register (i.e.  $m = n$ ), then this register will be post-incremented twice. The instruction will read two words from consecutive memory locations.

## Operation

MAC.W @Rm+, @Rn+



**MAC.W @Rm+, @Rn+**

```

macl ← ZeroExtend32(MACL);
mach ← ZeroExtend32(MACH);
s ← ZeroExtend1(S);
m_field ← ZeroExtend4(m);
n_field ← ZeroExtend4(n);
m_address ← SignExtend32(Rm);
n_address ← SignExtend32(Rn);
value2 ← SignExtend16(ReadMemory16(ZeroExtend32(n_address)));
n_address ← n_address + 2;
IF (n_field = m_field)
{
    m_address ← m_address + 2;
    n_address ← n_address + 2;
}
value1 ← SignExtend16(ReadMemory16(ZeroExtend32(m_address)));
m_address ← m_address + 2;
mul ← value2 × value1;
IF (s = 1)
{
    macl ← SignExtend32(macl) + mul;
    temp ← SignedSaturate32(macl);
    IF (macl = temp)
        result ← (mach << 32) ∨ ZeroExtend32(macl);
    ELSE
        result ← (0x1 << 32) ∨ ZeroExtend32(temp);
}
ELSE
    result ← ((mach << 32) + macl) + mul;
macl ← result;
mach ← result >> 32;
Rm ← Register(m_address);
Rn ← Register(n_address);
MACL ← ZeroExtend32(macl);
MACH ← ZeroExtend32(mach);

```



# MOV Rm, Rn

## Description

This instruction copies the value of  $R_m$  to  $R_n$ .

## Operation

**MOV Rm, Rn**

0110				n				m				0011			
15				12				8				0			
				11				7				3			
								4							

$op1 \leftarrow \text{ZeroExtend}_{32}(R_m);$   
 $op2 \leftarrow op1;$   
 $R_n \leftarrow \text{Register}(op2);$

## Note

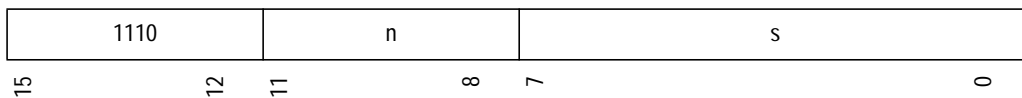
# MOV #imm, Rn

## Description

This instruction sign-extends the 8-bit immediate  $s$  and places the result in  $R_n$ .

## Operation

**MOV #imm, Rn**



```

imm ← SignExtend8(s);
op2 ← imm;
Rn ← Register(op2);

```

## Note

The '#imm' in the assembly syntax represents the immediate  $s$  after sign extension.

# MOV.B Rm, @Rn

## Description

This instruction stores a byte to memory using register indirect with zero-displacement addressing. The effective address is specified in  $R_n$ . The byte to be stored is held in the lowest 8 bits of  $R_m$ .

## Operation

**MOV.B Rm, @Rn**

0010	n	m	0000
15	12	11	8
			7
			4
			3
			0

```

op1 ← SignExtend32(Rm);
op2 ← SignExtend32(Rn);
address ← ZeroExtend32(op2);
WriteMemory8(address, op1);

```

## Exceptions

WADDERR, WTLBMISS, WRITEPROT, FIRSTWRITE

## Note

# MOV.B Rm, @-Rn

## Description

This instruction stores a byte to memory using register indirect with pre-decrement addressing.  $R_n$  is pre-decremented by 1 to give the effective address. The byte to be stored is held in the lowest 8 bits of  $R_m$ .

## Operation

**MOV.B Rm, @-Rn**

0010	n	m	0100
15	12	11	8
		7	4
			3
			0

```

op1 ← SignExtend32(Rm);
op2 ← SignExtend32(Rn);
address ← ZeroExtend32(op2 - 1);
WriteMemory8(address, op1);
op2 ← address;
Rn ← Register(op2);

```

## Exceptions

WADDERR, WTLBMISS, WRITEPROT, FIRSTWRITE

## Note

The effective address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.

# MOV.B Rm, @(R0, Rn)

## Description

This instruction stores a byte to memory using register indirect addressing. The effective address is formed by adding  $R_0$  to  $R_n$ . The byte to be stored is held in the lowest 8 bits of  $R_m$ .

## Operation

**MOV.B Rm, @(R0, Rn)**

0000	n	m	0100
15	12	11	8
		7	4
		3	0

```

r0 ← SignExtend32(R0);
op1 ← SignExtend32(Rm);
op2 ← SignExtend32(Rn);
address ← ZeroExtend32(r0 + op2);
WriteMemory8(address, op1);

```

## Exceptions

WADDERR, WTLBMISS, WRITEPROT, FIRSTWRITE

## Note

The effective address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.

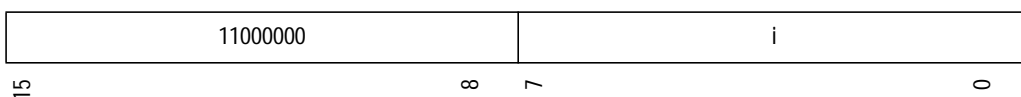
# MOV.B R0, @(disp, GBR)

## Description

This instruction stores a byte to memory using GBR-relative with displacement addressing. The effective address is formed by adding GBR to the zero-extended 8-bit immediate *i*. The byte to be stored is held in the lowest 8 bits of  $R_0$ .

## Operation

**MOV.B R0, @(disp, GBR)**



```

gbr ← SignExtend32(GBR);
r0 ← SignExtend32(R0);
disp ← ZeroExtend8(i);
address ← ZeroExtend32(disp + gbr);
WriteMemory8(address, r0);

```

## Exceptions

WADDERR, WTLBMISS, WRITEPROT, FIRSTWRITE

## Note

The effective address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.

The 'disp' in the assembly syntax represents the immediate *i* after zero extension.

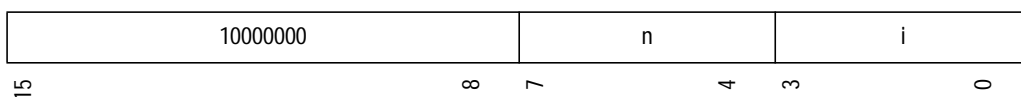
# MOV.B R0, @(disp, Rn)

## Description

This instruction stores a byte to memory using register indirect with displacement addressing. The effective address is formed by adding  $R_n$  and the zero-extended 4-bit immediate  $i$ . The byte to be stored is held in the lowest 8 bits of  $R_0$ .

## Operation

**MOV.B R0, @(disp, Rn)**



```

r0 ← SignExtend32(R0);
disp ← ZeroExtend4(i);
op2 ← SignExtend32(Rn);
address ← ZeroExtend32(disp + op2);
WriteMemory8(address, r0);

```

## Exceptions

WADDERR, WTLBMISS, WRITEPROT, FIRSTWRITE

## Note

The effective address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.

The 'disp' in the assembly syntax represents the immediate  $i$  after zero extension.

# MOV.B @Rm, Rn

## Description

This instruction loads a signed byte from memory using register indirect with zero-displacement addressing. The effective address is specified in  $R_m$ . The byte is loaded from the effective address, sign-extended and placed in  $R_n$ .

## Operation

**MOV.B @Rm, Rn**

0110	n	m	0000
15	12	11	8
		7	4
		3	0

```

op1 ← SignExtend32(Rm);
address ← ZeroExtend32(op1);
op2 ← SignExtend8(ReadMemory8(address));
Rn ← Register(op2);

```

## Exceptions

RADDERR, RTLBMIS, READPROT

## Note



# MOV.B @Rm+, Rn

## Description

This instruction loads a signed byte from memory using register indirect with post-increment addressing. The byte is loaded from the effective address specified in  $R_m$  and sign-extended.  $R_m$  is post-incremented by 1, and then the loaded byte is placed in  $R_n$ .

## Operation

**MOV.B @Rm+, Rn**

0110	n	m	0100
15	12	11	8
			7
			4
			3
			0

```

m_field ← ZeroExtend4(m);
n_field ← ZeroExtend4(n);
op1 ← SignExtend32(Rm);
address ← ZeroExtend32(op1);
op2 ← SignExtend8(ReadMemory8(address));
IF (m_field = n_field)
    op1 ← op2;
ELSE
    op1 ← op1 + 1;
Rm ← Register(op1);
Rn ← Register(op2);

```

## Exceptions

RADDERR, RTLBMISS, READPROT

## Note

If  $R_m$  and  $R_n$  refer to the same register (i.e.  $m = n$ ), the result placed in this register will be the sign-extended byte loaded from memory.

# MOV.B @(R0, Rm), Rn

## Description

This instruction loads a signed byte from memory using register indirect addressing. The effective address is formed by adding  $R_0$  to  $R_m$ . The byte is loaded from the effective address, sign-extended and placed in  $R_n$ .

## Operation

**MOV.B @(R0, Rm), Rn**

0000	n	m	1100
15	12	11	8
			7
			4
			3
			0

```

r0 ← SignExtend32(R0);
op1 ← SignExtend32(Rm);
address ← ZeroExtend32(r0 + op1);
op2 ← SignExtend8(ReadMemory8(address));
Rn ← Register(op2);

```

## Exceptions

RADDERR, RTLBMIS, READPROT

## Note

The effective address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.

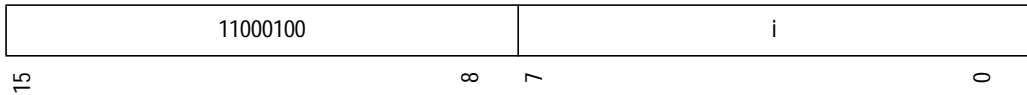
# MOV.B @(disp, GBR), R0

## Description

This instruction loads a signed byte from memory using GBR-relative with displacement addressing. The effective address is formed by adding GBR to the zero-extended 8-bit immediate *i*. The byte is loaded from the effective address, sign-extended and placed in  $R_0$ .

## Operation

**MOV.B @(disp, GBR), R0**



```

gbr ← SignExtend32(GBR);
disp ← ZeroExtend8(i);
address ← ZeroExtend32(disp + gbr);
r0 ← SignExtend8(ReadMemory8(address));
R0 ← Register(r0);

```

## Exceptions

RADDERR, RTLBMIS, READPROT

## Note

The effective address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.

The 'disp' in the assembly syntax represents the immediate *i* after zero extension.

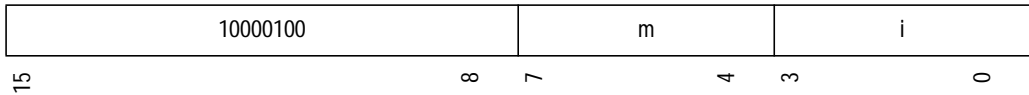
# MOV.B @(disp, Rm), R0

## Description

This instruction loads a signed byte from memory using register indirect with displacement addressing. The effective address is formed by adding  $R_m$  to the zero-extended 4-bit immediate  $i$ . The byte is loaded from the effective address, sign-extended and placed in  $R_0$ .

## Operation

**MOV.B @(disp, Rm), R0**



```

disp ← ZeroExtend4(i);
op2 ← SignExtend32(Rm);
address ← ZeroExtend32(disp + op2);
r0 ← SignExtend8(ReadMemory8(address));
R0 ← Register(r0);

```

## Exceptions

RADDERR, RTLBMIS, READPROT

## Note

The effective address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.

The 'disp' in the assembly syntax represents the immediate  $i$  after zero extension.

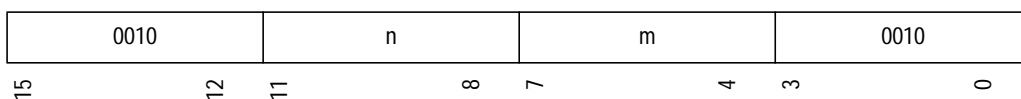
# MOV.L Rm, @Rn

## Description

This instruction stores a long-word to memory using register indirect with zero-displacement addressing. The effective address is specified in  $R_n$ . The long-word to be stored is held in  $R_m$ .

## Operation

**MOV.L Rm, @Rn**



```

op1 ← SignExtend32(Rm);
op2 ← SignExtend32(Rn);
address ← ZeroExtend32(op2);
WriteMemory32(address, op1);

```

## Exceptions

WADDERR, WTLBMISS, WRITEPROT, FIRSTWRITE

## Note

# MOV.L Rm, @-Rn

## Description

This instruction stores a long-word to memory using register indirect with pre-decrement addressing.  $R_n$  is pre-decremented by 4 to give the effective address. The long-word to be stored is held in  $R_m$ .

## Operation

**MOV.L Rm, @-Rn**

0010	n	m	0110
15	12	11	8
		7	4
		3	0

```

op1 ← SignExtend32(Rm);
op2 ← SignExtend32(Rn);
address ← ZeroExtend32(op2 - 4);
WriteMemory32(address, op1);
op2 ← address;
Rn ← Register(op2);

```

## Exceptions

WADDERR, WTLBMISS, WRITEPROT, FIRSTWRITE

## Note

The effective address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.

# MOV.L Rm, @(R0, Rn)

## Description

This instruction stores a long-word to memory using register indirect addressing. The effective address is formed by adding  $R_0$  to  $R_n$ . The long-word to be stored is held in  $R_m$ .

## Operation

**MOV.L Rm, @(R0, Rn)**

0000	n	m	0110
15	12	11	8
			7
			4
			3
			0

```

r0 ← SignExtend32(R0);
op1 ← SignExtend32(Rm);
op2 ← SignExtend32(Rn);
address ← ZeroExtend32(r0 + op2);
WriteMemory32(address, op1);

```

## Exceptions

WADDERR, WTLBMISS, WRITEPROT, FIRSTWRITE

## Note

The effective address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.

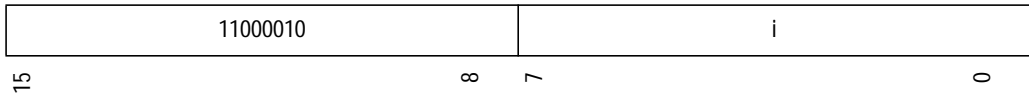
# MOV.L R0, @(disp, GBR)

## Description

This instruction stores a long-word to memory using GBR-relative with displacement addressing. The effective address is formed by adding GBR to the zero-extended 8-bit immediate *i* multiplied by 4. The long-word to be stored is held in  $R_0$ .

## Operation

**MOV.L R0, @(disp, GBR)**



```

gbr ← SignExtend32(GBR);
r0 ← SignExtend32(R0);
disp ← ZeroExtend8(i) << 2;
address ← ZeroExtend32(disp + gbr);
WriteMemory32(address, r0);

```

## Exceptions

WADDERR, WTLBMISS, WRITEPROT, FIRSTWRITE

## Note

The effective address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.

The 'disp' in the assembly syntax represents the immediate *i* after zero extension and scaling.



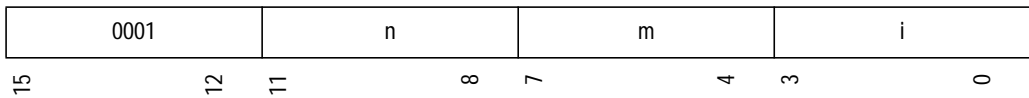
# MOV.L Rm, @(disp, Rn)

## Description

This instruction stores a long-word to memory using register indirect with displacement addressing. The effective address is formed by adding  $R_n$  to the zero-extended 4-bit immediate  $i$  multiplied by 4. The long-word to be stored is held in  $R_m$ .

## Operation

**MOV.L Rm, @(disp, Rn)**



```

op1 ← SignExtend32(Rm);
disp ← ZeroExtend4(i) << 2;
op3 ← SignExtend32(Rn);
address ← ZeroExtend32(disp + op3);
WriteMemory32(address, op1);

```

## Exceptions

WADDERR, WTLBMISS, WRITEPROT, FIRSTWRITE

## Note

The effective address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.

The 'disp' in the assembly syntax represents the immediate  $i$  after zero extension and scaling.

# MOV.L @Rm, Rn

## Description

This instruction loads a signed long-word from memory using register indirect with zero-displacement addressing. The effective address is specified in  $R_m$ . The long-word is loaded from the effective address and placed in  $R_n$ .

## Operation

**MOV.L @Rm, Rn**

0110	n	m	0010
15	12	11	8
		7	4
		3	0

```

op1 ← SignExtend32(Rm);
address ← ZeroExtend32(op1);
op2 ← SignExtend32(ReadMemory32(address));
Rn ← Register(op2);

```

## Exceptions

RADDERR, RTLBMIS, READPROT

## Note

# MOV.L @Rm+, Rn

## Description

This instruction loads a signed long-word from memory using register indirect with post-increment addressing. The long-word is loaded from the effective address specified in  $R_m$ .  $R_m$  is post-incremented by 4, and then the loaded long-word is placed in  $R_n$ .

## Operation

**MOV.L @Rm+, Rn**

0110	n	m	0110
15	12	11	8
		7	4
		3	0

```

m_field ← ZeroExtend4(m);
n_field ← ZeroExtend4(n);
op1 ← SignExtend32(Rm);
address ← ZeroExtend32(op1);
op2 ← SignExtend32(ReadMemory32(address));
IF (m_field = n_field)
    op1 ← op2;
ELSE
    op1 ← op1 + 4;
Rm ← Register(op1);
Rn ← Register(op2);

```

## Exceptions

RADDERR, RTLBMISS, READPROT

## Note

If  $R_m$  and  $R_n$  refer to the same register (i.e.  $m = n$ ), the result placed in this register will be the sign-extended byte loaded from memory.

# MOV.L @(R0, Rm), Rn

## Description

This instruction loads a signed long-word from memory using register indirect addressing. The effective address is formed by adding  $R_0$  to  $R_m$ . The long-word is loaded from the effective address and placed in  $R_n$ .

## Operation

**MOV.L @(R0, Rm), Rn**

0000	n	m	1110
15	12	11	8
			7
			4
			3
			0

```

r0 ← SignExtend32(R0);
op1 ← SignExtend32(Rm);
address ← ZeroExtend32(r0 + op1);
op2 ← SignExtend32(ReadMemory32(address));
Rn ← Register(op2);

```

## Exceptions

RADDERR, RTLBMIS, READPROT

## Note

The effective address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.

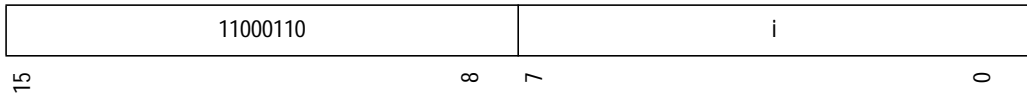
# MOV.L @(disp, GBR), R0

## Description

This instruction loads a signed long-word from memory using GBR-relative with displacement addressing. The effective address is formed by adding GBR to the zero-extended 8-bit immediate *i* multiplied by 4. The long-word is loaded from the effective address and placed in  $R_0$ .

## Operation

**MOV.L @(disp, GBR), R0**



```

gbr ← SignExtend32(GBR);
disp ← ZeroExtend8(i) << 2;
address ← ZeroExtend32(disp + gbr);
r0 ← SignExtend32(ReadMemory32(address));
R0 ← Register(r0);

```

## Exceptions

RADDERR, RTLBMIS, READPROT

## Note

The effective address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.

The 'disp' in the assembly syntax represents the immediate *i* after zero extension and scaling.

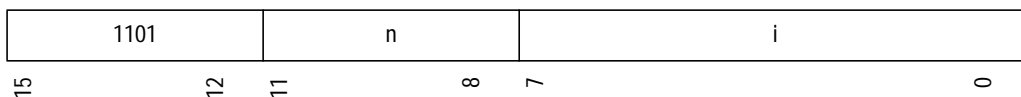
# MOV.L @(disp, PC), Rn

## Description

This instruction loads a signed long-word from memory using PC-relative with displacement addressing. The effective address is formed by calculating PC+4, clearing the lowest 2 bits, and adding the zero-extended 8-bit immediate *i* multiplied by 4. This address calculation ensures that the effective address is correctly aligned for a long-word access regardless of the PC alignment. The long-word is loaded from the effective address and placed in  $R_n$ .

## Operation

**MOV.L @(disp, PC), Rn**



```

pc ← SignExtend32(PC);
disp ← ZeroExtend8(i) << 2;
IF (IsDelaySlot())
    THROW ILLSLOT;
address ← ZeroExtend32(disp + ((pc + 4) & (~ 0x3)));
op2 ← SignExtend32(ReadMemory32(address));
Rn ← Register(op2);

```

## Exceptions

ILLSLOT, RADDERR, RTLBMIS, READPROT

## Note

The effective address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.

An ILLSLOT exception is raised if this instruction is executed in a delay slot.

The 'disp' in the assembly syntax represents the immediate *i* after zero extension and scaling.

# MOV.L @(disp, Rm), Rn

## Description

This instruction loads a signed long-word from memory using register indirect with displacement addressing. The effective address is formed by adding  $R_m$  to the zero-extended 4-bit immediate  $i$  multiplied by 4. The long-word is loaded from the effective address and placed in  $R_n$ .

## Operation

**MOV.L @(disp, Rm), Rn**

0101	n	m	i
15	12	11	8
		7	4
		3	0

```

disp ← ZeroExtend4(i) << 2;
op2 ← SignExtend32(Rm);
address ← ZeroExtend32(disp + op2);
op3 ← SignExtend32(ReadMemory32(address));
Rn ← Register(op3);

```

## Exceptions

RADDERR, RTLBMIS, READPROT

## Note

The effective address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.

The 'disp' in the assembly syntax represents the immediate  $i$  after zero extension and scaling.

# MOV.W Rm, @Rn

## Description

This instruction stores a word to memory using register indirect with zero-displacement addressing. The effective address is specified in  $R_n$ . The word to be stored is held in the lowest 16 bits of  $R_m$ .

## Operation

**MOV.W Rm, @Rn**

0010	n	m	0001
15	12	11	8
			7
			4
			3
			0

```

op1 ← SignExtend32(Rm);
op2 ← SignExtend32(Rn);
address ← ZeroExtend32(op2);
WriteMemory16(address, op1);

```

## Exceptions

WADDERR, WTLBMISS, WRITEPROT, FIRSTWRITE

## Note



# MOV.W Rm, @-Rn

## Description

This instruction stores a word to memory using register indirect with pre-decrement addressing.  $R_n$  is pre-decremented by 2 to give the effective address. The word to be stored is held in the lowest 16 bits of  $R_m$ .

## Operation

**MOV.W Rm, @-Rn**

0010	n	m	0101
15	12	11	8
		7	4
		3	0

```

op1 ← SignExtend32(Rm);
op2 ← SignExtend32(Rn);
address ← ZeroExtend32(op2 - 2);
WriteMemory16(address, op1);
op2 ← address;
Rn ← Register(op2);

```

## Exceptions

WADDERR, WTLBMISS, WRITEPROT, FIRSTWRITE

## Note

The effective address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.

# MOV.W Rm, @(R0, Rn)

## Description

This instruction stores a word to memory using register indirect addressing. The effective address is formed by adding  $R_0$  to  $R_n$ . The word to be stored is held in the lowest 16 bits of  $R_m$ .

## Operation

**MOV.W Rm, @(R0, Rn)**

0000	n	m	0101
15	12	11	8
		7	4
		3	0

```

r0 ← SignExtend32(R0);
op1 ← SignExtend32(Rm);
op2 ← SignExtend32(Rn);
address ← ZeroExtend32(r0 + op2);
WriteMemory16(address, op1);

```

## Exceptions

WADDERR, WTLBMISS, WRITEPROT, FIRSTWRITE

## Note

The effective address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.

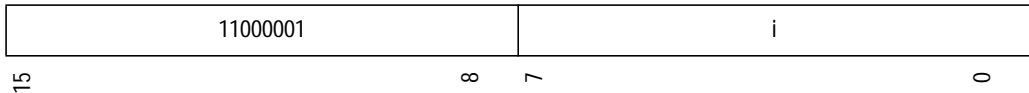
# MOV.W R0, @(disp, GBR)

## Description

This instruction stores a word to memory using GBR-relative with displacement addressing. The effective address is formed by adding GBR to the zero-extended 8-bit immediate *i* multiplied by 2. The word to be stored is held in the lowest 16 bits of R<sub>0</sub>.

## Operation

**MOV.W R0, @(disp, GBR)**



```

gbr ← SignExtend32(GBR);
r0 ← SignExtend32(R0);
disp ← ZeroExtend8(i) << 1;
address ← ZeroExtend32(disp + gbr);
WriteMemory16(address, r0);

```

## Exceptions

WADDERR, WTLBMISS, WRITEPROT, FIRSTWRITE

## Note

The effective address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.

The 'disp' in the assembly syntax represents the immediate *i* after zero extension and scaling.

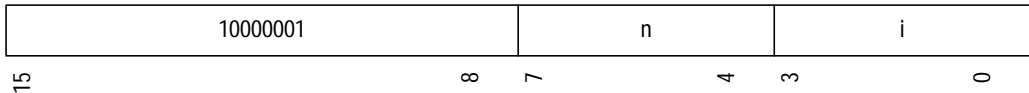
# MOV.W R0, @(disp, Rn)

## Description

This instruction stores a word to memory using register indirect with displacement addressing. The effective address is formed by adding  $R_n$  to the zero-extended 4-bit immediate  $i$  multiplied by 2. The word to be stored is held in the lowest 16 bits of  $R_m$ .

## Operation

**MOV.W R0, @(disp, Rn)**



```

r0 ← SignExtend32(R0);
disp ← ZeroExtend4(i) << 1;
op2 ← SignExtend32(Rn);
address ← ZeroExtend32(disp + op2);
WriteMemory16(address, r0);

```

## Exceptions

WADDERR, WTLBMISS, WRITEPROT, FIRSTWRITE

## Note

The effective address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.

The 'disp' in the assembly syntax represents the immediate  $i$  after zero extension and scaling.

# MOV.W @Rm, Rn

## Description

This instruction loads a signed word from memory using register indirect with zero-displacement addressing. The effective address is specified in  $R_m$ . The word is loaded from the effective address, sign-extended and placed in  $R_n$ .

## Operation

**MOV.W @Rm, Rn**

0110	n	m	0001
15	12	11	8
		7	4
		3	0

```

op1 ← SignExtend32(Rm);
address ← ZeroExtend32(op1);
op2 ← SignExtend16(ReadMemory16(address));
Rn ← Register(op2);

```

## Exceptions

RADDERR, RTLBMIS, READPROT

## Note

# MOV.W @Rm+, Rn

## Description

This instruction loads a signed word from memory using register indirect with post-increment addressing. The word is loaded from the effective address specified in  $R_m$  and sign-extended.  $R_m$  is post-incremented by 2, and then the loaded word is placed in  $R_n$ .

## Operation

**MOV.W @Rm+, Rn**

0110	n	m	0101
15	12	11	8
		7	4
		3	0

```

m_field ← ZeroExtend4(m);
n_field ← ZeroExtend4(n);
op1 ← SignExtend32(Rm);
address ← ZeroExtend32(op1);
op2 ← SignExtend16(ReadMemory16(address));
IF (m_field = n_field)
    op1 ← op2;
ELSE
    op1 ← op1 + 2;
Rm ← Register(op1);
Rn ← Register(op2);

```

## Exceptions

RADDERR, RTLBMIS, READPROT

## Note

If  $R_m$  and  $R_n$  refer to the same register (i.e.  $m = n$ ), the result placed in this register will be the sign-extended byte loaded from memory.

# MOV.W @(R0, Rm), Rn

## Description

This instruction loads a signed word from memory using register indirect addressing. The effective address is formed by adding  $R_0$  to  $R_m$ . The word is loaded from the effective address, sign-extended and placed in  $R_n$ .

## Operation

**MOV.W @(R0, Rm), Rn**

0000	n	m	1101
15	12	11	8
		7	4
		3	0

```

r0 ← SignExtend32(R0);
op1 ← SignExtend32(Rm);
address ← ZeroExtend32(r0 + op1);
op2 ← SignExtend16(ReadMemory16(address));
Rn ← Register(op2);

```

## Exceptions

RADDERR, RTLBMIS, READPROT

## Note

The effective address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.

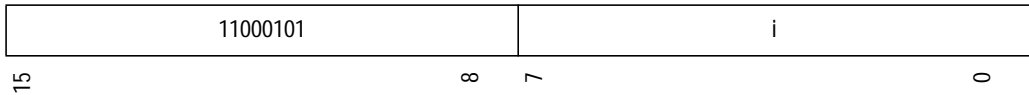
# MOV.W @(disp, GBR), R0

## Description

This instruction loads a signed word from memory using GBR-relative with displacement addressing. The effective address is formed by adding GBR to the zero-extended 8-bit immediate  $i$  multiplied by 2. The word is loaded from the effective address, sign-extended and placed in  $R_0$ .

## Operation

**MOV.W @(disp, GBR), R0**



```

gbr ← SignExtend32(GBR);
disp ← ZeroExtend8(i) << 1;
address ← ZeroExtend32(disp + gbr);
r0 ← SignExtend16(ReadMemory16(address));
R0 ← Register(r0);

```

## Exceptions

RADDERR, RTLBMIS, READPROT

## Note

The effective address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.

The 'disp' in the assembly syntax represents the immediate  $i$  after zero extension and scaling.



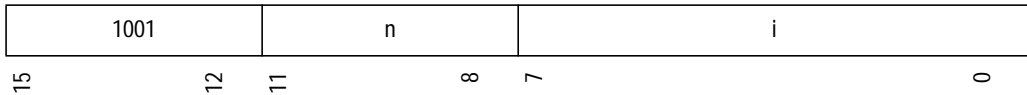
# MOV.W @(disp, PC), Rn

## Description

This instruction loads a signed word from memory using PC-relative with displacement addressing. The effective address is formed by calculating PC+4, and adding the zero-extended 8-bit immediate *i* multiplied by 2. The word is loaded from the effective address, sign-extended and placed in  $R_n$ .

## Operation

**MOV.W @(disp, PC), Rn**



```

pc ← SignExtend32(PC);
disp ← ZeroExtend8(i) << 1;
IF (IsDelaySlot())
    THROW ILLSLOT;
address ← ZeroExtend32(disp + (pc + 4));
op2 ← SignExtend16(ReadMemory16(address));
Rn ← Register(op2);

```

## Exceptions

ILLSLOT, RADDERR, RTLBMIS, READPROT

## Note

The effective address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.

An ILLSLOT exception is raised if this instruction is executed in a delay slot.

The 'disp' in the assembly syntax represents the immediate *i* after zero extension and scaling.

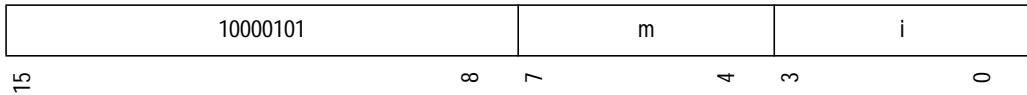
# MOV.W @(disp, Rm), R0

## Description

This instruction loads a signed word from memory using register indirect with displacement addressing. The effective address is formed by adding  $R_m$  to the zero-extended 4-bit immediate  $i$  multiplied by 2. The word is loaded from the effective address, sign-extended and placed in  $R_n$ .

## Operation

**MOV.W @(disp, Rm), R0**



```

disp ← ZeroExtend4(i) << 1;
op2 ← SignExtend32(Rm);
address ← ZeroExtend32(disp + op2);
r0 ← SignExtend16(ReadMemory16(address));
R0 ← Register(r0);

```

## Exceptions

RADDERR, RTLBMIS, READPROT

## Note

The effective address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.

The 'disp' in the assembly syntax represents the immediate  $i$  after zero extension and scaling.

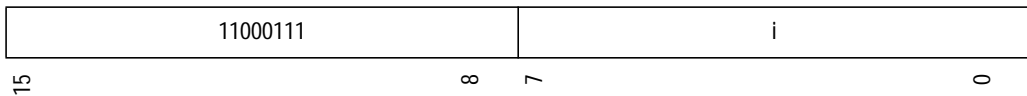
# MOVA @(disp, PC), R0

## Description

This instruction calculates an effective address using PC-relative with displacement addressing. The effective address is formed by calculating PC+4, clearing the lowest 2 bits, and adding the zero-extended 8-bit immediate *i* multiplied by 4. This address calculation ensures that the effective address is correctly aligned for a long-word access regardless of the PC alignment. The effective address is placed in R<sub>0</sub>.

## Operation

**MOVA @(disp, PC), R0**



```

pc ← SignExtend32(PC);
disp ← ZeroExtend8(i) << 2;
IF (IsDelaySlot())
    THROW ILLSLOT;
r0 ← disp + ((pc + 4) & (~ 0x3));
R0 ← Register(r0);

```

## Exceptions

ILLSLOT

## Note

The instructions only computes the effective address, no memory request is made.

An ILLSLOT exception is raised if this instruction is executed in a delay slot.

The 'disp' in the assembly syntax represents the immediate *i* after zero extension and scaling.

# MOVCA.L R0, @Rn

## Description

This instruction stores the long-word in  $R_0$  to memory at the effective address specified in  $R_n$ . It provides a hint to the implementation that it is not necessary to retrieve the data of this operand cache block from memory. It is implementation-specific as to whether the memory access will occur.

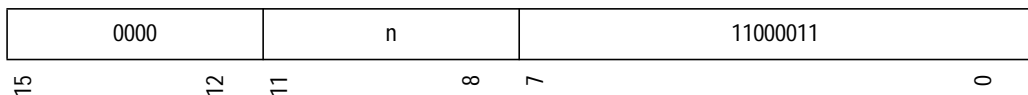
The effective address specified in  $R_n$  identifies a surrounding block of memory, which starts at an address aligned to the cache block size and has a size equal to the cache block size. The cache block size is implementation dependent.

MOVCA.L checks for address error, translation miss and protection exception cases.

Apart from the written long-word, the value of all other locations in the memory block targeted by a MOVCA.L becomes architecturally undefined. Programs must not rely on these values. For compatibility with other implementations, software must exercise care when using MOVCA.L.

## Operation

**MOVCA.L R0, @Rn**



```

r0 ← SignExtend32(R0);
op1 ← SignExtend32(Rn);
IF (AddressUnavailable(op1))
    THROW WADDERR, op1;
IF (MMU() AND DataAccessMiss(op1))
    THROW WTLBMISS, op1;
IF (MMU() AND WriteProhibited(op1))
    THROW WRITEPROT, op1;
IF (MMU() AND NOT DirtyBit(op1))
    THROW FIRSTWRITE, op1;
ALLOCO(op1);
address ← ZeroExtend32(op1);
WriteMemory32(op1, r0);
  
```

**Exceptions**

WADDERR, WTLBMISS, WRITEPROT, FIRSTWRITE

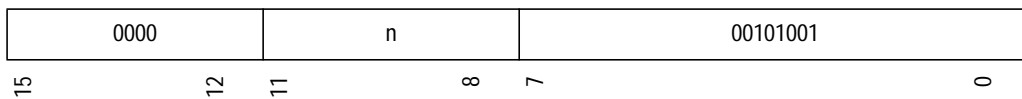
# MOVT Rn

## Description

This instruction copies the T-bit to  $R_n$ .

## Operation

### MOVT Rn



$t \leftarrow \text{ZeroExtend}_1(T);$   
 $op1 \leftarrow t;$   
 $R_n \leftarrow \text{Register}(op1);$

## Note

# MUL.L Rm, Rn

## Description

This instruction multiplies the 32-bit value in  $R_m$  by the 32-bit value in  $R_n$ , and places the least significant 32 bits of the result in MACL. The most significant 32 bits of the result are not provided, and MACH is not modified.

## Operation

### MUL.L Rm, Rn

0000	n	m	0111
15	12	11	8
			7
			4
			3
			0

```

op1 ← SignExtend32(Rm);
op2 ← SignExtend32(Rn);
mac1 ← op1 × op2;
MACL ← ZeroExtend32(mac1);

```

## Note

# MULS.W Rm, Rn

## Description

This instruction multiplies the signed lowest 16 bits of  $R_m$  by the signed lowest 16 bits of  $R_n$ , and places the full 32-bit result in MACL. MACH is not modified.

## Operation

**MULS.W Rm, Rn**

0010	n	m	1111
15	12	11	8
		7	4
		3	0

```

op1 ← SignExtend16(SignExtend32(Rm));
op2 ← SignExtend16(SignExtend32(Rn));
mac1 ← op1 × op2;
MACL ← ZeroExtend32(mac1);

```

## Note



# MULU.W R<sub>m</sub>, R<sub>n</sub>

## Description

This instruction multiplies the unsigned lowest 16 bits of R<sub>m</sub> by the unsigned lowest 16 bits of R<sub>n</sub>, and places the full 32-bit result in MACL. MACH is not modified.

## Operation

**MULU.W R<sub>m</sub>, R<sub>n</sub>**

0010	n	m	1110
15	12	11	8
		7	4
		3	0

```

op1 ← ZeroExtend16(SignExtend32(Rm));
op2 ← ZeroExtend16(SignExtend32(Rn));
mac1 ← op1 × op2;
MACL ← ZeroExtend32(mac1);

```

## Note

# NEG Rm, Rn

## Description

This instruction subtracts  $R_m$  from zero and places the result in  $R_n$ .

## Operation

**NEG Rm, Rn**

0110	n	m	1011
15	12	11	8
			7
			4
			3
			0

```

op1 ← SignExtend32(Rm);
op2 ← - op1;
Rn ← Register(op2);

```

## Note

# NEGC Rm, Rn

## Description

This instruction subtracts  $R_m$  and the T-bit from zero and places the result in  $R_n$ . The borrow from the subtraction is placed in the T-bit.

## Operation

**NEGC Rm, Rn**

0110	n	m	1010
15	12	11	8
		7	4
			3
			0

```

t ← ZeroExtend1(T);
op1 ← ZeroExtend32(Rm);
op2 ← (- op1) - t;
t ← op2 < 32 FOR 1 >;
Rn ← Register(op2);
T ← Bit(t);

```

## Note

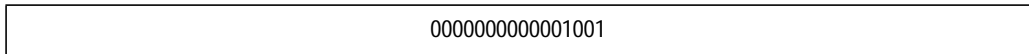
# NOP

## Description

This instruction performs no operation.

## Operation

**NOP**



15

0



# NOT Rm, Rn

## Description

This instruction performs a bitwise NOT on  $R_m$  and places the result in  $R_n$ .

## Operation

**NOT Rm, Rn**

0110				n				m				0111			
15				12				8				0			
				11				7				3			
								4							

```

op1 ← ZeroExtend32(Rm);
op2 ← ~ op1;
Rn ← Register(op2);

```

## Note

# OCBI @Rn

## Description

This instruction invalidates an operand cache block (if any) that corresponds to a specified effective address. If the data in the operand cache block is dirty, it is discarded without write-back to memory. Immediately after execution of OCBI, assuming no exception was raised, it is guaranteed that the targeted memory block in physical address space is not present in the operand cache.

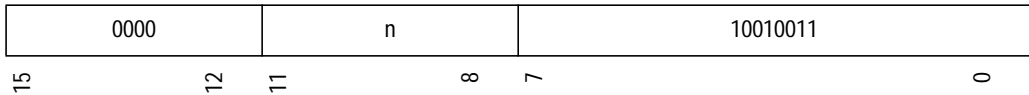
The effective address specified in  $R_n$  identifies a surrounding block of memory, which starts at an address aligned to the cache block size and has a size equal to the cache block size. The cache block size is implementation dependent.

OCBI invalidates an implementation-dependent amount of data. For compatibility with other implementations, software must exercise care when using OCBI.

OCBI checks for address error, translation miss and protection exception cases.

## Operation

### OCBI @Rn



```

op1 ← SignExtend32(Rn);
IF (AddressUnavailable(op1))
    THROW WADDERR, op1;
IF (MMU() AND DataAccessMiss(op1))
    THROW WTLBMISS, op1;
IF (MMU() AND WriteProhibited(op1))
    THROW WRITEPROT, op1;
IF (MMU() AND NOT DirtyBit(op1))
    THROW FIRSTWRITE, op1
OCBI(op1);
  
```

## Exceptions

WADDERR, WTLBMISS, WRITEPROT, FIRSTWRITE

## Note

# OCBP @Rn

## Description

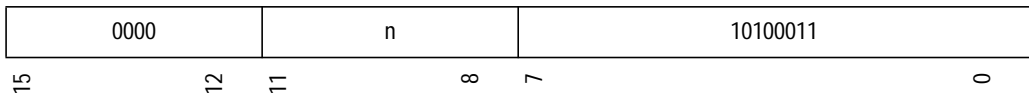
This instruction purges an operand cache block (if any) that corresponds to a specified effective address. If the data in the operand cache block is dirty, it is written back to memory before being discarded. Immediately after execution of OCBP, assuming no exception was raised, it is guaranteed that the targeted memory block in physical address space is not present in the operand cache.

The effective address specified in  $R_n$  identifies a surrounding block of memory, which starts at an address aligned to the cache block size and has a size equal to the cache block size. The cache block size is implementation dependent.

OCBP checks for address error, translation miss and protection exception cases.

## Operation

### OCBP @Rn



```

op1 ← SignExtend32(Rn);
IF (AddressUnavailable(op1))
    THROW RADDERR, op1;
IF (MMU() AND DataAccessMiss(op1))
    THROW RTLBMIS, op1;
IF (MMU() AND (ReadProhibited(op1) AND WriteProhibited(op1)))
    THROW READPROT, op1;
OCBP(op1);
  
```

## Exceptions

RADDERR, RTLBMIS, READPROT

## Note

# OCBWB @Rn

## Description

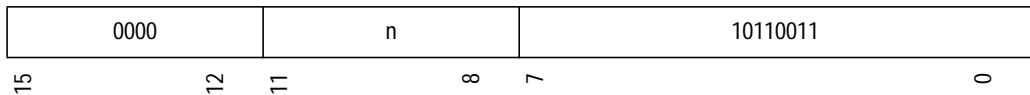
This instruction write-backs an operand cache block (if any) that corresponds to a specified effective address. If the data in the operand cache block is dirty, it is written back to memory but is not discarded. Immediately after execution of OCBWB, assuming no exception was raised, it is guaranteed that the targeted memory block in physical address space will not be dirty in the operand cache.

The effective address specified in  $R_n$  identifies a surrounding block of memory, which starts at an address aligned to the cache block size and has a size equal to the cache block size. The cache block size is implementation dependent.

OCBWB checks for address error, translation miss and protection exception cases.

## Operation

### OCBWB @Rn



```

op1 ← SignExtend32(Rn);
IF (AddressUnavailable(op1))
    THROW RADDERR, op1;
IF (MMU() AND DataAccessMiss(op1))
    THROW RTLBMIS, op1;
IF (MMU() AND (ReadProhibited(op1) AND WriteProhibited(op1)))
    THROW READPROT, op1;
OCBWB(op1);
  
```

## Exceptions

RADDERR, RTLBMIS, READPROT

## Note



# OR Rm, Rn

## Description

This instruction performs a bitwise OR of  $R_m$  with  $R_n$  and places the result in  $R_n$ .

## Operation

**OR Rm, Rn**

0010	n	m	1011
15	12	11	8
		7	4
		3	0

```

op1 ← ZeroExtend32(Rm);
op2 ← ZeroExtend32(Rn);
op2 ← op2 ∨ op1;
Rn ← Register(op2);

```

## Note

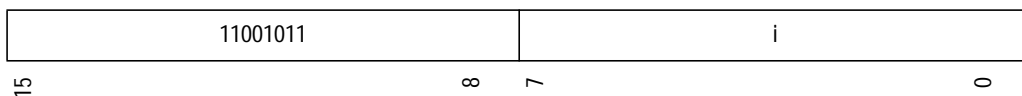
# OR #imm, R0

## Description

This instruction performs a bitwise OR of  $R_0$  with the zero-extended 8-bit immediate  $i$  and places the result in  $R_0$ .

## Operation

**OR #imm, R0**



```

r0 ← ZeroExtend32(R0);
imm ← ZeroExtend8(i);
r0 ← r0 ∨ imm;
R0 ← Register(r0);

```

## Note

The '#imm' in the assembly syntax represents the immediate  $i$  after zero extension.

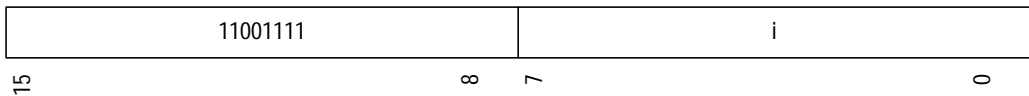
# OR.B #imm, @(R0, GBR)

## Description

This instruction performs a bitwise OR of an immediate constant with 8 bits of data held in memory. The effective address is calculated by adding  $R_0$  and GBR. The 8 bits of data at the effective address are read. A bitwise OR is performed of the read data with the zero-extended 8-bit immediate  $i$ . The result is written back to the 8 bits of data at the same effective address.

## Operation

**OR.B #imm, @(R0, GBR)**



```

r0 ← SignExtend32(R0);
gbr ← SignExtend32(GBR);
imm ← ZeroExtend8(i);
address ← ZeroExtend32(r0 + gbr);
value ← ZeroExtend8(ReadMemory8(address));
value ← value ∨ imm;
WriteMemory8(address, value);

```

## Exceptions

WADDERR, WTLBMISS, READPROT, WRITEPROT, FIRSTWRITE

## Note

The effective address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.

The '#imm' in the assembly syntax represents the immediate  $i$  after zero extension.

# PREF @Rn

## Description

This instruction indicates a software-directed data prefetch from the specified effective address. Software can use this instruction to give advance notice that particular data will be required. It is implementation-specific as to whether a prefetch will be performed.

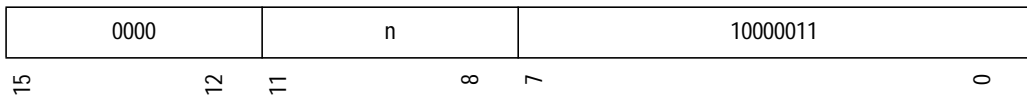
The effective address specified in  $R_n$  identifies a surrounding block of memory, which starts at an address aligned to the cache block size and has a size equal to the cache block size. The cache block size is implementation dependent.

Any OTLBMULTIHIT or RADDERR exception is delivered, other exceptions are discarded and the prefetch has no effect.

The semantics of a PREF instruction, when applied to an address in the store queues range (0xE0000000 to 0xE3FFFFFF) is quite different to that elsewhere. For details refer to [Section 4.6: Store queues on page 101](#).

## Operation

### PREF @Rn



```

op1 ← SignExtend32(Rn);
IF (AddressUnavailable(op1))
    THROW RADDERR, op1
IF (NOT (MMU() AND DataAccessMiss(op1)))
    IF (NOT (MMU() AND ReadProhibited(op1)))
        PREF(op1);
  
```

## Exceptions

RADDERR, OTLBMULTIHIT

## Note

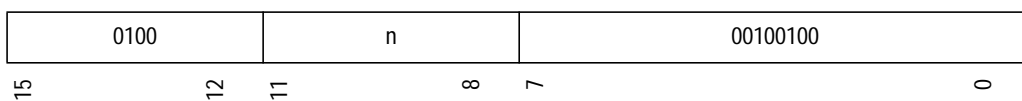
# ROTCL Rn

## Description

This instruction performs a one-bit left rotation of the bits held in  $R_n$  and the T-bit. The 32-bit value in  $R_n$  is shifted one bit to the left, the least significant bit is given the old value of the T-bit, and the bit that is shifted out is moved to the T-bit.

## Operation

### ROTCL Rn



```

t ← ZeroExtend1(T);
op1 ← ZeroExtend32(Rn);
op1 ← (op1 << 1) ∨ t;
t ← op1 < 32 FOR 1 >;
Rn ← Register(op1);
T ← Bit(t);

```

## Note

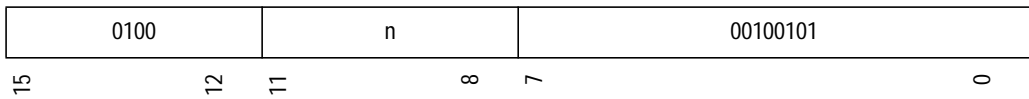
# ROTCR R<sub>n</sub>

## Description

This instruction performs a one-bit right rotation of the bits held in R<sub>n</sub> and the T-bit. The 32-bit value in R<sub>n</sub> is shifted one bit to the right, the most significant bit is given the old value of the T-bit, and the bit that is shifted out is moved to the T-bit.

## Operation

### ROTCR R<sub>n</sub>



```

t ← ZeroExtend1(T);
op1 ← ZeroExtend32(Rn);
oldt ← t;
t ← op1< 0 FOR 1 >;
op1 ← (op1 >> 1) ∨ (oldt << 31);
Rn ← Register(op1);
T ← Bit(t);

```

## Note

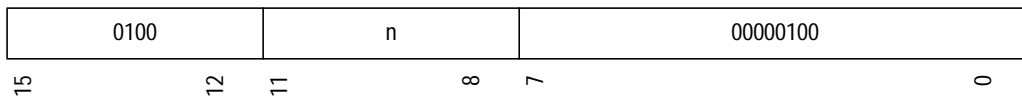
# ROTL Rn

## Description

This instruction performs a one-bit left rotation of the bits held in  $R_n$ . The 32-bit value in  $R_n$  is shifted one bit to the left, and the least significant bit is given the value of the bit that is shifted out. The bit that is shifted out of the operand is also copied to the T-bit.

## Operation

### ROTL Rn



```

op1 ← ZeroExtend32(Rn);
t ← op1< 31 FOR 1 >;
op1 ← (op1 << 1) ∨ t;
Rn ← Register(op1);
T ← Bit(t);

```

## Note

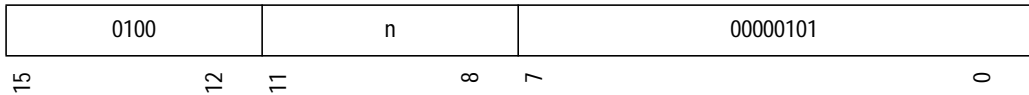
# ROTR R<sub>n</sub>

## Description

This instruction performs a one-bit right rotation of the bits held in R<sub>n</sub>. The 32-bit value in R<sub>n</sub> is shifted one bit to the right, and the most significant bit is given the value of the bit that is shifted out. The bit that is shifted out of the operand is also copied to the T-bit.

## Operation

### ROTR R<sub>n</sub>



```

op1 ← ZeroExtend32(Rn);
t ← op1<0 FOR 1>;
op1 ← (op1 >> 1) ∨ (t << 31);
Rn ← Register(op1);
T ← Bit(t);

```

## Note



# RTE

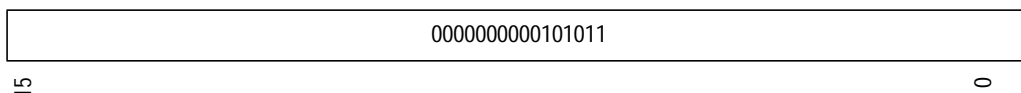
## Description

This instruction returns from an exception or interrupt handling routine by restoring the PC and SR values from SPC and SSR. Program execution continues from the address specified by the restored PC value.

RTE is a privileged instruction, and can only be used in privileged mode. Use of this instruction in user mode will cause an RESINST exception.

## Operation

### RTE



```

md ← ZeroExtend1(MD);
IF (md = 0)
    THROW RESINST;
ssr ← SignExtend32(SSR);
pc ← SignExtend32(PC)
IF (IsDelaySlot())
    THROW ILLSLOT;
target ← pc;
delayedpc ← target ∧ (~ 0x1);
PC" ← Register(delayedpc);

```

## Exceptions

RESINST, ILLSLOT

## Note

Since this is a delayed branch instruction, the instruction in the delay slot is executed before branching and must not generate an exception.

An ILLSLOT exception is raised if this instruction is executed in a delay slot.

Interrupts are not accepted between this instruction and the instruction in the delay slot.

The SR value defined prior to RTE execution is used to fetch the instruction in the RTE delay slot. However, the value of SR used during execution of the instruction in the delay slot, is that restored from SSR by the RTE instruction. It is recommended that, because of this feature, privileged instructions should not be placed in the delay slot.

If the branch target address is invalid then IADDERR trap is not delivered until after the instruction in the delay slot has executed and the PC has advanced to the target address, that is the exception is associated with the target instruction not the branch.

The behavior is architecturally undefined if the instruction in an RTE delay slot raises an exception. For this reason, it is recommended that only simple instructions that cannot generate exceptions are placed in RTE delay slots (unless considerable care is taken).

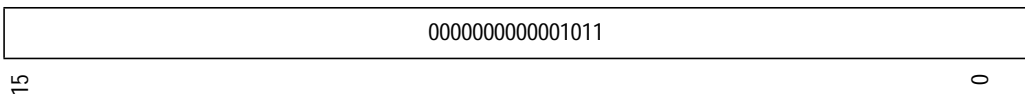
# RTS

## Description

This instruction is a delayed unconditional branch used for returning from a subroutine. The value in PR specifies the target address.

## Operation

### RTS



```
pr ← SignExtend32(PR);
IF (IsDelaySlot())
    THROW ILLSLOT;
target ← pr;
delayedpc ← target ∧ (~ 0x1);
PC" ← Register(delayedpc);
```

## Exceptions

### ILLSLOT

### Note

Since this is a delayed branch instruction, the delay slot is executed before branching. An ILLSLOT exception is raised if this instruction is executed in a delay slot.

If the branch target address is invalid then IADDERR trap is not delivered until after the instruction in the delay slot has executed and the PC has advanced to the target address, that is the exception is associated with the target instruction not the branch.

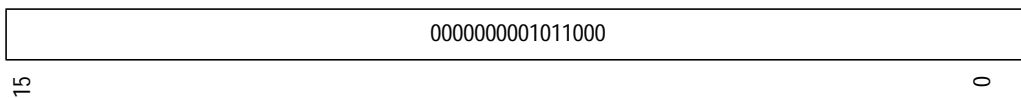
# SETS

## Description

This instruction sets the S-bit to 1.

## Operation

### SETS



$s \leftarrow 1;$   
 $S \leftarrow \text{Bit}(s);$

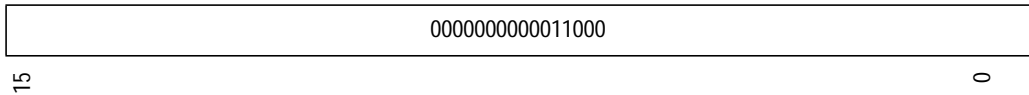
# SETT

## Description

This instruction sets the T-bit to 1.

## Operation

### SETT



$t \leftarrow 1;$   
 $T \leftarrow \text{Bit}(t);$

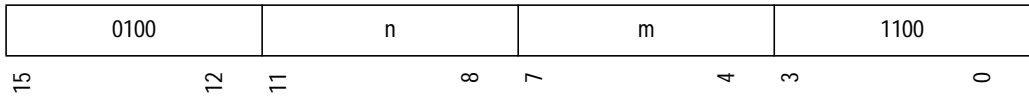
# SHAD R<sub>m</sub>, R<sub>n</sub>

## Description

This instruction performs an arithmetic shift of R<sub>n</sub>, with the dynamic shift direction and shift amount indicated by R<sub>m</sub>, and places the result in R<sub>n</sub>. If R<sub>m</sub> is zero, no shift is performed. If R<sub>m</sub> is greater than zero, this is a left shift and the shift amount is given by the least significant 5 bits of R<sub>m</sub>. If R<sub>m</sub> is less than zero, this is an arithmetic right shift and the shift amount is given by the least significant 5 bits of R<sub>m</sub> subtracted from 32. In the case where R<sub>m</sub> indicates an arithmetic right shift by 32, the result is filled with copies of the sign-bit of the original R<sub>n</sub>.

## Operation

### SHAD R<sub>m</sub>, R<sub>n</sub>



```

op1 ← SignExtend32(Rm);
op2 ← SignExtend32(Rn);
shift_amount ← ZeroExtend5(op1);
IF (op1 ≥ 0)
    op2 ← op2 << shift_amount;
ELSE IF (shift_amount ≠ 0)
    op2 ← op2 >> (32 - shift_amount);
ELSE IF (op2 < 0)
    op2 ← - 1;
ELSE
    op2 ← 0;
Rn ← Register(op2);

```

## Note

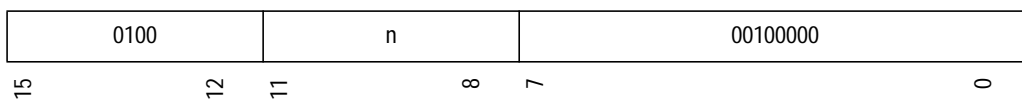
# SHAL Rn

## Description

Arithmetically shifts  $R_n$  to the left by one bit and places the result in  $R_n$ . The bit that is shifted out of the operand is moved to T-bit.

## Operation

### SHAL Rn



```

op1 ← SignExtend32(Rn);
t ← op1 < 31 FOR 1 >;
op1 ← op1 << 1;
Rn ← Register(op1);
T ← Bit(t);

```

## Note

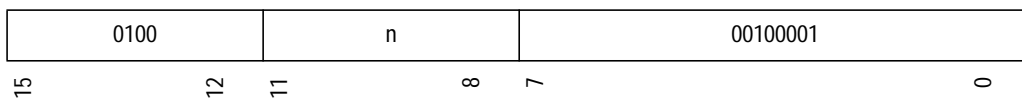
# SHAR Rn

## Description

Arithmetically shifts  $R_n$  to the right by one bit and places the result in  $R_n$ . The bit that is shifted out of the operand is moved to T-bit.

## Operation

### SHAR Rn



```

op1 ← SignExtend32(Rn);
t ← op1<0 FOR 1>;
op1 ← op1 >> 1;
Rn ← Register(op1);
T ← Bit(t);

```

## Note



# SHLD Rm, Rn

## Description

This instruction performs a logical shift of  $R_n$ , with the dynamic shift direction and shift amount indicated by  $R_m$ , and places the result in  $R_n$ . If  $R_m$  is zero, no shift is performed. If  $R_m$  is greater than zero, this is a left shift and the shift amount is given by the least significant 5 bits of  $R_m$ . If  $R_m$  is less than zero, this is a logical right shift and the shift amount is given by the least significant 5 bits of  $R_m$  subtracted from 32. In the case where  $R_m$  indicates a logical right shift by 32, the result is 0.

## Operation

### SHLD Rm, Rn

0100	n	m	1101
15	12	11	8
		7	4
		3	0

```

op1 ← SignExtend32(Rm);
op2 ← ZeroExtend32(Rn);
shift_amount ← ZeroExtend5(op1);
IF (op1 ≥ 0)
    op2 ← op2 << shift_amount;
ELSE IF (shift_amount ≠ 0)
    op2 ← op2 >> (32 - shift_amount);
ELSE
    op2 ← 0;
Rn ← Register(op2);

```

## Note

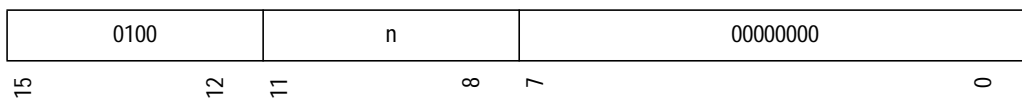
# SHLL Rn

## Description

This instruction performs a logical left shift of  $R_n$  by 1 bit and places the result in  $R_n$ . The bit that is shifted out is moved to the T-bit.

## Operation

### SHLL Rn



```

op1 ← ZeroExtend32(Rn);
t ← op1< 31 FOR 1 >;
op1 ← op1 << 1;
Rn ← Register(op1);
T ← Bit(t);

```

## Note

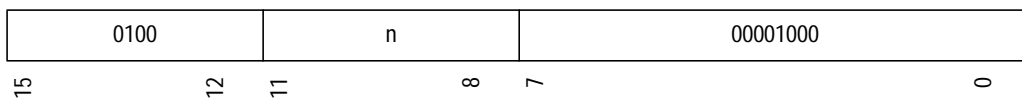
# SHLL2 Rn

## Description

This instruction performs a logical left shift of  $R_n$  by 2 bits and places the result in  $R_n$ . The bits that are shifted out are discarded.

## Operation

### SHLL2 Rn



```

op1 ← ZeroExtend32(Rn);
op1 ← op1 << 2;
Rn ← Register(op1);

```

## Note

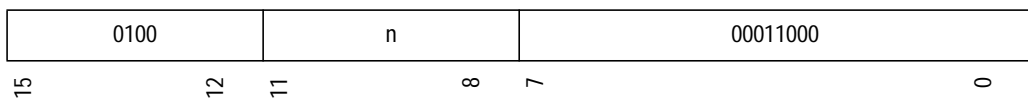
# SHLL8 Rn

## Description

This instruction performs a logical left shift of  $R_n$  by 8 bits and places the result in  $R_n$ . The bits that are shifted out are discarded.

## Operation

### SHLL8 Rn



```

op1 ← ZeroExtend32(Rn);
op1 ← op1 << 8;
Rn ← Register(op1);

```

## Note

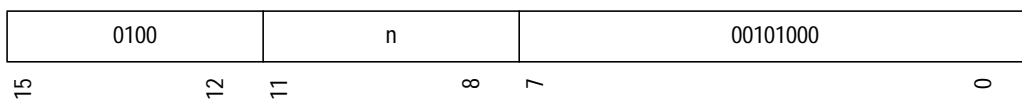
# SHLL16 Rn

## Description

This instruction performs a logical left shift of  $R_n$  by 16 bits and places the result in  $R_n$ . The bits that are shifted out are discarded.

## Operation

### SHLL16 Rn



```

op1 ← ZeroExtend32(Rn);
op1 ← op1 << 16;
Rn ← Register(op1);

```

## Note

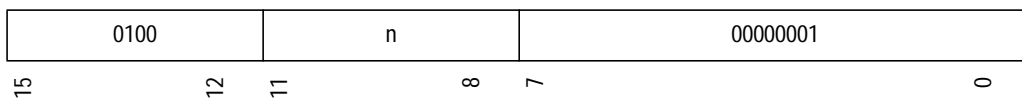
# SHLR Rn

## Description

This instruction performs a logical right shift of  $R_n$  by 1 bit and places the result in  $R_n$ . The bit that is shifted out is moved to the T-bit.

## Operation

### SHLR Rn



```

op1 ← ZeroExtend32(Rn);
t ← op1<0 FOR 1>;
op1 ← op1 >> 1;
Rn ← Register(op1);
T ← Bit(t);

```

## Note

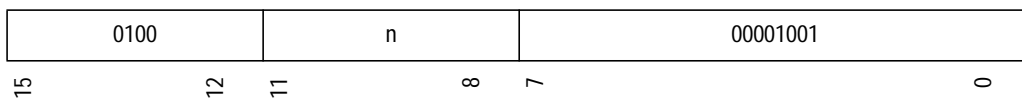
# SHLR2 Rn

## Description

This instruction performs a logical right shift of  $R_n$  by 2 bits and places the result in  $R_n$ . The bits that are shifted out are discarded.

## Operation

### SHLR2 Rn



```

op1 ← ZeroExtend32(Rn);
op1 ← op1 >> 2;
Rn ← Register(op1);

```

## Note

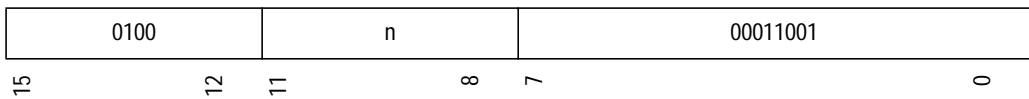
# SHLR8 Rn

## Description

This instruction performs a logical right shift of  $R_n$  by 8 bits and places the result in  $R_n$ . The bits that are shifted out are discarded.

## Operation

### SHLR8 Rn



```

op1 ← ZeroExtend32(Rn);
op1 ← op1 >> 8;
Rn ← Register(op1);

```

## Note



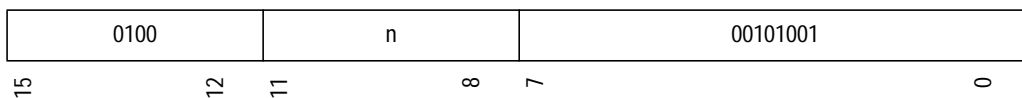
# SHLR16 Rn

## Description

This instruction performs a logical right shift of  $R_n$  by 16 bits and places the result in  $R_n$ . The bits that are shifted out are discarded.

## Operation

### SHLR16 Rn



```

op1 ← ZeroExtend32(Rn);
op1 ← op1 >> 16;
Rn ← Register(op1);

```

## Note

# SLEEP

## Description

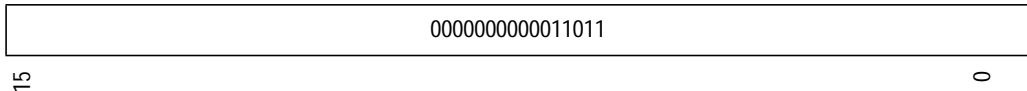
This instruction places the CPU in the power-down state.

In power-down mode, the CPU retains its internal state, but immediately stops executing instructions and waits for an interrupt request. The PC at the point of sleep is the address of the instruction immediately following the SLEEP instruction. This property ensures that when the CPU receives an interrupt request, and exits the power-down state, the SPC will contain the address of the instruction following the SLEEP.

SLEEP is a privileged instruction, and can only be used in privileged mode. Use of this instruction in user mode will cause an RESINST exception.

## Operation

### SLEEP



```
md ← ZeroExtend1(MD);
IF (md = 0)
    THROW RESINST;
SLEEP()
```

## Exceptions

RESINST

## Note

The effect of SLEEP upon rest of system depends upon the system architecture. Refer to the system architecture manual of the appropriate product for further details.

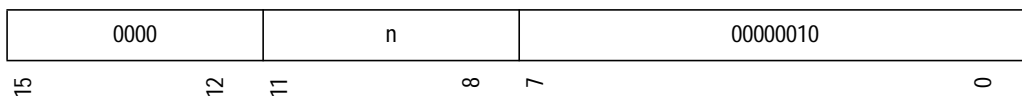
# STC SR, R<sub>n</sub>

## Description

This instruction copies SR to R<sub>n</sub>, it is a privileged instruction.

## Operation

**STC SR, R<sub>n</sub>**



```
md ← ZeroExtend1(MD);
IF (md = 0)
    THROW RESINST;
sr ← SignExtend32(SR);
op1 ← sr
Rn ← Register(op1);
```

## Exceptions

RESINST

## Note

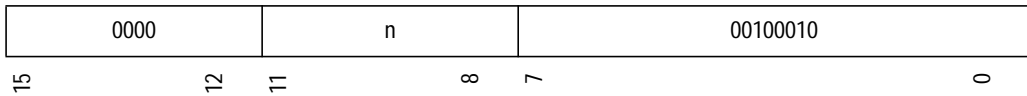
# STC VBR, R<sub>n</sub>

## Description

This instruction copies VBR to R<sub>n</sub>, it is a privileged instruction.

## Operation

**STC VBR, R<sub>n</sub>**



```

md ← ZeroExtend1(MD);
IF (md = 0)
    THROW RESINST;
vbr ← SignExtend32(VBR);
op1 ← vbr
Rn ← Register(op1);

```

## Exceptions

RESINST

## Note

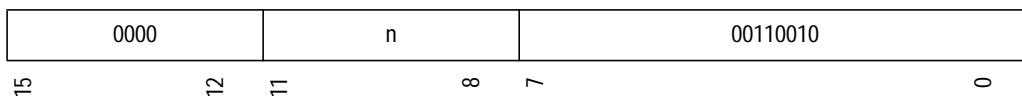
# STC SSR, R<sub>n</sub>

## Description

This instruction copies SSR to R<sub>n</sub>, it is a privileged instruction.

## Operation

**STC SSR, R<sub>n</sub>**



```
md ← ZeroExtend1(MD);
IF (md = 0)
    THROW RESINST;
ssr ← SignExtend32(SSR);
op1 ← ssr
Rn ← Register(op1);
```

## Exceptions

RESINST

## Note

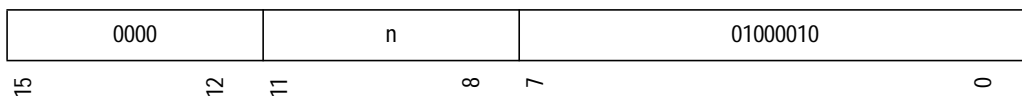
# STC SPC, R<sub>n</sub>

## Description

This instruction copies SPC to R<sub>n</sub>, it is a privileged instruction.

## Operation

**STC SPC, R<sub>n</sub>**



```

md ← ZeroExtend1(MD);
IF (md = 0)
    THROW RESINST;
spc ← SignExtend32(SPC);
op1 ← spc
Rn ← Register(op1);

```

## Exceptions

RESINST

## Note

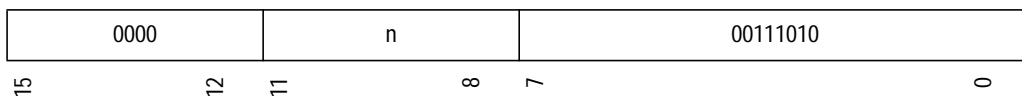
# STC SGR, R<sub>n</sub>

## Description

This instruction copies SGR to R<sub>n</sub>, it is a privileged instruction.

## Operation

**STC SGR, R<sub>n</sub>**



```
md ← ZeroExtend1(MD);
IF (md = 0)
    THROW RESINST;
sgr ← SignExtend32(SGR);
op1 ← sgr
Rn ← Register(op1);
```

## Exceptions

RESINST

## Note

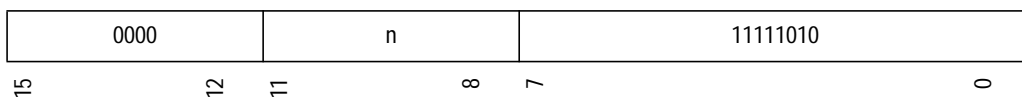
# STC DBR, R<sub>n</sub>

## Description

This instruction copies DBR to R<sub>n</sub>, it is a privileged instruction.

## Operation

**STC DBR, R<sub>n</sub>**



```

md ← ZeroExtend1(MD);
IF (md = 0)
    THROW RESINST;
dbr ← SignExtend32(DBR);
op1 ← dbr
Rn ← Register(op1);

```

## Exceptions

RESINST

## Note



# STC Rm\_BANK, Rn

## Description

This instruction copies Rm\_BANK to Rn, it is a privileged instruction.

## Operation

**STC Rm\_BANK, Rn**

0000				n			1	m		0010	
15	12		11	8		7	6	4		3	0

```

md ← ZeroExtend1(MD);
IF (md = 0)
    THROW RESINST;
op1 ← SignExtend32(Rm_BANK);
op2 ← op1;
Rn ← Register(op2);

```

## Exceptions

RESINST

## Note

# STC.L SR, @-Rn

## Description

This instruction stores SR to memory using register indirect with pre-decrement addressing.  $R_n$  is pre-decremented by 4 to give the effective address. The 32-bit value of SR is written to the effective address. This is a privileged instruction.

## Operation

**STC.L SR, @-Rn**

0100	n	00000011
15	12	11
	8	7
		0

```

md ← ZeroExtend1(MD);
IF (md = 0)
    THROW RESINST;
sr ← SignExtend32(SR);
op1 ← SignExtend32(Rn);
address ← ZeroExtend32(op1 - 4);
WriteMemory32(address, sr);
op1 ← address;
Rn ← Register(op1);

```

## Exceptions

RESINST, WADDERR, WTLBMISS, WRITEPROT, FIRSTWRITE

## Note

The effective address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.

# STC.L VBR, @-Rn

## Description

This instruction stores VBR to memory using register indirect with pre-decrement addressing.  $R_n$  is pre-decremented by 4 to give the effective address. The 32-bit value of VBR is written to the effective address. This is a privileged instruction.

## Operation

**STC.L VBR, @-Rn**

0100	n	00100011
15	12	0

```

md ← ZeroExtend1(MD);
IF (md = 0)
    THROW RESINST;
vbr ← SignExtend32(VBR);
op1 ← SignExtend32(Rn);
address ← ZeroExtend32(op1 - 4);
WriteMemory32(address, vbr);
op1 ← address;
Rn ← Register(op1);

```

## Exceptions

RESINST, WADDERR, WTLBMISS, WRITEPROT, FIRSTWRITE

## Note

The effective address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.

# STC.L SSR, @-Rn

## Description

This instruction stores SSR to memory using register indirect with pre-decrement addressing.  $R_n$  is pre-decremented by 4 to give the effective address. The 32-bit value of SSR is written to the effective address. This is a privileged instruction.

## Operation

### STC.L SSR, @-Rn

0100	n	00110011
15	12	0

```

md ← ZeroExtend1(MD);
IF (md = 0)
    THROW RESINST;
ssr ← SignExtend32(SSR);
op1 ← SignExtend32(Rn);
address ← ZeroExtend32(op1 - 4);
WriteMemory32(address, ssr);
op1 ← address;
Rn ← Register(op1);

```

## Exceptions

RESINST, WADDERR, WTLBMISS, WRITEPROT, FIRSTWRITE

## Note

The effective address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.

# STC.L SPC, @-Rn

## Description

This instruction stores SPC to memory using register indirect with pre-decrement addressing.  $R_n$  is pre-decremented by 4 to give the effective address. The 32-bit value of SPC is written to the effective address. This is a privileged instruction.

## Operation

### STC.L SPC, @-Rn

0100	n	01000011
15	12	11
	8	7
		0

```

md ← ZeroExtend1(MD);
IF (md = 0)
    THROW RESINST;
spc ← SignExtend32(SPC);
op1 ← SignExtend32(Rn);
address ← ZeroExtend32(op1 - 4);
WriteMemory32(address, spc);
op1 ← address;
Rn ← Register(op1);

```

## Exceptions

RESINST, WADDERR, WTLBMISS, WRITEPROT, FIRSTWRITE

## Note

The effective address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.

# STC.L SGR, @-Rn

## Description

This instruction stores SGR to memory using register indirect with pre-decrement addressing.  $R_n$  is pre-decremented by 4 to give the effective address. The 32-bit value of SGR is written to the effective address. This is a privileged instruction.

## Operation

### STC.L SGR, @-Rn

0100	n	00110010
15	12	0

```

md ← ZeroExtend1(MD);
IF (md = 0)
    THROW RESINST;
sgr ← SignExtend32(SGR);
op1 ← SignExtend32(Rn);
address ← ZeroExtend32(op1 - 4);
WriteMemory32(address, sgr);
op1 ← address;
Rn ← Register(op1);

```

## Exceptions

RESINST, WADDERR, WTLBMISS, WRITEPROT, FIRSTWRITE

## Note

The effective address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.

# STC.L DBR, @-Rn

## Description

This instruction stores DBR to memory using register indirect with pre-decrement addressing.  $R_n$  is pre-decremented by 4 to give the effective address. The 32-bit value of DBR is written to the effective address. This is a privileged instruction.

## Operation

### STC.L DBR, @-Rn

0100	n	11110010
15	12	11
	8	7
		0

```

md ← ZeroExtend1(MD);
IF (md = 0)
    THROW RESINST;
dbr ← SignExtend32(DBR);
op1 ← SignExtend32(Rn);
address ← ZeroExtend32(op1 - 4);
WriteMemory32(address, dbr);
op1 ← address;
Rn ← Register(op1);

```

## Exceptions

RESINST, WADDERR, WTLBMISS, WRITEPROT, FIRSTWRITE

## Note

The effective address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.

# STC.L Rm\_BANK, @-Rn

## Description

This instruction stores Rm\_BANK to memory using register indirect with pre-decrement addressing. R<sub>n</sub> is pre-decremented by 4 to give the effective address. The 32-bit value of Rm\_BANK is written to the effective address. This is a privileged instruction.

## Operation

**STC.L Rm\_BANK, @-Rn**

0100	n	1	m	0011
15	12	11	8	7
				0

```

md ← ZeroExtend1(MD);
IF (md = 0)
    THROW RESINST;
op1 ← SignExtend32(Rm_BANK);
op2 ← SignExtend32(Rn);
address ← ZeroExtend32(op2 - 4);
WriteMemory32(address, op1);
op2 ← address;
Rn ← Register(op2);

```

## Exceptions

RESINST, WADDERR, WTLBMISS, WRITEPROT, FIRSTWRITE

## Note

The effective address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.



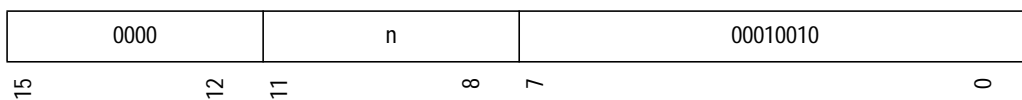
# STC GBR, R<sub>n</sub>

## Description

This instruction copies GBR to R<sub>n</sub>.

## Operation

### STC GBR, R<sub>n</sub>



```

gbr ← SignExtend32(GBR);
op1 ← gbr;
Rn ← Register(op1);

```

## Note

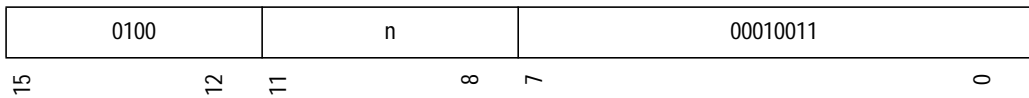
# STC.L GBR, @-Rn

## Description

This instruction stores GBR to memory using register indirect with pre-decrement addressing.  $R_n$  is pre-decremented by 4 to give the effective address. The 32-bit value of GBR is written to the effective address.

## Operation

**STC.L GBR, @-Rn**



```

gbr ← SignExtend32(GBR);
op1 ← SignExtend32(Rn);
address ← ZeroExtend32(op1 - 4);
WriteMemory32(address, gbr);
op1 ← address;
Rn ← Register(op1);

```

## Exceptions

WADDERR, WTLBMISS, WRITEPROT, FIRSTWRITE

## Note

The effective address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.

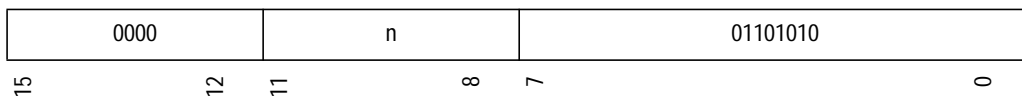
# STS FPSCR, R<sub>n</sub>

## Description

This floating-point instruction copies FPSCR to R<sub>n</sub>.

## Operation

### STS FPSCR, R<sub>n</sub>



```

sr ← ZeroExtend32(SR);
fps ← ZeroExtend32(FPSCR);
IF (FpulsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpulsDisabled(sr))
    THROW FPUDIS;
op1 ← fps;
Rn ← Register(op1);

```

## Exceptions

SLOTFPUDIS, FPUDIS

# STS.L FPSCR, @-Rn

## Description

This floating-point instruction stores FPSCR to memory using register indirect with pre-decrement addressing.  $R_n$  is pre-decremented by 4 to give the effective address. The 32-bit value of FPSCR is written to the effective address.

## Operation

### STS.L FPSCR, @-Rn

0100	n	01100010
15	12	0

```

sr ← ZeroExtend32(SR);
fps ← ZeroExtend32(FPSCR);
op1 ← SignExtend32(Rn);
IF (FpulsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpulsDisabled(sr))
    THROW FPUDIS;
value ← fps;
address ← ZeroExtend32(op1 - 4);
WriteMemory32(address, value);
op1 ← address;
Rn ← Register(op1);

```

## Exceptions

SLOTFPUDIS, FPUDIS, WADDERR, WTLBMISS, WRITEPROT, FIRSTWRITE

## Note

The effective address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.

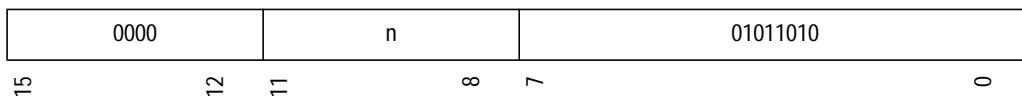
# STS FPUL, R<sub>n</sub>

## Description

This floating-point instruction copies FPUL to R<sub>n</sub>.

## Operation

### STS FPUL, R<sub>n</sub>



```

sr ← ZeroExtend32(SR);
fpul ← SignExtend32(FPUL);
IF (FpulsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpulsDisabled(sr))
    THROW FPUDIS;
op1 ← fpul;
Rn ← Register(op1);

```

## Exceptions

SLOTFPUDIS, FPUDIS

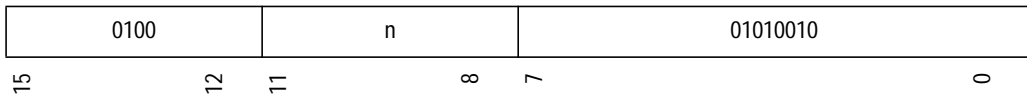
# STS.L FPUL, @-Rn

## Description

This floating-point instruction stores FPUL to memory using register indirect with pre-decrement addressing.  $R_n$  is pre-decremented by 4 to give the effective address. The 32-bit value of FPUL is written to the effective address.

## Operation

### STS.L FPUL, @-Rn



```

sr ← ZeroExtend32(SR);
fpul ← SignExtend32(FPUL);
op1 ← SignExtend32(Rn);
IF (FpulsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpulsDisabled(sr))
    THROW FPUDIS;
address ← ZeroExtend32(op1 - 4);
WriteMemory32(address, fpul);
op1 ← address;
Rn ← Register(op1);

```

## Exceptions

SLOTFPUDIS, FPUDIS, WADDERR, WTLBMISS, WRITEPROT, FIRSTWRITE

## Note

The effective address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.

# STS MACH, Rn

## Description

This instruction copies MACH to  $R_n$ .

## Operation

### STS MACH, Rn

0000	n	00001010
15	12	11
		8
		7
		0

```

mach ← SignExtend32(MACH);
op1 ← mach;
Rn ← Register(op1);

```

# STS.L MACH, @-Rn

## Description

This instruction stores MACH to memory using register indirect with pre-decrement addressing.  $R_n$  is pre-decremented by 4 to give the effective address. The 32-bit value of MACH is written to the effective address.

## Operation

**STS.L MACH, @-Rn**

0100	n	00000010
15	12	11
	8	7
		0

```

mach ← SignExtend32(MACH);
op1 ← SignExtend32(Rn);
address ← ZeroExtend32(op1 - 4);
WriteMemory32(address, mach);
op1 ← address;
Rn ← Register(op1);

```

## Exceptions

WADDERR, WTLBMISS, WRITEPROT, FIRSTWRITE

## Note

The effective address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.



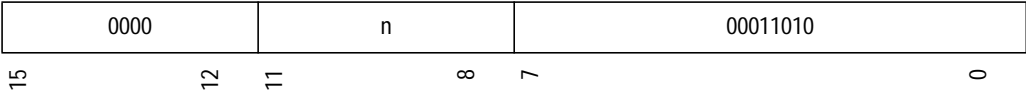
# STS MACL, Rn

## Description

This instruction copies MACL to R<sub>n</sub>.

## Operation

**STS MACL, Rn**



```
macl ← SignExtend32(MACL);  
op1 ← macl;  
Rn ← Register(op1);
```

# STS.L MACL, @-Rn

## Description

This instruction stores MACL to memory using register indirect with pre-decrement addressing.  $R_n$  is pre-decremented by 4 to give the effective address. The 32-bit value of MACL is written to the effective address.

## Operation

**STS.L MACL, @-Rn**

0100	n	00010010
15	12	11
	8	7
		0

```

mac1 ← SignExtend32(MACL);
op1 ← SignExtend32(Rn);
address ← ZeroExtend32(op1 - 4);
WriteMemory32(address, mac1);
op1 ← address;
Rn ← Register(op1);

```

## Exceptions

WADDERR, WTLBMISS, WRITEPROT, FIRSTWRITE

## Note

The effective address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.

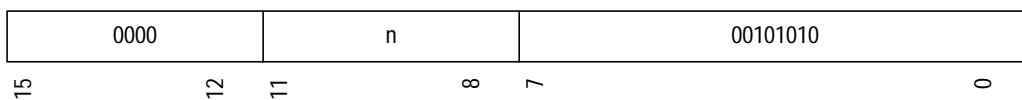
# STS PR, Rn

## Description

This instruction copies PR to  $R_n$ .

## Operation

**STS PR, Rn**



```
pr ← SignExtend32(PR');
op1 ← pr;
Rn ← Register(op1);
```

## Note

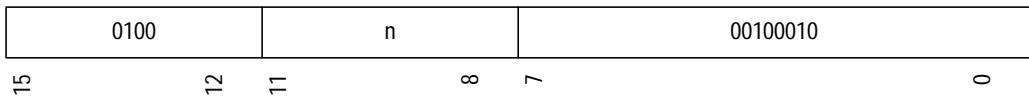
# STS.L PR, @-Rn

## Description

This instruction stores PR to memory using register indirect with pre-decrement addressing.  $R_n$  is pre-decremented by 4 to give the effective address. The 32-bit value of PR is written to the effective address.

## Operation

**STS.L PR, @-Rn**



```

pr ← SignExtend32(PR);
op1 ← SignExtend32(Rn);
address ← ZeroExtend32(op1 - 4);
WriteMemory32(address, pr);
op1 ← address;
Rn ← Register(op1);

```

## Exceptions

WADDERR, WTLBMISS, WRITEPROT, FIRSTWRITE

## Note

The effective address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.

# SUB R<sub>m</sub>, R<sub>n</sub>

## Description

This instruction subtracts R<sub>m</sub> from R<sub>n</sub> and places the result in R<sub>n</sub>.

## Operation

**SUB R<sub>m</sub>, R<sub>n</sub>**

0011	n	m	1000
15	12	11	8
		7	4
		3	0

```

op1 ← SignExtend32(Rm);
op2 ← SignExtend32(Rn);
op2 ← op2 - op1;
Rn ← Register(op2);

```

## Note

# SUBC Rm, Rn

## Description

This instruction subtracts  $R_m$  and the T-bit from  $R_n$  and places the result in  $R_n$ . The borrow from the subtraction is placed in the T-bit.

## Operation

**SUBC Rm, Rn**

0011	n	m	1010
15	12	11	8
		7	4
		3	0

```

t ← ZeroExtend1(T);
op1 ← ZeroExtend32(SignExtend32(Rm));
op2 ← ZeroExtend32(SignExtend32(Rn));
op2 ← (op2 - op1) - t;
t ← op2< 32 FOR 1 >;
Rn ← Register(op2);
T ← Bit(t);

```

## Note

# SUBV Rm, Rn

## Description

This instruction subtracts  $R_m$  from  $R_n$  and places the result in  $R_n$ . The T-bit is set to 1 if the subtraction result is outside the 32-bit signed range, otherwise the T-bit is set to 0.

## Operation

### SUBV Rm, Rn

0011	n	m	1011
15	12	11	8
		7	4
			3
			0

```

op1 ← SignExtend32(Rm);
op2 ← SignExtend32(Rn);
op2 ← op2 - op1;
t ← INT ((op2 < (- 231)) OR (op2 ≥ 231));
Rn ← Register(op2);
T ← Bit(t);

```

## Note

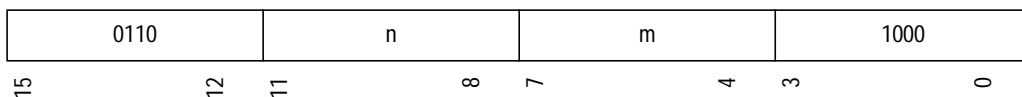
# SWAP.B Rm, Rn

## Description

This instruction swaps the values of the lower 2 bytes in  $R_m$  and places the result in  $R_n$ . Bits [0,7] take the value of bits [8,15]. Bits [8,15] take the value of bits [0,7]. Bits [16,31] are unchanged.

## Operation

**SWAP.B Rm, Rn**



```

op1 ← ZeroExtend32(Rm);
op2 ← ((op1 < 16 FOR 16 > << 16) ∨ (op1 < 0 FOR 8 > << 8)) ∨ op1 < 8 FOR 8 >;
Rn ← Register(op2);

```

## Note



# SWAP.W Rm, Rn

## Description

This instruction swaps the values of the 2 words in  $R_m$  and places the result in  $R_n$ . Bits [0,15] take the value of bits [16,31]. Bits [16,31] take the value of bits [0,15].

## Operation

### SWAP.W Rm, Rn

0110	n	m	1001
15	12	11	8
		7	4
		3	0

```

op1 ← ZeroExtend32(Rm);
op2 ← (op1<0 FOR 16> << 16) ∨ op1<16 FOR 16>;
Rn ← Register(op2);

```

## Note

# TAS.B @Rn

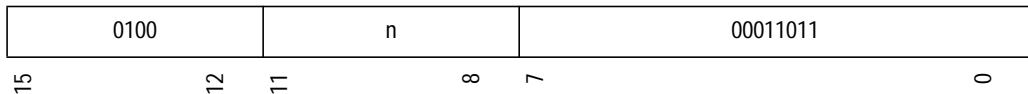
## Description

This instruction performs a test-and-set operation on the byte data at the effective address specified in  $R_n$ . It begins by purging the operand cache block containing the accessed memory location. The 8 bits of data at the effective address are read from memory. If the read data is 0 the T-bit is set, otherwise the T-bit is cleared. The highest bit of the 8-bit data (bit 7) is set, and the result is written back to the memory at the same effective address.

This test-and-set is atomic from the CPU perspective. This instruction cannot be interrupted during its operation.

## Operation

### TAS.B @Rn



```

op1 ← SignExtend32(Rn);
address ← ZeroExtend32(op1);
OCBP(address)
value ← ZeroExtend8(ReadMemory8(address));
t ← INT (value = 0);
value ← value ∨ (1 << 7);
WriteMemory8(address, value);
T ← Bit(t);

```

## Exceptions

WADDERR, WTLBMISS, READPROT, WRITEPROT, FIRSTWRITE

## Note

The TAS.B instruction guarantees atomicity of access to all components of the core but not necessarily the entire address space. Refer to the system architecture manual of the appropriate product to determine the properties of individual targets in the address map.

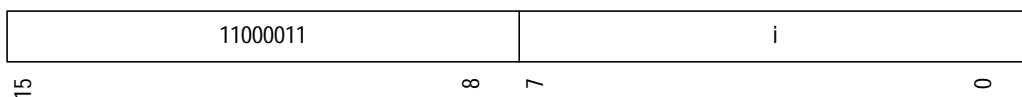
# TRAPA #imm

## Description

This instruction causes a pre-execution trap. The value of the zero-extended 8-bit immediate *i* is used by the handler launch sequence to characterize the trap.

## Operation

TRAPA #imm



```

imm ← ZeroExtend8(i);
IF (IsDelaySlot())
    THROW ILLSLOT;
    THROW TRAP, imm;
  
```

## Exceptions

ILLSLOT, TRAP

## Note

An ILLSLOT exception is raised if this instruction is executed in a delay slot.

The '#imm' in the assembly syntax represents the immediate *i* after zero extension.

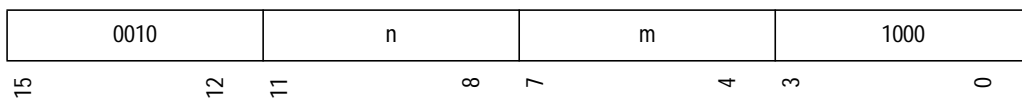
# TST Rm, Rn

## Description

This instruction performs a bitwise AND of  $R_m$  with  $R_n$ . If the result is 0, the T-bit is set, otherwise the T-bit is cleared.

## Operation

### TST Rm, Rn



```

op1 ← SignExtend32(Rm);
op2 ← SignExtend32(Rn);
t ← INT ((op1 ∧ op2) = 0);
T ← Bit(t);

```

## Note

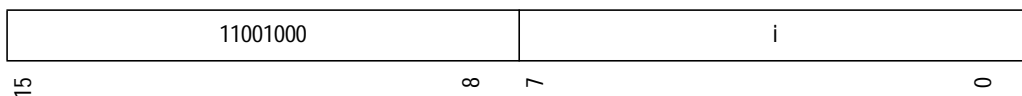
# TST #imm, R0

## Description

This instruction performs a bitwise AND of  $R_0$  with the zero-extended 8-bit immediate  $i$ . If the result is 0, the T-bit is set, otherwise the T-bit is cleared.

## Operation

**TST #imm, R0**



```

r0 ← SignExtend32(R0);
imm ← ZeroExtend8(i);
t ← INT ((r0 ∧ imm) = 0);
T ← Bit(t);

```

## Note

The '#imm' in the assembly syntax represents the immediate  $i$  after zero extension.

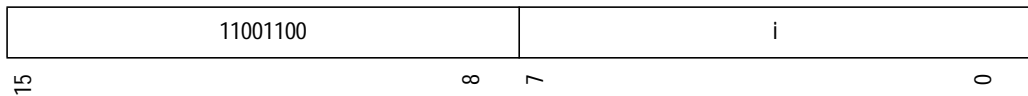
# TST.B #imm, @(R0, GBR)

## Description

This instruction performs a bitwise test of an immediate constant with 8 bits of data held in memory. The effective address is calculated by adding  $R_0$  and GBR. The 8 bits of data at the effective address are read. A bitwise AND is performed of the read data with the zero-extended 8-bit immediate  $i$ . If the result is 0, the T-bit is set, otherwise the T-bit is cleared.

## Operation

**TST.B #imm, @(R0, GBR)**



```

r0 ← SignExtend32(R0);
gbr ← SignExtend32(GBR);
imm ← ZeroExtend8(i);
address ← ZeroExtend32(r0 + gbr);
value ← ZeroExtend8(ReadMemory8(address));
t ← ((value ∧ imm) = 0);
T ← Bit(t);

```

## Exceptions

RADDERR, RTLBMIS, READPROT

## Note

The effective address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.

The '#imm' in the assembly syntax represents the immediate  $i$  after zero extension.

# XOR Rm, Rn

## Description

This instruction performs a bitwise XOR of  $R_m$  with  $R_n$  and places the result in  $R_n$ .

## Operation

**XOR Rm, Rn**

0010	n	m	1010
15	12	11	8
			7
			4
			3
			0

```

op1 ← ZeroExtend32(Rm);
op2 ← ZeroExtend32(Rn);
op2 ← op2 ⊕ op1;
Rn ← Register(op2);

```

## Note

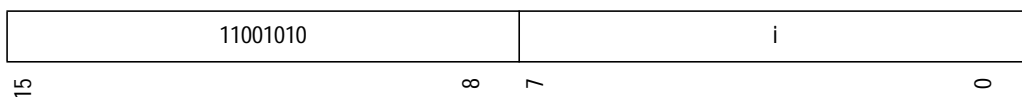
# XOR #imm, R0

## Description

This instruction performs a bitwise XOR of  $R_0$  with the zero-extended 8-bit immediate  $i$  and places the result in  $R_0$ .

## Operation

**XOR #imm, R0**



```

r0 ← ZeroExtend32(R0);
imm ← ZeroExtend8(i);
r0 ← r0 ⊕ imm;
R0 ← Register(r0);

```

## Note

The '#imm' in the assembly syntax represents the immediate  $i$  after zero extension.



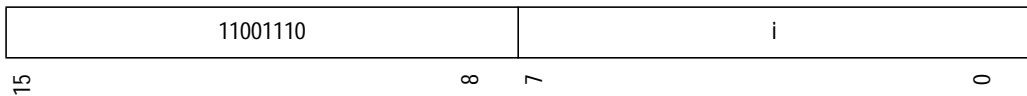
# XOR.B #imm, @(R0, GBR)

## Description

This instruction performs a bitwise XOR of an immediate constant with 8 bits of data held in memory. The effective address is calculated by adding  $R_0$  and GBR. The 8 bits of data at the effective address are read. A bitwise XOR is performed of the read data with the zero-extended 8-bit immediate  $i$ . The result is written back to the 8 bits of data at the same effective address.

## Operation

**XOR.B #imm, @(R0, GBR)**



```

r0 ← SignExtend32(R0);
gbr ← SignExtend32(GBR);
imm ← ZeroExtend8(i);
address ← ZeroExtend32(r0 + gbr);
value ← ZeroExtend8(ReadMemory8(address));
value ← value ⊕ imm;
WriteMemory8(address, value);

```

## Exceptions

WADDERR, WTLBMISS, READPROT, WRITEPROT, FIRSTWRITE

## Note

The effective address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.

The '#imm' in the assembly syntax represents the immediate  $i$  after zero extension.

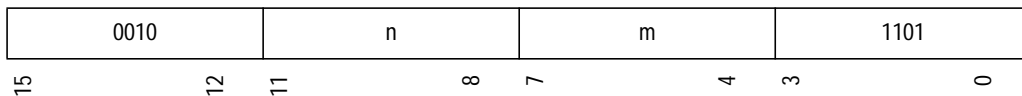
# XTRCT Rm, Rn

## Description

This instruction extracts the lower 16-bit word from  $R_m$  and the upper 16-bit word from  $R_n$ , swaps their order, and places the result in  $R_n$ . Bits [0,15] of  $R_n$  take the value of bits [16,31] of the original  $R_n$ . Bits [16,31] of  $R_n$  take the value of bits [0,15] of  $R_m$ .

## Operation

**XTRCT Rm, Rn**



```

op1 ← ZeroExtend32(Rm);
op2 ← ZeroExtend32(Rn);
op2 ← op2<16 FOR 16 > ∨ (op1<0 FOR 16 > << 16);
Rn ← Register(op2);

```

## Note



# 10

## Pipelining

The SH-4 CPU core is a dual-issue superscalar pipelining microprocessor. This section gives a high-level description of the way in which this particular implementation of the SH4 architecture executes instructions. Definitions in this section may not be applicable to SH-4 Series models other than the SH-4 CPU core.

### 10.1 Pipelines

*Figure 33* shows the basic pipelines. Normally, a pipeline consists of five or six stages: instruction fetch (I), decode and register read (D), execution (EX/SX/F0/F1/F2/F3), data access (NA/MA), and write-back (S/FS). An instruction is executed as a combination of basic pipelines. *Figure 34* to *Figure 38* show the instruction execution patterns.

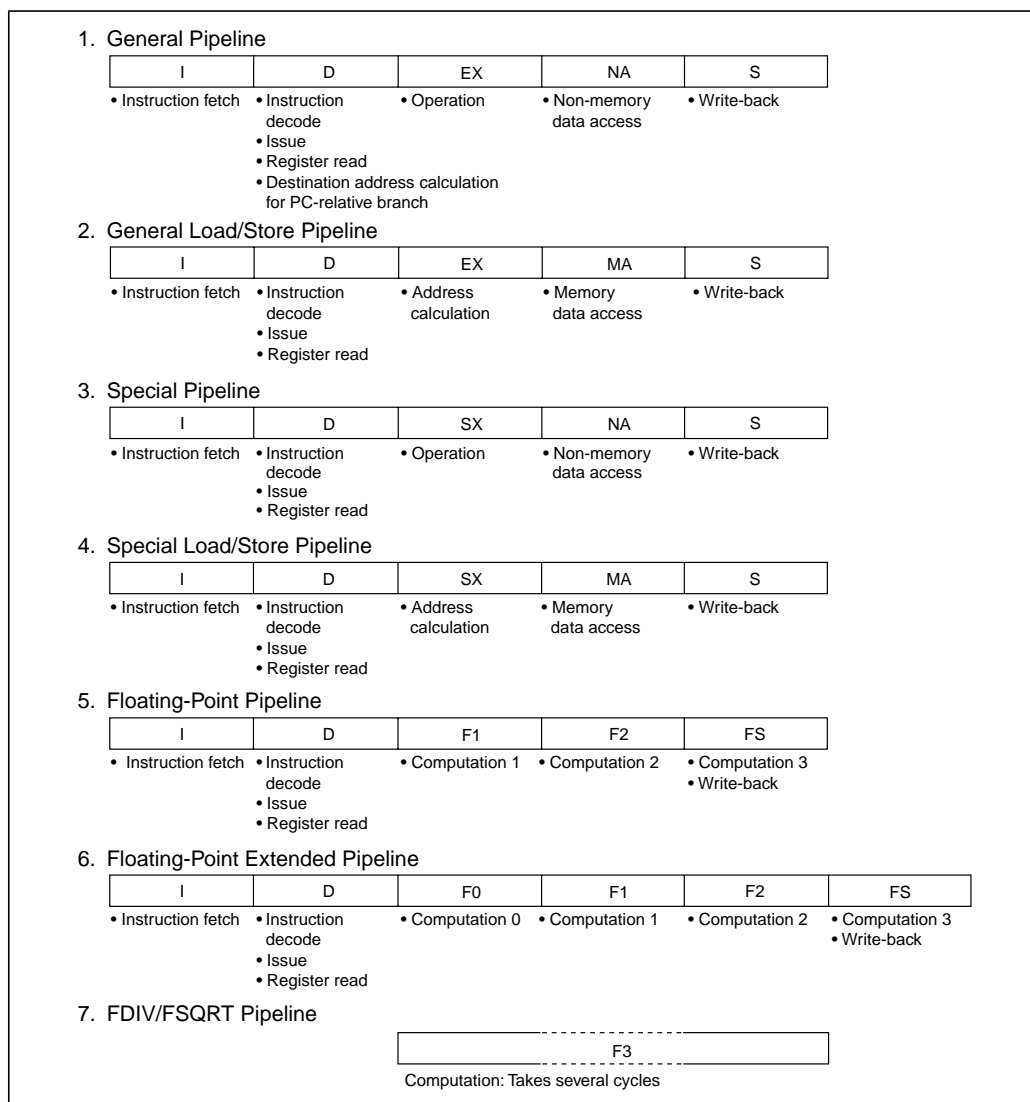


Figure 33: Basic pipelines

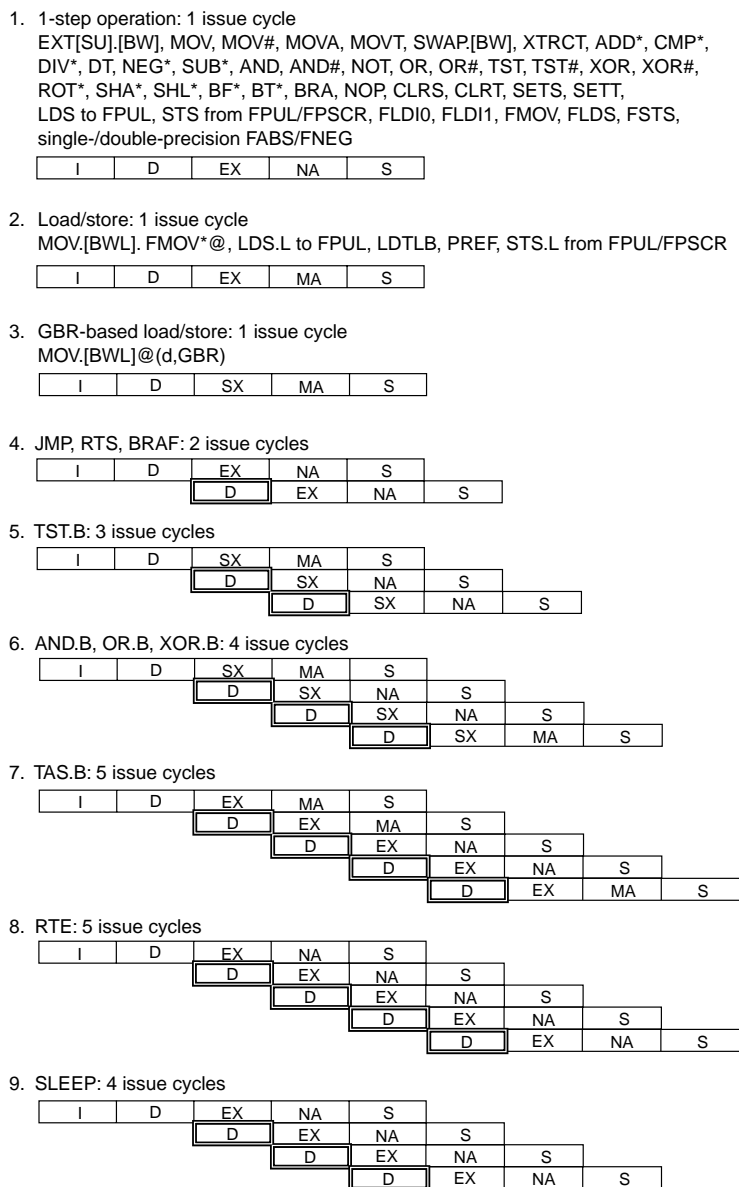


Figure 34: Instruction execution patterns

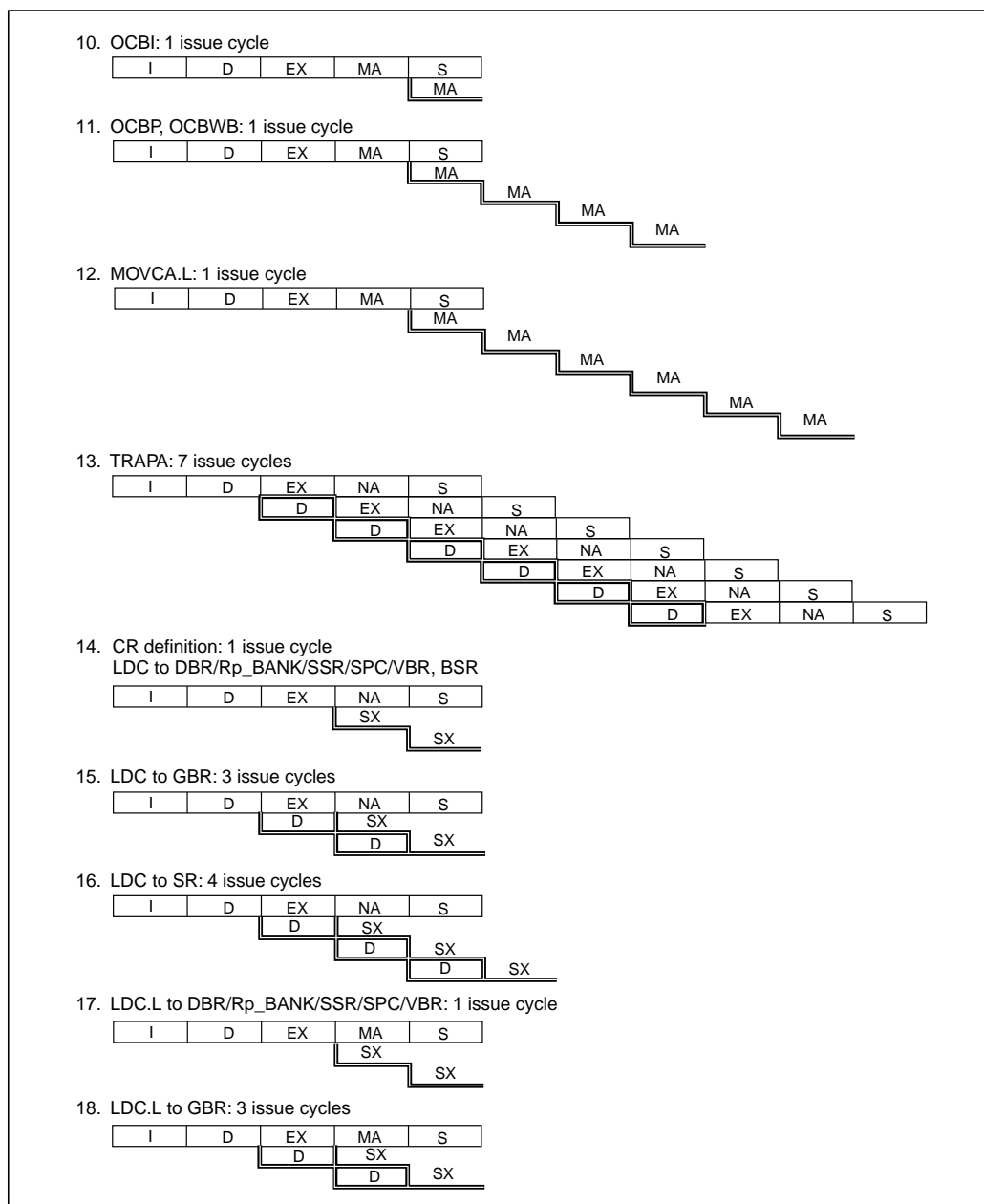


Figure 35: Instruction execution patterns (continued)

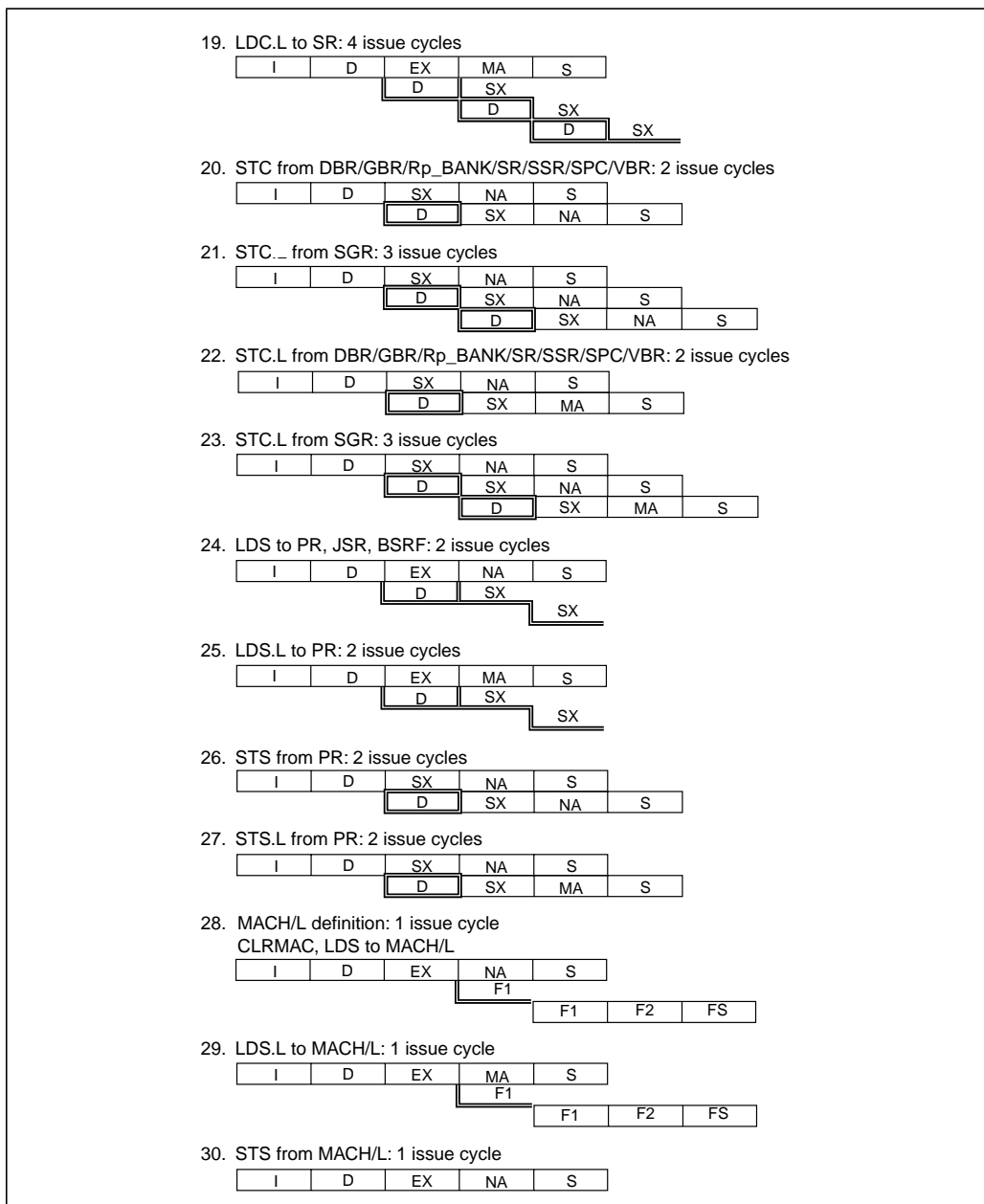
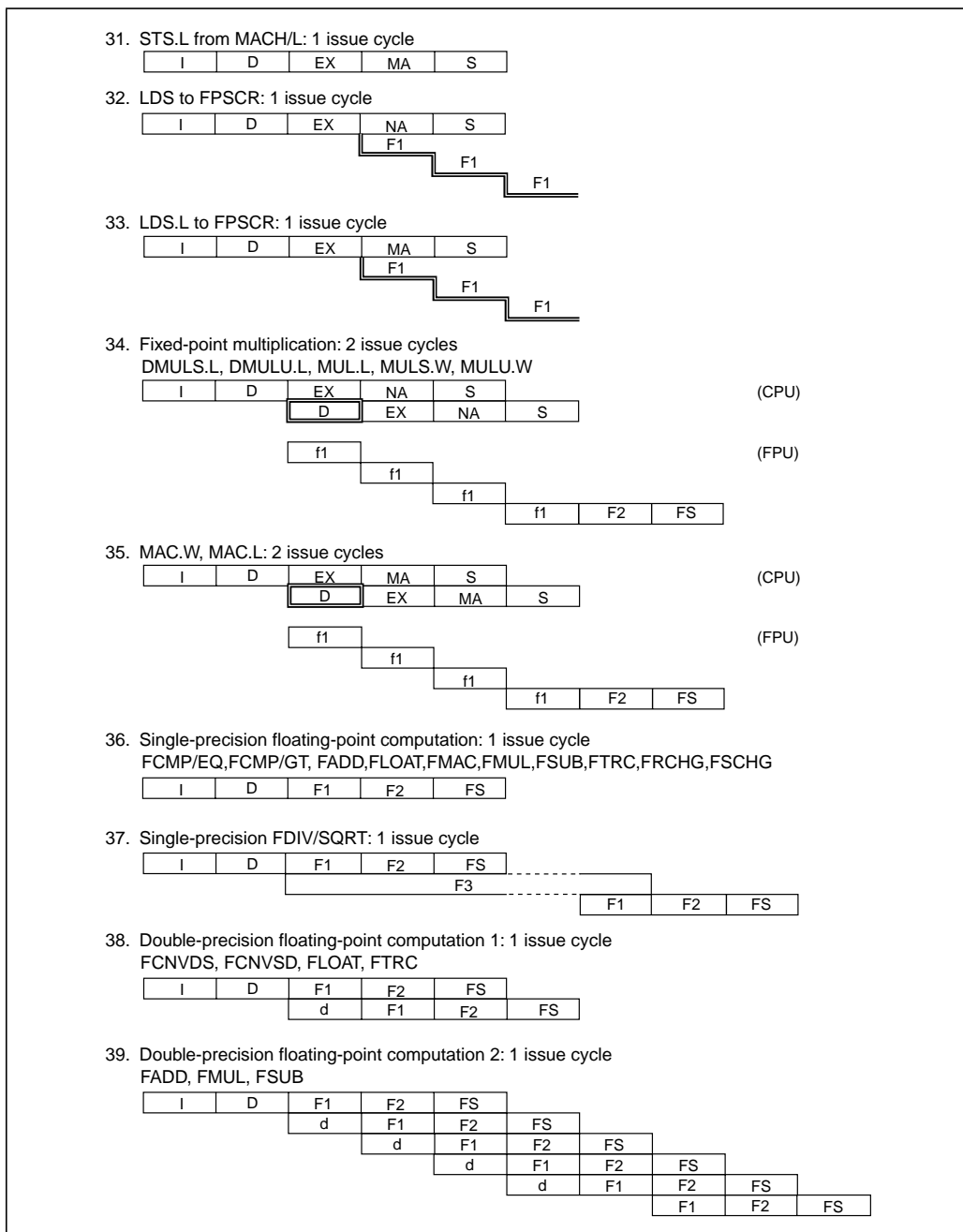


Figure 36: Instruction execution patterns (continued)





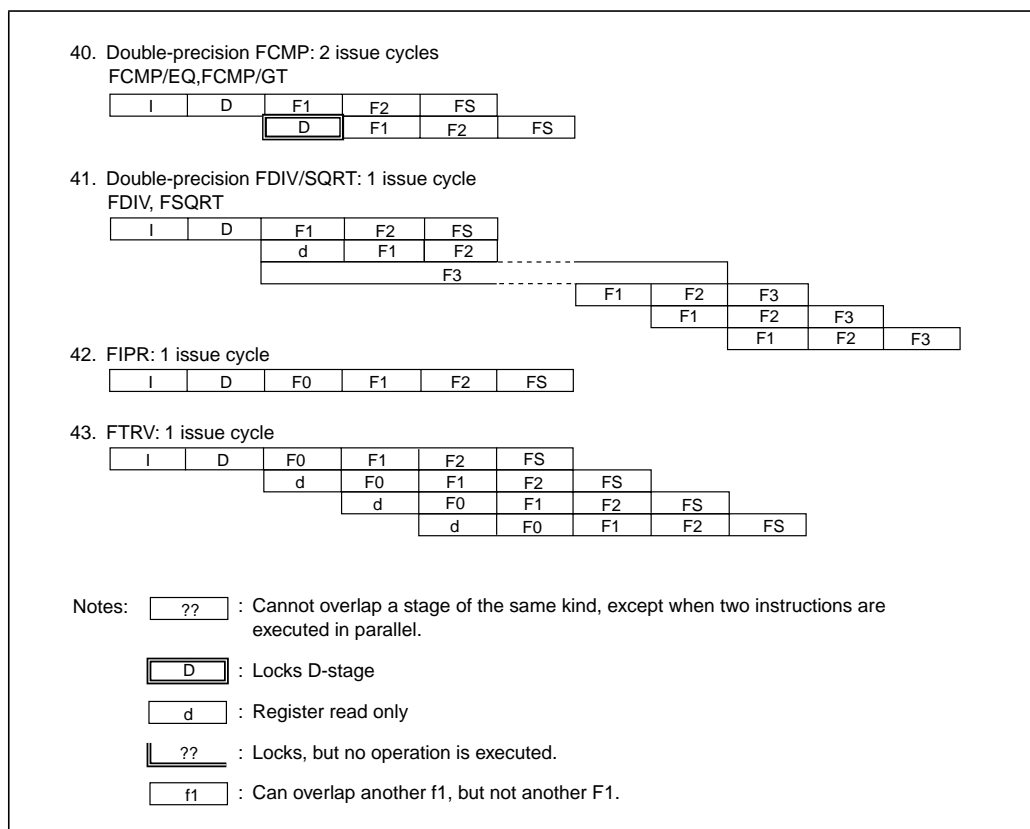


Figure 38: Instruction execution patterns (continued)

## 10.2 Parallel executables

Instructions are categorized into six groups according to the internal function blocks used, as shown in table 8.1. Table 8.2 shows the parallel executable pairs of instructions in terms of groups. For example, ADD in the EX group and BRA in the BR group can be executed in parallel.

### 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

Table 74: Instruction groups

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,@-Rn	MOV.L	@(disp,GBR),R0	OCBP	@Rn

**Table 74: Instruction groups**

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		

**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
CLRMACH		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

Table 74: Instruction groups

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 74: Instruction groups

		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

Table 75: Parallel executables

O: Can be executed in parallel

X: Cannot be executed in parallel

## 10.3 Execution cycles and pipeline stalling

Instruction execution cycles are summarized in *Table 76: Execution cycles on page 501*. Penalty cycles due to a pipeline stall or freeze are not considered in this table.

- Issue rate: Interval between the issue of an instruction and that of the next instruction
- Latency: Interval between the issue of an instruction and the generation of its result (completion)
- Instruction execution pattern (see *Figure 34* to *Figure 38*)
- Locked pipeline stages
- Interval between the issue of an instruction and the start of locking
- Lock time: Period of locking in machine cycle units

The instruction execution sequence is expressed as a combination of the execution patterns shown in [Figure 34](#) to [Figure 38](#). One instruction is separated from the next by the number of machine cycles for its issue rate. Normally, execution, data access, and write-back stages cannot be overlapped onto the same stages of another instruction; the only exception is when two instructions are executed in parallel under parallel executables conditions. Refer to (a) through (d) in [Figure 39](#) for some simple examples.

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

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

In the case of flow dependency, latency may be exceptionally increased or decreased, depending on the combination of sequential instructions ([Figure 40](#) (e)).

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

The number of cycles in a pipeline stall due to flow dependency will vary depending on the combination of interdependent instructions or the fetch timing (see [Figure 40](#) (e)).

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

For the stall cycles of an instruction with output dependency, the longest latency to the last write-back among all the destination operands must be applied instead of latency-2 (see [Figure 41 \(f\)](#)). A stall due to output dependency with respect to FPSCR, which reflects the result of an FP operation, never occurs. For example, when FADD follows FDIV with no dependency between FP registers, FADD is not stalled even if both instructions update the cause field of FPSCR.

Anti-flow dependency can occur only between a preceding double-precision FADD, FMUL, FSUB, or FTRV and a following FMOV, FLDI0, FLDI1, FABS, FNEG, or FSTS. See [Figure 41 \(g\)](#).

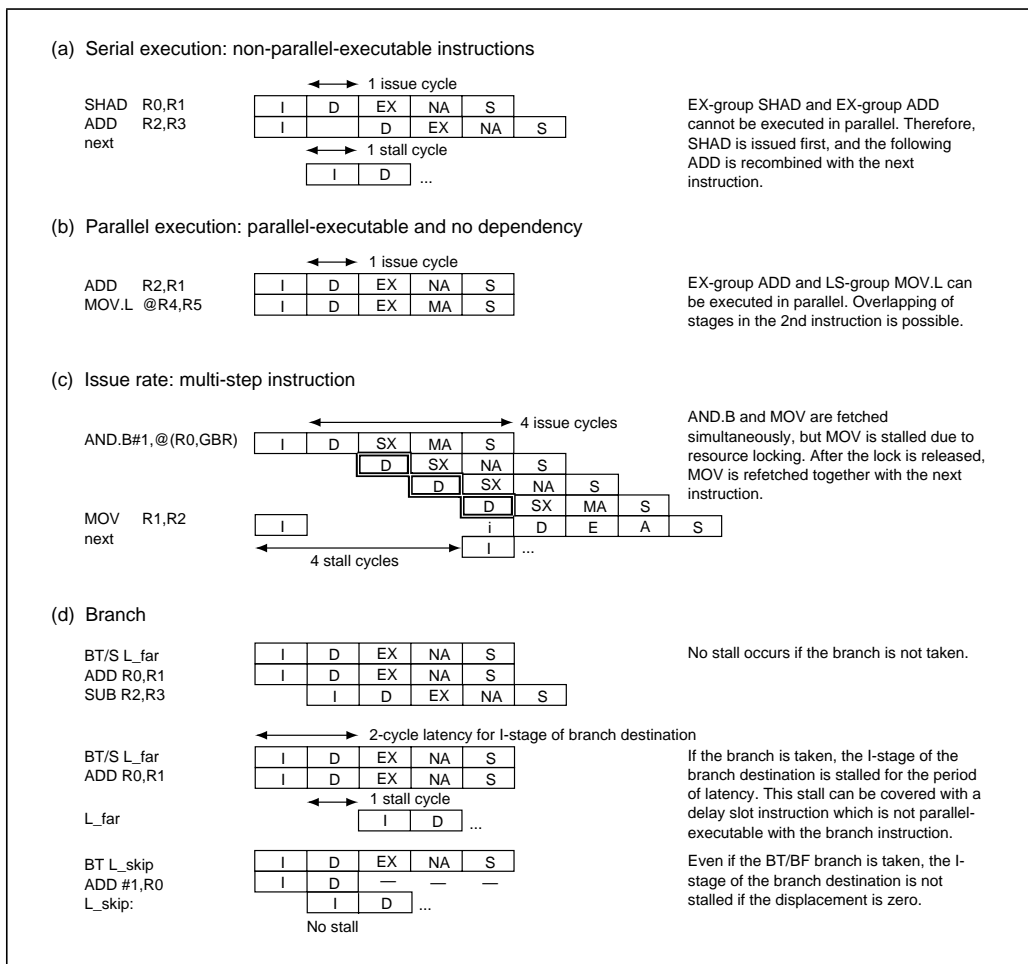
If an executing instruction locks any resource, i.e. a function block that performs a basic operation, a following instruction that happens to attempt to use the locked resource must be stalled ([Figure 42 \(h\)](#)). This kind of stall can be compensated by inserting one or more instructions independent of the locked resource to separate the interfering instructions. For example, when a load instruction and an ADD instruction that references the loaded value are consecutive, the 2-cycle stall of the ADD is eliminated by inserting three instructions without dependency. Software performance can be improved by such instruction scheduling.

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

- Instruction TLB miss: a penalty of 7 CPU clocks
- Instruction access to external memory (instruction cache miss, etc.)
- Data access to external memory (operand cache miss, etc.)
- Data access to a memory-mapped control register. The penalty differs from register to register, and depends on the kind of operation (read or write), the clock mode, and the bus use conditions when the access is made.

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





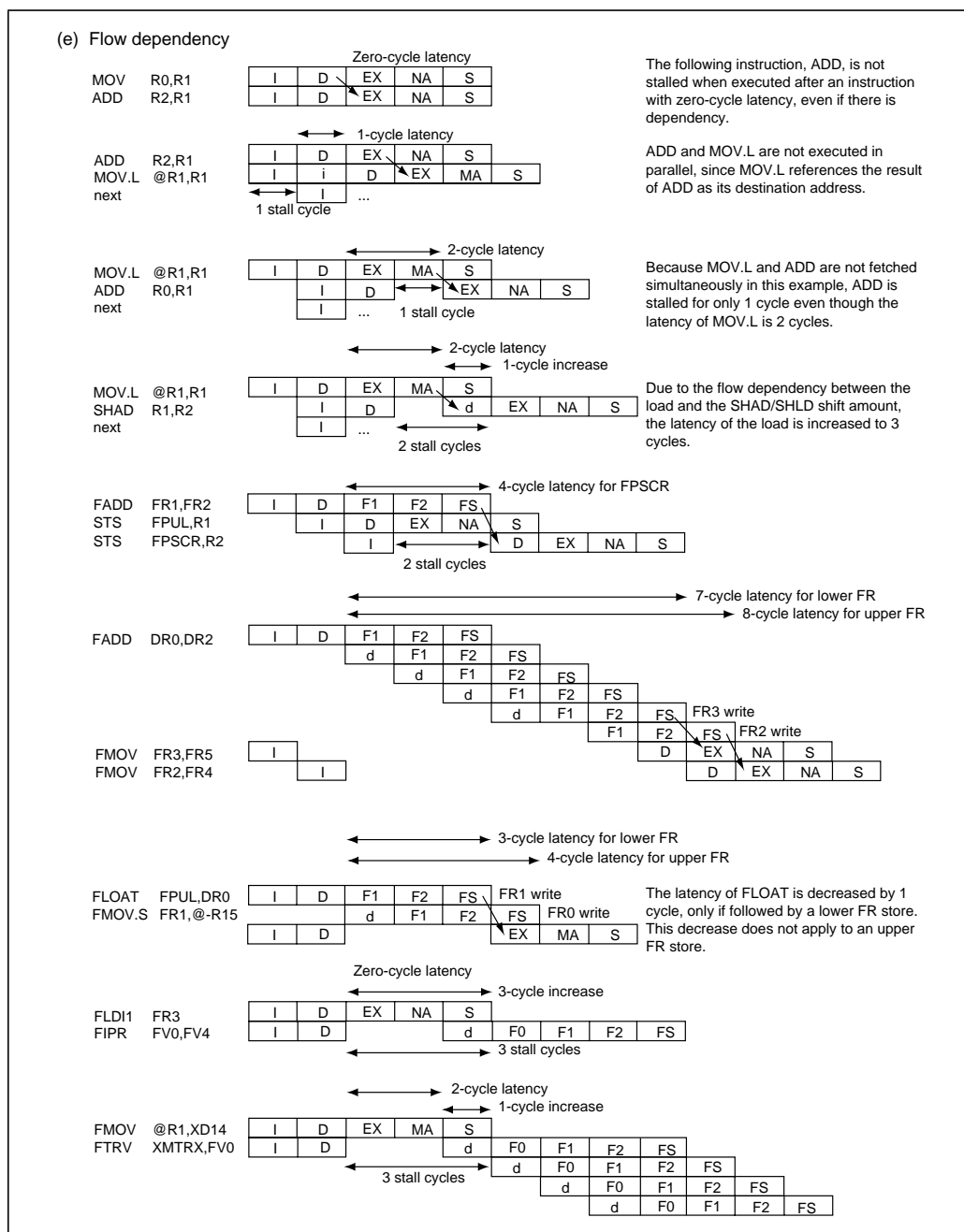


Figure 40: Examples of pipelined execution (continued)



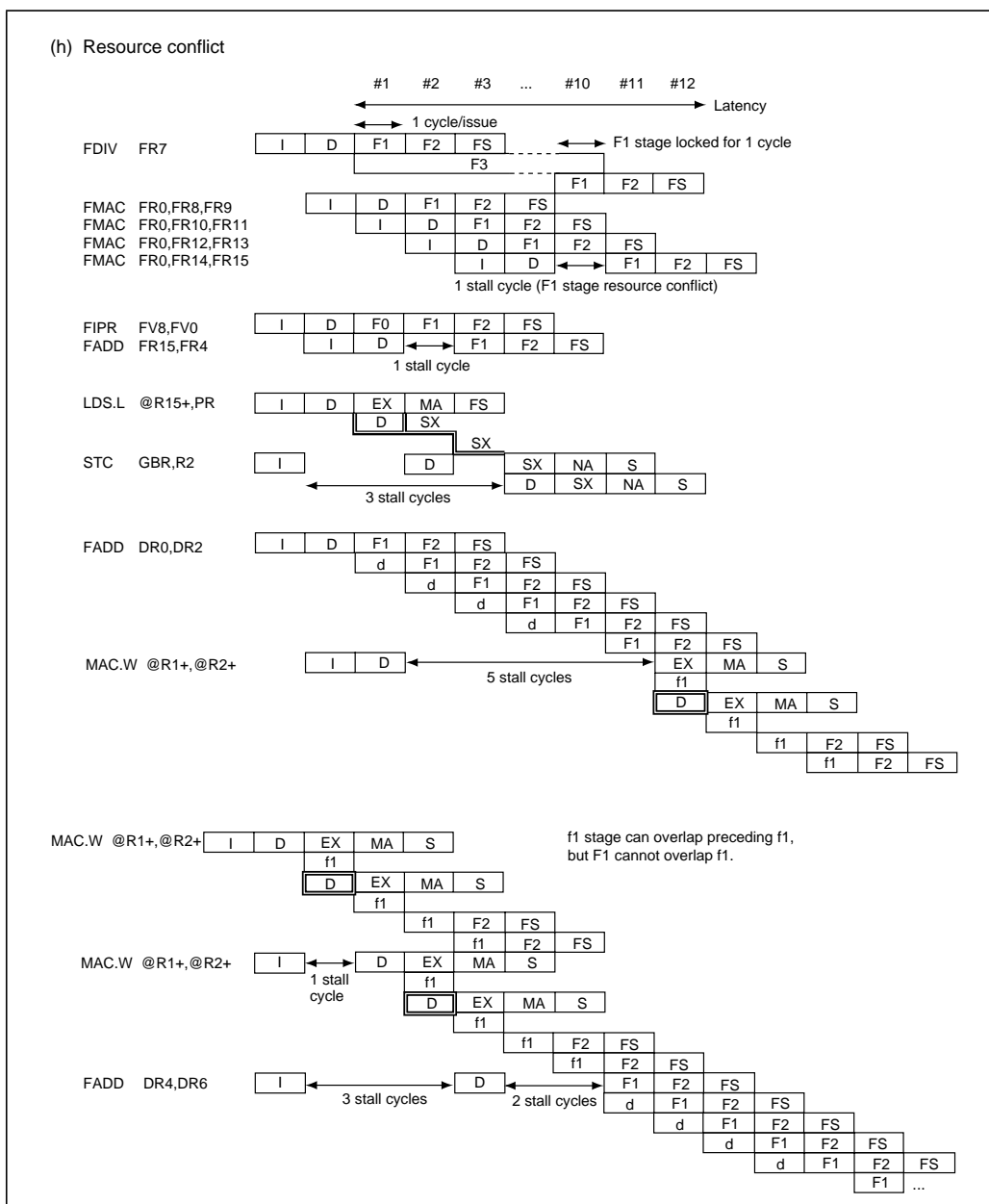


Figure 42: Examples of pipelined execution (continued)

Functional category	No	Instruction		Instruction group	Issue rate	Latency	Execution pattern	Lock		
								Stage	Start	Cycles
Data transfer instructions	1	EXTS.B	Rm,Rn	EX	1	1	#1	-	-	-
	2	EXTS.W	Rm,Rn	EX	1	1	#1	-	-	-
	3	EXTU.B	Rm,Rn	EX	1	1	#1	-	-	-
	4	EXTU.W	Rm,Rn	EX	1	1	#1	-	-	-
	5	MOV	Rm,Rn	MT	1	0	#1	-	-	-
	6	MOV	#imm,Rn	EX	1	1	#1	-	-	-
	7	MOVA	@(disp,PC),R0	EX	1	1	#1	-	-	-
	8	MOV.W	@(disp,PC),Rn	LS	1	2	#2	-	-	-
	9	MOV.L	@(disp,PC),Rn	LS	1	2	#2	-	-	-
	10	MOV.B	@Rm,Rn	LS	1	2	#2	-	-	-
	11	MOV.W	@Rm,Rn	LS	1	2	#2	-	-	-
	12	MOV.L	@Rm,Rn	LS	1	2	#2	-	-	-
	13	MOV.B	@Rm+,Rn	LS	1	1/2	#2	-	-	-
	14	MOV.W	@Rm+,Rn	LS	1	1/2	#2	-	-	-
	15	MOV.L	@Rm+,Rn	LS	1	1/2	#2	-	-	-
	16	MOV.B	@(disp,Rm),R0	LS	1	2	#2	-	-	-
	17	MOV.W	@(disp,Rm),R0	LS	1	2	#2	-	-	-
	18	MOV.L	@(disp,Rm),Rn	LS	1	2	#2	-	-	-
	19	MOV.B	@(R0,Rm),Rn	LS	1	2	#2	-	-	-
	20	MOV.W	@(R0,Rm),Rn	LS	1	2	#2	-	-	-
	21	MOV.L	@(R0,Rm),Rn	LS	1	2	#2	-	-	-
	22	MOV.B	@(disp,GBR),R0	LS	1	2	#3	-	-	-
	23	MOV.W	@(disp,GBR),R0	LS	1	2	#3	-	-	-

Table 76: Execution cycles

Functional category	No	Instruction		Instruction group	Issue rate	Latency	Execution pattern	Lock		
								Stage	Start	Cycles
Data transfer instructions	24	MOV.L	@(disp,GBR),R0	LS	1	2	#3	-	-	-
	25	MOV.B	Rm,@Rn	LS	1	1	#2	-	-	-
	26	MOV.W	Rm,@Rn	LS	1	1	#2	-	-	-
	27	MOV.L	Rm,@Rn	LS	1	1	#2	-	-	-
	28	MOV.B	Rm,@-Rn	LS	1	1/1	#2	-	-	-
	29	MOV.W	Rm,@-Rn	LS	1	1/1	#2	-	-	-
	30	MOV.L	Rm,@-Rn	LS	1	1/1	#2	-	-	-
	31	MOV.B	R0,@(disp,Rn)	LS	1	1	#2	-	-	-
	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	MOV.T	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	-	-	-

Table 76: Execution cycles

Functional category	No	Instruction		Instruction group	Issue rate	Latency	Execution pattern	Lock		
								Stage	Start	Cycles
Data transfer instructions	47	SWAP.W	Rm,Rn	EX	1	1	#1	-	-	-
	48	XTRCT	Rm,Rn	EX	1	1	#1	-	-	-
Fixed-point arithmetic instructions	49	ADD	Rm,Rn	EX	1	1	#1	-	-	-
	50	ADD	#imm,Rn	EX	1	1	#1	-	-	-
	51	ADDC	Rm,Rn	EX	1	1	#1	-	-	-
	52	ADDV	Rm,Rn	EX	1	1	#1	-	-	-
	53	CMP/EQ	#imm,R0	MT	1	1	#1	-	-	-
	54	CMP/EQ	Rm,Rn	MT	1	1	#1	-	-	-
	55	CMP/GE	Rm,Rn	MT	1	1	#1	-	-	-
	56	CMP/GT	Rm,Rn	MT	1	1	#1	-	-	-
	57	CMP/HI	Rm,Rn	MT	1	1	#1	-	-	-
	58	CMP/HS	Rm,Rn	MT	1	1	#1	-	-	-
	59	CMP/PL	Rn	MT	1	1	#1	-	-	-
	60	CMP/PZ	Rn	MT	1	1	#1	-	-	-
	61	CMP/STR	Rm,Rn	MT	1	1	#1	-	-	-
	62	DIV0S	Rm,Rn	EX	1	1	#1	-	-	-
	63	DIV0U		EX	1	1	#1	-	-	-
	64	DIV1	Rm,Rn	EX	1	1	#1	-	-	-
	65	DMULS.L	Rm,Rn	CO	2	4/4	#34	F1	4	2
	66	DMULU.L	Rm,Rn	CO	2	4/4	#34	F1	4	2
	67	DT	Rn	EX	1	1	#1	-	-	-
	68	MAC.L	@Rm+,@Rn+	CO	2	2/2/4/4	#35	F1	4	2
	69	MAC.W	@Rm+,@Rn+	CO	2	2/2/4/4	#35	F1	4	2

Table 76: Execution cycles

Functional category	No	Instruction		Instruction group	Issue rate	Latency	Execution pattern	Lock		
								Stage	Start	Cycles
Fixed-point arithmetic instructions	70	MUL.L	Rm,Rn	CO	2	4/4	#34	F1	4	2
	71	MULS.W	Rm,Rn	CO	2	4/4	#34	F1	4	2
	72	MULU.W	Rm,Rn	CO	2	4/4	#34	F1	4	2
	73	NEG	Rm,Rn	EX	1	1	#1	-	-	-
	74	NEGC	Rm,Rn	EX	1	1	#1	-	-	-
	75	SUB	Rm,Rn	EX	1	1	#1	-	-	-
	76	SUBC	Rm,Rn	EX	1	1	#1	-	-	-
	77	SUBV	Rm,Rn	EX	1	1	#1	-	-	-
Logical instructions	78	AND	Rm,Rn	EX	1	1	#1	-	-	-
	79	AND	#imm,R0	EX	1	1	#1	-	-	-
	80	AND.B	#imm,@(R0,GBR)	CO	4	4	#6	-	-	-
	81	NOT	Rm,Rn	EX	1	1	#1	-	-	-
	82	OR	Rm,Rn	EX	1	1	#1	-	-	-
	83	OR	#imm,R0	EX	1	1	#1	-	-	-
	84	OR.B	#imm,@(R0,GBR)	CO	4	4	#6	-	-	-
	85	TAS.B	@Rn	CO	5	5	#7	-	-	-
	86	TST	Rm,Rn	MT	1	1	#1	-	-	-
	87	TST	#imm,R0	MT	1	1	#1	-	-	-
	88	TST.B	#imm,@(R0,GBR)	CO	3	3	#5	-	-	-
	89	XOR	Rm,Rn	EX	1	1	#1	-	-	-
	90	XOR	#imm,R0	EX	1	1	#1	-	-	-
	91	XOR.B	#imm,@(R0,GBR)	CO	4	4	#6	-	-	-

Table 76: Execution cycles



Functional category	No	Instruction		Instruction group	Issue rate	Latency	Execution pattern	Lock		
								Stage	Start	Cycles
Shift instructions	92	ROTL	Rn	EX	1	1	#1	-	-	-
	93	ROTR	Rn	EX	1	1	#1	-	-	-
	94	ROTCL	Rn	EX	1	1	#1	-	-	-
	95	ROTCR	Rn	EX	1	1	#1	-	-	-
	96	SHAD	Rm,Rn	EX	1	1	#1	-	-	-
	97	SHAL	Rn	EX	1	1	#1	-	-	-
	98	SHAR	Rn	EX	1	1	#1	-	-	-
	99	SHLD	Rm,Rn	EX	1	1	#1	-	-	-
	100	SHLL	Rn	EX	1	1	#1	-	-	-
	101	SHLL2	Rn	EX	1	1	#1	-	-	-
	102	SHLL8	Rn	EX	1	1	#1	-	-	-
	103	SHLL16	Rn	EX	1	1	#1	-	-	-
	104	SHLR	Rn	EX	1	1	#1	-	-	-
	105	SHLR2	Rn	EX	1	1	#1	-	-	-
	106	SHLR8	Rn	EX	1	1	#1	-	-	-
	107	SHLR16	Rn	EX	1	1	#1	-	-	-
Branch instructions	108	BF	disp	BR	1	2 (or 1)	#1	-	-	-
	109	BF/S	disp	BR	1	2 (or 1)	#1	-	-	-
	110	BT	disp	BR	1	2 (or 1)	#1	-	-	-
	111	BT/S	disp	BR	1	2 (or 1)	#1	-	-	-
	112	BRA	disp	BR	1	2	#1	-	-	-
	113	BRAF	Rn	CO	2	3	#4	-	-	-
	114	BSR	disp	BR	1	2	#14	SX	3	2

Table 76: Execution cycles

Functional category	No	Instruction		Instruction group	Issue rate	Latency	Execution pattern	Lock		
								Stage	Start	Cycles
Branch instructions	115	BSRF	Rn	CO	2	3	#24	SX	3	2
	116	JMP	@Rn	CO	2	3	#4	-	-	-
	117	JSR	@Rn	CO	2	3	#24	SX	3	2
	118	RTS		CO	2	3	#4	-	-	-
System control instructions	119	NOP		MT	1	0	#1	-	-	-
	120	CLRMAC		CO	1	3	#28	F1	3	2
	121	CLRS		CO	1	1	#1	-	-	-
	122	CLRT		MT	1	1	#1	-	-	-
	123	SETS		CO	1	1	#1	-	-	-
	124	SETT		MT	1	1	#1	-	-	-
	125	TRAPA	#imm	CO	7	7	#13	-	-	-
	126	RTE		CO	5	5	#8	-	-	-
	127	SLEEP		CO	4	4	#9	-	-	-
	128	LDTLB		CO	1	1	#2	-	-	-
	129	LDC	Rm,DBR	CO	1	3	#14	SX	3	2
	130	LDC	Rm,GBR	CO	3	3	#15	SX	3	2
	131	LDC	Rm,Rp_BANK	CO	1	3	#14	SX	3	2
	132	LDC	Rm,SR	CO	4	4	#16	SX	3	2
	133	LDC	Rm,SSR	CO	1	3	#14	SX	3	2
	134	LDC	Rm,SPC	CO	1	3	#14	SX	3	2
	135	LDC	Rm,VBR	CO	1	3	#14	SX	3	2
	136	LDC.L	@Rm+,DBR	CO	1	1/3	#17	SX	3	2
	137	LDC.L	@Rm+,GBR	CO	3	3/3	#18	SX	3	2

Table 76: Execution cycles

Functional category	No	Instruction		Instruction group	Issue rate	Latency	Execution pattern	Lock		
								Stage	Start	Cycles
	138	LDC.L	@Rm+,Rp_BANK	CO	1	1/3	#17	SX	3	2
	139	LDC.L	@Rm+,SR	CO	4	4/4	#19	SX	3	2
	140	LDC.L	@Rm+,SSR	CO	1	1/3	#17	SX	3	2
	141	LDC.L	@Rm+,SPC	CO	1	1/3	#17	SX	3	2
	142	LDC.L	@Rm+,VBR	CO	1	1/3	#17	SX	3	2
	143	LDS	Rm,MACH	CO	1	3	#28	F1	3	2
	144	LDS	Rm,MACL	CO	1	3	#28	F1	3	2
	145	LDS	Rm,PR	CO	2	3	#24	SX	3	2
	146	LDS.L	@Rm+,MACH	CO	1	1/3	#29	F1	3	2
	147	LDS.L	@Rm+,MACL	CO	1	1/3	#29	F1	3	2
	148	LDS.L	@Rm+,PR	CO	2	2/3	#25	SX	3	2
	149	STC	DBR,Rn	CO	2	2	#20	-	-	-
	150	STC	SGR,Rn	CO	3	3	#21	-	-	-
	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	-	-	-

Table 76: Execution cycles

Functional category	No	Instruction		Instruction group	Issue rate	Latency	Execution pattern	Lock		
								Stage	Start	Cycles
	161	STC.L	SR,@-Rn	CO	2	2/2	#22	-	-	-
	162	STC.L	SSR,@-Rn	CO	2	2/2	#22	-	-	-
	163	STC.L	SPC,@-Rn	CO	2	2/2	#22	-	-	-
	164	STC.L	VBR,@-Rn	CO	2	2/2	#22	-	-	-
	165	STS	MACH,Rn	CO	1	3	#30	-	-	-
	166	STS	MACL,Rn	CO	1	3	#30	-	-	-
	167	STS	PR,Rn	CO	2	2	#26	-	-	-
	168	STS.L	MACH,@-Rn	CO	1	1/1	#31	-	-	-
	169	STS.L	MACL,@-Rn	CO	1	1/1	#31	-	-	-
	170	STS.L	PR,@-Rn	CO	2	2/2	#27	-	-	-
Single-precision floating-point instructions	171	FLDI0	FRn	LS	1	0	#1	-	-	-
	172	FLDI1	FRn	LS	1	0	#1	-	-	-
	173	FMOV	FRm,FRn	LS	1	0	#1	-	-	-
	174	FMOV.S	@Rm,FRn	LS	1	2	#2	-	-	-
	175	FMOV.S	@Rm+,FRn	LS	1	1/2	#2	-	-	-
	176	FMOV.S	@(R0,Rm),FRn	LS	1	2	#2	-	-	-
	177	FMOV.S	FRm,@Rn	LS	1	1	#2	-	-	-
	178	FMOV.S	FRm,@-Rn	LS	1	1/1	#2	-	-	-
	179	FMOV.S	FRm,@(R0,Rn)	LS	1	1	#2	-	-	-
	180	FLDS	FRm,FPUL	LS	1	0	#1	-	-	-
	181	FSTS	FPUL,FRn	LS	1	0	#1	-	-	-
	182	FABS	FRn	LS	1	0	#1	-	-	-
	183	FADD	FRm,FRn	FE	1	3/4	#36	-	-	-

Table 76: Execution cycles

Functional category	No	Instruction		Instruction group	Issue rate	Latency	Execution pattern	Lock		
								Stage	Start	Cycles
	184	FCMP/EQ	FRm,FRn	FE	1	2/4	#36	-	-	-
	185	FCMP/GT	FRm,FRn	FE	1	2/4	#36	-	-	-
	186	FDIV	FRm,FRn	FE	1	12/13	#37	F3	2	10
								F1	11	1
	187	FLOAT	FPUL,FRn	FE	1	3/4	#36	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
	192	FSUB	FRm,FRn	FE	1	3/4	#36	-	-	-
	193	FTRC	FRm,FPUL	FE	1	3/4	#36	-	-	-
	194	FMOV	DRm,DRn	LS	1	0	#1	-	-	-
	195	FMOV	@Rm,DRn	LS	1	2	#2	-	-	-
	196	FMOV	@Rm+,DRn	LS	1	1/2	#2	-	-	-
	197	FMOV	@(R0,Rm),DRn	LS	1	2	#2	-	-	-
	198	FMOV	DRm,@Rn	LS	1	1	#2	-	-	-
	199	FMOV	DRm,@-Rn	LS	1	1/1	#2	-	-	-
	200	FMOV	DRm,@(R0,Rn)	LS	1	1	#2	-	-	-
Double-precision floating-point instructions	201	FABS	DRn	LS	1	0	#1	-	-	-
	202	FADD	DRm,DRn	FE	1	(7, 8)/9	#39	F1	2	6
	203	FCMP/EQ	DRm,DRn	CO	2	3/5	#40	F1	2	2
	204	FCMP/GT	DRm,DRn	CO	2	3/5	#40	F1	2	2

Table 76: Execution cycles

Functional category	No	Instruction		Instruction group	Issue rate	Latency	Execution pattern	Lock		
								Stage	Start	Cycles
	205	FCNVDS	DRm,FPUL	FE	1	4/5	#38	F1	2	2
	206	FCNVSD	FPUL,DRn	FE	1	(3, 4)/5	#38	F1	2	2
	207	FDIV	DRm,DRn	FE	1	(24, 25)/26	#41	F3	2	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	1	(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 instructions	214	LDS	Rm,FPUL	LS	1	1	#1	-	-	-
	215	LDS	Rm,FPSCR	CO	1	4	#32	F1	3	3
	216	LDS.L	@Rm+,FPUL	CO	1	1/2	#2	-	-	-
	217	LDS.L	@Rm+,FPSCR	CO	1	1/4	#33	F1	3	3
	218	STS	FPUL,Rn	LS	1	3	#1	-	-	-
	219	STS	FPSCR,Rn	CO	1	3	#1	-	-	-
	220	STS.L	FPUL,@-Rn	CO	1	1/1	#2	-	-	-
	221	STS.L	FPSCR,@-Rn	CO	1	1/1	#2	-	-	-
Graphics acceleration instructions	222	FMOV	DRm,XDn	LS	1	0	#1	-	-	-
	223	FMOV	XDm,DRn	LS	1	0	#1	-	-	-
	224	FMOV	XDm,XDn	LS	1	0	#1	-	-	-

Table 76: Execution cycles

Functional category	No	Instruction		Instruction group	Issue rate	Latency	Execution pattern	Lock		
								Stage	Start	Cycles
	225	FMOV	@Rm, XDn	LS	1	2	#2	-	-	-
	226	FMOV	@Rm+, XDn	LS	1	1/2	#2	-	-	-
	227	FMOV	@(R0, Rm), XDn	LS	1	2	#2	-	-	-
	228	FMOV	XDm, @Rn	LS	1	1	#2	-	-	-
	229	FMOV	XDm, @-Rm	LS	1	1/1	#2	-	-	-
	230	FMOV	XDm, @(R0, Rn)	LS	1	1	#2	-	-	-
	231	FIPR	FVm, FVn	FE	1	4/5	#42	F1	3	1
	232	FRCHG		FE	1	1/4	#36	-	-	-
	233	FSCHG		FE	1	1/4	#36	-	-	-
	234	FTRV	XMTRX, FVn	FE	1	(5, 5, 6, 7)/8	#43	F0	2	4
								F1	3	4

Table 76: Execution cycles

Note: 1 See [Table 74](#) for the instruction groups.

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

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

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

4 Conditional branch latency “2 (or 1)”: The latency is 2 for a nonzero displacement, and 1 for a zero displacement.

5 Double-precision floating-point instruction latency “(L1, L2)/L3”: L1 is the latency for FR [n+1], L2 that for FR [n], and L3 that for FPSCR.

6 FTRV latency “(L1, L2, L3, L4)/L5”: L1 is the latency for FR [n], L2 that for FR [n+1], L3 that for FR [n+2], L4 that for FR [n+3], and L5 that for FPSCR.

7 Latency “L1/L2/L3/L4” of MAC.L and MAC.W instructions: L1 is the latency for Rm, L2 that for Rn, L3 that for MACH, and L4 that for MACL.

8 Latency “L1/L2” of MUL.L, MULS.W, MULU.W, DMULS.L, and DMULU.L instructions: L1 is the latency for MACH, and L2 that for MACL.

9 Execution pattern: The instruction execution pattern number (see figure 8.2)

10 Lock/stage: Stage locked by the instruction

11 Lock/start: Locking start cycle; 1 is the first D-stage of the instruction.

12 Lock/cycles: Number of cycles locked.

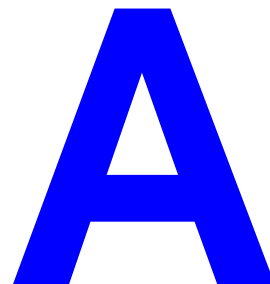
### Exceptions:

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

Example: In the case of FMOV FR4,FR0 and FIPR FV0,FV4, FIPR is stalled for 2 cycles.

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





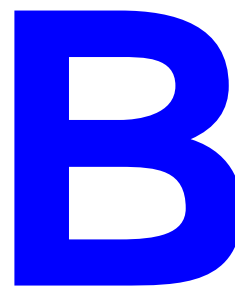
# Address list

Module	Register	P4 address	Area 7 address <sup>a</sup>	Size	Power-on reset	Manual reset	Sleep	Standby	Sync clock
CCN	PTEH	0xFF00 0000	0x1F00 0000	32	0x0000 0000	0x0000 0000	Held	Held	Iclk
CCN	PTEL	0xFF00 0004	0x1F00 0004	32	0x0000 0000	0x0000 0000	Held	Held	Iclk
CCN	TTB	0xFF00 0008	0x1F00 0008	32	0x0000 0000	0x0000 0000	Held	Held	Iclk
CCN	TEA	0xFF00 000C	0x1F00 000C	32	0x0000 0000	0x0000 0000	Held	Held	Iclk
CCN	MMUCR	0xFF00 0010	0x1F00 0010	32	0x0000 0000	0x0000 0000	Held	Held	Iclk
CCN	BASRA	0xFF00 0014	0x1F00 0014	8	Undefined	Held	Held	Held	Iclk
CCN	BASRB	0xFF00 0018	0x1F00 0018	8	Undefined	Held	Held	Held	Iclk
CCN	CCR	0xFF00 001C	0x1F00 001C	32	0x0000 0000	0x0000 0000	Held	Held	Iclk
CCN	TRA	0xFF00 0020	0x1F00 0020	32	0x0000 0000	0x0000 0000	Held	Held	Iclk
CCN	EXPEVT	0xFF00 0024	0x1F00 0024	32	0x0000 0000	0x0000 0020	Held	Held	Iclk
CCN	INTEVT	0xFF00 0028	0x1F00 0028	32	0x0000 0000	Held	Held	Held	Iclk
CCN	QACR0	0xFF00 0038	0x1F00 0038	32	Undefined	Undefined	Held	Held	Iclk
CCN	QACR1	0xFF00 003C	0x1F00 003C	32	Undefined	Undefined	Held	Held	Iclk
UBC	BARA	0xFF20 0000	0x1F20 0000	32	Undefined	Held	Held	Held	Iclk
UBC	BAMRA	0xFF20 0004	0x1F20 0004	8	Undefined	Held	Held	Held	Iclk

Table 77: Address list

- a. With control registers, the above addresses in the physical page number field can be accessed by means of a TLB setting. When these addresses are referenced directly without using the TLB, operations are limited.

*Note: The address map for peripheral devices is contained in the system manual for the part.*



# Instruction prefetch side effects

The SH-4 is provided with an internal buffer for holding pre-read instructions, and always performs pre-reading. Therefore, program code must not be located in the last 20-byte area of any memory space. If program code is located in these areas, the memory area will be exceeded and a bus access for instruction pre-reading may be initiated. A case in which this is a problem is shown below.

Address		
0x03FFFFFF8	ADD R1,R4	PC (program counter)
0x03FFFFFFA	JMP @R2	
Area 0 0x03FFFFFFC	NOP	
0x03FFFFFFE	NOP	
<hr/>		
Area 1 0x040000000		
0x40000002		Instruction prefetch address

Table 78: Example

*Table 78* illustrates a case in which the instruction (ADD) indicated by the program counter (PC) and the address 0x0400002 instruction prefetch are executed simultaneously. Note that the program branches to an area outside Area 1 after executing the following JMP instruction and delay slot instruction.

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

**Instruction prefetch side effects**

- 1 It is possible that an external bus access caused by an instruction prefetch may result in misoperation of an external device, such as a FIFO, connected to the area concerned.
- 2 If there is no device to reply to an external bus request caused by an instruction prefetch, hangup will occur.

**Remedies**

- 1 These illegal instruction fetches can be avoided by using the MMU.
- 2 The problem can be avoided by not locating program code in the last 20 bytes of any area.



# Index

## A

ADD 209, 214-215, 261  
ADDC 216  
ADDV 217  
AND 186, 199-201, 218-220  
AND.B 220

## B

Backus-Naur Form xiii  
BF 221, 223  
BNF. See Backus-naur Form.  
BRA 225  
BRAf 226  
BREAK 227  
BRK 227  
BSR 209, 228  
BSRF 209, 230  
BT 231, 233

## C

CMPGT 267

## D

DIV0S 247  
DIV1 249

DMULS.L 250  
DMULU.L 251  
DT 252

## E

ELSE 192  
EXTS.B 253  
EXTS.W 254  
EXTU.B 255  
EXTU.W 256

## F

FABS 257-258  
FABS.D 204  
FABS.S 204  
FADD 211, 259-261  
FADD.D 204  
FADD.S 204  
FCMPEQ.D 205  
FCMPEQ.S 205  
FCMPGT.D 205  
FCMPGT.S 205  
FCNV.DS 205  
FCNV.SD 205  
FCNVDS 268, 270

FCNVSD 269-270  
FDIV 271-273  
FDIV.D 204  
FDIV.S 204  
FIPR 275-276, 278  
FIPR.S 206, 276  
FLDI 280-281  
FLDS 279  
FLOAT 282-284  
FLOAT.LD 205  
FLOAT.LS 205  
FMAC 285  
FMAC.S 205, 286-287  
FMOV 290-293, 295-298, 300-304,  
306-315  
FMOV.S 296-298, 300, 310-312  
FMUL 316-318  
FMUL.D 204  
FMUL.S 204  
FNEG 319-320  
FNEG.D 205  
FNEG.S 205  
FOR 184-185, 188, 192, 196, 199, 201  
FPU 194, 203, 261, 264, 267, 270, 273,  
276-277, 284, 286, 318, 325, 329,  
331-332, 334-335  
FPUDIS 257-260, 262-263, 265-266,  
268-269, 271-272, 276, 279-283, 285,  
290-292, 294-297, 299-303, 305-317,  
319-324, 326-328, 330-331, 334,  
353-356, 459-462  
FPUExc 212  
FPUL 194, 268-269, 279, 282-284, 326,  
330-331, 355-356, 461-462  
FROM 192  
FSQRT 323-325  
FSQRT.D 205  
FSQRT.S 205  
FSTS 326  
FSUB 327-329  
FSUB.D 204  
FSUB.S 204  
FTRC 330-331  
FTRC.DL 205  
FTRC.SL 205  
FTRV 333-334, 336  
FTRV.S 206, 334  
Function  
  Bit(i) 188  
  DataAccessMiss(address) 197, 200  
  ExecuteProhibited(address) 197  
  FABS\_D 204  
  FABS\_S 204  
  FADD\_D 204  
  FADD\_S 204, 212  
  FCMPEQ\_D 205  
  FCMPEQ\_S 205  
  FCMPGT\_D 205  
  FCMPGT\_S 205  
  FCNV\_DS 205  
  FCNV\_SD 205  
  FDIV\_D 204  
  FDIV\_S 204  
  FIPR\_S 206  
  FLOAT\_LD 205  
  FLOAT\_LS 205  
  FloatRegister32(i) 189  
  FloatRegister64(i) 189  
  FloatRegisterMatrix32(a) 189  
  FloatRegisterPair32(a) 189  
  FloatRegisterVector32(a) 189  
  FloatValue32(r) 189  
  FloatValue64(r) 189  
  FloatValueMatrix32(r) 189  
  FloatValuePair32(r) 189  
  FloatValueVector32(r) 189  
  FMAC\_S 205  
  FMUL\_D 204  
  FMUL\_S 204

FNEG\_D 205  
FNEG\_S 205  
FpuCauseE0 203  
FpuCauseI0 203  
FpuCauseO0 203  
FpuCauseU0 203  
FpuCauseV0 203  
FpuCauseZ0 203  
FpuEnableI0 203  
FpuEnableO0 203  
FpuEnableU0 203  
FpuEnableV0 203  
FpuEnableZ0 203  
FpuFlagI0 203  
FpuFlagO0 203  
FpuFlagU0 203  
FpuFlagV0 203  
FpuFlagZ0 203  
FpuIsDisabled0 203  
FSQRT\_D 205  
FSQRT\_S 205  
FSUB\_D 204  
FSUB\_S 204  
FTRC\_DL 205  
FTRC\_SL 205  
FTRV\_S 206  
InstFetchMiss(address) 197  
InstInvalidateMiiss(address) 197  
IsLittleEndian0 198  
MalformedAddress(address) 197,  
199-201  
MMU0 197, 199-201  
OCBI(address) 202  
OCBP(address) 202  
OCBWB(address) 202  
PrefetchMemory(address) 200  
PREFO(address) 200, 202  
ReadMemoryLown(address) 200  
ReadMemoryrn(address) 198-199  
ReadMemoryPairn(address) 198-199  
ReadProhibited(address) 197, 199-200  
Register(i) 188

SignExtendn(i) 187  
WriteControlRegister(index, value) 201  
WriteMemoryLown(address, value) 201  
WriteMemoryrn(address, value) 200-201  
WriteMemoryPairn(address, value)  
200-201  
WriteProhibited(address) 197, 201  
ZeroExtendn(i) 187

## I

IADDERR 226, 230  
IF 192, 199-201, 212  
ILLSLOT 208-209, 221-223, 225-226,  
228, 230-233, 337-338, 390, 401, 403,  
425, 427, 442, 475  
INT 186  
ISA 207-208, 226, 230, 338

## J

JMP 337  
JSR 209, 338

## L

LDC 339-352, 443-449  
LDC.L 340, 346-352  
LDS 209, 348, 353-362  
LDS.L 354, 356, 358, 360, 362

## M

MAC.L 365  
MAC.W 367  
MACH 194, 235, 250-251, 357-358, 365,  
367, 407-409, 463-464  
MACL 194, 235, 250-251, 359-360, 365,  
367, 407-409, 465-466  
MD 194  
MEM 195-196, 199, 201  
MMU 197-201

MOV 369-402  
 MOV.B 371-380  
 MOV.L 381-391  
 MOV.W 392-402  
 MOVA 403  
 MOVCA.L 404  
 MOVT 406  
 MUL.L 407  
 MULS.W 408  
 MULU.W 409

**N**

NEG 410  
 NEGC 411  
 NOT 186, 200, 413

**O**

OCBI 202, 414  
 OCBP 202, 415  
 OCBWB 202, 416  
 OR 186, 199, 201, 417-419  
 OR.B 419

**P**

P0 204-206  
 PC 194-195, 207-209, 221, 223, 225-226,  
 228, 230-231, 233, 390, 401, 403  
 PR 194, 207-209, 228, 230, 338,  
 361-362, 427, 467-468  
 PREF 420  
 PREFO 200, 202

**R**

RADDERR 199  
 READPROT 199  
 Register  
 DR 195

FPSCR 194, 202-206, 212, 259-261,  
 263, 266, 268-273, 276, 282-283,  
 285-286, 316-318, 321-325,  
 327-331, 334-335, 353-354,  
 459-460  
 FPSCR.CAUSE.E 203  
 FPSCR.CAUSE.I 203  
 FPSCR.CAUSE.O 203  
 FPSCR.CAUSE.U 203  
 FPSCR.CAUSE.V 203  
 FPSCR.CAUSE.Z 203  
 FPSCR.DN 261, 263, 266, 270, 273,  
 276, 286, 318, 325, 329, 331,  
 334-335  
 FPSCR.ENABLE.I 203  
 FPSCR.ENABLE.O 203  
 FPSCR.ENABLE.U 203  
 FPSCR.ENABLE.V 203  
 FPSCR.ENABLE.Z 203  
 FPSCR.FLAG.I 203  
 FPSCR.FLAG.O 203  
 FPSCR.FLAG.U 203  
 FPSCR.FLAG.V 203  
 FPSCR.FLAG.Z 203  
 FPSCR.FR 321  
 FPSCR.RM 259-260, 268-269,  
 271-272, 282-283, 285, 316-317,  
 323-324, 327-328, 330-331  
 FPSCR.SZ 322  
 FR 285, 321  
 GBR 194, 220, 339-352, 374, 379, 384,  
 389, 395, 400, 419, 443-458, 478,  
 481  
 MTRX 195  
 R 219-220, 239, 295, 300, 306, 309,  
 312, 315, 373-375, 378-380,  
 383-384, 388-389, 394-396,  
 399-400, 402-404, 418-419,  
 477-478, 480-481  
 Rm 214, 216-218, 238, 240-243,  
 246-247, 249-251, 253-256,  
 307-315, 339-362, 365, 367, 369,



371-373, 376-378, 380-383,  
385-388, 391-394, 396-399, 402,  
407-411, 413, 417, 430, 433,  
443-449, 469-473, 476, 479, 482

SR 194, 202-203

SR.FD 203

REPEAT 192

ROTCL 421

ROTCR 422

ROTL 423

ROTR 424

RTLBMIS 199

## S

SHAD 430

SHAL 431

SHAR 432

SHLD 433

SHLL 434-437

SHLR 438-442

SLEEP 201-202

SLOTFPUDIS 257-260, 262-263,  
265-266, 268-269, 271-272, 276,  
279-283, 285, 290-292, 294-297,  
299-303, 305-317, 319-324, 326-328,  
330, 334, 353-356, 459-462

STC 450-458

STC.L 450-456, 458

STEP 192

STS 209, 459-468

STS.L 460, 462, 464, 466, 468

SUB 329, 469

SUBC 470

SUBV 471

SuperH SH-Series  
documentation suite  
notation xiii

SWAP.B 472

SWAP.W 473

SZ 322

## T

TAS.B 474

The appendix 515

THROW 193, 199, 201, 212

TRAPA 475

TST 476-478

TST.B 478

## U

UNDEFINED 190-191

## W

WRITEPROT 201

WTLBMIS 201

## XYZ

XMTRX 333-334

XOR 186, 479-481

XOR.B 481

XTRCT 482

